Mining Association Rules: Anti-Skew Algorithms *

Jun-Lin Lin and Margaret H. Dunham
Department of Computer Science and Engineering
Southern Methodist University
Dallas, Texas 75275-0122, USA

Abstract

Mining association rules among items in a large database has been recognized as one of the most important data mining problems. All proposed approaches for this problem require scanning the entire database at least or almost twice in the worst case. In this paper we propose several techniques which overcome the problem of data skew in the basket data. These techniques reduce the maximum number of scans to less than 2, and in most cases find all association rules in about 1 scan. Our algorithms employ prior knowledge collected during the mining process and/or via sampling, to further reduce the number of candidate itemsets and identify false candidate itemsets at an earlier stage.

1 Introduction

Recently, data mining has attracted much attention in the database community [3, 4]. One of the most investigated topics in data mining is the problem of discovering *association rules* over *basket data* [1, 6, 8, 9]. Basket data, which typically contains items purchased by a customer, are collected at the point-of-sales system in a retail business. Each basket data defines a *transaction*. With the advance of bar-code technology, businesses can effectively gather a vast amount of basket data. However, to effectively extract useful knowledge over such basket data to assist decision making is a major challenge.

Mining association rules over basket data was first introduced in [1]. Since then, association rules have been applied to other databases as well, for examples, telecommunication alarm data and university course enrollment data [6]. An association rule has three parts: the *head*, the *body*, and the *confidence* of the rule. Both the head and the body are a set of items. For example, if 90% of the customers who purchase A and B, also purchase C and D, then it can be represented as an association rule as follows.

$$\{A, B\} \rightarrow \{C, D\} \text{ with } confidence = 90\%$$

In order to generate reliable results, the size of the database must be large [9]. Thus, an efficient algorithm to discover the association rules must try to reduce I/O as much as possible. Many algorithms have been proposed in the literature [1, 6, 8, 9]. Among them, the *Partition* algorithm [8] and the *Sampling* algorithm [9] require the least amount of I/O to accomplish the task. Both the Partition algorithm and the Sampling algorithm require two complete scans over the database in the worst case and one complete scan in the best case. Also, a variation of the Partition algorithm called SPINC [7] requires $\frac{2n-1}{n}$ scans in the worst case, where n is the number of partitions.

In this paper we propose and investigate a family of algorithms called *Anti-Skew Counting Partition Algorithms (AS-CPA)*. Like SPINC, AS-CPA only needs to scan the database once in the best case, and $\frac{2n-1}{n}$ times in the worst case. Several techniques have been incorporated into AS-CPA to reduce the possibility of worst case behavior. In case a random sample can be drawn, a Random Sampling version of AS-CPA (denoted as RSAS-CPA) outperforms the Sampling algorithm whenever the Sampling algorithm fails to find all association rules during the first scan. RSAS-CPA has the same best case performance (i.e., 1 scan) as the Sampling algorithm. In case a random sample can not be drawn without scanning the database from the beginning, a Sequential Sampling version of AS-CPA (denoted as SSAS-CPA) can be applied. It is important to note that our techniques employ prior knowledge to remove false candidates in an earlier stage which in turn reduces the CPU and memory overhead.

The rest of this paper is organized as follows. Section 2 gives a formal description of the problem, and describes previous work. Sections 3 and 4 present AS-CPA and Sampling AS-CPA, respectively. Section 5 gives a performance comparison among these algorithms, and Section 6 concludes this paper.

2 Mining Association Rules

This section gives a formal description of the problem of mining association rules over basket data, mostly based on the description in [1].

Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of literals called *items*. Let $\mathcal{D}$ be a set of transactions. Each transaction is a non-empty subset of $\mathcal{I}$, and is associated with a unique identifier called *TID*.

A set of items is called an *itemset*. An itemset with k items in it, is called a k-itemset. Each itemset $X \subset \mathcal{I}$ has a measurement of statistical significance

*This research is based upon work supported by a Massive Digital Data System (MDDS) effort sponsored by the Advanced Research and Development Committee of the Community Management Staff.

in $\mathcal{D}$, called *support*, where $support(X, \mathcal{D})$ equals the fraction of transactions in $\mathcal{D}$ containing X.

An *association rule* is an implication of the form $X \Rightarrow Y$, where $X, Y \subset \mathcal{I}$, and $X \cap Y = \emptyset$. Each rule $X \Rightarrow Y$ has a measurement of strength, called *confidence*, where $confidence(X \Rightarrow Y) = \frac{support(X \cup Y, \mathcal{D})}{support(X, \mathcal{D})}$.

The problem of mining association rules is to find *all* the association rules of the form $X \Rightarrow Y$ that satisfy the following two conditions:

1. $support(X \cup Y, \mathcal{D}) \geq min\_support$
2. $confidence(X \Rightarrow Y) \geq min\_confidence$

Here, $min\_confidence$ and $min\_support$ are user specified. This problem can be decomposed into two subproblems:

1. Find all the itemsets that have support above $min\_support$. These itemsets are called the *large* itemsets.
2. For each large itemset X and any $Y \subset X$, check if rule $(X - Y) \Rightarrow Y$ has confidence above $min\_confidence$.

After finding all large itemsets, the second subproblem can be solved in a straightforward manner. However, finding all large itemsets is not trivial because the size of $\mathcal{D}$ and the number of subsets of $\mathcal{I}$ can be very large. Thus, most previous work focuses on how to find all the large itemsets efficiently. The rest of this paper will also focus on this subproblem only.

Several algorithms have been proposed on finding large itemsets [1, 6, 8, 9]. All previous approaches to finding large itemsets construct a set of *candidate* itemsets, and verify if a candidate itemset is a large itemset. Since the size of the database can be very large, most algorithms aim at reducing the number of scans needed over the entire database and the number of candidate itemsets.

2.1 Level-Wise Algorithms

The idea behind level-wise algorithms is based on the following observation. If X is a large itemset, then any non-empty subset of X is also a large itemset.

Let L_k be the set of all large k-itemsets, and C_k be the set of candidate k-itemsets. Level-wise algorithms [4, 6] use L_k to generate C_{k+1} so that C_{k+1} is a superset of L_{k+1}. Then, the algorithms construct L_{k+1} by scanning the database once to check if c is a large itemset, for each $c \in C_{k+1}$. The same process repeats for the next level until no large itemset is found. As a result, if the largest large itemset is a j-itemset, then the level-wise algorithms need to scan the database at most $(j + 1)$ times.

2.2 Partition Algorithm

The idea behind the Partition algorithm [8] is based on lemma 1.

Lemma 1 *If X is a large itemset in database $\mathcal{D}$, which is divided into n partitions $p_1, p_2, \ldots, p_n$, then X must be a large itemset in at least one of the n partitions.*

Proof: Prove by contrapositive. Assume that X is not a large itemset in any of the n partitions, and prove that X must be a small itemset in $\mathcal{D}$. □

The Partition algorithm divides $\mathcal{D}$ into n partitions, and processes one partition in main memory at a time. The algorithm first scans partition p_i, for $i = 1$ to n, to find the set of all local large itemsets in p_i, denoted as L^{p_i}. Then, by taking the union of L^{p_i} for $i = 1$ to n, a set of candidate itemsets over $\mathcal{D}$ is constructed, denoted as C^G. Based on Lemma 1, C^G is a superset of the set of all large itemsets in $\mathcal{D}$. Finally, the algorithm scans each partition for the second time to calculate the support of each itemset in C^G and to find out which candidate itemsets are really large itemsets in $\mathcal{D}$. Thus, only two scans are needed to find all the large itemsets in $\mathcal{D}$. Only one scan is needed if all partitions have identical local large itemsets.

SPINC is a variation of the Partition algorithm proposed in [7]. Instead of constructing C^G by taking the union of L^{p_i} for $i = 1$ to n at the end of the first scan as in the Partition algorithm, it constructs C^G incrementally by adding L^{p_i} to C^G whenever L^{p_i} is available. SPINC starts counting the number of occurrences for each candidate itemset $c \in C^G$ as soon as c is added to C^G. Thus, after the first scan, if SPINC added the last candidate itemset to C^G when processing partition p_j, then it only needs to scan up to partition p_{j-1} for the second scan. Thus, in the best case only 1 scan is needed, and in the worst case $\frac{2n-1}{n}$ scans are needed.

2.3 Sampling Algorithm

Using sampling to assist finding all association rules has been discussed in [6, 9, 10]. Among them, the Sampling algorithm proposed by [9] has the best performance. The Sampling algorithm [9] first takes a random sample of the database $\mathcal{D}$, and finds the set of large itemsets (denoted as S) in the sample using a smaller $min\_support$ than the one the user specified. Then, the algorithm calculates the set of itemsets, which are in the $Bd^-(S)$. $Bd^-(S)$ is the set of minimal itemsets $X \subset \mathcal{I}$ that X is not in S [9]. The algorithm scans $\mathcal{D}$ to check if c is a large itemset in $\mathcal{D}$, for each itemset $c \in S \cup Bd^-(S)$.

If there is no large itemset in $Bd^-(S)$, the algorithm has found all the large itemsets. Otherwise, the algorithm constructs a set of candidate itemsets (denoted as C^G) by expanding the negative border of $S \cup Bd^-(S)$ recursively until the negative border is empty. Finally, the algorithm scans $\mathcal{D}$ for the second time to check if c is a large itemset for each $c \in C^G$. In the best case, this algorithm needs only 1 scan over $\mathcal{D}$. In the worst case, two scans are needed. It should be noted that the number of candidate itemsets generated for the second scan can be very large if a bad sample was drawn for the first scan. As a result, the second scan can be very inefficient. Also, using a smaller $min\_support$ for the sample and expanding with the negative border could result in a large C^G, making the first scan very inefficient.

3 Anti-Skew Counting Partition Algorithms

One of the major problems of the Partition algorithm (and the SPINC algorithm as well) is data skew, which refers to the irregularity of data distribution over the entire database $\mathcal{D}$. An example from [8] is severe weather conditions causing the sales of some items to increase rapidly only for a short period of time. Data skew can cause both algorithms to generate many false candidate itemsets. One way to overcome data skew is to increase the randomness of data across all partitions [8]. However, this conflicts with the goal of exploiting sequential I/O to speed up reading the database.

Even without data skew, unless each item is distributed rather uniformly over $\mathcal{D}$, and the size of each partition is large enough to capture this uniformness, the chance of a local large itemset (i.e., a candidate itemset) being a global large itemset can be small, and the number of candidate itemsets generated can be very large. It is obvious that the above problems can be mitigated with large partitions. However, this conflicts with the main idea of the Partition algorithm: processing one partition in memory at a time to avoid multiple scans over $\mathcal{D}$ from disk.

In what follows, we propose a variation of the SPINC algorithm, called *Anti-Skew Counting Partition Algorithm (AS-CPA)*, that makes use of the cumulative *count* of each candidate itemset to achieve the illusion of a large partition. Like the Partition Algorithm, AS-CPA also divides the database $\mathcal{D}$ into n disjoint partitions, and processes one partition at a time. A partition is a subset of transactions contained in $\mathcal{D}$. The notation shown in Table 1 is partly based on the notation in [8]. An itemset x is a *local* large itemset in partition p_i if $support(x, p_i) \geq min\_support$. An itemset x is a *global* large itemset if $support(x, \mathcal{D}) \geq min\_support$.

Like the SPINC algorithm, AS-CPA adds L^i to C^G as soon as L^i is available during the first scan. AS-CPA starts counting the number of occurrences for each candidate itemset c as soon as c is added to C^G. Each candidate itemset $c \in C^G$ has two attributes: $c.age$ contains the partition number when c was added to C^G, and $c.count$ contains the number of occurrences of c since c was added to C^G. At the end of the first scan, we already have the number of occurrences over $\mathcal{D}$ for each candidate itemset $c \in C^G$ with $c.age = 1$, thus we can check if c is a large itemset immediately. After that, we remove those itemsets with $age = 1$ from C^G. Then, we start the second scan on partition p_1, and count the number of occurrence over p_1 for each itemset $c \in C^G$ and add it to $c.count$. At this moment, we have the the number of occurrences over $\mathcal{D}$ for each candidate itemset $c \in C^G$ with $c.age = 2$, thus we can check if c is a large itemset and remove c from C^G. Repeat the same procedure for partitions $p_2, p_3, \ldots, p_i$ until C^G is empty.

So far, the description of AS-CPA is just the same as SPINC. The main feature that separates AS-CPA from SPINC is that AS-CPA provides several effective techniques to filter out false candidate itemsets at an earlier stage. This reduces the probability of worse case behavior. We discuss these techniques in the following three subsections.

DB^i	Set of all 1-itemsets in partition p_i and their tidlists
L_1^i	Set of all local large 1-itemsets in partition p_i
L^i	Set of all local large itemsets (not including 1-itemsets) in partition p_i
$L_1^{1\ldots i}$	Set of all local large 1-itemsets in $\bigcup_{j=1\ldots i} p_j$
B_1^i	Set of all better local large 1-itemsets in partition p_i
B^i	Set of all better local large itemsets (not including 1-itemsets) in partition p_i
C^G	Set of all global candidate itemsets
L^G	Set of all global large itemsets

Table 1: Notation

3.1 Early Local Pruning

To generate the set L^i of all local large itemsets in a partition p_i, both the Partition algorithm and SPINC first find the set of all local large 1-itemsets in p_i and use a level-wise algorithm to find all the local large itemsets in p_i. With *early local pruning*, we take a similar but more efficient approach.

The idea of *early local pruning* is as follows. When reading a partition p_i to generate L_1^i, we record and accumulate the number of occurrences for each item. Thus, after reading partition p_i, we know both the number of occurrences for each item in partition p_i, and the number of occurrences for each item in the *big* partition $\bigcup_{j=1,2,\ldots,i} p_j$. With this information, we know both the set L_1^i of all local large 1-itemsets in p_i, and the set $L_1^{1\ldots i}$ of all local large 1-itemsets in the big partition $\bigcup_{j=1,2,\ldots,i} p_j$. We define B_1^i as the set of all *better* local large 1-itemsets in partition p_i, where $B_1^i = L_1^i \cap L_1^{1\ldots i}$. With *early local pruning*, we use B_1^i instead of L_1^i to start the level-wise algorithm to construct the set B^i of all *better* local large itemsets in partition p_i. Since the size of B_1^i is usually smaller than that of L_1^i, B^i can be constructed faster than L^i.

With Partition algorithm, Lemma 1 ensures that $C^G = \bigcup_{i=1,2,\ldots,n} L^{p_i}$ is a superset of the set L^G of all global large itemsets. With early local pruning, Lemma 2 shows that $C^G = \bigcup_{i=1,2,\ldots,n} B^{p_i}$ is also a superset of L^G.

Lemma 2 *Let the database $\mathcal{D}$ be divided into n partitions, $p_1, p_2, \ldots, p_n$. Let $C^G = \bigcup_{i=1,2,\ldots,n} B^i$. Then, C^G is a superset of L^G.*

Proof: prove by mathematical induction. We denote the set of all global large itemsets of a database $\mathcal{D}$ as $L^G(\mathcal{D})$.

BASE CASE: $\mathcal{D}$ contains only 1 partition, Lemma 2 is true (trivial).
INDUCTION HYPOTHESIS: if $\mathcal{D} = \bigcup_{i=1,2,\ldots,n} p_i$, then $\bigcup_{i=1,2,\ldots,n} B^i$ is a superset of $L^G(\mathcal{D})$. That is,

$$L^G(\mathcal{D}) \subset \bigcup_{i=1,2,\ldots,n} B^i$$

INDUCTION STEP: Consider a new database $\mathcal{D}' = \mathcal{D} \cup p_{n+1}$, and prove that $\bigcup_{i=1,2,\ldots,n,(n+1)} B^i$ is a superset of $L^G(\mathcal{D}')$. Consider $\mathcal{D}'$ as a database with 2 partitions: $\mathcal{D}$ and p_{n+1}. According to Lemma 1 and the Induction Hypothesis, we have,

$$\begin{aligned} L^G(\mathcal{D}') &\subset L^G(\mathcal{D}) \cup L^{n+1} \\ &\subset \bigcup_{i=1,2,\ldots,n} B^i \cup L^{n+1} \end{aligned}$$

(case 1) $L_1^{n+1} \subset L_1^{1\ldots(n+1)}$:

$$\begin{aligned} & B_1^{n+1} = L_1^{n+1} \cap L_1^{1\ldots(n+1)} = L_1^{n+1} \\ \Longrightarrow \quad & B^{n+1} = L^{n+1} \\ \Longrightarrow \quad & L^G(\mathcal{D}') \subset \bigcup_{i=1,2,\ldots,n,(n+1)} B^i \end{aligned}$$

(case 2) L_1^{n+1} is not a subset of $L_1^{1\ldots(n+1)}$: there exists some 1-itemset $l \in L_1^{n+1}$ but not in $L_1^{1\ldots(n+1)}$. Since l is not in $L_1^{1\ldots(n+1)}$, any superset of l is not in $L^G(\mathcal{D}')$. Thus, we can remove all such l from L_1^{n+1}, and eventually L_1^{n+1} will become a subset of $L_1^{1\ldots(n+1)}$; case 2 degrades to case 1. □

3.2 First Global Anti-Skew

The *first global anti-skew* technique aims at removing *possibly* false global candidate itemsets during the first scan over the database $\mathcal{D}$. It can be used with or without the early local pruning technique. In what follows, we first discuss how it is used without the early local pruning technique, and then discuss how it can be used with the early local pruning technique.

Lemma 3 *Let the database $\mathcal{D}$ be divided into n partitions, $p_1, p_2, \ldots, p_n$. Assume that itemset x is not a local large itemset in either of the following two big partitions: $\bigcup_{j=1,2,\ldots,k-1} p_j$, and $\bigcup_{j=k,k+1,k+2,\ldots,i} p_j$. If x is a large itemset over $\mathcal{D}$, x must be a local large itemset in at least one of the following partitions: $p_{i+1}, p_{i+2}, \ldots, p_n$.*

Proof: Lemma 3 is the direct result of Lemma 1. □

Without early local pruning, C^G is defined as $\bigcup_{i=1,2,\ldots,n} L^{p_i}$ as in the Partition algorithm, and the first global anti-skew technique is based on Lemma 3. Note that for each $c \in C^G$, c is not a local large itemset in $\bigcup_{j=1,2,\ldots,c.age-1} p_j$ because c was added to C^G when processing partition $p_{c.age}$. During the first scan, after processing partition p_i, for each itemset $c \in C^G$, $c.count$ contains the number of occurrences of c in $\bigcup_{j=c.age,c.age+1,\ldots,i} p_j$. Thus, we can check if c is a local large itemset in the big partition $BIG = \bigcup_{j=c.age,c.age+1,\ldots,i} p_j$. If c is not a local large itemset in BIG, then c is a *possibly* false candidate itemset, and the *first global anti-skew* technique removes c from C^G. If c is really a global large itemset in $\mathcal{D}$, according to Lemma 3, c must be a local large itemset in at least one of the following partitions: $p_{i+1}, p_{i+2}, \ldots, p_n$, and thus c will be added back to C^G again.

Notice here we can also remove any superset s of c from C^G when we remove c from C^G. The reason is as follows. Since c was added to C^G when processing partition $p_{c.age}$, c is not a local large itemset in $\bigcup_{j=1,2,\ldots,c.age-1} p_j$. This, together with the fact that c is not a local large itemset in $\bigcup_{j=c.age,c.age+1,\ldots,i} p_j$, indicates that c is not a local large itemset in $\bigcup_{j=1,2,\ldots,i} p_j$, according to Lemma 1. Since s is a superset of c, s is not a local large itemset in $\bigcup_{j=1,2,\ldots,i} p_j$ either. If we remove s from C^G when processing partition p_i but s is actually a global large itemset in $\mathcal{D}$, according to Lemma 1, s must be a local large itemset in at least one of the following partitions: $p_{i+1}, p_{i+2}, \ldots, p_n$, and thus s will be added back to C^G again.

To avoid removing and adding the same itemset repeatedly and to reduce the added CPU overhead of the *first global anti-skew* technique, this procedure is applied to each $c \in C^G$ just once as soon as c is considered *old* enough. That is, if k partitions are considered large enough to capture the uniformness of the database $\mathcal{D}$, then when $c.count$ contains the number of occurrences of c in k partitions, this procedure is applied to c, i.e. after processing partition $p_{c.age+k-1}$.

With early local pruning, C^G is defined as $\bigcup_{i=1,2,\ldots,n} B^{p_i}$, and the first anti-skew technique is based on Lemma 4 below.

Lemma 4 *Let the database $\mathcal{D}$ be divided into n partitions, $p_1, p_2, \ldots, p_n$. Assume that itemset x is not a better local large itemset in either of the following two big partitions: $\bigcup_{j=1,2,\ldots,k-1} p_j$, and $\bigcup_{j=k,k+1,k+2,\ldots,i} p_j$. If x is a large itemset over $\mathcal{D}$, x must be a better local large itemset in at least one of the following partitions: $p_{i+1}, p_{i+2}, \ldots, p_n$.*

Proof: According to Lemma 2, if $c \in C^G$ is a global large itemset in $\mathcal{D}$, then c must be a *better* local large itemset in at least one of the partitions of $\mathcal{D}$. Lemma 4 is the direct result of Lemma 2. □

3.3 Second Global Anti-Skew

The *second global anti-skew* technique is used to remove false global candidate itemsets during the second scan. It can be used with or without the earlier local pruning and the first global anti-skew techniques. In what follows, we first show how the second anti-skew technique is used without the earlier local pruning and the first anti-skew techniques. Then, we show how to use the second anti-skew technique with the first anti-skew technique.

Lemma 5 *Let the database $\mathcal{D}$ be divided into n partitions, $p_1, p_2, \ldots, p_n$. If itemset x is not a local large itemset in any of the following partitions: $p_1, p_2, \ldots, p_{k-1}$, and $\bigcup_{j=k,k+1,k+2,\ldots,n} p_j \cup \bigcup_{j=1,2,3,\ldots,i} p_j$, where $i < k$, then x is not a large itemset in $\mathcal{D}$.*

Proof: Since x is not a local large itemset in any of the following partitions: $p_1, p_2, \ldots, p_{k-1}$, x is not a local large itemset in $\bigcup_{j=i+1,i+2,\ldots,k-1} p_j$. Imagine there are only two partitions in $\mathcal{D}$: $\bigcup_{j=k,k+1,k+2,\ldots,n} p_j \cup \bigcup_{j=1,2,3,\ldots,i} p_j$ and $\bigcup_{j=i+1,i+2,\ldots,k-1} p_j$. Lemma 5 is the direct result of Lemma 1. □

During the second scan, after processing partition p_i, for each itemset $c \in C^G$, $c.count$ contains the number of occurrences of c in $\bigcup_{j=c.age,c.age+1,\ldots,n} p_j \cup \bigcup_{j=1,2,\ldots,i} p_j$. Since c is added to C^G when processing partition $p_{c.age}$, c is not a local large itemset in any of the following partitions: $p_1, p_2, \ldots, p_{c.age-1}$. Here, we assume the first anti-skew and early local pruning techniques are not used. According to Lemma 5, if c is not a large itemset in $\bigcup_{j=c.age,c.age+1,\ldots,n} p_j \cup \bigcup_{j=1,2,\ldots,i} p_j$, c is not a large itemset in $\mathcal{D}$, and thus we can remove c and all supersets of c from C^G.

A less restricted version of Lemma 5 is needed in order to support the combination of the *first* and *second* anti-skew techniques. With first anti-skew technique, a candidate itemset c can be added to and removed from C^G many times so that c might be a local large itemset in one of the following partitions: $p_1, p_2, \ldots, p_{c.age-1}$. Thus, we can no longer apply Lemma 5 to support the second anti-skew technique. Lemma 6 shows that both anti-skew techniques can coexist.

Lemma 6 *Let the database $\mathcal{D}$ be divided into n partitions, $p_1, p_2, \ldots, p_n$. Assume that, at the end of the first scan, an itemset $c \in C^G$ has $c.age = k$. If c is not a local large itemset in $\bigcup_{j=k,k+1,k+2,\ldots,n} p_j \cup \bigcup_{j=1,2,3,\ldots,i} p_j$, where $i < k$, then c is not a large itemset in $\mathcal{D}$.*

The proof for Lemma 6 can be found in [5]. Both Lemmas 5 and 6 are based on Lemma 1. Since Lemma 1 to local large itemsets is just the same as Lemma 2 to better local large itemsets, we can derive from Lemma 2 in a similar fashion to prove that the second anti-skew technique can be used with the early local pruning technique as well.

3.4 Algorithms

AS-CPA is a family of algorithms. Each uses one or more techniques to improve its performance, as shown in Table 2. Here we only list the algorithm for AS-CPA-12E. AS-CPA has adopted some procedures from their counterparts in the Partition algorithm [8]. As in [2, 8], we assume that a transaction in the database is in the form of $< TID, i_j, i_k, \ldots, i_x >$, and the items in a transaction are in lexicographic order.

Name	early local pruning	1st global anti-skew	2nd global anti-skew
AS-CPA-12	no	yes	yes
AS-CPA-2	no	no	yes
AS-CPA-12E	yes	yes	yes
AS-CPA-2E	yes	no	yes

Table 2: Algorithms

procedure AS-CPA-12E

1. P = partition_database($\mathcal{D}$)
2. n = Number of partitions
3. m = Number of transactions in $\mathcal{D}$
4. k = number of partitions to be considered as a big partition
5. **for** $i = 1$ to n
6. $\quad m[i]$ = Number of transactions in partition p_i
7. $C^G = \emptyset$
8. **for** $i = 1$ to n **begin** // *phase I*
9. $\quad acc[i] = m[i]$
10. $\quad$ **for** $j = 1$ to $(i-1)$
11. $\quad\quad acc[j] = acc[j] + m[i]$
12. $\quad DB^i$ = read_and_transpose($p_i \in P$)
13. $\quad$ add_count(C^G, DB^i)
14. $\quad B_1^i$ = gen_better_large_1-itemsets(DB^i)
15. $\quad B^i$ = gen_large_itemsets(B_1^i)
16. $\quad$ **for** each itemset $l \in B^i - C^G$ **begin**
17. $\quad\quad l.age = i$
18. $\quad\quad l.count = |l.tidlist|$
19. $\quad\quad C^G = C^G \cup \{l\}$
20. $\quad$ **end**
21. $\quad C^G$ = first_anti_skew($C^G, acc[1..i]$)
22. **end**
23. C^G = second_anti_skew($C^G, acc[i,n]$) // *phase II*
24. $L^G = \{c \in C^G | c.age == 1\}$
25. $C^G = C^G - L^G$
26. $i = 1$
27. **while** ($C^G \neq \emptyset$) **begin**
28. $\quad DB^i$ = read_and_transpose($p_i \in P$)
29. $\quad$ add_count(C^G, DB^i)
30. $\quad$ **for** $j = (i+1)$ to n
31. $\quad\quad acc[j] = acc[j] + m[i]$
32. $\quad i = i + 1$
33. $\quad C^G$ = second_anti_skew($C^G, acc[i,n]$)
34. $\quad L^G = L^G \cup \{c \in C^G | c.age == i\}$
35. $\quad C^G = C^G - \{c \in C^G | c.age == i\}$
36. **end**
37. **return** L^G

Initially, the database $\mathcal{D}$ is partitioned into n partitions by executing procedure **partition_database**, and C^G is empty. During the *phase I*, the algorithm processes one partition at a time for all partitions. When processing partition p_i, the content of p_i is read and transformed into DB^i, which contains a *tidlist* data structure for each item in p_i. Here, the *tidlist* of an item x contains the TID of all the transactions in partition p_i that contains x. Thus, the length of the *tidlist* of an item x, denoted as $|x.tidlist|$, equals to the number of occurrences of x over partition p_i. With DB^i, the algorithm then calls procedure **add_count** to count the number of occurrences for each itemset $c \in C^G$, and add it to $c.count$. [1]

[1] The *tidlist* data structure is proposed in [8]. By taking the intersection of the *tidlist* of all items in an itemset, the length of the intersection gives the number of occurrences of the itemset.

In AS-CPA-12E, we use the *early local pruning* technique, thus we also accumulate the number of occurrences for each item and construct the set $L_1^{1...i}$ of all local large 1-itemsets in $\bigcup_{j=1,2,...,i} p_j$ in procedure **add_count**. Then, the algorithm calls procedure **gen_better_large_1-itemsets** to generate the set B_1^i of all the better local large 1-itemsets and their associated *tidlist*. Again, due to the *early local pruning* technique, the B_1^i generated here is $L_1^i \cap L_1^{1...i}$. At this stage, the content of DB^i is no longer needed.

procedure add_count(C^G, DB^i)

1. $L_1^{1...i} = \emptyset$
2. **for** each $x \in \mathcal{I}$ **begin**
3. $x.count = x.count + |DB_x^i.tidlist|$
4. **if** $\frac{x.count}{acc[1]} \geq min\_support$
5. $L_1^{1...i} = L_1^{1...i} \cup \{x\}$
6. **end**
7. **for** each $c \in C^G$ **begin**
8. num = number of occurrences of c in partition p_i
9. $c.count = c.count + num$
10. **end**

procedure gen_better_large_1-itemsets(DB^i)

1. $L_1^i = \emptyset$
2. **for** each $x \in \mathcal{I}$ **begin**
3. **if** $\frac{|DB_x^i.tidlist|}{acc[i]} \geq min\_support$
4. $L_1^i = L_1^i \cup \{x\}$
5. **end**
6. **return** $L_1^i \cap L_1^{1...i}$

The algorithm then calls procedure **gen_large_itemsets** to generate the set of all the local better large itemsets in partition p_i, denoted as B^i. Then, the algorithm starts to add itemsets from B^i to C^G. When adding an itemset l from B^i to C^G, we differentiate those in B^i that are already in C^G (i.e., old candidate itemsets) and those in B^i that are not yet in C^G (i.e., new candidate itemsets). If l is a old candidate, do nothing since its number of occurrences in partition p_i has been already added to $l.count$ when calling procedure **add_count**. If l is a new candidate, we assign partition number i to $l.age$, assign the number of occurrences of l in p_i to $l.count$, and add l to C^G. Then, the algorithm calls procedure **first_anti_skew** to remove possibly false candidate itemsets and their supersets in C^G.

procedure first_anti_skew ($C^G, acc[i..j]$)

1. **for** each $c \in C^G$ with $c.age == i - k + 1$ **begin**
2. **if** $\frac{c.count}{acc[c.age]} < min\_support$
3. $C^G = C^G - \{s \in C^G | c \subseteq s\}$
4. **end**

The same process repeats for the rest of the partitions, then the algorithm enters the *phase II*. Note that, at this moment, for each candidate itemset $c \in C^G$, $c.count$ contains the number of occurrences of c from partition $p_{c.age}$ to partition p_n.

During the *phase II*, the algorithm calls procedure **second_anti_skew** to remove some false candidate itemsets in C^G. Then, the algorithm moves each $c \in C^G$ with $c.age = 1$ to L^G. Then, the algorithm reads in partition 1, and updates the count for each itemset $c \in C^G$. The same process repeats until C^G is empty.

procedure second_anti_skew ($C^G, acc[i..j]$)

1. **for** each $c \in C^G$ **begin**
2. **if** $\frac{c.count}{acc[c.age]} < min\_support$
3. $C^G = C^G - \{s \in C^G | c \subseteq s\}$
4. **end**

Note that, the only complex procedure called by CPA is **gen_large_itemsets**. Since AS-CPA uses the same *tidlist* data structure to count the number of occurrences of an itemset as in Partition algorithm, the same counterpart in Partition algorithm can be used in CPA with very little modification. Please refer [8] for **gen_large_itemsets**.

4 Sampling Anti-Skew Counting Partition Algorithm

In this section, we present a sequential sampling version of AS-CPA called *SSAS-CPA*, that uses the first few partitions as a sample to locate all large itemsets at an earlier stage. With SSAS-CPA, the user also specifies how many partitions are used as the sample. If the first k partitions are used as sample, then SSAS-CPA uses a *min_support* smaller than the one the user specified to determine the local large itemsets for those partitions. The rest of the algorithm is just the same as AS-CPA.

The idea of using a smaller *min_support* for the sample was proposed in the Sampling algorithm [9]. However, instead of calculating the negative border of S to check if all large itemsets have been found at the end of the first scan as in [9], SSAS-CPA simply executes procedure *second_anti_skew* and checks if there are any itemsets in C^G with age greater than 1. The Sampling algorithm in [9] needs a second *complete* scan over $\mathcal{D}$ if it finds any large itemsets not in S. SSAS-CPA needs a second *partial* scan over $\mathcal{D}$ if it finds any large itemsets with age greater than 1.

There are several advantages in SSAS-CPA. First, the sampling phase of SSAS-CPA is actually part of the first scan, and thus it does not add extra overhead like the Sampling algorithm [9]. Second, in some cases, a random sample cannot be drawn without scanning the database $\mathcal{D}$ from the beginning. An example for this kind of database is a database with variant length transactions, and there is no delimiter between transactions. In these cases, a random sampling approach could require an extra scan over the database, and thus a sequential sampling approach like SSAS-CPA is a better way to go. In case a random sample can be drawn without too much overhead, we can simply use algorithm AS-CPA but first add all local large

$\|\mathcal{D}\|$	number of transactions
$\|T\|$	average number of items per transaction
$\|I\|$	average number of items of maximal potentially large itemsets
$\|L\|$	number of maximal potentially large itemsets
N	number of items

Table 3: Parameters

Name	$\|T\|$	$\|I\|$	$\|\mathcal{D}\|$	N
T5.I2.100K	5	2	100K	1000
T10.I4.100K	10	4	100K	1000

Table 4: Parameter Settings

itemsets (with age = 1) found in the sample to C^G. We denote the random sampling version of AS-CPA as RSAS-CPA. The algorithm for RSAS-CPA can be found in [5].

5 Performance Results

We have implemented the various AS-CPA algorithms and examined their performance using a standard set of synthetic test data [2]. Since the second global anti-skew technique detects false candidate itemsets with little overhead, it is treated as a required part of AS-CPA. We treat the first global anti-skew and the early local pruning techniques as the optional parts of AS-CPA. We also apply the early local pruning technique to SPINC, and the resulting algorithm is denoted as SPINC-E. Below we briefly examine some of the results.

The results of two data sets are discussed here. Table 3 shows the notation for each parameter, and Table 4 shows the name and parameter setting for each data set. All algorithms in this experiment divides the database into 5 partitions.

Tables 5 and 6 show number of scans for the various algorithms with each data set. The columns are labeled with the *min_support* value. We were not able to actually implement the Sampling algorithm as the sampling code is proprietary, so we have shown a value of 1^+. This is because the best behavior for the Sampling algorithm is one scan. The worst is 2. The actual number of scans required depends on the sampling as well as the data. In [9] experiments were repeated 100 times for each test database and the number of scans was reported as the average of these. The values plotted in this work were slightly above 1. The overall percent of retrievals with misses was 0.0038. When a miss occurs the database must be read again completely. This requires another scan. In addition, the Sampling algorithm requires extra I/O to do the sampling. Experiments done in [9] use a sample size from 20% to 80% of the database. Thus coming up with an exact number here is difficult. As can be seen, though, the sequential sampling AS-CPA algorithms has about the same performance as the Sampling algorithm based on number of scans. Certainly further work with more data and real data (rather than synthetic data) will help to determine the best overall approach. The SSAS-CPA algorithms, however, do appear to be quite promising.

	0.0025	0.0050	0.0075	0.0100
Partition	2	2	2	1
SPINC	1.8	1.8	1.8	1
AS-CPA-2	1.8	1.8	1.2	1
AS-CPA-12	1.8	1.8	1.6	1
SPINC-E	1.8	1.8	1.8	1
AS-CPA-2E	1.8	1.8	1.2	1
AS-CPA-12E	1.8	1.8	1.6	1
SSAS-CPA-2	1	1	1	1
SSAS-CPA-2E	1	1	1	1
Sampling	1^+	1^+	1^+	1^+

Table 5: Number of Scans for T5.I2.D100K

	0.0025	0.0050	0.0075	0.0100
Partition	2	2	2	2
SPINC	1.8	1.8	1.8	1.8
AS-CPA-2	1.8	1.8	1.4	1.2
AS-CPA-12	1.8	1.8	1.8	1.2
SPINC-E	1.8	1.8	1.8	1.8
AS-CPA-2E	1.8	1.8	1.4	1.2
AS-CPA-12E	1.8	1.8	1.8	1.2
SSAS-CPA-2	1	1	1	1
SSAS-CPA-2E	1	1	1	1
Sampling	1^+	1^+	1^+	1^+

Table 6: Number of Scans for T10.I4.D100K

Figure 1 shows the number of global candidate itemsets generated by the various algorithms with the two data sets. The early local pruning versions of AS-CPA are not shown in Figure 1 since their number of global candidate itemsets is about the same as their counterpart without early local pruning. The early local pruning technique aims at reducing the number of local candidate itemsets, not the global candidate itemsets.

During the first phase (scanning the 5 partitions the first time), AS-CPA-12 generates the smallest number of global candidate itemset. During the second phase, AS-CPA-2 has the smallest. Certainly since SPINC generates a smaller global candidate itemset size than the regular Partition algorithm our anti-skew approaches are also better than this. Compared with non-sampling algorithms, SSAS-CPA-2 has a much large number of global candidate itemsets. Although not shown, we know that the Sampling algorithm would have a much larger number of global candidate itemsets due to the fact that it adds those in the negative border. Notice that for these test data SSAS-CPA only had one scan in most case, but less candidate itemsets were generated.

Using the number of global candidate itemsets as a performance measure allows us to estimate CPU and memory utilization of the various algorithms. Certainly more work is needed to investigate these further.

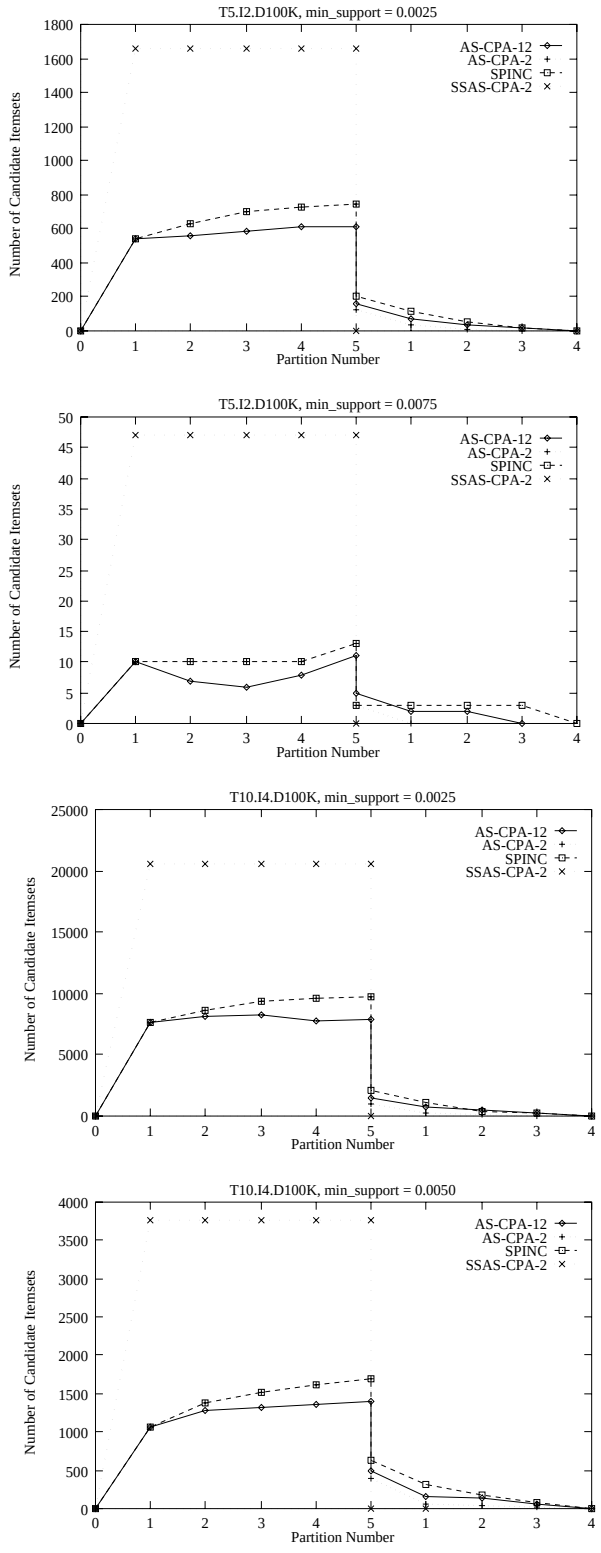

Figure 1: Number of Global Candidate Itemsets.

6 Conclusions and Future Work

In this paper, we propose several techniques to speed up the process of finding association rules among basket data. Our techniques employ prior or accumulated knowledge of the data processed, to prune false candidates at an early stage. Future work includes further performance studies of AS-CPA, early identification of possibly large itemsets before scanning, parallel version of AS-CPA on a share-nothing architecture, and rule maintenance with AS-CPA after adding data to or removing data from the database.

References

[1] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1993.

[2] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *Proceedings of the 20th International Conference on Very Large Databases*, 1994.

[3] M. Chen, J. Han, and P. S. Yu. Data mining: An overview from a database perspective. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):866–883, December 1996.

[4] U. M. Fayyad, G. P. Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. AAAI Press, 1996.

[5] J. Lin and M. H. Dunham. Mining association rules: Anti-skew algorithms. Technical Report 97-CSE-4, Southern Methodist University, Computer Science Department, 1997. *Available from http://www.seas.smu.edu/~jun/icde98.ps.gz.*

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

[7] A. Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. Technical Report CS-TR-3515, Dept. of Computer Science, Univ. of Maryland, College Park, MD, 1995.

[8] A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the 21st International Conference on Very Large Databases*, 1995.

[9] H. Toivonen. Sampling large databases for association rules. In *Proceedings of the 22nd International Conference on Very Large Databases*, 1996.

[10] M. J. Zaki, S. Parthasarathy, W. Li, and M. Ogihara. Evaluation of sampling for data mining of association rules. Technical Report TR 617, University of Rochester, Computer Science Department, 1996.