Parallel FIM using Hadoop

Suraj Ghadge[1], PravinDurge[2], Vishal Bhosale[3], Sumit Mishra[4]
[1, 2, 3, 4] Department of Computer Engineering
[1, 2, 3, 4] JSPM's ICOER

Abstract- *Frequent itemsets mining (FIM)is the initial step for deriving association rules. The problem related to FIM is its significant amount of time consumption. When datasets become excessively large, single machine algorithms do not perform up to the mark. Speeding up FIM is indispensable and several algorithms have been developed to do the same. Existing algorithms lack a mechanism that brings automatic parallelization, load balancing, data distribution and fault tolerance. In this paper, we introduce an algorithm for FIM process which solves the performance problem of existing algorithms when exposed to huge datasets. This algorithm incorporates parallelism with the use of Hadoop'smapreduce.*

Keywords- Frequent Itemsets Mining, Parallel Mining Algorithms, Data Mining, MapReduce, Hadoop

I. INTRODUCTION

Data mining is a process of discovering and deriving previously unknown and useful information from huge 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 mining, 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 FIM is the most time-consuming procedure, it plays an essential rolein mining association rules.

The two well-known categories of existing FIM algorithms are Apriori [1] and FP growth[2] algorithms. Aprioriis an algorithm using the generate-and-test process that generates a large number of candidate itemsets. Apriorihas to repeatedly scan an entire database which is indeed time consuming. To reduce the time required for scanning databases, FP-growth algorithm was developed which avoids generating candidate itemsets.

The scalability problem was addressed by Apriori and FP-growth like parallel FIM algorithms. However, they had their limitations too. Apriori like parallel FIM algorithms suffer potential problems of high I/O and synchronization overhead, which makes it strenuous to scale up these parallel algorithms. And the major disadvantage of FP-growth like parallel algorithms lies in the infeasibility to construct in-memory FP trees to accommodate large-scale databases.

Rather than considering Apriori and FP-growth, we introduce a parallel algorithm using Hadoop'smapreduce[3]-[8]which solves the problem related scalability and performance.

II. BACKGROUND

In this section, we briefly explain association rules. Then basic idea of Hadoop'sMapreduce programming model.

A. Association Rules

Association rules mining(ARM) provides a strategic resource for decision support by extracting the most important frequent patterns that frequently occur in large transactional database. A typical example of ARM is market basket analysis. An association rule, for example, can be "if a customer buys *A* and *B*, then 70% of them also buy *C*." In this example, 70% is the confidence of the rule.

Apart from confidence, support is another measure of association rules, each of which is an implication in the form of$X \Rightarrow Y$. Here, X and Y are two itemsets, and $X \cap Y = \varnothing$. The confidence of a rule $X \Rightarrow Y$ is defined as a ratio between support$(X \cup Y)$ and support(X). Note that, an itemset$X$has support s if s% of transactions contain the itemset. We denote s = support(X); the support of the rule $X \Rightarrow Y$ is support $(X \cup Y)$.

The ultimate objective of ARM is to discover all rules that satisfy a user-defined minimum support and minimum confidence. The ARM process can be divided into two phases:1) identifying all frequent itemsets whose support is greater than the minimum support and 2) forming conditional implication rules among the frequent itemsets. The first phase is more challenging and complicated than the second one. As such, most prior studies are primarily focused on the issue of discovering frequent itemsets.

B. MapReduce Framework

MapReduce is a parallel and scalable programming model for data-intensive applications and scientific analysis. A MapReduce program expresses a large distributed computation as a sequence of parallel operations on datasets of key/value pairs. A MapReduce computation has two phases, namely, the Map and Reduce phases. The Map phase splits the input data into a large number of fragments, which are evenly distributed to Map tasks across the nodes of a cluster to process. Each Map task takes in a key-value pairand then generates a set of intermediate key-value pairs. After the MapReduce runtime system groups and sorts all the intermediate values associated with the same intermediate key, the runtime system delivers the intermediate values to Reduce tasks. Each Reduce task takes in all intermediate pairs associated with a particular key and emits a final set of key value pairs. Both input pairs of Map and the output pairs of Reduce are managed by an underlying distributed file system.
MapReduce greatly improves programmability by offering automatic data management, highly scalable, and transparent fault-tolerant processing. Also, MapReduce is running on clusters of cheap commodity servers—an increasingly attractive alternative to expensive computing platforms. Thanks to the aforementioned advantages, MapReduce has been widely adopted by companies like Google, Yahoo, Microsoft, and Facebook. Hadoop—one of the most popular MapReduce implementations—is running on clusters where Hadoop distributed file system (HDFS) stores data to provide high aggregate I/O bandwidth. At the heart of HDFS is a single NameNode—a master server that manages the file system namespace and regulates access to files. The Hadoop runtime system establishes two processes called JobTracker and TaskTracker. JobTracker is responsible for assigning and scheduling tasks; each TaskTracker handles Map or Reduce tasks assigned by JobTracker.

III. MAPREDUCE-BASED ALGORITHM

In light of the Hadoop'sMapReduce programming model, we introduce a parallel FIM algorithm that enables automatic parallelization, load balancing and data distribution for parallel mining of frequent itemsets on large clusters.

Aiming to improve data storage efficiency and to avert building conditional pattern bases, this algorithm incorporates the concept of ultrametric tree rather than traditional FP trees.

A. First MapReduce Job

The first MapReduce job discovers all frequent items or frequent one-itemsets (see Algorithm 1). In this phase, the input of Map tasks is a database, and the output of Reduce tasks is all frequent one-itemsets.

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 >minsupportthen
18: F − list ← the (1-itemset, sum) //F-list is a CacheFile
is   storingfrequent 1-itemsets and their count.
19: end if
20: end function
```

B. Second MapReduce Job

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 frefrquent
   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
   itemset
20: end function
```

C. Third MapReduce Job

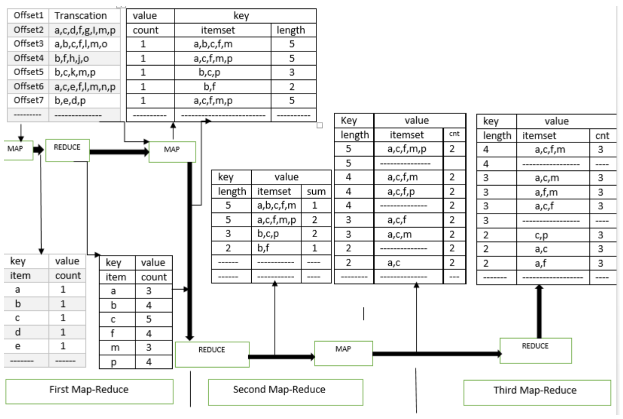
The second phase of FIUT involving the construction of a *k*-FIU(ultrametric) tree and the discovery of frequent *k*-itemsets is handled by a third MapReduce job, in which *h*-itemsets ($2 <= h <= M$) are directly decomposed into a list of ($h - 1$)-itemsets, ($h - 2$)-itemsets,*etc.*In the third MapReduce 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 decompose each
De-itemset into t-itemsets (t is    f   from 2 to 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-Iteitemset(i.e. t ≤
k);
6: for all ( t-itemset ) do
7: $t - FIU - tree$ ← t-FIU-tree generation(local- F   I  FIU
tree, 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   each
mapper;
15: for all (each leaf with item name v in t-FIU-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
```

The decomposition procedure of each mapper is independent of other mappers and this fact makes third mapreduce task highly scalable. In other words multiple mappers can perform decomposition process parallel. This improves data storage efficiency and I/O performance.[9] Figure 1 illustrates the overview of the algorithm with sample example.



Offset1	Transcation
Offset2	a,c,d,f,g,I,m,p
Offset3	a,b,c,f,l,m,o
Offset4	b,f,h,j,o
Offset5	b,c,k,m,p
Offset6	a,c,e,f,l,m,n,p
Offset7	b,e,d,p
---------	--------------

value	key	
count	itemset	length
1	a,b,c,f,m	5
1	a,c,f,m,p	5
1	b,c,p	3
1	b,f	2
1	a,c,f,m,p	5
----------	------------------------	----------

key	value
item	count
a	1
b	1
c	1
d	1
e	1
-------	------

key	value
item	count
a	3
b	4
c	5
f	4
m	3
p	4

key	value	
length	itemset	sum
5	a,b,c,f,m	1
5	a,c,f,m,p	2
3	b,c,p	2
2	b,f	1
------	-----------	----
------	-----------	----

Key	value	
length	itemset	cnt
5	a,c,f,m,p	2
5	--------------	
4	a,c,f,m	2
4	a,c,f,p	2
4	-------------	2
3	a,c,f	2
3	a,c,m	2
2	-------------	2
2	a,c	2
--------	--------------	---

key	value	
length	itemset	cnt
4	a,c,f,m	3
4	----------------	----
3	a,c,m	3
3	a,f,m	3
3	a,c,f	3
3	----------------	----
2	c,p	3
2	a,c	3
2	a,f	3
-------	-----------------	----

Fig1. Overview of MapReduce-based algorithm

IV. RELATED WORK

Parallel frequent itemsets mining algorithms based on *Apriori* can be classified into two camps, namely, count distribution (e.g., count distribution (CD), fast parallel mining, and parallel data mining (PDM)) and data distribution (e.g., data distribution (DD)and intelligent data distribution). 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-growth based parallel algorithms were proposed as a replacement of *Apriori*-based parallel algorithms. A few parallel FP-growth-based parallel algorithms 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.

V. 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 processorhas to scan a database multiple times and to exchange anexcessive 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.

A major disadvantage of FP-growth like parallel algorithms, however, lies in the infeasibility to constructin-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[10] to develop a parallel frequent itemsets mining algorithm. Itachieves compressed storage and avoiding the necessity to build conditional pattern bases. This algorithm seamlessly integrates three MapReduce jobs to accomplish parallel mining of frequent itemsets. The third MapReduce jobplays an important role in parallel mining; its mappers independently decompose itemsets whereas its reducers construct small ultrametric trees to be separately mined and hence improving performance for FIM.

REFERENCES

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

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

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

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

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

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

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

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

[9] D. W. Cheung, S. D. Lee, and Y. Xiao, "Effect of data skewness andworkload balance in parallel data mining," IEEE Trans. Knowl. DataEng., vol. 14, no. 3, pp. 498–514, May/Jun. 2002.

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