An Efficient Algorithm for Mining Association Rules in Large Databases

Ashok Savasere Edward Omiecinski Shamkant Navathe

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

e-mail: {ashok,edwardo,sham}@cc.gatech.edu

Abstract

Mining for association rules between items in a large database of sales transactions has been described as an important database mining problem. In this paper we present an efficient algorithm for mining association rules that is fundamentally different from known algorithms. Compared to previous algorithms, our algorithm not only reduces the I/O overhead significantly but also has lower CPU overhead for most cases. We have performed extensive experiments and compared the performance of our algorithm with one of the best existing algorithms. It was found that for large databases, the CPU overhead was reduced by as much as a factor of four and I/O was reduced by almost an order of magnitude. Hence this algorithm is especially suitable for very large size databases.

1 Introduction

Database mining is motivated by decision support problems faced by most business organizations and is described as an important area of research [12, 13]. One of the main challenges in database mining is developing fast and efficient algorithms that can handle large volumes of data because most mining algorithms perform computation over the entire database and often the databases are very large.

Discovering association rules between items over *basket* data was introduced in [1]. Basket data typically consists of items bought by a customer along with the date of transaction, quantity, price, etc. Such data may be collected, for example, at supermarket checkout counters. Association rules identify the set of items that are most often purchased with another set of items. For example, an association rule may state that "95% of customers who bought items A and B also bought C and D." Association rules may be used for catalog design, store layout, product placement, target marketing, etc.

Many algorithms have been discussed in the literature for discovering association rules [1, 8, 2]. One of the key features of all the previous algorithms is that they require multiple passes over the database. For disk resident databases, this requires reading the database completely for each pass resulting in a large number of disk I/Os. In these algorithms, the effort spent in performing just the I/O may be considerable for large databases. Apart from poor response times, this approach also places a huge burden on the I/O subsystem adversely affecting other users of the system. The problem can be even worse in a client-server environment.

In this paper, we describe an algorithm called *Partition*, that is fundamentally different from all the previous algorithms in that it reads the database at most two times to generate all significant association rules. Contrast this with the previous algorithms, where the database is not only scanned multiple times but the number of scans cannot even be determined in advance. Surprisingly, the savings in I/O is not achieved at the cost of increased CPU overhead. We have performed extensive experiments and compared our algo-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 21st VLDB Conference
Zurich, Swizerland, 1995

rithm with one of the best previous algorithms. Our experimental study shows that for computationally intensive cases, our algorithm performs better than the previous algorithm in terms of both CPU and I/O overhead.

Other related, but not directly applicable work in database mining are reported in [7, 10, 6, 9, 3, 14, 15].

The paper is organized as follows: in the next section, we give a formal description of the problem. In Section 2, we describe the problem and give an overview of the previous algorithms. In section 3, we describe our algorithm. Performance results are described in section 4. Section 5 contains conclusion and future work.

2 Problem Description

This section is largely based on the description of the problem in [1] and [2]. Formally, the problem can be stated as follows: Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of m distinct literals called *items*[1]. $\mathcal{D}$ is a set of variable length transactions over $\mathcal{I}$. Each transaction *contains* a set of items $i_i, i_j, \ldots, i_k \subset \mathcal{I}$. A transaction also has an associated unique identifier called *TID*. An *association rule* is an implication of the form $X \Longrightarrow Y$, where $X, Y \subset \mathcal{I}$, and $X \cap Y = \emptyset$. X is called the antecedent and Y is called the consequent of the rule.

In general, a set of items (such as the antecedent or the consequent of a rule) is called an *itemset*. The number of items in an itemset is called the *length* of an itemset. Itemsets of some length k are referred to as k–itemsets. For an itemset $X \cdot Y$, if Y is an m–itemset then Y is called an *m-extension* of X.

Each itemset has an associated measure of statistical significance called *support*. For an itemset $X \subset \mathcal{I}$, *support(X)* $= s$, if the fraction of transactions in $\mathcal{D}$ containing X equals s. A rule has a measure of its strength called *confidence* defined as the ratio *support(X* $\cup$ *Y) / support(X)*.

The problem of mining association rules is to generate all rules that have support and confidence greater than some user specified minimum support and minimum confidence thresholds, respectively. This problem can be decomposed into the following subproblems:

1. All itemsets that have support above the user specified minimum support are generated. These itemset are called the *large* itemsets. All others are said to be *small*.

2. For each large itemset, all the rules that have minimum confidence are generated as follows: for a large itemset X and any $Y \subset X$, if *support(X)/support(X* $-$ *Y)* $\geq$ *minimum_confidence*, then the rule $X - Y \Longrightarrow Y$ is a valid rule.

For example, let $T_1 = \{A, B, C\}$, $T_2 = \{A, B, D\}$, $T_3 = \{A, D, E\}$ and $T_4 = \{A, B, D\}$ be the only transactions in the database. Let the minimum support and minimum confidence be 0.5 and 0.8 respectively. Then the large itemsets are the following: $\{A\}, \{B\}, \{D\}, \{AB\}, \{AD\}$ and $\{ABD\}$. The valid rules are $B \Longrightarrow A$ and $D \Longrightarrow A$.

The second subproblem, i.e., generating rules given all large itemsets and their supports, is relatively straightforward. However, discovering all large itemsets and their supports is a nontrivial problem if the cardinality of the set of items, $\mid \mathcal{I} \mid$, and the database, $\mathcal{D}$, are large. For example, if $\mid \mathcal{I} \mid = m$, the number of possible distinct itemsets is 2^m. The problem is to identify which of these large number of itemsets has the minimum support for the given set of transactions. For very small values of m, it is possible to setup 2^m counters, one for each distinct itemset, and count the support for every itemset by scanning the database once. However, for many applications m can be more than 1,000. Clearly, this approach is impractical. To reduce the combinatorial search space, all algorithms exploit the following property: any subset of a large itemset must also be large. Conversely, all extensions of a small itemset are also small. This property is used by all existing algorithms for mining association rules as follows: initially support for all itemsets of length 1 (1-itemsets) are tested by scanning the database. The itemsets that are found to be small are discarded. A set of 2-itemsets called *candidate itemsets* are generated by extending the large 1-itemsets generated in the previous pass by one (1-extensions) and their support is tested by scanning the database. Itemsets that are found to be large are again extended by one and their support is tested. In general, some kth iteration contains the following steps:

1. The set of candidate k-itemsets is generated by 1-extensions of the large $(k-1)$-itemsets generated in the previous iteration.

2. Supports for the candidate k–itemsets are generated by a pass over the database.

3. Itemsets that do not have the minimum support are discarded and the remaining itemsets are called large k-itemsets.

This process is repeated until no more large itemsets are found.

[1] In this paper we use the terminology introduced in [1]

2.1 Previous Work

The problem of generating association rules was first introduced in [1] and an algorithm called *AIS* was proposed for mining all association rules. In [8], an algorithm called *SETM* was proposed to solve this problem using relational operations. In [2], two new algorithms called *Apriori* and *AprioriTid* were proposed. These algorithms achieved significant improvements over the previous algorithms. The rule generation process was also extended to include multiple items in the consequent and an efficient algorithm for generating the rules was also presented.

The algorithms vary mainly in (a) how the candidate itemsets are generated; and (b) how the supports for the candidate itemsets are counted. In [1], the candidate itemsets are generated on the fly during the pass over the database. For every transaction, candidate itemsets are generated by extending the large itemsets from previous pass with the items in the transaction such that the new itemsets are contained in that transaction. In [2] candidate itemsets are generated in a separate step using only the large itemsets from the previous pass. It is performed by joining the set of large itemsets with itself. The resulting candidate set is further pruned to eliminate any itemset whose subset is not contained in the previous large itemsets. This technique produces a much smaller candidate set than the former technique.

Supports for the candidate itemsets are determined as follows. For each transaction, the set of all candidate itemsets that are contained in that transaction are identified. The counts for these itemsets are then incremented by one. In [1] the authors do not describe the data structures used for this subset operation. Apriori and AprioriTid differ based on the data structures used for generating the supports for candidate itemsets.

In Apriori, the candidate itemsets are compared with the transactions to determine if they are contained in the transaction. A hashtree structure is used to restrict the set of candidate itemsets compared so that subset testing is optimized. Bitmaps are used in place of transactions to make the testing fast. In AprioriTid, after every pass, an encoding of all the large itemsets contained in a transaction is used in place of the transaction. In the next pass, candidate itemsets are tested for inclusion in a transaction by checking whether the large itemsets used to generate the candidate itemset are contained in the encoding of the transaction. In Apriori, the subset testing is performed for every transaction in each pass. However, in AprioriTid, if a transaction does not contain any large itemsets in the current pass, that transaction is not considered in subsequent passes. Consequently, in later passes, the size of the encoding can be much smaller than the actual database. A hybrid algorithm is also proposed which uses Apriori for initial passes and switches to AprioriTid for later passes.

3 Partition Algorithm

The idea behind Partition algorithm is as follows. Recall that the reason the database needs to be scanned multiple number of times is because the number of possible itemsets to be tested for support is exponentially large if it must be done in a single scan of the database. However, suppose we are given a small set of potentially large itemsets, say a few thousand itemsets. Then the support for them can be tested in one scan of the database and the actual large itemsets can be discovered. Clearly, this approach will work only if the given set contains all actual large itemsets.

Partition algorithm accomplishes this in two scans of the database. In one scan it generates a set of all potentially large itemsets by scanning the database once. This set is a superset of all large itemsets, i.e., it may contain false positives. But no false negatives are reported. During the second scan, counters for each of these itemsets are set up and their actual support is measured in one scan of the database.

The algorithm executes in two phases. In the first phase, the Partition algorithm logically divides the database into a number of non-overlapping partitions. The partitions are considered one at a time and all large itemsets for that partition are generated. At the end of phase I, these large itemsets are merged to generate a set of all potential large itemsets. In phase II, the actual support for these itemsets are generated and the large itemsets are identified. The partition sizes are chosen such that each partition can be accommodated in the main memory so that the partitions are read only once in each phase.

We assume the transactions are in the form $\langle TID, i_j, i_k, \ldots, i_n \rangle$. The items in a transaction are assumed to be kept sorted in the lexicographic order. Similar assumption is also made in [2]. It is straightforward to adapt the algorithm to the case where the transactions are kept normalized in $\langle TID, item \rangle$ form. We also assume that the *TID*s are monotonically increasing. This is justified considering the nature of the application. We further assume the database resides on secondary storage and the approximate size of the database in blocks or pages is known in advance.

Definition A *partition* $p \subseteq \mathcal{D}$ of the database refers to any subset of the transactions contained in the database $\mathcal{D}$. Any two different partitions are non-overlapping, i.e., $p_i \cap p_j = \emptyset, i \neq j$. We define *local support* for an itemset as the fraction of transactions containing that itemset in a partition. We define a

C_k^p	Set of local candidate k–itemsets in partition p
L_k^p	Set of local large k–itemsets in partition p
L^p	Set of all local large itemsets in partition p
C_k^G	Set of global candidate k–itemsets
C^G	Set of all global candidate itemsets
L_k^G	Set of global large k–itemsets

Table 1: Notation

```
1)  $P$ = partition_database($\mathcal{D}$)
2)  $n$ = Number of partitions
3)  for $i = 1$ to $n$ begin // Phase I
4)      read_in_partition($p_i \in P$)
5)      $L^i$ = gen_large_itemsets($p_i$)
6)  end
7)  for ($i = 2$; $L_i^j \neq \emptyset$, $j = 1, 2, \ldots, n$; $i$++) do
8)      $C_i^G = \cup_{j=1,2,\ldots,n} L_i^j$ // Merge Phase
10) for $i = 1$ to $n$ begin // Phase II
11)     read_in_partition($p_i \in P$)
12)     for all candidates $c \in C^G$ gen_count($c$, $p_i$)
13) end
14) $L^G = \{c \in C^G | c.\text{count} \geq minSup\}$
```

Figure 1: Partition Algorithm

local candidate itemset to be an itemset that is being tested for minimum support within a given partition. A *local large itemset* is an itemset whose local support in a partition is at least the user defined minimum support[2]. A local large itemset may or may not be large in the context of the entire database. We define *global support*, *global large itemset*, and *global candidate itemset* as above except they are in the context of the entire database $\mathcal{D}$. Our goal is to find all global large itemsets.

We use the notation shown in Table 1 in this paper. Individual itemsets are represented by small letters and sets of itemsets are represented by capital letters. When there is no ambiguity we omit the partition number when referring to a local itemset. We use the notation $c[1]\cdot c[2]\cdots c[k]$ to represent a k-itemset c consisting of items $c[1]$, $c[2]$, ..., $c[k]$.

Algorithm The Partition algorithm is shown in Figure 1. Initially the database $\mathcal{D}$ is logically partitioned into n partitions. Phase I of the algorithm takes n iterations. During iteration i only partition p_i is considered. The function **gen_large_itemsets** takes a partition p_i as input and generates local large itemsets of all lengths, $L_1^i, L_2^i, \ldots, L_l^i$ as the output. In the merge phase the local large itemsets of same lengths from all n partitions are combined to generate the global candidate itemsets. In phase II, the algorithm sets up counters for each global candidate itemset and counts their support for the entire database and generates the global large itemsets. The algorithm reads the entire database once during phase I and once during phase II.

[2]The minimum support is specified as a ratio, e.g., 2 %, 0.0037, etc

```
    procedure gen_large_itemsets($p$: database partition)
1)  $L_1^p$ = {large 1–itemsets along with their tidlists}
2)  for ( $k = 2$; $L_k^p \neq \emptyset$; $k$++) do begin
3)      forall itemsets $l_1 \in L_{k-1}^p$ do begin
4)          forall itemsets $l_2 \in L_{k-1}^p$ do begin
5)              if $l_1[1] = l_2[1] \wedge l_1[2] = l_2[2] \wedge \ldots \wedge$
                    $l_1[k-2] = l_2[k-2] \wedge l_1[k-1] < l_2[k-1]$ then
6)                  $c = l_1[1] \cdot l_1[2] \cdots l_1[k-1] \cdot l_2[k-1]$
7)                  if $c$ cannot be pruned then
8)                      $c$.tidlist = $l_1$.tidlist$\cap l_2$.tidlist
9)                      if $|c.\text{tidlist}| \,/\, |p| \geq minSup$ then
10)                         $L_k^p = L_k^p \cup \{c\}$
11)         end
12)     end
13) end
14) return $\cup_k L_k^p$
```

Figure 2: Procedure gen_large_itemsets

Correctness The key to correctness of the above algorithm is that any potential large itemset appears as a large itemset in at least one of the partitions. A more formal proof is given in [11].

3.1 Generation of Local Large Itemsets

The procedure **gen_large_itemsets** takes a partition and generates all large itemsets (of all lengths) for that partition. The procedure is shown in Figure 2. Lines 3–8 show the candidate generation process. The prune step is performed as follows:

```
prune($c$: $k$–itemset)
forall ($k-1$)–subsets $s$ of $c$ do
    if $s \notin L_{k-1}$ then
        return "$c$ can be pruned"
```

The prune step eliminates extensions of $(k-1)$-itemsets which are not found to be large, from being considered for counting support. For example, if L_3^p is found to be {{1 2 3}, {1 2 4}, {1 3 4}, {1 3 5}, {2 3 4}}, the candidate generation initially generates the itemsets {1 2 3 4} and {1 3 4 5}. However, itemset {1 3 4 5} is pruned since {1 4 5} is not in L_3^p. This technique is same as the one described in [2] except in our case, as each candidate itemset is generated, its count is determined immediately.

The counts for the candidate itemsets are generated as follows. Associated with every itemset, we define

```
1) forall 1-itemsets do
2)     generate the tidlist
3) for( $k = 2$; $C_k^G \neq \emptyset$; $k$++) do begin
4)     forall $k$-itemset $c \in C_k^G$ do begin
5)         templist = c[1].tidlist ∩c[2].tidlist ∩ ... ∩ c[k].tidlist
6)         c.count = c.count + | templist |
7)     end
8) end
```

Figure 3: Procedure gen_final_counts

a structure called as *tidlist*. A tidlist for itemset l contains the *TID*s of all transactions that contain the itemset l within a given partition. The *TID*s in a tidlist are kept in sorted order. Clearly, the cardinality of the tidlist of an itemset divided by the total number of transactions in a partition gives the support for that itemset in that partition.

Initially, the tidlists for 1-itemsets are generated directly by reading the partition. The tidlist for a candidate k-itemset is generated by joining the tidlists of the two $(k-1)$-itemsets that were used to generate the candidate k-itemset. For example, in the above case the tidlist for the candidate itemset {1 2 3 4} is generated by joining the tidlists of itemsets {1 2 3} and {1 2 4}.

Correctness It has been shown in [2] that the candidate generation process correctly produces all potential large candidate itemsets. It is easy to see that the intersection of tidlists gives the correct support for an itemset.

3.2 Generation of Final Large Itemsets

The global candidate set is generated as the union of all local large itemsets from all partitions. In phase II of the algorithm, global large itemsets are determined from the global candidate set. This phase also takes n (number of partitions) iterations. Initially, a counter is set up for each candidate itemsets and initialized to zero. Next, for each partition, tidlists for all 1-itemsets are generated. The support for a candidate itemset in that partition is generated by intersecting the tidlists of all 1-subsets of that itemset. The cumulative count gives the global support for the itemsets. The procedure `gen_final_counts` is given in Figure 3. Any other technique such as the subset operation described in [2], can also be used to generate global counts in phase II.

Correctness Since the partitions are non-overlapping, a cumulative count over all partitions gives the support for an itemset in the entire database.

3.3 Discovering Rules

Once the large itemsets and their supports are determined, the rules can be discovered in a straight forward manner as follows: if l is a large itemset, then for every subset a of l, the ratio ***support(l) / support (a)*** is computed. If the ratio is at least equal to the user specified minimum confidence, them the rule $a \Longrightarrow (l - a)$ is output. A more efficient algorithm is described in [2].

As mentioned earlier, generating rules given the large itemsets and their supports is much simpler compared to generating the large itemsets. Hence we have not attempted to improve this step further.

3.4 Size of the Global Candidate Set

The global candidate set contains many itemsets which may not have global support (false candidates). The fraction of false candidates in the global candidate set must be as small as possible otherwise much effort may be wasted in finding the global supports for those itemsets. The number of false candidates depends on many factors such as the characteristics of the data, how the data is partitioned, number of partitions, and so on. In this section we study the effects of partition size and data skew on the size of the global candidate set.

As the number of partitions is increased, the number of false candidates also increases and hence the global candidate size also increases. However, its size is bounded by n times the size of the largest set of local large itemsets, where n is the number of partitions.

The local large itemsets are generated for the same minimum support as specified by the user. Hence this is equivalent to generating large itemsets with that minimum support for a database which is same as the partition. So, for sufficiently large partition sizes, the number of local large itemsets is likely to be comparable to the number of large itemsets generated for the entire database.

Additionally, if the data characteristics are uniform across partitions, then a large number of the itemsets generated for individual partitions may be common. Hence the global candidate set may be smaller than the above limit.

In Table 2 we show the variation in the size of the local large itemsets and the global candidate sets for varying the number of partitions from 2 to 30. The database contained 100,000 transactions[3]. The minimum support was set at 0.75 %. It can be seen from the table that as the number of partitions increases, both the variation in the sizes of local large sets and the size of the global candidate set increases. However, there is a large overlap among the local large itemsets.

[3]The dataset used was T10.I4.100K described in Section 4.1

Number of Partitions	Size of Largest L^p	Average size of L^p	Size of C^G
2	91	89.0	93
4	100	82.5	108
7	131	97.0	144
10	149	109.1	170
20	273	211.9	381
30	463	344.1	673

Table 2: Variation of Global and Local sets against the number of partitions.

For example, consider the case where number of partitions is set to 10. The number of large itemsets for all partitions combined is 109.1 $\times$ 10 = 1091. However, the union of these itemsets (global candidate set) is only 170.

It should be noted that when the partition sizes are sufficiently large, the local large itemsets and the global candidate itemsets are likely to be very close to the actual large itemsets as it tends to eliminate the effects of local variations in data. For example, when the number of partitions is 30 in Table 2, each partition contains 100,000 / 30 = 3,333 transactions, which is too small and hence the large variations.

3.4.1 Effect of Data Skew

The sizes of the local and global candidate sets may be susceptible to data skew. A gradual change in data characteristics, such the average length of transactions, can lead to the generation of a large number of local large sets which may not have global support. For example, due to severe weather conditions, there may be an abnormally high sales of certain items which may not be bought during the rest of the year. If a partition comprises of data from only this period, then certain itemsets will have high support for that partition, but will be rejected during phase II due to lack of global support. A large number of such spurious local large itemsets can lead to much wasted effort. Another problem is that fewer itemsets will be found common between partitions leading to a larger global candidate set.

The effect of data skew can be eliminated to a large extent by randomizing the data allocated to each partition. This is done by choosing the data to be read in a partition randomly from the database. However, to exploit sequential I/O, the minimum unit of data read is equal to the extent size. Given the size of the database in number of extents and the number of partitions, the algorithm initially assigns extents randomly to the partitions. No extent appears in more than one partition.

The effect of sequentially reading the data vs. randomly picking the blocks for a highly skewed dataset is shown in Table 3. To simulate data skew, the average lengths of transactions are varied from 5 to 20. The size of the database is about 41 Mbytes containing about 600,000 transactions. The minimum support was fixed at 0.75 %. In the first set of experiments, we generated the partitions by reading the blocks sequentially. In the second set, the partitions are generated by choosing the blocks randomly from the database. The number of partitions was varied from 5 to 30. The table shows the sum of local large itemset for all partitions and the size of the global candidate set. It is clear that randomly reading the pages from the database is extremely effective in eliminating data skew.

	Reading Sequential blocks		Reading Random blocks	
Number of partitions	Sum of all L^p	Size of C^G	Sum of all L^p	Size of C^G
5	3961	3831	69	26
10	9281	6166	150	39
15	15800	7618	304	75
20	22871	8228	439	87
25	29245	8961	599	101
30	36311	9598	760	121

Table 3: Effect of data skew: generating partitions sequentially vs. randomly

3.5 Data Structures and Implementation

In this section we describe the data structures and the implementation of our algorithm.

3.5.1 Generating Local Large Itemsets

To efficiently generate the candidate itemsets by joining the large itemsets, we store the itemsets in sorted order. We also store references to the itemsets in a hash table for performing pruning efficiently.

For computation of the intersection, the tidlists are maintained in sorted order and sort-merge join algorithm is used. The resulting tidlists are also in the sorted order. The intersection operation in this case involves only the cost of traversing the two lists once.

3.5.2 Generating the Global Candidate Set

Initially the global candidate set is empty. All local large itemsets of the first partition are added to the global candidate set. For subsequent partitions the local large itemsets are added only if the itemset is not already included. The candidate itemsets are kept in a hash table to perform this operation efficiently.

It is possible to prune the global candidate set by eliminating (a) itemsets for which the global support is known and (b) itemsets which cannot possibly have the required global support. The first case arises when an itemset is reported as large in every partition. Since the counts for that itemset in every partition is known, its global support is already known. The second case arises when an itemset is reported as large only in very few partitions and further their supports in those partitions are only slightly above the minimum support. Many of these itemsets cannot possibly have the global support. For example, suppose there are 20 partitions and the minimum support in each partition is 500. If an itemset is found to be large in only one partition and its support in that partition is 510, then it cannot have global support because its support in all other partitions can be at most 499 so that its global support is less than 10,000. For each local large itemset, we maintain its cumulative support and the number of partitions it was reported as large during merging. These counts are used to perform the pruning as described in [11].

3.5.3 Generating Final Counts

The data structures used for the final counting phase are similar to those used during phase I. Initially, a counter is set up for each itemset in the global candidate set. The tidlists for all 1-itemsets are generated directly by reading in a partition. The local count for an itemset is generated by joining the tidlists of all 1-itemsets contained in that itemset. For example, to generate the count for {1 2 3 4} the tidlists of itemsets {1}, {2}, {3} and {4} are joined. The cumulative count from all partitions gives the support for the itemset in the database.

To optimize the number of joins performed during this step, the counts for the longest itemsets are generated first. The intermediate join results are used to set the counts for the corresponding itemsets. For example, while generating the count for {1 2 3 4}, the counts for itemsets {1 2} and {1 2 3} are also set. The itemsets are kept in a hash table to facilitate efficient lookup.

Unlike phase I, the partitions for this phase can be obtained by reading the database blocks sequentially. Additionally, the size of the partitions may be different from those used in phase I.

3.6 Buffer Management

A key objective of the Partition algorithm is to reduce disk I/O as much as possible. To achieve this objective, the partitions are chosen such that all data structures can be accommodated in the main memory. However, the number of large itemsets that will be generated cannot be estimated accurately. In some situations it may be necessary to write the temporary data to disk.

The buffer management technique in phase I is similar to the one described in [2]. However, in Partition algorithm there is no separate step for counting the supports. As each local candidate k-itemset is generated, its count is also immediately generated. Hence in some iteration k, we need storage for the large $(k-1)$-itemsets that were generated in the previous iteration and their associated tidlists. Among these, only those itemsets for which the first $k-2$ items are the same are needed in main memory.

For the merge phase, we need space for at least those global candidate itemsets and local large itemsets that are of same length and have items in common. For phase II, we need space for the tidlists of only 1-itemsets and the the global candidate set. We try to choose the partition sizes such that they can be accommodated in the available buffer space.

3.7 Choosing the Number of Partitions

We have described how partitioning can be effectively used for reducing the disk I/O. However, how do we choose the number of partitions? In this section we describe how to estimate the partition size from system parameters and compute the number of partitions for a given database size.

For a small database, we may process the entire database as a single partition. As the database size grows, the size of the tidlists also grows and we may no longer be able to fit in main memory the tidlists that are being joined. This leads to thrashing and degradation in performance. We must choose the partition size such that at least those itemsets (and their tidlists) that are used for generating the new large itemsets can fit in main memory.

As noted in Section 3.6, in iteration k we need to keep in main memory at least all large $(k-1)$-itemsets in which the first $k-2$ items are common. We assume the number of such itemsets is at most a few thousand. We use heuristics to estimate the partition size based on the available main memory size and the average length of transactions.

Sampling can also be used to estimate the number of large itemsets and their average support which can be used to compute the partition size. We are exploring this approach as part of the future work.

4 Performance Comparison

In this section we describe the experiments and the performance results of our algorithm. We also compare the performance with the Apriori algorithm. The experiments were run on a Silicon Graphics Indy R4400SC workstation with a clock rate of 150 MHz and 32 Mbytes of main memory. The data resided on a 1 GB SCSI disk. All the experiments were run on

```
1) $L_1$ = {large 1-itemsets};
2) for ( $k = 2$; $L_{k-1} \neq \emptyset$; $k$++ ) do begin
3)     $C_k$ = apriori-gen($L_{k-1}$);
4)     forall transactions $t \in \mathcal{D}$ do begin
5)         $C_t$ = subset($C_k$, $t$);
6)         forall candidates $c \in C_t$ do
7)             $c$.count++;
8)     end
9)     $L_k = \{c \in C_k | c.\text{count} \geq \text{MinSup}\}$
10)end
11)Answer = $\cup_k L_k$;
```

Figure 4: Algorithm Apriori

synthetic data. For the performance comparison experiments, we used the same synthetic data sets as in [2].

Both Apriori and AprioriTid algorithms were implemented as described in [2]. Our initial experiments showed that the performance of Apriori is superior to that of AprioriTid confirming the results reported in [2]. Hence, in the following experiments we have limited the comparison to Apriori algorithm. The synthetic data generation procedure is described in detail in [2]. In the following section, we describe Apriori algorithm and the synthetic data generation procedure for the sake of completeness.

The Apriori algorithm is shown in Figure 4. The procedure **apriori-gen** is similar to the candidate generation step described earlier. The **subset** operation is performed using bit fields and hashtree structure as described in [2].

4.1 Synthetic Data

The synthetic data is said to simulate a customer buying pattern in a retail environment. The length of a transaction is determined by poisson distribution with mean μ equal to $|T|$. The transaction is repeatedly assigned items from a set of potentially maximal large itemsets, $\mathcal{T}$ until the length of the transaction does not exceed the generated length.

The length of an itemset in $\mathcal{T}$ is determined according to poisson distribution with mean μ equal to $|I|$. The items in an itemset are chosen such that a fraction of the items are common to the previous itemset determined by an exponentially distributed random variable with mean equal to a correlation level. The remaining items are randomly picked. Each itemset in $\mathcal{T}$ has an exponentially distributed weight that determines the probability that this itemset will be picked. Not all items from the itemset picked are assigned to the transaction. Items from the itemset are dropped as long as an uniformly generated random number between 0 and 1 is less than a corruption level, c. The corruption level for itemset is determined by a normal distribution with mean 0.5 and variance 0.1.

$\|D\|$	Number of transactions
$\|T\|$	Average size of transactions
$\|I\|$	Average size of maximal potentially large itemsets
$\|L\|$	Number of maximal potentially large itemsets
N	Number of items

Table 4: Parameters

Name	$\|T\|$	$\|I\|$	$\|D\|$	Size in MB
T5.I2.100K	5	2	100K	2.4
T10.I2.100K	10	2	100K	4.4
T10.I4.100K	10	4	100K	
T20.I2.100K	20	2	100K	8.4
T20.I4.100K	20	4	100K	
T20.I6.100K	20	6	100K	

Table 5: Parameter settings

4.2 Experiments

Six different data sets were used for performance comparison. Table 5 shows the names and parameter settings for each data set. For all data sets N was set to 1,000 and $|L|$ was set to 2,000. These datasets are same as those used in [2] for the experiments.

Figure 5 shows the execution times for the six synthetic datasets for decreasing values of minimum support. Since the datasets contained about 100,000 transactions with the largest dataset only about 8.4 MB, we could run the Partition algorithm setting the number of partitions to 1. However, for comparison, we also ran the experiments setting the number of partitions to 10. These results are indicated as Partition–1 and Partition–10 in the figure. Since we have not implemented complete buffer and disk management, we did not include disk I/O times in the execution times to keep the comparison uniform.

The execution times increase for both Apriori and Partition algorithms as the minimum support is reduced because the total number of large and candidate itemsets increase. Also, as the average length of transactions increase, the number of large and candidate itemsets also increase.

For these datasets, Partition–1 performed better than partition–10 in all cases as expected. The reason is that the Partition–10 tests support for more itemsets which have only local support but are discarded in phase II. Except for cases where the minimum support is high, Partition-1 performed better than Apriori. Even Partition–10 performed better than Apriori in most cases for low minimum support settings. The reason why Apriori performs better for higher mini-

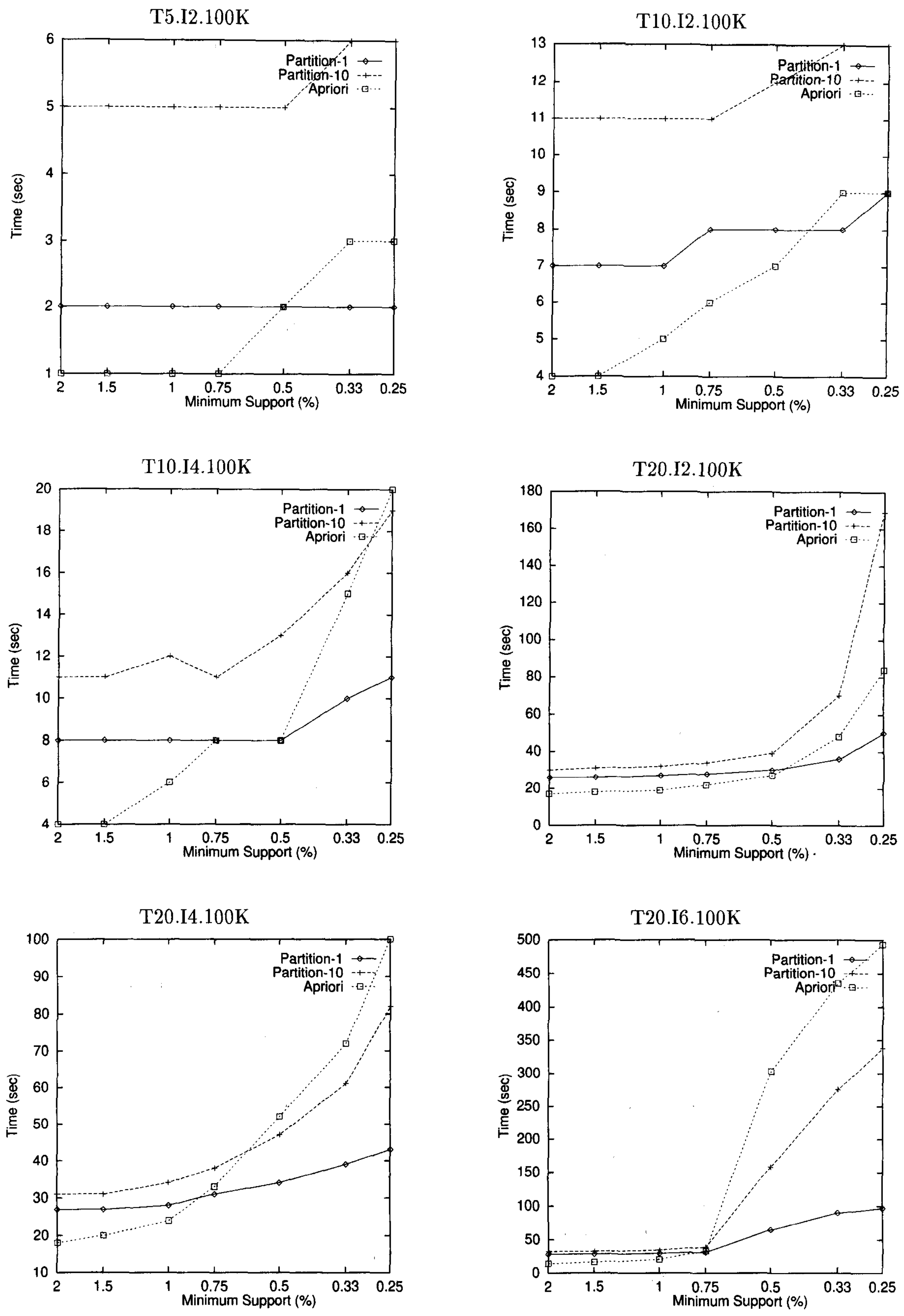

Figure 5: Execution times

mum support settings is that Partition has the overhead of setting up the tidlist data structures. However, at these minimum supports the large and candidate itemsets are very few and in some cases none at all. So, Partition does not benefit from setting up the data structures. Partition–10 performed worse than Apriori for the dataset T20.I2.100K at minimum support of 0.25 %. The reason was that a large number of itemsets were found to be locally large which later turned out to be small. However, this behavior did not repeat for any other case. We attribute it to the characteristics of that particular dataset.

At the lowest minimum support setting, the least improvement was 10 % (10 seconds for Apriori vs. 9 for Partition for T10.I2.100K). The best improvement was about 81 % (707 seconds for Apriori vs. 97 for Partition for T20.I6.100K). This is an improvement by a factor of 5.

It should be noted that the improvement in the execution times for Partition shown in Figure 5 is mainly due to the reduction in the CPU overhead and not due to the reduction in I/O. The reason is that the database is only 8.4 Mbytes which is too small to significantly affect the total execution time.

4.2.1 Explanation of Performance

As both Apriori and Partition use same candidate itemset generation technique, the improvement is mainly due to better technique for generating the counts. In Apriori algorithm counts are generated by the subset operation where itemsets from a candidate set are compared with all transactions during each pass for inclusion to determine their counts. The cost of subset operation per itemset increases in later passes as the length of the itemsets increase. As an illustration of the amount of work done for subset step consider the following example. Assume the number of candidate itemsets is 1,000 and that there are 1 million transactions in the database. Further, assume that the hashtree structure eliminates 99 % of the candidate itemsets and on an average 4 comparisons are required to determine if an itemset is contained in a transaction. This requires 0.01 × 1,000 × 1 million × 4, or 40 million basic integer compare operations. The cost of traversing the hashtree and initializing the bit field for every transaction can add substantially to this figure.

The partitioned approach in our algorithm allows us to use more efficient data structures for computing the counts for the itemsets. The cost of generating the support decreases during later passes as the lengths of the tidlists become smaller. To illustrate the efficiency of counting using tidlists, consider the above example. For the purpose of illustration, assume that the number of partitions is 1. Then, in our algorithm the operation of counting supports involves performing just 1,000 intersection operations. Assume that each transaction contains on an average 10 items and that there are 1,000 distinct items. Then on an average the length of a tidlist is 1 million × 10 / 1,000, or about 10,000. So the overall cost is about 1,000 × 10,000, or about 10 million basic integer compare operations. However, if the number of partitions is more than 1, this value can be much larger. The above example assumes a very simple scenario and does not include the cost of setting up the data structures. The actual comparisons depend on the parameters used for building the hashtree, characteristics of the data, etc. However, it explains why the Partition algorithm performs better than Apriori.

Algorithm	Minimum Support	
	0.75%	0.25%
Apriori	412,904	10,495,190
Partition	58,452	1,978,077

Table 6: Number of comparison operations

We have compared the actual number of comparisons performed by Partition and Apriori algorithms for some different support levels for T10.I4.100K. The results are shown in Table 6. It should be noted that the actual execution times also include generation of data structures, generation of candidate itemsets, and in the case of Apriori, traversing the hashtree, etc. and hence do not reflect the figures shown in the table which compares only the cost of generating supports.

4.2.2 Improvement in Disk I/O

The Partition algorithm was motivated by the need to reduce disk I/O. In this aspect it has a clear advantage over the Apriori algorithm. The Partition algorithm reads the database at most twice irrespective of (a) the minimum support and (b) the number of partitions. Apriori reads the database multiple number of times[4]. The exact number depends on the minimum support and the data characteristics and cannot be determined in advance.

We measured the number of read requests for data for both the algorithms for the datasets described in Table 5. The page size was set to 4Kbytes. The results are shown in Figure 6. The best improvement we found was about 87 % for T20.I6.100K at minimum support of 0.25 %. This is an improvement by a factor of 8. The least improvement for this minimum support was 60 % representing an improvement by a

[4]Actually when the minimum support is set very high, no large itemsets are generated. In this case, both algorithms read the database only once.

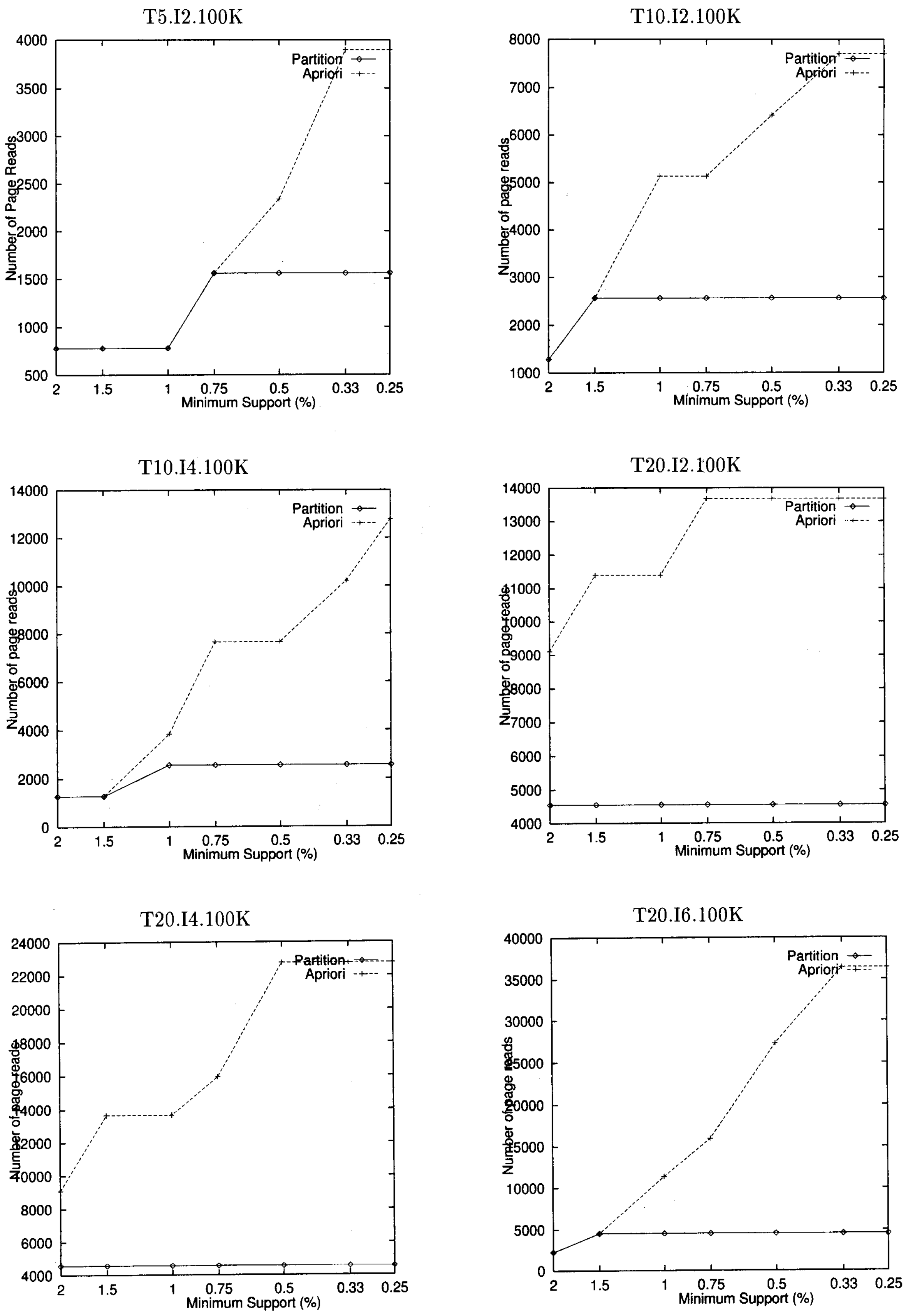

Figure 6: Number of database read requests

factor of 2.5. Even at the median minimum support of 0.75 %, Partition showed an improvement over Apriori, except for T5.I2.100K in which both algorithms read the database twice. When the support level is set very high no itemsets are found to have the required support. In such cases, both algorithms read the database only once.

4.3 Scale-up Experiments

We have studied the scale-up characteristics of the Partition algorithm by varying the number of transactions from 100,000 to 10 million. All other parameter settings were same as T10.I4.100K. The results are shown in Figure 7. The number partitions was varied from 1 for 100K transactions to 100 for 10M transactions for the Partition algorithm. The execution times are first normalized with respect to the size of the database and then with respect execution times taken by the Partition algorithm for 100,000 transactions. The initial jump in the execution time (from about 1.3 to 1.9) is due to the increase in the number of partitions from 1 to 4. As expected, this increases the size of the global candidate set and hence an increase in the execution time. However, as the number of partitions is increased, size of the global candidate set does not increase correspondingly as more and more local large itemsets are common. The execution time was relatively linear from 400,000 transactions to 10 million transactions.

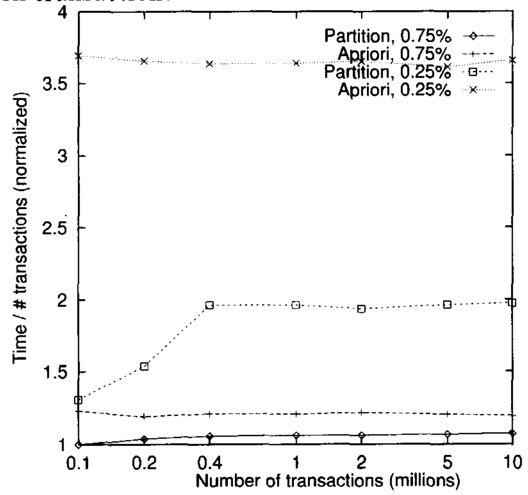

Figure 7: Number of transactions scale-up

We also studied the performance of the algorithm for average transaction size and the average size of maximal potentially large itemset scale-up. For this experiment, we varied the transaction length from 5 to 50. The size of $|I|$ was varied from 2 to 6. The physical size of the database was kept roughly constant by keeping the product of the number of transaction and the average transaction size constant. The number of transactions varied from 200,000 for the

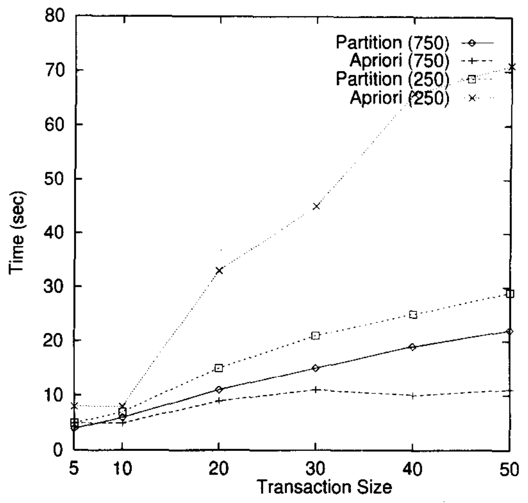

Figure 8: Transaction size scale-up

database with an average transaction length of 5 to 20,000 for the database with the average transaction length of 50. The minimum support level was fixed in terms of the number of transactions. We ran the experiments for minimum support levels of 750 and 250. The results are shown in Figure 8. Partition exhibits marginally inferior scale-up compared to Apriori when the minimum support is high (750) as it spends more and more time initializing the data structures without deriving much benefit in processing cost. However, for lower minimum support (i.e., high processing cost), the scale-up is superior to Apriori because the processing cost increases slower than that of Apriori.

5 Conclusions

We have described an algorithm which is not only efficient but also fast for discovering association rules in large databases. An important contribution of our approach is that it drastically reduces the I/O overhead associated with previous algorithms. This feature may prove useful for many real-life database mining scenarios where the data is most often a centralized resource shared by many user groups, and may even have to support on-line transactions. Interestingly, this improvement in disk I/O is not achieved at the cost of CPU overhead. We have demonstrated with extensive experiments that the CPU overhead is actually less than the best existing algorithm for low minimum supports (i.e., cases which are computationally more expensive). In addition, the algorithm has excellent scale-up property.

The problem of accurately estimating the number of partitions given the available memory, however, needs further work. We are currently addressing this problem. We are also exploring the possibility of combining our algorithm with the previous algorithms to develop a hybrid approach which performs best for all cases. In future, we also plan to extend this work by parallelizing the algorithm for a shared nothing multiprocessor machine.

Acknowledgment

We wish to thank Dr. Rakesh Agrawal at IBM Almaden Research Center for providing the synthetic data for running the experiments. The first author also wishes to thank Sreenivas Gukal for his comments and suggestions.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data,* pages 207–216, Washington, DC, May 26-28 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 Data Bases*, Santiago, Chile, August 29-September 1 1994.

[3] T. M. Anwar, S. B. Navathe, and H. W. Beck. Knowledge mining in databases: A unified approach through conceptual clustering. Technical report, Georgia Institute of Technology, May 1992.

[4] J. Han, Y. Cai, and N. Cercone. Knowledge discovery in databases: an attribute-oriented approach. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 547–559, Vancouver, Canada, 23-27, August 1992.

[5] M. Holsheimer and A. Siebes. Data mining: The search for knowledge in databases. Technical Report CS-R9406, CWI, Amsterdam, The Netherlands, 1993.

[6] M. Houtsma and A. Swami. Set-oriented mining of association rules. In *Proceedings of the International Conference on Data Engineering*, Taipei, Taiwan, March 1995.

[7] R. Krishnamurthy and T. Imielinski. Practitioner problems in need of database research. *ACM SIGMOD Record*, 20(3):76–78, September 1991.

[8] G. Piatetsky-Shapiro and W. J. Frawley, editors. *Knowledge Discovery in Databases.* MIT Press, 1991.

[9] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. Technical Report GIT-CC-95-04, Georgia Institute of Technology, Atlanta, GA 30332, January 1995.

[10] A. Silberschatz, M. Stonebraker, and J. Ullman. Database systems: achievements and opportunities. *Communications of the ACM*, 34(10):110–120, October 1991.

[11] M. Stonebraker, R. Agrawal, U. Dayal, E. Nuehold, and A. Reuter. Database research at a crossroads: The vienna update. In *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 688–192, Dublin, Ireland, August 1993.

[12] S. Tsur. Data dedging. *IEEE Data Engineering Bulletin*, 13(4):58–63, December 1990.

[13] J. T-L. Wang, G-W. Chirn, T. G. Marr, B. Shapiro, D. Shasha, and K. Zhang. Cobinatorial pattern discovery for scientific data: some preliminary results. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 115–125, Minneapolis, MN, May 24-27 1994.