Effect of Data Skewness and Workload Balance in Parallel Data Mining*

David W. Cheung† S. D. Lee†(李守敦)
Yongqiao Xiao†

†Department of Computer Science and Information Systems, The University of Hong Kong, Hong Kong.

Email: {dcheung, sdlee, yqxiao}@csis.hku.hk.

Abstract

To mine association rules efficiently, we have developed a new parallel mining algorithm FPM on a distributed share-nothing parallel system in which data are partitioned across the processors. FPM is an enhancement of the FDM algorithm, which we proposed previously for distributed mining of association rules [8]. FPM requires fewer rounds of message exchanges than FDM and hence has a better response time in a parallel environment. The algorithm has been experimentally found to outperform CD, a representative parallel algorithm for the same goal [2]. The efficiency of FPM is attributed to the incorporation of two powerful candidate sets pruning techniques : distributed and global prunings. The two techniques are sensitive to two data distribution characteristics, data skewness and workload balance. Metrics based on entropy are proposed for these two characteristics. The prunings are very effective when both the skewness and balance are high. In order to increase the efficiency of FPM, we have developed methods to partition a database so that the resulting partitions have high balance and skewness. Experiments have shown empirically that our partitioning algorithms can achieve these aims very well, in particular, the results are consistently better than a random partitioning. Moreover, the partitioning algorithms incur little overhead. So, using our partitioning algorithms and FPM together, we can mine association rules from a database efficiently.

Keywords: Association Rules, Data Mining, Data Skewness, Workload Balance, Parallel Mining, Partitioning.

*This research is supported in part by RGC (the Hong Kong Research Grants Council) grant, project number HKU 7023/98E.

1 Introduction

Mining *association rules* in large databases is an important problem in data mining research [1, 2, 4, 6, 11, 12, 15, 17, 19, 20, 24]. It can be reduced to finding *large itemsets* with respect to a given *support threshold* [1, 2]. The problem demands a lot of CPU resources and disk I/O to solve. It needs to scan all the transactions in a database which introduces much I/O, and at the same time search through a large set of candidates for large itemsets which requires a lot of CPU computation. Thus, parallel mining may deliver an effective solution to this problem. In this paper, we study the behavior of parallel association rule mining algorithms in parallel systems with a share-nothing memory. In this model, the database is partitioned and distributed across the local disks of the processors. We investigate how the partitioning method affects the performance of the algorithms, and then propose new partitioning methods to exploit this finding to speed up the parallel mining algorithms.

The prime activity in finding large itemsets is the computation of *support counts* of candidate itemsets. Two different paradigms have been proposed for parallel system with distributed memory for this purpose. The first one is *count distribution* and the second one is *data distribution* [3]. Algorithms that use the count distribution paradigm include CD (Count Distribution) [3] and PDM (Parallel Data Mining) [18]. Algorithms which adopt the data distribution paradigm include DD (Data Distribution) [3], IDD (Intelligent Data Distribution) [10] and HPA (Hash Based Parallel) [23].

In the count distribution paradigm, each processor is responsible for computing the *local support counts* of *all* the candidates, which are the support counts in its partition. By exchanging the local support counts, all processors then compute the *global support counts* of the candidates, which are the total support counts of the candidates from all the partitions. Subsequently, large itemsets are computed by each processor independently. The merit of this approach is the simple communication scheme: the processors need only one round of communication in every iteration. This makes it very suitable for parallel system when considering response time. CD [3] is a representative algorithm in count distribution. It was implemented on an IBM SP2. PDM [18] is a modification of CD with the inclusion of the direct hashing technique proposed in [17]. In the count distribution approach, every processor is required to keep the local support counts of *all* the candidates at each iteration, one possible problem is the space required to maintain the local support counts of a large number of candidate sets.

In the data distribution paradigm, to ensure enough memory for the candidates, each processor is responsible for keeping the support counts of *only a subset* of the candidates. However, transactions (or their subsets) in different partitions must then be sent to other processors for counting purpose. Comparing with sending support counts, sending transaction data requires a lot more communication bandwidth. DD

(Data Distribution) is the first proposed data distribution algorithm [3]. It has been implemented on an IBM SP2. The candidates in DD are distributed equally over all the processors in a round-robin fashion. Then every processor ships its database partition to all other processors for support counts computing. It generates a lot of redundant computation, because every transaction is processed as many times as the number of processors. In addition, it requires a lot of communication, and its performance is worse than CD [3]. IDD (Intelligent Data Distribution) and its variant HD (Hybrid Distribution) are important improvements on DD [10]. They partition the candidates across the processors based on the first item of a candidate. Therefore, each processor only needs to handle the subsets of a transaction which begin with the items assigned to the processor. This reduces significantly the redundant computation in DD. HPA (Hash Based Parallel), which is very similar to IDD, uses a hashing technique to distribute the candidates to different processors [23].

One problem of data distribution that needs to be noted: it requires at least two rounds of communication in each iteration at each processor — to send its transaction data to other processors; to broadcast the large itemsets found subsequently to other processors for candidate sets generation of the next iteration. This two-round communication scheme puts data distribution in an unfavorable situation when considering response time.

In this work, we investigate parallel mining employing the count distribution approach. This approach requires less bandwidth and has a simple one-round communication scheme. To tackle the problem of large number of candidate sets in count distribution, we adopt two effective techniques, *distributed pruning* and *global pruning* to prune and reduce the number of candidates in each iteration. These two techniques make use of the local support counts of large itemsets found in an iteration to prune candidates for the next iteration. These two pruning techniques have been adopted in a mining algorithm FDM (Fast Distributed Mining) previously proposed by us for distributed databases [7, 8]. However, FDM is not suitable for parallel environment, it requires at least two rounds of message exchanges in each iteration that increases the response time significantly. We have adopted the two pruning techniques to develop a new parallel mining algorithm FPM (Fast Parallel Mining), which requires only one round of message exchange in each iteration. Its communication scheme is as simple as that in CD, and it has a much smaller number of candidate sets due to the pruning. In the rare case that the set of candidates are still too large to fit into the memory of each processor even after the pruning, we can integrate the pruning techniques with the algorithm HD into a 2-level cluster algorithm. This approach will provide the scalability to handle candidate sets of any size and at the same maintain the benefit of effective pruning. (For details, please see discussion in Section 7.2).

In this paper, we focus on studying the performance behavior of FPM and CD. It depends heavily on

the distribution of data among the partitions of the database. To study this issue, we first introduce two metrics, *skewness* and *balance* to describe the distribution of data in the databases. Then, we analytically study their effects on the performance of the two mining algorithms, and verify the results empirically. Next, we propose algorithms to produce database partitions that give "good" skewness and balance values. In other words, we propose algorithms to partition the database so that good skewness and balance values are obtained. Finally, we do experiments to find out how effective these partitioning algorithms are.

We have captured the distribution characteristics in two factors : *data skewness* and *workload balance*. Intuitively, a partitioned database has high data skewness if most globally large itemsets[1] are locally large only at a very few partitions. Loosely speaking, a partitioned database has a high workload balance if all the processors have similar number of locally large itemsets.[2] We have defined quantitative metrics to measure data skewness and workload balance. We found out that both distributed and global prunings have super performance in the best case of high data skewness and high workload balance. The combination of high balance with moderate skewness is the second best case. Inspired by this finding, we investigate the feasibility of planning the partitioning of the database. We want to divide the data into different partitions so as to maximize the workload balance and yield high skewness. Mining a database by partitioning it appropriately and then employing FPM thus gives us excellent mining performance. We have implemented FPM on an IBM SP2 parallel machine with 32 processors. Extensive performance studies have been carried out. The results confirm our observation on the relationship between pruning effectiveness and data distribution.

For the purpose of partitioning, we have proposed four algorithms. We have implemented these algorithms to study their effectiveness. K-means clustering, like most clustering algorithm, provides good skewness. However, it in general would destroy the balance. Random partitioning in general can deliver high balance, but very low skewness. We introduce an optimization constraint to control the balance factor in the k-means clustering algorithm. This modification, called $Bk\mu$ (balanced k-means clustering), produces results which exhibit as good a balance as the random partitioning and also high skewness. In conclusion, we found that $Bk\mu$ is the most favorable partitioning algorithms among those we have studied.

We summarize our contributions as follows:

1. We have enhanced FDM to FPM for mining association rules on distributed share-nothing parallel system which requires fewer rounds of message communication.

[1] An itemset is *locally large* at a processor if it is large within the partition at the processor. It is *globally large* if it is large with respect to the whole database [7, 8]. Note that every globally large itemset must be locally large at some processor. Refer to Section 3 for details.

[2] More precise definitions of skewness and workload balance will be given in Section 4.

2. We have shown analytically that the performance of the pruning techniques in FPM are very sensitive to the data distribution characteristics of skewness and balance, and proposed entropy-based metrics to measure these two characteristics.

3. We have implemented FPM on an SP2 parallel machine and experimentally verified its performance behavior with respect to skewness and balance.

4. We have proposed four partitioning algorithms and empirically verified that $Bk\mu$, among the four, is the most effective in introducing balance and skewness into database partitions.

The rest of this paper is organized as follows. Section 2 overviews the parallel mining of association rules. The techniques of distributed and global prunings, together with the FPM algorithm are described in Section 3. In the same section, we also investigate the relationship between the effectiveness of the prunings and the data distribution characteristics. In Section 4, we define two metrics to measure the data skewness and workload balance of a database partitioning and then present the results of an experimental study on the performance behavior of FPM and CD. Basing on the results of Section 4, we introduce algorithms in Section 5 to partition database so as to improve the performance of FPM. This is achieved by arranging the tuples in the database carefully to increase skewness and balance. Experiments in Sections 6 evaluates the effectiveness of the partitioning algorithms. In Section 7, we discuss a few issues including possible extensions of FPM to enhance its scalability and we give our conclusions in Section 8.

2 Parallel Mining of Association Rules

2.1 Association Rules

Let $I = \{i_1,\ i_2,\ \cdots\ ,\ i_m\}$ be a set of items and D be a database of transactions, where each transaction T consists of a set of items such that $T \subseteq I$. An *association rule* is an implication of the form $X \Rightarrow Y$, where $X \subseteq I$, $Y \subseteq I$ and $X \cap Y = \phi$. An association rule $X \Rightarrow Y$ has support s in D if the probability of a transaction in D contains both X and Y is s. The association rule $X \Rightarrow Y$ holds in D with confidence c if the probability of a transaction in D which contains X also contains Y is c. The task of mining association rules is to find all the association rules whose support is larger than a given minimum support threshold and whose confidence is larger than a given minimum confidence threshold. For an itemset X, we use $X_{.sup}$ to denote its *support count* in database D, which is the number of transactions in D containing X. An itemset $X \subseteq I$ is *large* if $X_{.sup} \geq minsup \times |D|$, where $minsup$ is the given minimum support threshold. For the purpose of presentation, we sometimes just use *support* to stand for support count of an itemset.

It has been shown that the problem of mining association rules can be decomposed into two subproblems

[1]: (1) find all large itemsets for a given minimum support threshold, and (2) generate the association rules from the large itemsets found. Since (1) dominates the overall cost, research has been focused on how to efficiently solve the first subproblem.

In the parallel environment, it is useful to distinguish between the two different notions of *locally large* and *global large* itemsets. Suppose the entire database D is partitioned into $D_1, D_2, \cdots, D_n$ and distributed over n processors. Let X be an itemset, the *global support* of X is the support $X_{.sup}$ of X in D. When refering to a partition D_i, the *local support* of X at processor i, denoted by $X_{.sup(i)}$, is the support of X in D_i. X is *globally large* if $X_{.sup} \geq minsup \times |D|$. Similarly X is *locally large* at processor i if $X_{.sup(i)} \geq minsup \times |D_i|$. *Note that in general, an itemset X which is locally large at some processor i may not necessary be globally large. On the contrary, every globally large itemset X must be locally large at some processor i.* This result and its application have been discussed in details in [7].

For convenience, we use the short form k-itemset to stand for size-k itemset, which consist of exactly k items. And use L_k to denote the set of globally large k-itemsets. We have pointed out above that there is distinction between locally and globally large itemsets. For discussion purpose, we will call a globally large itemset which is also locally large at processor i, *gl-large* at processor i. We will use $GL_{k(i)}$ to denote the gl-large k-itemsets at processor i. Note that $GL_{k(i)} \subseteq L_k$, $\forall i, 1 \leq i \leq n$.

2.2 Count Distribution Algorithm for Parallel Mining

Apriori is the most well known serial algorithm for mining association rules [2]. It relies on the apriori_gen function to generate the candidate sets at each iteration. CD (Count Distribution) is a parallelized version of Apriori for parallel mining [3]. The database D is partitioned into $D_1, D_2, \cdots, D_n$ and distributed across n processors. In the first iteration of CD, every processor i scans its partition D_i to compute the local supports of all the size-1 itemsets. All processors are then engage in one round of support counts exchange. After that, they independently find out global support counts of all the items and then the large size-1 itemsets. For the other iteration k, $(k > 1)$, each processor i runs the program fragment in Figure 1. In step 1, it computes the candidate set C_k by applying the aprior_gen function on L_{k-1}, the set of large itemsets found in the previous iteration. In step 2, local support counts of candidates in C_k are computed by a scanning of D_i. In step 3, local support counts are exchanged with all other processors to get global support counts. In step 4, the globally large itemsets L_k are computed independently by each processor. In the next iteration, CD increases k by one and repeats steps 1–4 until no more candidate is found.

1. C_k = apriori_gen(L_{k-1});
2. scan partition D_i to find the local support counts $X_{.sup(i)}$ for all $X \in C_k$;
3. exchange $\{X_{.sup(i)} \mid X \in C_k\}$ with all other processors to find out the global support counts $X_{.sup}$, for all $X \in C_k$;
4. $L_k = \{X \in C_k \mid X_{.sup} \geq minsup \times |D|\}$

Figure 1: The Count Distribution Algorithm

3 Pruning Techniques and the FPM Algorithm

CD has not taken advantage of the data partitioning in the parallel setting to prune its candidate sets. We propose a new parallel mining algorithm FPM which has adopted the distributed and global prunings proposed first in [7].

3.1 Candidate Pruning Techniques

3.1.1 Distributed Pruning

In step 4 of CD (Figure 1), after the support counts exchange in the k-th iteration, each processor can find out not only the large itemsets L_k, *but also the processors at which an itemset X is locally large, for all itemsets $X \in L_k$*. In other words, the subsets $GL_{k(i)}$, $1 \leq i \leq n$, of L_k can be identified at every processor. These information of locally large itemsets turn out to be very valuable in developing a pruning technique to reduce the number of candidates generated in CD.

Suppose the database is partitioned into D_1 and D_2 on processors 1 and 2. Further assume that both A and B are two size-1 globally large itemsets. In addition, A is gl-large at processor 1 but not processor 2, and B is gl-large at processor 2 but not processor 1. Then AB can never be globally large, and hence does not need to be considered as a candidate. A simple proof of this result in the following. If AB is globally large, it must be locally large (i.e. gl-large) at some processor. Assume that it is gl-large at processor 1, then its subset B must also be gl-large at processor 1, which is contradictory to the assumption. Similarly, we can prove that AB cannot be gl-large at processor 2. Hence AB cannot be globally large at all.

Following the above result, no two 1-itemsets which are not gl-large together at the same processor can be combined to form a size-2 globally large itemset. This observation can be generalized to size-k candidates. The subsets $GL_{k-1(i)}$, $(1 \leq i \leq n)$, together form a partition of L_{k-1}, (some of them may overlap). For $i \neq j$, no candidate need to be generated by joining sets from $GL_{k-1(i)}$ and $GL_{k-1(j)}$. In other words, candidates can be generated by applying apriori_gen on each $GL_{k-1(i)}$, $(1 \leq i \leq n)$, separately, and then take their union. The set of size-k candidates generated with this technique is equal to $CG_k = \cup_{i=1}^{n} CG_{k(i)} = \cup_{i=1}^{n}$apriori_gen$(GL_{k-1(i)})$. Since $GL_{k-1(i)} \subseteq L_{k-1}$, $\forall i, 1 \leq i \leq n$, the number of

candidates in CG_k could be much less than that in $C_k = \text{apriori\_gen}(L_{k-1})$, the candidates in the Apriori and CD algorithms.

Based on the above result, we can prune away a size-k candidate if there is no processor at which *all* its size-$(k-1)$ subsets are gl-large. This pruning technique is called *distributed pruning* [7]. The following example taken from [7] shows that the distributed pruning is more effective in reducing the candidate sets than the pruning in CD.

Example 1 Assuming there are 3 processors which partitions the database D into D_1, D_2 and D_3. Suppose the set of large 1-itemsets (computed at the first iteration) $L_1 = \{A, B, C, D, E, F, G, H\}$, in which A, B, and C are locally large at processor 1, B, C, and D are locally large at processor 2, and E, F, G, and H are locally large at processor 3. Therefore, $GL_{1(1)} = \{A, B, C\}$, $GL_{1(2)} = \{B, C, D\}$, and $GL_{1(3)} = \{E, F, G, H\}$.

Based on the above discussion, the set of size-2 candidate sets from processor 1 is $CG_{2(1)} = \text{apriori\_gen}(GL_{1(1)}) = \{AB, BC, AC\}$. Similarly, $CG_{2(2)} = \{BC, CD, BD\}$, and $CG_{2(3)} = \{EF, EG, EH, FG, FH, GH\}$. Hence, the set of size-2 candidate sets is $CG_{(2)} = CG_{2(1)} \cup CG_{2(2)} \cup CG_{2(3)}$, total 11 candidates.

However, if apriori_gen is applied to L_1, the set of size-2 candidate sets $C_2 = \text{apriori\_gen}(L_1)$ would have 28 candidates. This shows that it is very effective to use the distributed pruning to reduce the candidate sets. ∎

3.1.2 Global Pruning

As a result of count exchange at iteration $k-1$, (step 3 of CD in Figure 1), the local support counts $X_{.sup(i)}$, for all large $(k-1)$-itemsets, for all processor i, $(1 \leq i \leq n)$, are available at every processor. Another powerful pruning technique called *global pruning* is developed by using this information.

Let X be a candidate k-itemset. At each processor i, $X_{.sup(i)} \leq Y_{.sup(i)}$, if $Y \subset X$. Thus, $X_{.sup(i)}$ will always be smaller than $\min\{Y_{.sup(i)} \mid Y \subset X \text{ and } |Y| = k-1\}$. Hence

$$X_{.maxsup} = \sum_{i=1}^{n} \min\{Y_{.sup(i)} \mid Y \subset X \text{ and } |Y| = k-1\}$$

is an upper bound of $X_{.sup}$. X can then be pruned away if $X_{.maxsup} < \text{minsup} \times |D|$. This technique is called *global pruning*. Note that the upper bound is computed from the local support counts resulting from the previous count exchange.

Table 1 (page 6) is an example that global pruning could be more effective than distributed pruning. In this example, the global support count threshold is 15 and the local support count threshold at each

processor is 5. Distributed pruning cannot prune away CD, as C and D are both gl-large at processor 2. Whereas global pruning can prune away CD, as $CD_{.maxsup} = 1 + 12 + 1 < 15$.

In fact, global pruning subsumes distributed pruning, and it is shown in the following theorem.

Theorem 1 *If X is a k-itemset ($k > 1$) which is pruned away in the k-th iteration by distributed pruning, then X is also pruned away by global pruning in the same iteration.*

Proof: If X can be pruned away by distributed pruning, then there does not exist a processor at which all the size $(k-1)$ subsets of X are gl-large. Thus, at each processor i, there exists a size $(k-1)$ subset Y of X such that $Y_{.sup(i)} < minsup \times |D_i|$. Hence $X_{.maxsup} < \sum_{i=1}^{n}(minsup \times |D_i|) = minsup \times |D|$. Therefore, X is pruned away by global pruning. ∎

The reverse of Theorem 1 is not necessarily true, however. From the above discussions, it can be seen that the three pruning techniques, the one in apriori_gen, the distributed and global prunings, have increasing pruning power, and the latter ones subsume the previous ones.

Items	A	B	C	D	E	F
local support at processor 1	13	33	1	2	2	1
local support at processor 2	1	3	12	34	1	4
local support at processor 3	2	1	2	1	12	33
global support	16	37	15	37	15	38
gl-large at processor 1	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$
gl-large at processor 2	$\times$	$\times$	$\surd$	$\surd$	$\times$	$\times$
gl-large at processor 3	$\times$	$\times$	$\times$	$\times$	$\surd$	$\surd$

Table 1: High data skewness and high workload balance case

3.2 Fast Parallel Mining Algorithm (FPM)

We present the FPM algorithm in this section. It improves CD by adopting the two pruning techniques. The first iteration of FPM is the same as CD. Each processor scans its partition to find out local support counts of all size-1 itemsets and use one round of count exchange to compute the global support counts. At the end, in addition to L_1, each processor also finds out the gl-large itemsets $GL_{1(i)}$, for $1 \leq i \leq n$. Starting from the second iteration, prunings are used to reduce the number of the candidate sets. Figure 2 is the program fragment of FPM at processor i for the k-th ($k > 1$) iteration. In step 1, distributed pruning is used and apriori_gen is applied to the sets $GL_{k-1(i)}$, $\forall i = 1, \cdots, n$, instead of to the set L_{k-1}.[3] In step 2, global pruning is applied to the candidates that survive the distributed pruning. The remaining steps are the same as those in CD. As has been discussed, FPM in general enjoys a smaller candidate sets than

[3]Compare this step with step 1 of Figure 1 would be useful in order to see the difference between FPM and CD.

1. compute the candidate sets $CG_k = \cup_{i=1}^{n} \text{apriori\_gen}(GL_{k-1(i)})$; (distributed pruning)
2. apply global pruning to prune the candidates in CG_k;
3. scan partition D_i to find out the local support counts $X_{.sup(i)}$ for all remaining candidates $X \in CG_k$;
4. exchange $\{X_{.sup(i)} \mid X \in CG_k\}$ with all other processors to find out the global support counts $X_{.sup}$, for all $X \in CG_k$;
5. compute $GL_{k(i)} = \{X \in CG_k \mid X_{.sup} \geq minsup \times |D| \text{ and } X_{.sup(i)} \geq minsup \times |D_i|\}$ and exchange the result with all other processors;
6. return $L_k = \cup_{i=1}^{n} GL_{k(i)}$.

Figure 2: The FPM Algorithm

CD. Furthermore, it uses a simple one-round message exchange scheme same as CD. If we compare FPM with the FDM algorithm proposed in [7], we will see that this simple communication scheme makes FPM more suitable than FDM in terms of response time in a parallel system.

Note that since the number of processors n would not be very large, the cost of generating the candidates with the distributed pruning in step 1 (Figure 2) should be on the same order as that in CD. As for global pruning, since all local support counts are available at each processor, no additional count exchange is required to perform the pruning. Furthermore, the pruning in step 2 (Figure 2) is performed only on the remaining candidates after the distributed pruning. Therefore, cost for global pruning is small comparing with database scanning and count updates.

3.3 Data Skewness and Workload Balance

In a partitioned database, two data distribution characteristics, *data skewness* and *workload balance*, affect the effectiveness of the pruning and hence performance of FPM. Intuitively, the data skewness of a partitioned database is high if most large itemsets are locally large only at a few processors. It is low if a high percentage of the large itemsets are locally large at most of the processors. For a partitioning with high skewness, even though it is highly likely that each large itemset will be locally large at only a small number of partitions, the set of large itemsets together can still be distributed either evenly or extremely skewed among the partitions. In one extreme case, most partitions can have similar number of locally large itemsets, and the workload balance is high. In the other extreme case, the large itemsets are concentrated at a few partitions and hence there are large differences in the number of locally large itemsets among different partitions. In this case, the workload is unbalance. These two characteristics have important bearing on the performance of FPM.

Example 2 Table 1 (page 6) is a case of high data skewness and high workload balance. The supports

of the itemsets are clustered mostly in one partition, and the skewness is high. On the otherhand, every partition has the same number (two) of locally large itemsets.[4] Hence, the workload balance is also high. CD will generate $\binom{6}{2} = 15$ candidates in the second iteration, while distributed pruning will generate only three candidates AB, CD and EF, which shows that the pruning has good effect.

Items	A	B	C	D	E	F
local support at processor 1	6	12	4	13	5	12
local support at processor 2	6	12	5	12	4	13
local support at processor 3	4	13	6	12	6	13
global support	16	37	15	37	15	38
gl-large at processor 1	$\surd$	$\surd$	$\times$	$\surd$	$\surd$	$\surd$
gl-large at processor 2	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\surd$
gl-large at processor 3	$\times$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

Table 2: High workload balance and low data skewness case

Table 2 is an example of high workload balance and low data skewness. The support counts of the items A, B, C, D, E and F are almost equally distributed over the 3 processors. Hence, the data skewness is low. However, the workload balance is high, because every partition has the same number (five) of locally large itemsets. Both CD and distributed pruning generate the same 15 candidate sets in the second iteration. However, global pruning can prune away the candidates AC, AE and CE. FPM still exhibits a 20% of improvement over CD in this pathological case of high balance and low skewness. ∎

We will define formal metrics for measuring data skewness and workload balance for partitions in Section 4. We will also see how high values of balance and skewness can be obtained by suitably and carefully partitioning the database in Section 5. In the following, we will show the effects of distributed pruning analytically for some special cases.

Theorem 2 *Let L_1 be the set of size-1 large itemsets, $C_{2(c)}$ and $C_{2(d)}$ be the size-2 candidates generated by CD and distributed pruning, respectively. Suppose that each size-1 large itemset is gl-large at one and only one processor, and the size-1 large itemsets are distributed evenly among the processors, i.e., the number of size-1 gl-large itemsets at each processor is $|L_1|/n$, where n is the number of processors. Then $\frac{|C_{2(d)}|}{|C_{2(c)}|} = \frac{(|L_1|/n)-1}{|L_1|-1} \approx \frac{1}{n}$.*

Proof: Since CD will generate all the combinations in L_1 as size-2 candidates, $|C_{2(c)}| = \binom{|L1|}{2} = \frac{|L_1|}{2}(|L_1|-1)$. As for distributed pruning, each processor will generate candidates independently from the gl-large size-1 itemsets at the processor. The total number of candidates it will generate is $|C_{2(d)}| = \binom{|L_1|/n}{2} \times n =$

[4] As has been mentioned above, the support threshold in Table 1 for globally large itemsets is 15, while that for locally large itemset is 5 for all the partitions.

$\frac{|L_1|}{2}(|L_1|/n-1)$. Since n is the number of processor, which is much smaller than $|L_1|$, we have $|L_1| \gg n \geq 1$. Hence, we can take the approximations $|L_1|-1 \approx |L_1|$ and $|L_1|/n-1 \approx |L_1|/n$. So, $\frac{|C_{2(d)}|}{|C_{2(c)}|} = \frac{(|L_1|/n)-1}{|L_1|-1} \approx \frac{1}{n}$. ∎

Theorem 2 shows that distributed pruning can dramatically prune away almost $\frac{n-1}{n}$ size-2 candidates generated by CD in the high balance and good skewness case.

We now consider a special case for the k-th iteration ($k > 2$) of FPM. In general, if A_{k-1} is the set of size-$(k-1)$ large itemsets, then the maximum number of size-k candidates that can be generated by applying apriori_gen on A_{k-1} is equal to $\binom{m}{k}$, where m is the smallest integer such that $\binom{m}{k-1} = |A_{k-1}|$. In the following, we use this maximal case to estimate the number of candidates that can be generated in the k-th iteration. Let $C_{k(c)}$ and $C_{k(d)}$ be the set of size-k candidates generated by CD and distributed pruning, respectively. Similar to Theorem 2, we investigate the case in which all gl-large $(k-1)$-itemsets are locally large at only one processor, and the number of gl-large itemsets at each processor is the same. Let m be the smallest integer such that $\binom{m}{k-1} = |L_{k-1}|$. Then, we have $|C_{k(c)}| = \binom{m}{k}$. Hence $|C_{k(c)}| = \frac{m-k+1}{k} \times |L_{k-1}|$. Similarly, let m' be the smallest integer such that $\binom{m'}{k-1} = \frac{|L_{k-1}|}{n}$. Then $|C_{k(d)}| = \binom{m'}{k} \times n$. Hence $|C_{k(d)}| = \frac{m'-k+1}{k} \times \frac{|L_{k-1}|}{n} \times n$. Therefore, $\frac{|C_{k(d)}|}{|C_{k(c)}|} = \frac{m'-k+1}{m-k+1}$. When $k = 2$, this result becomes Theorem 2. In general, $m' \ll m$, which shows that distributed pruning has significant effect in almost all iterations. However, the effect will decrease when m' converges to m as k increases.

4 Metrics for Data Skewness and Workload Balance

In this section, we define metrics to measure data skewness and workload balance. Then, we present empirical results of the effects of skewness and balance on the performance of FPM and CD.

It is important to note that the simple intuition of defining balance as a measure on the evenness of distributing transactions among the partitions is not suitable in our studies. The performance of the prunings is linked to the distribution of the large itemsets, not that of the transactions. Furthermore, the metrics on skewness and balance should be consistent between themselves. In the following, we explain our entropy-based metrics defined for these two notions.

4.1 Data Skewness

We develop a skewness metric based on the well established notion of entropy [5]. Given a random variable X, it's entropy is a measurement on how even or uneven its probability distribution is over its values. If a database is partitioned over n processors, the value $p_X(i) = \frac{X_{.sup(i)}}{X_{.sup}}$ can be regarded as the

probability that a transaction containing itemset X comes from partition D_i, $(1 \leq i \leq n)$. The *entropy* $H(X) = -\sum_{i=1}^{n}(p_X(i) \times \log(p_X(i)))$ is an indication of how even the supports of X are distributed over the partitions.[5] For example, if X is skewed completely into a single partition D_k, $(1 \leq k \leq n)$, i.e., it only occurs in D_k, then $p_X(k) = 1$ and $p_X(i) = 0$, $\forall i \neq k$. The value of $H(X) = 0$ is the minimal in this case. On the other hand, if X is evenly distributed among all the partitions, then $p_X(i) = \frac{1}{n}$, $1 \leq i \leq n$, and the value of $H(X) = \log(n)$ is the maximal in this case. Therefore, the following metric can be used to measure the skewness of a database partitioning.

Definition 1 *Given a database with n partitions, the skewness $S(X)$ of an itemset X is defined by $S(X) = \frac{H_{max} - H(X)}{H_{max}}$, where $H(X) = -\sum_{i=1}^{n}(p_X(i) \times \log(p_X(i)))$ and $H_{max} = \log(n)$.*

The skewness $S(X)$ has the following properties:

- $S(X) = 0$ when all $p_X(i)$ $(1 \leq i \leq n)$, are equal. So the skewness is at its lowest value when X is distributed evenly in all partitions.
- $S(X) = 1$, when a $p_X(i)$ equals 1 and all the others are 0. So the skewness is at its highest value when X occurs only in one partition.
- $0 < S(X) < 1$, in all the other cases.

Following the property of entropy, higher values of $S(X)$ corresponds to higher skewness for X. The above definition gives a metric on the skewness of an itemset. In the following, we define the skewness of a partitioned database as a weighted sum of the skewness of all its itemsets.

Definition 2 *Given a database D with n partitions, the skewness $TS(D)$ is defined by*

$$TS(D) = \sum_{X \in IS} S(X) \times w(X)$$

where IS is the set of all the itemsets, $w(X) = \frac{X.sup}{\sum_{Y \in IS} Y.sup}$ is the weight *of the support of X over all the itemsets, and $S(X)$ is the skewness of itemset X.*

$TS(D)$ has some properties similar to those of $S(X)$.

- $TS(D) = 0$, when the skewness of all the itemsets are at its minimal value.

[5] In the computation of $H(X)$, some of the probability values $p_X(i)$ may be zero. In that case, we take $0 \log 0 = 0$, in the sense that $\lim_{h \to 0} h \log h = 0$.

- $TS(D) = 1$, when the skewness of all the itemsets are at its maximal value.
- $0 < TS(D) < 1$, in all the other cases.

We can compute the skewness of a partitioned database according to Definition 2. However, the number of itemsets may be very large in general. One approximation is to compute the skewness over the set of globally large itemsets only, and take the approximation $p_X(i) \approx 0$ if X is not gl-large in D_i. In the generation of candidate itemsets, only globally large itemsets will be joined together to form new candidates. Hence, only their skewness would impact the effectiveness of pruning. Therefore, this approximation is a reasonable and practical measure.

4.2 Workload Balance

Workload balance is a measurement on the distribution of the total weights of the locally large itemsets among the processors. Based on the definition of $w(X)$ in Definition 2, we define $W_i = \sum_{X \in IS} w(X) \times p_X(i)$ to be the *itemset workload* of partition D_i, where IS is the set of all the itemsets. Note that $\sum_{i=1}^{n} W_i = 1$. A database has high workload balance if the W_i's are the same for all partitions D_i, $1 \leq i \leq n$. On the other hand, if the values of W_i exhibit large differences among themselves, the workload balance is low. Thus, our definition of the *workload balance* metric is also based on the entropy measure.

Definition 3 *For a database D with n partitions, the workload balance factor (workload balance, for short) $TB(D)$ is defined as* $TB(D) = \frac{-\sum_{i=1}^{n} W_i \log(W_i)}{\log(n)}$.

The metric $TB(D)$ has the following properties:

- $TB(D) = 1$, when the workload across all processors are the same;
- $TB(D) = 0$, when the workload is concentrated at one processor;
- $0 < TB(D) < 1$, in all the other cases.

Similar to the skewness metric, we can approximate the value of $TB(D)$ by only considering globally large itemsets.

The data skewness and workload balance are not independent of each other. Theoretically, each one of them may attain values between 0 and 1, inclusively. However, some combinations of their values are not admissible. For instance, we cannot have a database partitioning with very low balance and very low skewness. This is because a very low skewness would accompany with a high balance while a very low balance would accompany with a high skewness.

Theorem 3 *Let $D_1, D_2, \cdots, D_n$ be the partitions of a database D.*

1. *If $TS(D) = 1$, then the admissible values of $TB(D)$ ranges from 0 to 1. Moreover, if $TS(D) = 0$, then $TB(D) = 1$.*
2. *If $TB(D) = 1$, then the admissible values of $TS(D)$ ranges from 0 to 1. Moreover, if $TB(D) = 0$, then $TS(D) = 1$.*

Proof:

1. By definition $0 \leq TB(D) \leq 1$. What we need to prove is that the boundary cases are admissible when $TS(D) = 1$. $TS(D) = 1$ implies that $S(X) = 1$, for all large itemsets X. Therefore, each large itemset is large at one and only one partition. If all the large itemsets are large at the same partition D_i, then $W_i = 1$ and $W_k = 0, (1 \leq k \leq n, k \neq i)$. Thus $TB(D) = 0$ is admissible. On the other hand, if every partition has the same number of large itemsets, then $W_i = \frac{1}{n}, (1 \leq i \leq n)$, and hence $TB(D) = 1$. Furthermore, if $TS(D) = 0$, then $S(X) = 0$ for all large itemsets X. This implies that W_i are the same for all $1 \leq i \leq n$. Hence $TB(D) = 1$.
2. It follows from the first result of this theorem that both $TS(D) = 0$ and $TS(D) = 1$ are admissible when $TB(D) = 1$. Therefore the first part is proved. Furthermore, if $TB(D) = 0$, there exists a partition D_i such that $W_i = 1$ and $W_k = 0, (1 \leq k \leq n, k \neq i)$. This implies that all large itemsets are locally large at only D_i. Hence $TS(D) = 1$.

∎

Even though, $0 \leq TS(D) \leq 1$ and $0 \leq TB(D) \leq 1$, not all possible combinations are admissible. In general, the admissible combinations is a subset of the unit square, represented by the shaded region in Figure 6. It always contains the two line segments $TS(D) = 1$ (S = 1 in Figure 6) and $TB(D) = 1$ (B = 1 in Figure 6), but not the origin, (S = 0, B = 0). After defining the metrics and studying their characteristics, we can experimentally validate our analysis (see Section 3.3) on the relationship between data skewness, workload balance and performance of FPM and CD.

We would like to note that the two metrics are based on total entropy which is a good model to measure evenness (or unevenness) of data distribution. Also, they are consistent with each other.

4.3 Performance Behaviors of FPM and CD

We study the performance behaviors of FPM and CD in response to various skewness and balance values on an IBM SP2 parallel processing machine with 32 nodes. Each node consists of a POWER2 processor with

a CPU clock rate of 66.7 MHz and 64 MB of main memory. The system runs the AIX operating system. Communication between processors are done through a high performance switch with an aggregated peak bandwidth of 40 MBps and a latency about 40 microseconds. The appropriate database partition is downloaded to the local disk of each processor before mining starts. The databases used for the experiments are all synthesized according the a model which is an enhancement of the model adopted in [2]. Due to insufficient paper space, the description is omitted.

4.3.1 Improvement of FPM over CD

In order to compare the performance of FPM and CD, we have generated a number of databases. The size of every partition of these databases is about 100MB, and the number of partitions is 16, i.e., $n = 16$.[6] We also set $N = 1000$, $L = 2000$, correlation level to 0.5. The name of each database is in the form Dx.Ty.Iz.Sr.Bl, where x is the average number of transactions per partition, y is the average size of the transactions, z is the average size of the itemsets. These three values are the control values for the database generation. On the other hand, the two values r and l are the control values (in percentage) of the skewness and balance, respectively. They are added to the name, in the sense that they are intrinsic properties of the database.

We ran FPM and CD on various databases. The minimum support threshold is 0.5%. The improvement of FPM over CD in response time on these databases are recorded in Table 3. In the table, each entry corresponds to the results of one database, and the value of the entry is the speedup ratio of FPM to CD, i.e. the response time of CD over that of FPM. Entries corresponding to the same skewness value are put onto the same row, while entries under the same column correspond to databases with the same balance value. The result is very encouraging. FPM is consistently faster than CD in all cases. Comparing the figures for databases D2016K.T10.I4 against those of D3278K.T5.I2.S10 and comparing D1140K.T20.I6.S90 against D2016K.T10.I4.S10, we observe that the larger the transaction sizes and longer the large-itemsets, the more significant the improvements are in general. Obviously, the local pruning and global pruning adopted in FPM are very effective. The sizes of the candidate sets are significantly reduced, and hence FPM outperforms CD significantly.

[6] Even though the SP2 we use has 32 nodes, because of administration policy, we can only use 16 nodes in our experiments.

Database	Speedup (FPM/CD)					
	B100	B90	B70	B50	B30	B10
D3278K.T5.I2.S90	2.10	1.69	1.36	1.23	1.14	1.06
D3278K.T5.I2.S70	2.07	1.41	1.23	1.13	1.06	—
D3278K.T5.I2.S50	1.88	1.22	1.11	1.06	—	—
D3278K.T5.I2.S30	1.55	1.17	1.08	—	—	—
D3278K.T5.I2.S10	1.36	1.09	—	—	—	—
D2016K.T10.I4.S90	2.33	1.59	1.38	1.35	1.15	1.07
D2016K.T10.I4.S70	2.30	1.47	1.29	1.14	1.08	—
D2016K.T10.I4.S50	2.08	1.28	1.11	1.09	—	—
D2016K.T10.I4.S30	1.55	1.19	1.10	—	—	—
D2016K.T10.I4.S10	1.27	1.10	—	—	—	—
D1140K.T20.I6.S90	3.21	2.10	1.51	1.26	1.12	1.06
D1140K.T20.I6.S70	3.09	1.81	1.35	1.16	1.07	—
D1140K.T20.I6.S50	2.53	1.42	1.18	1.09	—	—
D1140K.T20.I6.S30	1.87	1.28	1.10	—	—	—
D1140K.T20.I6.S10	1.50	1.12	—	—	—	—

Table 3: Performance Improvement of FPM over CD

4.3.2 Performance of FPM with High Workload Balance

Figure 3 shows the response time of the FPM and CD algorithms for databases with various skewness values and a high balance value of B = 100 [7]. FPM outperforms CD significantly even when the skewness is in the moderate range, $(0.1 \leq s \leq 0.5)$. This trend can be read also from Table 3. When B = 100, FPM is 36% to 110% faster than CD. In the more skewed cases, i.e., $70 \leq S \leq 90$, FPM is at least 107% faster than CD. When skewness is moderate ($S = 30$), FPM is still 55% faster than CD. When B = 90, FPM maintains the performance lead over CD. The results clearly demonstrate that given a high workload balance, FPM outperforms CD significantly when the skewness is in the range of high to moderate.

4.3.3 Performance of FPM with High Skewness

Figure 4 plots the response time of FPM and CD for databases with various balance values. The skewness is maintained at S = 90. In this case, FPM behaves slightly differently from the high workload balance case presented in the previous section. FPM performs much better than CD when the workload balance is relatively high ($b > 0.5$). However, its performance improvement over CD in the moderate balance range, $(0.1 \leq b \leq 0.5)$, is marginal. This implies that FPM is more sensitive to workload balance than skewness.

[7] We use B and S to represent the control value of the balance and skewness in the databases in Table 3. Hence their unit is in percentage and their value is in the range of [0, 100]. On the other hand, in Figures 3 to 6, we use s and b to represent the skewness and balance of the databases, which are values in the range of $[0, 1]$. In real term, both B and b, S and s have the same value except different units.

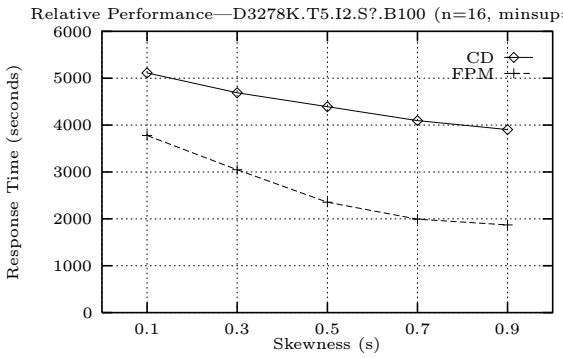


Figure 3: (left) Relative Performance on Databases with High Balance

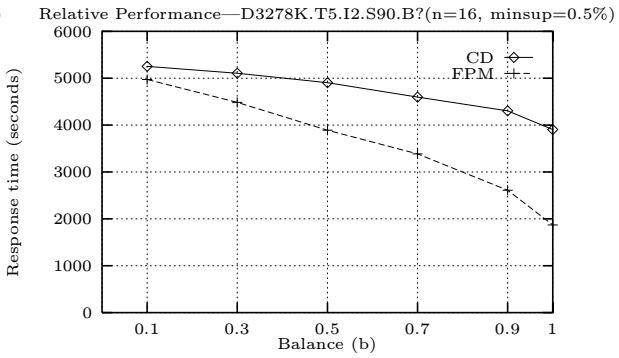


Figure 4: (right) Relative Performance on Databases with High Skewness

In other words, if the workload balance has dropped to a moderate value, even a high skewness cannot stop the degradation in performance improvement.

This trend can also be inferred from Table 3. When S = 90, FPM is 6% to 110% faster than CD depending on the workload balance. In the more balanced databases, i.e., $90 \leq B \leq 100$, FPM is at least 69% faster than CD. In the moderate balance case ($50 \leq B \leq 70$), the performance gain drops to the 23% to 36% range. This result shows that a high skewness has to be accompanied by a high workload balance in order for FPM to deliver a good improvement. The effect of a high skewness with a moderate balance is not as good as that of a high balance with a moderate skewness.

4.3.4 Performance of FPM with Moderate Skewness and Balance

In Figure 5, we vary both the skewness and balance together from a low values combination to a high values combination. The trend shows that the improvement of FPM over CD increases from a low percentage at $s = 0.5$, $b = 0.5$ to a high percentage at $s = 0.9$, $b = 0.9$. Reading Table 3, we find that the performance gain of FPM increases from around 6% (S = 50, B = 50) to 69% (S = 90, B = 90) as skewness and balance increase simultaneously. The combination of S = 50, B = 50 in fact is a point of low performance gain of FPM in the set of all admissible combinations in our experiments.

4.3.5 Summary of the Performance Behaviors of FPM and CD

We have done some other experiments to study the effects of skewness and balance on the performance of FPM and CD. Combining these results with our observations in the above three cases, we can divide the admissible area into several regions, as shown in Figure 6. Region A is the region in which FPM outperforms CD the most. In this region, the balance is high and the skewness varies from high to

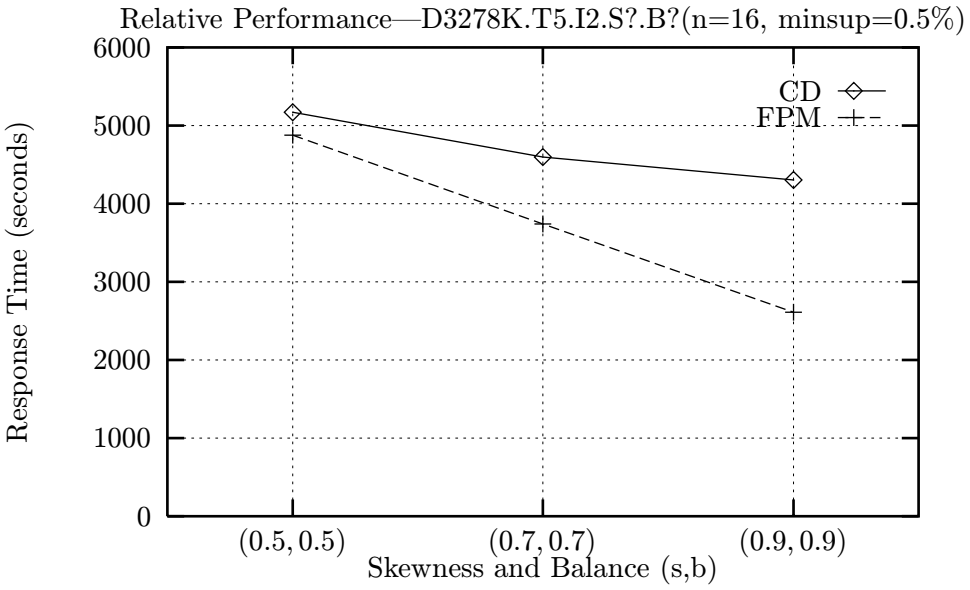
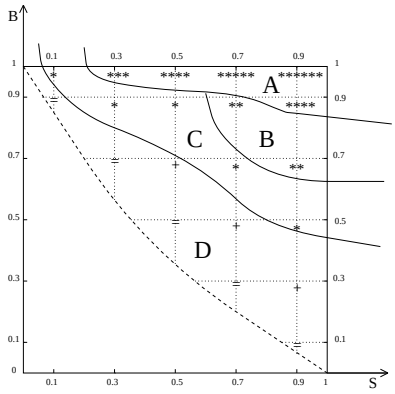


Figure 5: Relative Performance on Databases when both Skewness and Balance are varied



legend	FPM/CD
**** ... ******	(1.60, 3.21]
***	(1.45, 1.60]
**	(1.30, 1.45]
*	(1.15, 1.30]
+	(1.10, 1.15]
=	(1.00, 1.10]

Figure 6: Division of the admissible regions according to the performance improvement of FPM over CD (FPM/CD)

moderate, and FPM performs 45% to 221% faster than CD. In region B, the workload balance value has degraded moderately and the skewness remains high. The change in workload balance has brought the performance gain in FPM down to a lower range of 35% to 45%. Region C covers combinations that have undesirable workload balance. Even though the skewness could be rather high in this region, because of the low balance value, the performance gain in FPM drops to a moderate range of 15% to 30%. Region D contains those combinations on the bottom of the performance ladder in which FPM only has marginal performance gain.

Thus, our empirical study clearly shows that high values of skewness and balance favors FPM. Furthermore, between skewness and balance, workload balance is more important than skewness. The study here not only has shown that the performance of the prunings is very sensitive to the data distribution,

it also has demonstrated that the two metrics are useful in distinguishing "favorable" distributions from "unfavorable" distributions. In Figure 6, regions C & D may cover more than half of the whole admissible area. For database partitions fall in these regions, FPM may not be much better than CD. Because of this, it is important to partition a database in such a way that the resulted partitions would be in a more favorable region. In Sections 5 and 6, we will show that this is in fact possible by using partitioning algorithms we will proposed.

5 Partitioning of the Database

Suppose now that we have a centralized database and want to mine it for association rules. We have to divide the database into partitions and then run FPM on the partitions. If we can divide the database in a way to yield high balance and skewness across the partitions, we can have much more savings on resources by using FPM. Even for an already partitioned database, redistributing the data among the partitions may also increase the skewness and balance, thus benefiting FPM. So, how to divide the database into partitions to achieve high balance and skewness becomes an interesting and important problem.

Note that not all databases can be divided into a given number of partitions to yield high skewness and balance. If the data in the database is already highly uniform and homogeneous, with not much variations, then any method of dividing it into the given number of partitions would produce similar skewness and balance. However, most real-life database are not uniform, and there are many variations within them. It is possible to find a wise way of dividing the data tuples into different partitions to give a very high balance and skewness than an arbitrary partition. Therefore, if a database intrinsically has non-uniformness, we may dig out such non-uniformness and exploit it to partition the database.

So, it would be beneficial to partition the database carefully. Ideally, the partitioning method should maximize the skewness and balance metrics for any given database. However, doing such an optimization would be no easier than finding out the association rules. The overhead of this would be too high to worth doing. So, instead of optimizing the skewness and balance values, we would use low-cost algorithms that produce reasonably high balance and skewness values. These algorithms should be simple enough so that not much overhead is incurred. Such small overhead would be far compensated by the subsequent savings in running FPM.

5.1 Framework of the Partitioning Algorithms

To make the partitioning algorithms simple, we will base on the following framework. The core part of the framework is a clustering algorithm, for which we will plug in different clustering algorithms to give

different partitioning algorithms. The framework can be divided into three steps.

Conceptually the first step of the framework divides the transactions in the database into equal-sized chunks C_i. Each chunk contains the same number y of transactions. So, there will be a total of $z = \lceil \frac{|D|}{y} \rceil$ chunks. For each chunk, we define a signature γ_i, which is an $|I|$-dimensional vector. The j-th ($j = 1, 2, \ldots, |I|$) element of the signature, $\gamma_{i,j}$, is the support count of the 1-itemset containing item number j in chunk C_i. Note that each signature γ_i is a vector. This allows us to use functions and operations on vectors to describe our algorithms. The signatures γ_i are then used as representatives of their corresponding chunks C_i. All the γ_i's can be computed by scanning the database once. Moreover, we can immediately deduce the support counts of all 1-itemsets as $\sum \gamma_i$. This can indeed be exploited by FPM to avoid the first iteration, thus saving one scan in FPM (see Section 7.1 for details). So, overall, we can obtain the signatures without any extra database scans.

The second step is to divide the signatures γ_i into n groups G_k, where n is the number of partitions to be produced. Each group corresponds to a partition in the resulting partitioning. The number of partitions n should be equal to the number of processors to be used for running FPM. A good partitioning algorithm should assign the signatures to the groups according to the following criteria. To increase the resulting skewness, we should put the signatures into groups so that the distance[8] between signatures in the same group is small, but the distance between signatures in different groups is high. This would tend to make the resulting partitions more different from one another, and make the transactions within each partition more similar to themselves. Hence, it would have higher skewness. To increase the workload balance, each group should have similar signature sum. One way to achieve this is to assign more or less the same number of signatures to each group such that the total signatures in each group are very close.

The third step of the framework distributes the transactions to different partitions. For each chunk C_i, we check which group its signature γ_i was assigned to in step 2. If it was assigned to group G_k, then we send all transactions in that chunk to partition k. After sending out all the chunks, the whole database is partitioned.

It can be easily noticed that step 2 is the core part of the partitioning algorithm framework. Steps 1 and 3 are simply the pre-processing and post-processing parts. By using a suitable chunk size, we can reduce the total number of chunks, and hence signatures, to a suitable value, so that they can all be processed in RAM by the clustering algorithm in step 2. This effectively reduces the amount of information that our clustering algorithm have to handle, and hence the partitioning algorithms are very efficient.

We like to note that in our approach, we have only made use of the signatures from size-1 itemsets. We could have used those of larger size itemsets. However, that would cost more and eventually it becomes

[8] Any valid distance function in the $|I|$-dimensional space may be used.

the problem of finding the large itemsets. Using size-1 itemsets is a good trade-off, and the resources used in finding them as we have noted is not wasted. Furthermore, our empirical results (in Section 6) have shown that by just using the size-1 itemsets, we can already achieve reasonable skewness and high balance.

5.2 The Clustering Problem

With this framework, we have reduced the partitioning problem to a clustering algorithm. Our original problem is to partition a given database so that the resulting partitions give high skewness and balance values, with balance receiving more attention. Now, we have turned it into a clustering problem, which is stated as follows.

Problem Given z I-dimensional vectors γ_i $(i = 1, 2, \ldots, z)$, called "*signatures*", assign them to n groups G_j $(j = 1, 2, \ldots, n)$ so as to achieve the following to criteria:

1. (Skewness) $\sum_{i=1}^{z} \sum_{j=1}^{n} ||\gamma_i - \theta_j||^2 \cdot \delta(i,j)$ is minimized, where $\theta_j = \frac{1}{\sigma_j} \sum_{i=1}^{z} \gamma_i \cdot \delta(i,j)$ and $\delta(i,j) = 1$ if γ_i is assigned to G_j and $\delta(i,j) = 0$ otherwise and $\sigma_j = \sum_{i=1}^{z} \delta(i,j)$
2. (Balance) $\sigma_{j_1} = \sigma_{j_2}$ for all $j_1, j_2 = 1, 2, \ldots, n$

Note that here, the vector θ_j is the geometric centroid of the signatures assigned to group G_j, while σ_j is the number of signatures assigned to that group.[9] The notation $||x||$ denotes the distance of vector x from the origin, measured with the chosen distance function.

The first criterion above says that we want to minimize the distance between the signatures assigned to the same group. This is to achieve high skewness. The second criterion reads that each group shall be assigned the same number of signatures, so as to achieve high balance.

Note that it is non-trivial to meet both criteria at the same time. So, we develop algorithms that attempt to find approximate solutions. Below, we will give several clustering algorithms, that are to be plugged into step 2 of the framework to give various partitioning algorithms.

5.3 Some Straightforward Approaches

The simplest idea to solve the clustering problem is to assign the signatures γ_i to the groups G_j randomly.[10] For each signature, we choose a uniformly random integer r between 1 and n (the number of partitions) and

[9] In the subsequent sections, the index j will be used for the domain of groups. So, it will *implicitly* take values from 1 to n.

[10] In the subsequent sections, the index i will be used for the domain of signatures. So, it will *implicitly* take values $1, 2, \ldots, z$.

assign the signature to group G_r. As a result, each group would eventually receive roughly the same amount of signatures, and hence chunks and transactions. This satisfies the balance criterion (see Section 5.2), but leaves the skewness criterion unattacked. So, this clustering method should yield high balance, which is close to unity. However, skewness is close to zero, because of the completely random assignment of signatures to groups. With this clustering algorithm, we get a partitioning algorithm, which we will refer to "random partitioning".

To achieve good skewness, we shall assign signatures to groups such that signatures near to one another should go to the same group. Many clustering algorithms with such a goal have been developed. Here, we shall use one of the most famous ones: the k-means algorithm [14]. (We refer the readers to relevant publications for a detailed description of the algorithm.) Since the k-means algorithm minimizes the sum of the distances of the signatures to the geometric centroids of their corresponding groups, it meets the skewness criterion (see Section 5.2). However, the balance criterion is completely ignored, since we impose no restrictions on the size of each group G_j. Consequently some groups may get more signatures then the others and hence the corresponding partitions will receive more chunks of transactions. Thus, this algorithm should yield high skewness, but does not guarantee good workload balance. In the subsequent discussions, we shall use the symbol "$k\mu$" to denote the partitioning algorithm employing the k-means clustering algorithm.

Note that the random partitioning algorithm yields high balance but poor skewness, while $k\mu$ yields high skewness but low balance. They do not achieve our goal of getting high skewness *as well as* balance. Nonetheless, these algorithms does give us an idea of how high a balance or skewness value can be achieved by suitably partitioning a given database. The result of the random algorithm suggests the highest achievable balance of a database, while the $k\mu$ algorithm gives the highest achievable skewness value. Thus, they give us reference values to evaluate the effectiveness of the following two algorithms.

5.4 Sorting by the Highest-Entropy Item (SHEI)

To achieve high skewness, we may use sorting. This idea comes from the fact that sorting decreases the degree of disorder and hence entropy. So, the skewness measure should, according to Definitions 1 and 2, increase. If we sort the signatures γ_i $(i = 1, 2, \ldots, z)$ in ascending order of the k_1-th coordinate value (i.e. γ_{i,k_1}), which is the support count of item number k_1, and then divide the sorted list evenly into n equal-length consecutive sublists, and then assign each sublist to a group G_j, we can obtain a partitioning with good skewness. Since each sublist has the same length, the resulting groups have the same number of signatures. Consequently, the partitions generated will have equal amount of transactions. This should give a balance better then $k\mu$.

group	signature	coordinate values				
		$\cdots$	k_1	$\cdots$	k_2	$\cdots$
G_1	γ_1	$\cdots$	0	$\cdots$	1	$\cdots$
	γ_6	$\cdots$	3	$\cdots$	1	$\cdots$
	γ_3	$\cdots$	3	$\cdots$	1	$\cdots$
G_2	γ_2	$\cdots$	3	$\cdots$	0	$\cdots$
	γ_5	$\cdots$	5	$\cdots$	0	$\cdots$
	γ_4	$\cdots$	7	$\cdots$	2	$\cdots$

Table 4: An example demonstrating the idea of SHEI

This idea is illustrated with the example database in Table 4. This database has only 6 signatures and we divide it into 2 groups ($z = 6$, $n = 2$). The table shows the coordinate values of the items k_1 and k_2 of each signature. Using the idea mentioned above, we sort the signatures according to the k_1-th coordinate values. This gives the resulting ordering as shown in the table. Next, we divide the signatures into two groups (since $n = 2$), each of size 3, with the order of the signature being preserved. So, the first 3 signatures are assigned to group G_1, while the last 3 signatures are assigned to G_2. This assignment is also shown in the table. The database is subsequently partitioned by delivering the corresponding chunks C_i to the corresponding partitions. Observe how the sorting has brought the transactions that contribute to the support count of item k_1 to the partition 2. Since the coordinate values are indeed support counts of the corresponding 1-itemsets, we know that partition 1 has a support count of $0 + 3 + 3 = 6$ for item k_1, while partition 2 has a support count of $3 + 5 + 7 = 15$. Thus, the item k_1 has been made to be skewed towards partition 2.

Now, why did we choose item k_1 for the sort key, but not item k_2? Indeed, we should not choose an arbitrary item for the sort key, because not all items can give good skewness after the sorting. For example, if the item k_1 occurs equally frequently in each chunk C_i, then all signatures γ_i will have the same value in the coordinate corresponding to the support count of item k_1. Sorting the signatures using this key will not help increasing the skewness. On the other hand, if item k_1 has very uneven distribution among the chunks C_i, sorting would tend to deliver chunks with higher occurrence of k_1 to the same or near-by groups. In this case, skewness can be increased significantly by sorting. So, we shall choose the items with uneven distribution among the chunks C_i for the sort key. To measure the unevenness, we use the statistical entropy measure again. For every item X, we evaluate its statistical entropy value among the signature coordinate values $\gamma_{i,X}$ over all chunks C_i. The item with the highest entropy value is the most unevenly distributed. Besides considering the unevenness, we have to consider the support count of item X, too. If the support count of X is very small, then we gain little by sorting on X. So, we should consider both the unevenness and the support count of each item X in order to determine the sort key. We multiply the entropy value with the total support count of the item in the whole database (which can

be computed by summing up all signature vectors). The product gives us a measure of how frequent and how uneven an item is in the database. The item which gets the highest value for this product is chosen for the sort key, because it is both frequent and unevenly distributed in the database. In other words, we choose as the sort key the item X which has the largest value of $\pi_X = (\text{entropy}_{i=1}^{z} \gamma_{i,X}) \cdot (\sum_{i=1}^{z} \gamma_{i,X})$ The item with the second highest value for this product is used for the secondary sort key. We can similarly determine tertiary and quaternary sort keys, etc.

Sorting the signatures according to keys so selected will yield reasonably high skewness. However, the balance factor is not good. The balance factor is primarily guaranteed by the equal size of the groups G_j. However, this is not sufficient. This is because the sorting will tend to move the signatures with large coordinate values and hence the large itemsets to the same group. The "heavier" signatures will be concentrated in a few groups, while the "lighter" signatures are concentrated in a few other groups. Since the coordinate values are indeed support counts, the sorting would thus distribute the workloads unevenly. To partly overcome this problem, we sort on the primary key in ascending order of coordinate values (which is the support count of the item), the secondary key in descending order of the coordinate values, etc. By alternating the direction of sorting in successive keys, our algorithm can distribute the workload quite evenly, while maintaining a reasonably high skewness.

This idea of using auxiliary sort keys and alternating sort orders is illustrated in Table 4. Here, the primary sort key is k_1, while k_2 is the secondary sort key. The primary sort key is processed in ascending order, while the secondary key is processed in descending order. Note how the secondary sort key has helped assigning γ_6 and γ_3 to one group and γ_2 to another, when they have the same value in the primary sort key. This has helped in increasing the skewness by gathering the support counts of k_2 for those signatures sharing the same value in the primary sort key. Note also how the alternating sort order has slightly improved the balance, by assigning one less unit of support count, carried by C_2 (corresponding to γ_2), to G_2.

So, this resulting partitioning algorithm, which we shall call "Sorting by Highest Entropy Item", abbreviated as "SHEI", should thus give high balance and reasonably good skewness.

5.5 Balanced k-means Clustering ($Bk\mu$)

The balanced k-means clustering algorithm, which we will abbreviate as "$Bk\mu$", is a modification of the k-means algorithm. The $k\mu$ algorithm achieves the skewness criterion. However, it does not pay any effort to achieve to the balance criterion. In $Bk\mu$, we remedy this by assigning the signatures γ_i to the groups G_j while minimizing the value of the following expression. We also add the *constraint* that each group receives

the same number of signatures. The problem is stated as follows.

$$\begin{array}{ll} \text{Minimize} & F = \frac{1}{\lambda+\mu}\left(\lambda \sum_{i=1}^{z}\sum_{j=1}^{n} ||\gamma_i - \alpha_j||^2 \cdot \delta(i,j) + \mu \sum_{j=1}^{n} \sigma_j \cdot ||\alpha_j - E||^2\right) \\ \text{subject to} & \begin{cases} \sigma_{j_1} = \sigma_{j_2} \qquad (j_1, j_2 = 1, 2, \ldots, n) \\ \delta(i,j) = 0 \text{ or } 1 \qquad (i = 1, 2, \ldots, z; \quad j = 1, 2, \ldots, n) \end{cases} \end{array}$$

where $E = \frac{1}{z}\sum_{j=1}^{n} \sigma_j \theta_j$ is constant (which depends on the whole database) and λ, μ are constant control parameters. Actually, E is the arithmetic mean of all the signatures. (Note that σ_j and θ_j, $j = 1, \ldots, n$, have been defined in Section 5.2.) All α_j's and $\delta(i,j)$'s are variables in the above problem. Each α_j represents the geometric centroid of each group G_j, while each $\delta(i,j)$ takes the value of 1 or 0 accordingly as whether the signature γ_i is currently assigned to group G_j or not.

Note that the first term inside parenthesis is exactly the skewness criterion when we set $\alpha_j = \theta_j$. Thus, minimizing this term brings about high skewness. The second term inside parenthesis is introduced so as to achieve balance. Since the vector E is the average of all signatures, it gives the *ideal* value of θ_j (the position of the geometric centroids of each group of signatures) for high balance. The second term measures how far away the actual values of α_j are from this ideal value. Minimizing this term would bring us balance. Therefore, minimizing the above expression would achieve both high balance and skewness. The values of λ and μ let us control the weight of each criterion. A higher value of λ gives more emphasis on the skewness criterion, while a higher value of μ would make the algorithm focus on achieving high balance. In our experiments (see Section 6), we set $\lambda = \mu = 1.0$. In addition, we use the Euclidean distance function in the calculations.

This minimization problem is not trivial. Therefore, we take an iterative approach, based on the framework of the $k\mu$ algorithm. We first make an arbitrary initial assignment of the signatures to the groups, thus giving an initial value for each $\delta(i,j)$. Then, we iteratively improve this assignment to lower the value of the objective function.

Each iteration is divided into two steps. In the first step, we treat the values of $\delta(i,j)$ as constants and try to minimize F by assigning suitable values to each α_j. In the next step, we treat all α_j as constants and adjust the values of $\delta(i,j)$ to minimize F. Thus, the values of α_j and $\delta(i,j)$ are adjusted alternatively to reduce minimize value of F. The details are as follows. In each iteration we first use the same approach as $k\mu$ to calculate the geometric centroids θ_j for each group. To reduce the value of the second term (balance consideration) in the objective function F, we treat temporarily the values of $\delta(i,j)$ as constants and find out the partial derivatives of the object function w.r.t. each α_j. Solving $\frac{\partial F}{\partial \alpha_j} = 0$ $(j = 1, 2, \ldots, n)$, we find that we shall make the assignments $\alpha_j = \frac{\lambda\theta_j + \mu E}{\lambda+\mu}$ where $j = 1, 2, \ldots, n$ in order to minimize the objective function F. After determining α_j, we next adjust the values of $\delta(i,j)$, treating the values of α_j as constants. Since the second term inside parenthesis now does not involve any variable $\delta(i,j)$, we may

reduce the minimization problem to the following problem:

$$\begin{array}{ll} \text{Minimize} & \sum_{i=1}^{z}\sum_{j=1}^{n}||\gamma_i - \alpha_j||^2 \cdot \delta(i,j) \\ \text{subject to} & \begin{cases} \sigma_{j_1} = \sigma_{j_2} & (j_1, j_2 = 1, 2, \ldots, n) \\ \delta(i,j) = 0 \text{ or } 1 & (i = 1, 2, \ldots, z; \quad j = 1, 2, \ldots, n) \end{cases} \end{array}$$

where $\delta(i,j)$ are the variables. Note that this is a linear programming problem. We shall call it the "generalized assignment problem". It is indeed a generalization of the Assignment Problem and a specialization of the Transportation Problem in the literature of linear programming. There are many efficient algorithms for solving such problems. The Hungarian algorithm [9], which is designed for solving the Assignment Problem, has been extended to solve the generalized assignment problem. This extended Hungarian algorithm is incorporated as a part of the $Bk\mu$ clustering algorithm. Like $k\mu$, $Bk\mu$ iteratively improves its solution. The iterations are stopped when the assignment becomes stable.

Since the $Bk\mu$ algorithm imposes the constraint that each group gets assigned the same number[11] of signatures, workload balanced is guaranteed in the final partitioning. Under this constraint, it strives to maximize the skewness (by minimizing signature-centroid distance) like the $k\mu$ algorithm does. So, the skewness is in a reasonably high level. The $Bk\mu$ algorithm actually produces very high balance, and while maintaining such high balance workload, it attempts to maximize skewness. So, essentially the balance factor is given primary consideration. This should suit FPM well.

6 Experimental Evaluation of the Partitioning Algorithms

To find out whether the partitioning algorithms introduced in Section 5 are effective, we have done two sets of experiments. In these experiments, we first generate synthetic databases. The generated databases are already partitioned, with the desired skewness and workload balance. So, the databases are intrinsically non-uniform. This is, however, not suitable for the experiments for evaluating whether our partitioning algorithms can dig out the skewness and workload balance from a database. So, we "destroy" the apparent skewness and workload balance that already exist among the partitions, by concatenating the partitions to form a centralized database and then shuffling the transactions in the concatenated database. The shuffling would destroy the orderness of the transaction, so that arbitrary partitioning of the resulting database would give a partitioning with low balance and skewness. We can then test whether our partitioning algorithms can produce partitions that give higher workload balance and skewness than an arbitrary partitioning.

The databases used here are similar to those generated in Section 4.3. The number of partitions in the databases is 16. Each partition has 100,000 transactions. Chunk size is set to 1000 transactions;

[11] Practically, the constraint is relaxed to allow up to a difference of 1 between the σ_j's, due to the remainder when the number of signatures is divided by the number of groups

hence each partition has 100 chunks, and the total number of chunks is 1600. In order to evaluate the effectiveness of the partitioning algorithms, we have to compare the skewness and workload balance of the resulting partitions against the skewness and balance intrinsic in the database. For this purpose, we take the skewness and workload balance *before* concatenation as the *intrinsic* values. All the skewness and balance values reported below are obtained by measurement on the partitions before the concatenation as well as after the partitioning, not the corresponding control values for the data generation. So, they reflect the actual values for those metrics.

For each generated database, we run all the four partitioning algorithms given in Section 5. The skewness and workload balance of the resulting partitions are noted and compared with one another together with the intrinsic values. As discussed in Section 5.3, the result of the random algorithm suggests the highest achievable balance value, while the result of $k\mu$ gives the highest achievable skewness value.

We did two series of experiments : (1) the first series varied the intrinsic skewness while keeping the intrinsic balance at a high level; (2) the second series varied the intrinsic balance value while keeping the intrinsic skewness almost constant.

6.1 Effects of High Intrinsic Balance and Varied Intrinsic Skewness

The first series of experiments we did was to find out how the intrinsic balance and skewness values would affect the effectiveness of SHEI and $Bk\mu$ given that the intrinsic balance is in a high value and the skewness changes from high to low. Figure 7 shows the results for the skewness of the resulting partitionings. The vertical axis gives the skewness values of the resulting databases. Every four points on the same vertical line represent the results of partitioning the same initial database. The intrinsic skewness of the database is given on the horizontal axis. For your reference, the intrinsic balance values are given in parenthesis directly under the skewness value of the database. Different curves in the figure show the results of different partitioning algorithms.

The $k\mu$ algorithm, as explained before gives the highest skewness achievable by a suitable partitioning. Indeed, the resulting skewness values of $k\mu$ are very close to the intrinsic values. Both SHEI and $Bk\mu$ do not achieve this skewness value. This is primarily because they put more emphasis on balance than skewness. Yet, the results show that some of the intrinsic skewness can be recovered. According to the figure, the resulting skewness of $Bk\mu$ is almost always twice of that of SHEI. So, the $Bk\mu$ algorithm performs better than SHEI in terms of resulting skewness. This is due to the fact that $Bk\mu$ uses a more sophisticated method of achieving high skewness. Most importantly, the skewness achieved by $Bk\mu$ is between 50% to 60% of that of the benchmark algorithm $k\mu$, which indicates that $Bk\mu$ can maintain a significant degree of the intrinsic skewness.

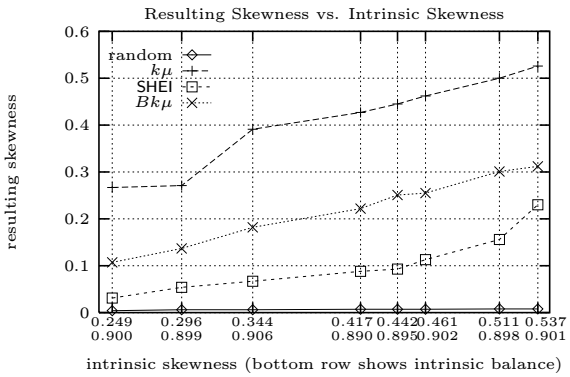


Figure 7: (left) High intrinsic balance and varied intrinsic skewness – resulting skewness

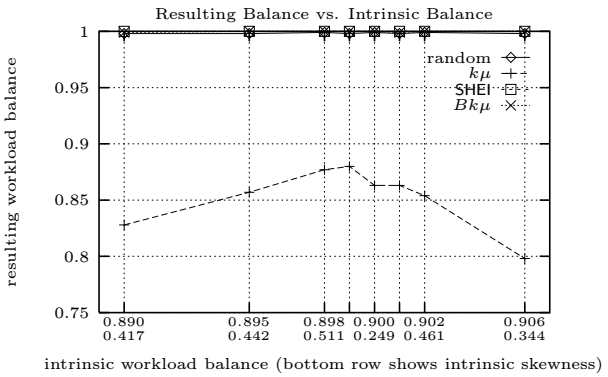


Figure 8: (right) High intrinsic balance and varied intrinsic skewness – resulting balance

Figure 8 shows the workload balance values for the same partitioned databases. This time, the vertical axis shows the resulting balance. Again, every four points on the same vertical line represent the partitioning of the same original database. The horizontal axis gives the intrinsic balance values of the databases, with the intrinsic skewness values of the corresponding databases given in parenthesis.

The generated databases all have an intrinsic balance very close to 0.90. Random partitioning of course yields a high balance value very close to 1.0, which can be taken as the highest achievable balance value. It is encouraging to discover that SHEI and $Bk\mu$ also give good balance values which are very close to that of the random partitioning.

From this series of experiments, we can conclude that: given a database with good intrinsic balance, even if the intrinsic skewness is not high, both $Bk\mu$ and SHEI can increase the balance to a level as good as that in a random partitioning; in addition, $Bk\mu$ can at the same time deliver a good level of skewness, much better than that from the random partitioning, the skewness achieved by $Bk\mu$ is also better than SHEI and is in an order comparable to what can be achieved by the benchmark algorithm $k\mu$.

Another way to look at the result of these experiments is to fit the intrinsic skewness and balance value pairs (those on the horizontal axis of Figure 7) into the regions in Figure 6. These pairs, which represent the intrinsic skewness and balance of the initial partitions, all fall into region C. Combining the results from Figures 7 and 8, among the resulting partitions from $Bk\mu$, five of them have moved to region A, the most favorable region. For the other three, even though their workload balance have been increased, their skewness have not been increased enough to move them out of region C. In summary, a high percentage of the resulting partitions have been benefited substantially from using $Bk\mu$.

6.2 Effects of Reducing Intrinsic Balance

Our second series of experiments attempted to find out how SHEI and $Bk\mu$ would be affected when the intrinsic balance is reduced to a lower level.

Figure 9 presents the resulting skewness values against the intrinsic skewness. The numbers in parenthesis show the intrinsic balance values for the corresponding databases. Figure 10 shows the resulting balance values of the four algorithms on the same partitioning results.

Again, the random algorithm suggests the highest achievable balance value, which is close to 1.0 in all cases. Both SHEI and $Bk\mu$ are able to achieve the same high balance value which is the most important requirement (Figure 10). Thus, they are very good at yielding a good balance even if the intrinsic balance is low. As for the resulting skewness, the results of the $k\mu$ algorithm gives the highest achievable skewness values, which are very close to the intrinsic values (Figure 9). Both SHEI and $Bk\mu$ can recover parts of the intrinsic skewness. However, the skewness is reduced more when the intrinsic balance is in the less favorable range (< 0.7). These results are consistent with our understanding. When the intrinsic balance is low, spending effort in re-arranging the transactions in the partitions to achieve high balance would tend to reduce the skewness.

Both $Bk\mu$ and SHEI are low-cost algorithms, they spend more effort in achieving a better balance while at the same time trying to maintain certain level of skewness. Between $Bk\mu$ and SHEI, the resulting skewness of $Bk\mu$ is at least twice of that of SHEI in all cases. This shows that $Bk\mu$ is better than SHEI in both cases of high and low intrinsic balance. It is also important to note that the skewness achieved by $Bk\mu$ is always better than that of the random partitioning.

Again, we can fit the skewness and balance values of the initial databases and their resulting partitions (Figures 9 and 10) into the regions in Figure 6. What we have found out is that: after the partitioning performed by $Bk\mu$, four partitionings have moved from region C to A, two from region D to C, two others remain unchanged. This again is very encouraging and shows the effectiveness of $Bk\mu$ — more than 70% of the databases would have their performance improved substantially by using $Bk\mu$ and FPM together.

6.3 Summary of the Experimental Results

The above results show that our clustering algorithms SHEI and $Bk\mu$ are very good preprocessors, which prepare a partitioned database that can be mined by FPM efficiently. It is encouraging to note that both $Bk\mu$ and SHEI can achieve a balance as good as random partitioning and also a much better skewness. Between themselves, $Bk\mu$ in general gives much better skewness values than SHEI. Also, the results are true for a wide range of intrinsic balance and skewness values. Referring back to Section 4.3, what we

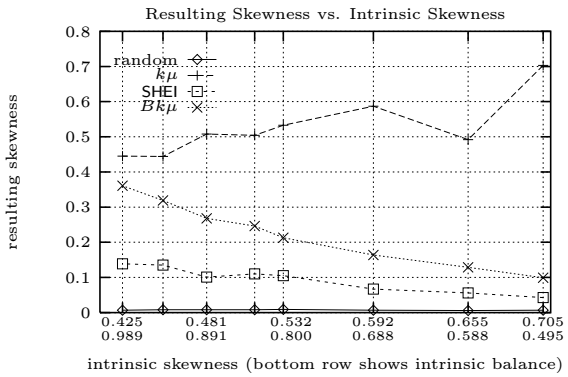


Figure 9: (left) Reducing intrinsic balance – resulting skewness

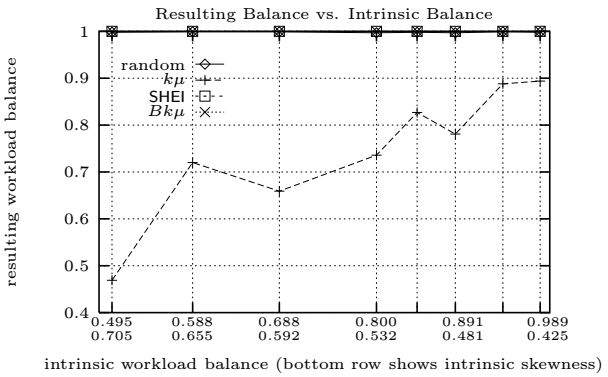


Figure 10: (right) Reducing intrinsic balance – resulting balance

have achieved is being able to partition a database such that the workload balance would fall into the ideal high level range while at the same time maintaining certain level of skewness. Therefore, the resulting partitioning would fall into the favorable regions in Figure 6. Given this result, we recommend using $Bk\mu$ for the partitioning (or repartitioning) of the database before running FPM.

Note that we did no study on the time performance of the partitioning algorithms. This is primarily because the algorithms are so simple that they consume negligible amounts of CPU time. In our experiments, the amount of CPU time is no more than 5% of the time spent by the subsequent running of FPM. As for I/O overhead, the general framework of the partitioning algorithms (see Section 5.1) require only one extra scan of the database, whose purpose is to calculate the signatures γ_i. This cost can be compensated by the saving of the first database scan of FPM (see Section 7.1). After that, no more extra I/O is required. We assume that the chunk size y, specified by the user, is large enough so that the total number of chunks z is small enough to allow all the signatures γ_i be handled in main memory. In this case, the total overhead of the partitioning algorithms is far compensated by the subsequent resource savings. For a more detailed discussion on the overhead of these partitioning algorithms, please refer to Section 7.1.

7 Discussion

To restrict the search of large itemset in a small set of candidates is very essential to the performance of mining association rules. After a database is partitioned over a number of processors, we have information on the support counts of the itemsets at a finer granularity. This enables us to use distributed and global prunings discussed in Section 3. However, the effectiveness of these pruning techniques are very dependent on the distribution of transactions among the partitions. We discuss two issues here related to the database partitioning and performance of FPM.

7.1 Overhead of Using the Partitioning Algorithms

We have already shown that FPM benefits over CD the most when the database is partitioned in a way such that the skewness and workload balance measures are high. Consequently, we suggest that partitioning algorithms such as $Bk\mu$ and SHEI be used before FPM, so as to increase the skewness and workload balance for FPM to work faster. But this suggestion is good only if the overhead of the partitioning is not high. We will make this claim by dividing the overhead of the partitioning algorithms into two parts for analysis.

The first part is CPU cost. First, the partitioning algorithms calculate the signatures of the data chunks. This involves only simple arithmetic operations. Next, the algorithms call a clustering algorithm to divide the signatures into groups. Since the amount of signatures is much smaller than the number of transactions in the whole database, the algorithms process much less information than a mining algorithm. Moreover, the clustering algorithms are designed to be simple, so that they are computationally not costly. Finally, the program delivers transaction in the database to different partitions. This involves little CPU cost. So, overall, the CPU overhead of the partitioning algorithms is very low. Experimental results show that it is no more than 5% of the CPU cost of the subsequent run of FPM.

The second part is the I/O cost. The partitioning algorithms in Section 5 all read the original database twice and write the partitioned database to disk once. In order to enjoy the power of parallel machines for mining, we have to partition the database anyway. Comparing with the simplest partitioning algorithm, which must inevitably read the original database once and write the partitioned database to disk once, our partitioning algorithms only does one extra database scan. But it shall be remarked that in our clustering algorithm, this extra scan is for computing the signatures of the chunks. Once the signatures are found, the support counts of *all* 1-itemsets can be deduced by summation, which involves no extra I/O overhead. So, we can indeed find out the support counts of all 1-itemsets essentially for free. This can be exploited to eliminate the first iteration of FPM, so that FPM can start straight into the second iteration to find large 2-itemsets. This saves one database scan from FPM, and hence as a whole, the 1-scan overhead of the partitioning algorithm is compensated.

Thus, the partitioning algorithms essentially introduce negligible CPU and I/O overhead to the whole mining activity. It is therefore worthwhile to employ our partitioning algorithms to partition the database before running FPM. The great savings by running FPM on a carefully partitioned database far compensates the overhead of our partitioning algorithms.

7.2 Scalability in FPM

Our performance studies of FPM were carried out on a 32-processor SP2 (section 4.3). If the number of processors n is very large, global pruning may need a large memory to store the local support counts from all the partitions for all the large itemsets found in an iteration. Also, there could be cases that candidates generated after pruning is still too large to fit into the memory. We suggest to use a cluster approach to solve this problem. The n processors can be grouped into p clusters, ($p \ll n$), so that each cluster would have $\frac{n}{p}$ processors. In the top level, support counts will be exchanged between the p clusters instead of the n processors. The counts exchanged in this level will be the sum of the supports from the processors within each cluster. Both distributed and global prunings can be applied by treating the data in a cluster together as a partition. Within a cluster, the candidates are distributed across the processors, and the support counts in this second level can be computed by count exchange among the processors inside the cluster. In this approach, we only need to ensure that the total distributed memory of the processors in each cluster is large enough to hold the candidates. From the setting, this approach is highly scalable. In fact, we can regard this approach as an integration of our pruning techniques into the parallel algorithm HD (section 1). Our pruning techniques is orthogonal to the parallelization technique in HD and can be used to enhance its performance.

8 Conclusions

A parallel algorithm FPM for mining association rules has been proposed. FPM is a modification of FDM and it requires fewer rounds of message exchanges. Performance studies carried out on an IBM SP2 shared-nothing memory parallel system show that FPM consistently outperforms CD. The gain in performance in FPM is due mainly to the pruning techniques incorporated.

It has been found that the effectiveness of the pruning techniques depend highly on two data distribution characteristics: data skewness and workload balance. An entropy-based metric has been proposed to measure these two characteristics. Our analysis and experiment results show that the pruning techniques are very sensitive to workload balance, though good skewness will also have important positive effects. The techniques are very effective in the best case of high balance and high skewness. The combination of high balance and moderate skewness is the second best case.

This is our motivation to introduce algorithms to partition database in a wise way, so as to get higher balance and skewness values. We have compared four partitioning algorithms. With the balanced k-means ($Bk\mu$) clustering algorithm, we can achieve a very high workload balance, while at the same time a reasonably good skewness. Our experiments have demonstrated that many unfavorable partitions can be

repartitioned by $Bk\mu$ into partitions that allow FPM to perform more efficiently.

Moreover, the overhead of the partitioning algorithms is negligible, and can be compensated by saving one database scan in the mining process. Therefore, we can obtain very high association rule mining efficiency by partitioning a database with $Bk\mu$, and then mining it with FPM. We have also discussed a cluster approach which can bring scalability to FPM.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. of 1993 ACM-SIGMOD Int. Conf. On Management of Data*, Washington, D.C., 1993.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th VLDB Conference*, Santiago, Chile, 1994.

[3] R. Agrawal and J.C. Shafer. Parallel mining of association rules: Design, implementation and experience. Technical Report TJ10004, IBM Research Division, Almaden Research Center, 1996.

[4] S. Brin, R. Motwani, J. Ullman, S. Tsur. Dynamic itemsets counting and implication rules for market basket data. In *Proc. of 1997 ACM-SIGMOD Int. Conf. On Management of Data*, 1997

[5] T. M. Cover, T. A. Thomas. Elements of information theory. John Wiley & Sons, Inc., 1991.

[6] D. W. Cheung, J. Han, V. Ng and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proc. of the 12th Int. Conf. on Data Eng.*, New Orleans, Louisiana, 1996.

[7] D. W. Cheung, J. Han, V. T. Ng, A. W. Fu, Y. Fu. A fast distributed algorithm for mining association rules. In *Proc. of 4th Int. Conf. on Parallel and Distributed Information Systems*, 1996

[8] D. W. Cheung, V. T. Ng, A. W. Fu, and Y. Fu. Efficient Mining of Association Rules in Distributed Databases. Special Issue in Data Mining, *IEEE Trans. on Knowledge and Data Engineering*, IEEE Computer Society, V8, N6, December 1996, pp. 911–922.

[9] S.K. Gupta, "*Linear Programming and Netword Models*", Affiliated East-West Press Private Limited, New Delhi, Madras HyderBrabad Bangalore (ISBN: 81-85095-08-6)

[10] E. Han, G. Karypis and V. Kumar. Scalable parallel data mining for association rules. In *Proc. of 1997 ACM-SIGMOD Int. Conf. On Management of Data*, 1997

[11] J Han and Y Fu. Discovery of multiple-level association rules from large databases. In *Proc. of the 21th VLDB Conference*, Zurich, Switzerland, 1995.

[12] M. A. W. Houtsma and A. N. Swami. Set-oriented mining for association rules in relational databases. In *Proc. of the 11th Int. Conf. on Data Eng.*, Taipei, 1995.

[13] Int'l Business Machines. *Scalable POWERparallel Systems*, GA23-2475-02 edition, February 1995

[14] L. Kaufman, and P.J. Rousseeuw. *Finding Groups in Data : An Introduction to Cluster Analysis.*, John Wiley & Sons, 1990.

[15] H. Mannila, H. Toivonen, and A. I. Verkamo. Efficient algorithms for discovering association rules. In *KDD-94: AAAI Workshop on Knowledge Discovery in Databases*, July 1994.

[16] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, May 1994.

[17] J. S. Park, M . S. Chen, and P. S. Yu, An effective hash-based algorithm for mining association rules. In *Proc. of 1995 ACM-SIGMOD Int. Conf. on Management of Data*, San Jose, CA, May 1995.

[18] J. S. Park, M. S. Chen, and P. S. Yu, Efficient parallel mining for association rules. In *Proc. of the 4th Int. Conf. on Information and Knowledge Management*, Baltimore, Maryland, 1995.

[19] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of the 21th VLDB Conference*, Zurich, Switzerland, 1995.

[20] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proc. of the 21th VLDB Conference*, Zurich, Switzerland, 1995.

[21] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. of the 5th Int. Conf. on Extending Database Technology*, Avignon, France, 1996.

[22] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In *Proc. 1996 ACM-SIGMOD Int. Conf. on Management of Data*, Montreal, Canada, 1996.

[23] T. Shintani, M. Kitsuregawa. Hash based parallel algorithms for mining association rules. In *Proc. of 4th Int. Conf. on Parallel and Distributed Information Systems*, 1996

[24] H. Toivonen. Sampling large databases for mining association rules. In *Proc. of the 22th VLDB Conference*, 1996

[25] M. J. Zaki, M. Ogihara, S. Parthasarathy, and W. Li, Parallel data mining for association rules on shared-memory multi-processors. Technical Report 618, Computer Science Department, The University of Rochester, May 1996.