American Journal of Applied Sciences 2 (5): 926-931, 2005
ISSN 1546-9239

Model for Load Balancing on Processors in Parallel Mining of Frequent Itemsets

[1]Ravindra Patel, [2]S.S. Rana and [3]K.R. Pardasani
[1]Department of Master in Computer Applications
Government Geetanjali Girl's College, Bhopal (M.P.) India
[2]University Institute of Computer Science and Applications
Rani Durgawati University Jabalpur (M.P.) India
[3]Department of Mathematics and Computer Applications
Maulana Azad National Institute of Technology, Bhopal (M.P), India

Abstract: The existence of many large transactions distributed databases with high data schemas, the centralized approach for mining association rules in such databases will not be feasible. Some distributed algorithms have been developed [FDM, CD], but none of them have considered the problem of data skews in distributed mining of association rules. The skewness of datasets reduces the workload balancing between processors involved in distributed mining of association rules. It is important to invent an efficient approach for distributed mining of association rules which have the ability to generate homogeneous partitions of the whole data sets; hence the supports of most large item sets are distributed evenly across the processors. We proposed an efficient stratified sampling based partitioned technique, which generate homogeneous partitions on which processors works in parallel and generate their local concepts approximately simultaneously.

Key words: Association rules, Data Mining, Data Skewness, Workload Balance, Parallel Mining, Partitioning, Stratified Sampling

INTRODUCTION

Association rule mining finds interesting associative or correlative relationships among a large set of data items. The problem was formulated originally in the context of transaction data at the supermarket.

This market basket data consisst of transactions made by each customer. Each transaction contains items bought by the customer. The goal is to see if the occurrence of certain items in a transaction can be used to deduce occurrence of other items or in other words, to find associative relationships between items. If such interesting relationships are found, then they can be put to various profitable uses such as self management, inventory management, etc. Thus association rules were born[1].

Let $I = \{ I_1, I_2, \ldots\ldots\ldots, I_m\}$ be a set of items. Let *D*, be a set of database transactions where each transaction *T* is a set of items such that $T \subseteq I$. Each transaction is associated with an identifier, called *T_ID* (transaction identity).

Let *A* be a set of items. A transaction *T* is said to contain A if and only if $A \subseteq T$. An association rule is an implication of the form A =>B, where $A \subset I$, $B \subset I$ and $A \cap B = \varnothing$.

The rule $A => B$ holds in the transaction set *D* with supports, where *s* is the percentage of transactions in *D* that contain $A \cup B$. This is taken to be the probability, $P(A \cup B)$.

The rule $A => B$ has confidence *c* in the transaction set D if *c* is the percentage of transactions in D containing A that also contain B. This is taken to be the conditional probability, $P(B/A)$. That is:

$$support\ (A => B) = P(A \cup B)$$

$$confidence\ (A => B) = P(B/A)$$

Rules that satisfy both a minimum support threshold *(min_sup)* and a minimum confidence threshold *(min_conf*) are called strong. An utmost that contain k items is a *k-item sets*. The set, {bread, butter} is a 2-item set. The occurrence frequency of an itemset is the number of transactions that contain the itemset. This is also known as frequency of support count. An itemset satisfies minimum support, if the occurrence frequency of the utmost is greater than or equal to the product of min_sup and the total number of transactions in D. If an item set satisfies minimum support, then it is a frequent item set. The set of frequent k-item sets are commonly denoted by L_k. Association rule mining is a two-step process. Find all frequent itemsets and generate strong association rules from the frequent itemsets.

1. F_1 = {frequent 1-itemsets};
2. for (k = 2; $F_{k-1} \neq \phi$; k++) {
3. C_k = apriori_gen (F_{k-1})
4. for all transactions t ∈ T {
5. subset (C_k, t)
6. }
7. $F_k = \{c \in C_k \mid c.count \geq minsup\}$
8. }
9. Answer = $\bigcup F_k$

Fig.1: Apriori Algorithm

In this study, we concentrate on the most time consuming process, which is the discovery of frequent item set. The first algorithm that handled the problem of generation of the frequent item set was the *Apriori algorithm*[2]. This algorithm used a very fundamental property for the support of item sets: An item set of size k can meet the minimum level of support only if all of its subsets also meet the minimum level of support. This property used to systematically prune the search space of desired itemsets, by increasing the length of the itemsets being discovered. In an iteration k, all candidate k-itemsets are formed such that all its (k-1) subsets are frequent. The numbers of occurrences of these candidates are then counted in the transaction database. Efficient data structures are used to perform the fast counting. Since its conception, many others algorithm[3-10] have emerged that improve upon the runtime, I/O and scalability performance of the Apriori algorithm by various efficient means of pruning the itemset search space and counting the candidate occurrences in large databases.

We assume that the database is a transactional database with high data skewness. The database consists of the huge amount of transaction records, each with a transaction identifier (TID) and a set of data items. The data mining in such databases requires substantial processing power and parallel system is a possible solution. This observation motivates us to study efficient parallel algorithms for mining association rules in large databases. The database is partitioned 'horizontally' (i.e., grouped by transactions) and each partition generated by using *stratified sampling* to select a sample of transactions for a partition. Allocate these partitions to the processors of sites in distributed system which communicates via a fast network. It has been well known that the major cost of mining association rules is the computation of the set of large itemsets (i.e. Frequently occurring sets of items) in the database. An itemset (a set of items) is large if the percentage of transactions that containing all these items is greater than a given threshold.

Sequential Mining of Association Rules:

A priori Algorithm: The Apriori algorithm consists of a number of passes Initially F_1 contains all the items (i.e., Item set of size one) that satisfy the minimum support requirement. During pass *k*, the algorithm finds the set of frequent itemsets F_k of size *k* that satisfy the minimum support requirement. The algorithm terminates when F_k has satisfied the minimum support requirement. The algorithm terminates when F_k is empty. In each pass, the algorithm first generates C_k the candidate itemsets of size *k*. Function *apriori_gen* (F_{k-1}) constructs C_k by extending frequent itemsets of size k – 1. This ensures that all the subsets of size $k-1$ of a new candidate itemset are in F_{k-1}. Once the candidate itemsets are found, their frequencies are computed by counting how many transactions contain these candidate itemsets.

Finally, F_k is generated by pruning C_k to eliminate itemsets with frequencies smaller than the minimum support. The union of the frequent itemsets, $\bigcup F_k$, is the frequent itemsets from which we generate association rules.

Computing the counts of the candidate itemsets is the most computationally expensive steps of the algorithm.

Parallel and Distributed Mining: The Count Distribution (CD) algorithm is a simple data parallelization algorithm. The database D is positioned horizontally into *D1, D2..Dn* and distributed across n processors P_i ($1 \leq i \leq n$). It uses sequential Apriori algorithms on each partition. The CD algorithm's main advantage is that it does not exchange data tuples between processors, it only exchange counts. In the first database scan, each processor generates its local candidate itemsets depending on the items present in its local partition. The algorithm obtains global support counts by exchanging local support count with all other processors. The algorithm communication overhead is O (|c|. n) at each phase, where |c| and n are the size of candidate itemsets and the number of data sets, respectively.

Researchers proposed FDM (Fast Distributed Mining) algorithms to mine association rules from distributed datasets partitioned among different sites[8]. At each site, FDMK find the local support counts and prunes locally in frequent itemsets. After completing local pruning, each site broadcasts messages containing all the remaining candidate sets to all other sites to collect their support counts. It then decides whether locally large itemsets are globally large and generates the candidate itemsets from those globally large itemsets.

The FDM's main advantages over CD is that it reduces the communication overhead to O ($|c_p|$. n), where $|c_p|$ and n are number of large itemsets and the number of sites. FDM generates fewer candidate itemsets compared to CD, when the number of disjoint candidate itemsets among various sites is large. However, we can achieve this when different sites have non homogenous data sets. The FDM's message optimization technique requires some functions to

determine the polling site, which could cause extra computational cost when each site has numerous local frequent itemsets.

All of the parallel approaches optimize message to reduce communication costs, but none of the parallel algorithm has considered the problem of partitioned database with high data schemas. Whenever, the partitioned database with high data skews increases computational cost and reduces workload balancing of processors and hence in such situation a parallel algorithm works look likes a sequential. Hence without consideration of problems with data schemas, we can't achieve the advantages of parallelization of a mining algorithm.

Proposed algorithm, WBDM (Workload Balanced Distributed Mining) deals with the problem of data skews and workload balancing by using a stratified sampling method to partition the database.

Data Skewness and Workload Balance: A partioned database has high data skewness if most globally large itemsets are locally large only at a very few partitions. It is low if most globally large itemsets are locally large evenly across the processors. When the clustering of different large itemsets distributed evenly across the processors; hence each processor would have similar numbers of locally large itemsets. This case characterizes as high workload balance. When the clustering of different large itemsets concentrated on a few processors; hence some processes would have much more locally large itemsets than the others. This is a case of low workload balance. When the clustering of different large itemsets distributed not evenly across the processors, then the pruning effects would be reduced significantly and the work of computing the large itemsets would be concentrated on a few processors which is a very troublesome issue of parallel computation.

For example Table 1 shows an example of high data skews and low workload balance. The global threshold is 15 and the local support threshold at each processor is 5.

Table 1: High Data Skewness and Low Workload Balance Case

Items	A	B	C	D	E	F
Local support at proc-1	13	33	12	34	2	1
Local support at proc-2	1	3	1	2	1	4
Local support at proc-3	2	1	2	1	12	33
Global support	16	37	15	37	15	38
Globally large at proc-1	√	√	√	√	×	×
Globally large at proc- 2	×	×	×	×	×	×
Globally large at proc-3	×	×	×	×	√	√

In this case, distributed pruning will generate 7 sizes-2 candidates, namely AB, AC, AD, BC, BD, CD and EF, while the CD will have 15 candidates. Thus distributed pruning to be very effective, but most globally large itemsets are locally large only at processor1, hence have lower high workload balance.

For example Table2 shows an example of low data skews and high workload balance. The support counts of items A, B, C, D, E and F are almost equally distributed over three processors. Hence the data skews are low. On the other hand the workload balance is high, because the number of locally large itemsets in each processor is almost the same. In this case, both CD and distributed pruning will generate the same 15 candidate sets; however, global pruning can prune away the three candidates AC, AE and CE. Hence FPM still has 20% of improvement over CD in the case of Low data skews and high workload balance.

Stratified Sampling Based Partitions: With stratified random sampling, the whole database is divided into a number of parts or 'strata' according to some characteristic. Simple random samples are then selected from each stratum. The same proportion will be selected within each stratum, making the sample a proportionate stratified random sample. Stratified sampling can be used as a data partition technique, it allows a high skewed data set can be partitioned as homogeneous portions

Let *DB* be a database with *D* transactions. Assume that there is *N* processors $P_1, P_2 \ldots P_N$ in a distributed environment. The database divided into *N* stratum $DB_1, DB_2 \ldots DB_N$ each with *D/N* transactions.

Simple random samples $S_{i,j}$ $(j=1..N)$, each with D/N^2 transactions selected from each stratum DB_i (i=1..N). Thus N partitions DS_i, with homogeneous data of size DI (=*D/N*) for i=1..N, can be generated as:

$$DS_i = S_{i,1} \bigcup S_{i,2} \ldots \bigcup S_{i,N\ (i=1..N)}$$

Such that:

$$DS_1 \bigcup DS_2 \ldots\ldots \bigcup DS_N = DB \text{ and}$$
$$DS_1 \cap DS_2 \ldots\ldots \cap DS_N = \phi$$

Maps these N partitions DS_i (i=1..N) to processors P_i $(i=1..N)$ respectively.

Table 2: Low Data Skewness and High Workload Balance Case

Items	A	B	C	D	E	F
Local support at proc-1	6	12	4	13	5	12
Local support at proc-2	6	12	5	12	4	13
Local support at proc-3	4	13	6	12	6	13
Global support	16	37	15	37	15	38
Globally large at proc-1	√	√	×	√	√	√
Globally large at proc- 2	√	√	√	√	×	√
Globally large at proc-3	×	√	√	√	√	√

In this technique the database of size *D* is divided into *N* mutually disjoint parts called strata, each of size *D/N*, a stratified sampling partition can be generated by obtaining a simple random sample of size *(D/N) /N (=D/N²)* from each stratum and N samples of size D/N^2 makes a sample of size *D/N* (= (D/N^2) *N) with homogeneous data. This helps to ensure a representative sample, especially when the data are skewed.

Distributed Approach for Generating Frequent Itemsets: Let the size of partitions DB_i be D_i *(=D/N)* for *i=1..N*. Let *X. sup* and *X. spy* be the support counts of an item sets *X* in *DB* and DS_i, respectively. *X.sup* is called global support count and $X.sup_i$ is called local support count of *X* at processor P_i. for a given minimum support threshold *s*, *X* is globally the largest if $X.sup \geq s \times D$ and X is locally large at P_i if $X.sup_i \geq s \times DS_i$.

Notations:

D Number of transactions in DB
s Support threshold min-sup
$L_{(k)}$ Globally large k-itemsets
$C_{(k)}$ Candidate sets generated from $L_{(k)}$
X. sup Global support count of X
$L_{i(k)}$ Locally large k-itemsets at S_i
$X.sup_i$ Local support count of X at S_i
$CG_{(k)}$ Candidate sets generated from $L_{(k-1)}$
Ti (j) Data structure to maintain the item set an their support count at the site S_i in jth iteration
D_i Number of transactions in DS_i

There is an important relationship between large itemsets and the partitions in allocating to distributed system: every globally large itemsets must be locally large at some partitions DS_i. If an itemset X is both globally large and locally large at a partition *DSi*, X is called gl_large at DS_i. Notice that at each partition DS_i, if a candidate set $X \in CG_{(k)}$ is not locally large at partition DS_i , there is no need for DS_i to find out its global support count to determine whether it is globally the largest. This is because in this case, either X is small (not globally large). Or it will be locally large at some other partition DS_i and hence only the partition DS_i at which *X* is locally large needed to be responsible to find the global support count of X. In the proposed approach since each processor has homogeneous data, hence generate approximately equal number of locally large itemsets simultaneously.

Workload Balanced Distributed Algorithm:
//Phase-I : Generation of homogeneous partitions:

Partition the database DB into N partitions DB_i (I = 1, 2……. N) each of size D/N
for (i = 1 to N)
{
for (j = 1 to N)
{
$DS_{i\,(j)}$ =random sample from DB_j of size D/N^2
}
$DS_i = DS_{i(1)} \cup DS_{i(2)} \cup \ldots\ldots\ldots \cup DS_{i(N)}$
}

//Phase-II: Generation of frequent itemsets:

if k = 1 then
$Tie_{(1)}$ = get_local_count (DS_i , ϕ, 1)
else
{
$CG_{(k)} = \bigcup_{i=1}^{N} CGi(k)$
$= \bigcup_{i=1}^{N} Apriory\_gen(GLi(k-1))$;
$Ti_{(k)}$ = get_local_count (DS_i , $CG_{(k)}$, i);
}
for_all X ∈ $T_{i(k)}$ do

if $X.sup_i \geq s \times D/N$ then
for j = 1 to N do
if polling_site (X) = P_i then
insert <X, $X.sup_i$) into $LL_{i,j(k)}$;
for j =1, ….. , N do
Send $LL_{i,\,j\,(k)}$ to processor P_i;
for j =1, ….. , N do
{
Receive $LL_{i,\,j\,(k)}$;
for_all X ∈ $LL_{i,\,j\,(k)}$ do
{
if X ∉ $LP_{i(k)}$ then
Insert X into $LP_{i\,(k)}$;
update X. large_processors ;
}
}
for_all X ∈ $LP_{i(k)}$ do
send_polling_request (X);
reply_polling_request ($T_{i\,(k)}$);
for_all X ∈ $LP_{i(k)}$ do
{
receive $X.sup_j$ from the processors P_j ,
where Pj ∉ X. large_processors;
$X.sup = X.sup_1 + X.\,sup_{2\ldots\ldots +}\, X.\,sup_N$;
if $X.sup \geq s \times D$ then
insert X into Gi(k) ;
}
broadcast $G_{i(k)}$;
receive $G_{j(k)}$ from all processors P_j (j ≠ i);
$L_{(k)} = \bigcup_{i=1}^{N} Gi(k)$.
Divide $L_{(k)}$ into GLi (k), (i=1,…, N);
return L(k)

Explanation of Algorithm:

* Phase1 create *N (N* is the no. of processors i.e. Sites) homogeneous partitions from the large high skew database.

*. **Home Site: Generate Candidate Sets and Submit Them to Polling Sites:** In the first iteration, the site S_i calls *get_local_count* to scan the partition DS_i once and store the local support counts of all the 1-itemsets found in the array $T_{i(1)}$. At the *k-th* (for k>1) iteration, S_i first computes the set of candidate set $CG_{(k)}$ and then scan DS_i to build the hash tree $T_{i(k)}$ containing the local support counts of all the sets in $CG_{(k)}$ by traversing $T_{i(k)}$, S_i finds out all locally large *k-itemsets* and group them according to their polling sites. Finally, it sends the candidate sets with their local support counts to their polling sites.

* **Polling Site: Receive Candidate Sets And Send Polling Requests:** As a polling site, the site S_i receives candidate sets from the other sites and insert them in $LP_{i(k)}$. For each candidate set $X \in LP_{i(k)}$, S_i stores all its "home " sites in *X. large_sites*, which contains all those sites from which X is sent to S_i for polling. In order to perform count exchange for *X*, S_i calls *send_polling_request* to send *X* to those sites not in the list *X. large_sites* to collect the remaining support counts.

* **Remote Site: Return Support Counts to Polling Site:** When S_i receives polling requests from the other sites, it acts as a remote site. For each candidate sets *Y* it receives from a polling site, it retrieves *Y. sup*$_i$ from the hash tree $T_{i(k)}$ and returns it to the polling site.

* **Polling Site: Receive Support Counts and Find Large Itemsets:** As a polling site, S_i receives local support counts for the candidate sets in $LP_{i(k)}$.. Following that it computes the global support counts of all these candidate set and find out the globally large itemsets among them. These globally large *k-itemsets* are stored in the set $GI_{(k)}$. Finally, S_i broadcasts the set $G_{i(k)}$ to all the other sites.

* **Home Site: Receives Large Itemsets:** As a "home "site, S_i receives the sets of globally large k-*itemsets* $G_{i(k)}$ from all the polling sites. By taking the union of $G_{i(k)}$, *(i=1,…..N)*, S_i finds out the set L_k of all the size-k large itemsets. Further S_i finds out from L_k the set $GL_{i(k)}$ of gl-large itemsets for each site by using the site list in *X. large_sites*. The set $GL_{i\ (k)}$ will be used for candidate set generation in the next iteration.

Comparison: Now let a sequential approach takes T time for support count of candidate sets in any iteration. And let processors, *Pi* (*i=1...N*) takes, T_i time to calculate support counts in allocating partition. Then on the proposed distributed approach each processor works on homogeneous partitions (have equal number of locally large itemsets), hence each processor P_i, performed their processing at same time *T/N* for *i=1..N*. Now since no any other distributed approach has considered a homogeneous partition technique of a database with high skins, hence must be most of globally large itemsets are locally large only on few processors. Thus the time required for processing in any other distributed approaches like Count Distribution (CD), Fast Distributed Algorithm (FDM) equal to Max (T_i , *i=1..N*), which will be greater than *T/N*.

CONCLUSION

We considered the problem of mining frequent itemsets on a shared–nothing multiprocessor environment on which data has been partitioned, across the nodes, by using stratified random sampling. An advantage of sampling for data partition is that the cost of obtaining a sample is propositional to the size of the sample, S, rather than the size of the datasets, D. Other data partition techniques can require at least one complete pass through D. This algorithm also attempts to minimize communication by allocating homogeneous partitions to each processor.

This algorithm is more efficient to mining frequent itemsets for those databases, whose size is very large and have high data skewness. Any parallel algorithm working on database with high data skews could not achieve the advantages of parallel processing, because most globally large itemsets clustered on few processors. The stratified random sampling used as partitioning approach balanced work load on each processor in a distributed environment.

REFERENCES

1. Agrawal, R., T. Imielinski and A. Swami, 1993. Mining association rules between sets of items in large databases. In Proc. Of 1993 ACM-SIGMOD Intl. Conf. On Management of Data, Washington, D.C., pp: 207-216.
2. Agrawal, R. and R. Srikant, 1994. Fast algorithms for mining association rules. In Proc. Of the 20th VLDB Conf., Santiago, Chile, pp: 487-499.
3. Park, J., Chen and P. Yu, 1995. An effective hash-based algorithm for mining association rules. In Proc. Of 1995 ACM-SIGMOD Int. Conf. On Management of Data, pp: 175-186.
4. Savasere, A., E. Omiecinski and S. Navathe, 1995. An efficient algorithm for mining association rules in large databases. In Proc. Of the 21st VLDB Conf., Zurich, Switzerland pp: 432-443.

5. Toivonen, H., 1996. Sampling large databases for association rules. In Proc. On the 22nd VLDB Conference, pp: 134-145.
6. Cheung, D.W. and Y. Xiao, 1998. Effect of data skews in parallel mining of association rules. In Proc. Of the 4th Pacific-Asia Conf. On Knowledge Discovery and Data Mining, New York, USA., pp: 48-60.
7. Zaki, M.J., S. Parthasarathy, M. Ogihara and W. Li, 1997. New algorithms for fast discovery of association rules. In Proc. Of the 3rd Int'l Conference on Knowledge Discovery and Data Mining AAAI Press.
8. Cheung, D.W., 1996. A fast distributed algorithm for mining association rules. In Proc. Parallel and Distributed Information Systems, IEEE CS Press, pp: 31-42.
9. Brin, S., R. Motwani, J.D. Ullman and S. Tsur, 1997. Dynamic itemset counting and implication rules for market basket data. In Proc. Of ACM-SIGMOD Intl. Conf. On Management of Data, Tucson, Arizona, pp: 255-264.
10. Agrawal, R.C., C. Agarwal and V.V.V. Prasad, 2000. A tree projection algorithm for generation of frequent itemsets. J. Parallel and Distributed Computing (Special Issue on High Performance Data Mining).