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

Technical Report No. GIT-CC-95-04

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 the previous algorithms, our algorithm reduces *both* CPU *and* I/O overheads. In our experimental study it was found that for large databases, the CPU overhead was reduced by as much as a factor of seven and I/O was reduced by almost an order of magnitude. Hence this algorithm is especially suitable for very large size databases. The algorithm is also ideally suited for parallelization. We have performed extensive experiments and compared the performance of the algorithm with one of the best existing algorithms.

1 Introduction

Increasingly, business organizations are depending on sophisticated decision-making information to maintain their competitiveness in today's demanding and fast changing marketplace. Inferring valuable high-level information based on large volumes of routine business data is becoming critical for making sound business decisions. For example, customer buying patterns and preferences, sales trends, etc, can be learned from analyzing point-of-sales data at supermarkets. This information may be used for retaining market leadership by tuning to the needs of customers. Database mining is motivated by such decision support problems and is described as an important area of research [11, 12].

One of the most difficult problems in database mining is the large volume of data that needs to be handled. In a medium sized business, it is not uncommon to collect hundreds of megabytes to a few gigabytes of data. Database mining applications often perform long-running, complex data analysis over the entire database. Given the large database sizes, one of the main challenges in database mining is developing fast and efficient algorithms that can handle large volumes of data.

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." This type of information may be used to decide 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 reads. In these algorithms, the effort spent in performing just the I/O may be considerable for large databases. For example, a 1 GB database will require roughly 125,000 block reads for a single pass (for a block size of 8KB). If the algorithm takes, say, 10 passes, this results in 1,250,000 block reads. Assuming an average readtime of 12 ms per page, the time spent in just performing the I/O is 1,250,000 $\times$ 12 ms, or roughly, 4 hours.

Apart from poor response times, this approach places a huge burden on the I/O subsystem. Usually, the data is collected by online transaction processing (OLTP) systems running hundreds or thousands of transactions per second. Running these algorithms under such workloads will adversely affect the transaction response times and may even disrupt the daily operation. If the data is retrieved from a central database server over a network such as a LAN, it will create network congestion problems and poor resource utilization.

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. Hence our algorithm can potentially achieve orders of magnitude savings in I/O. Surprisingly, the savings in I/O is not achieved at the cost of increased CPU overhead. The Partition algorithm has much lower CPU overhead compared to the previous algorithms. We have performed extensive experiments and compared our algorithm 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.

Some additional advantages of the Partition algorithm are as follows:

1. Our algorithm can generate approximate results in half the time with no false negatives. This may be useful when such results are sufficient.

2. The algorithm is inherently parallel in nature and can be parallelized with minimal communication and synchronization between the processing nodes.

Other related, but not directly applicable work in database mining are reported in [7, 10, 6, 9, 3, 13, 14].

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. An approach to parallelizing our algorithm is described in Section 5. Section 6 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 the *confidence*. The confidence of a rule $X \Longrightarrow Y$ is computed 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. It should be noted that only a very small fraction of this exponentially large number of itemsets will have minimum support. Hence, it is not necessary to test the support for every itemset. Even if practically feasible, testing support for every possible itemset results in much wasted effort. To reduce the combinatorial search space, all algorithms exploit the following property: any subset of a large itemset must also be large. For example, if a transaction contains itemset *ABCD*, then it also contains *A*, *AB*, *BC*, *ABC*, etc.

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

Conversely, all extensions of a small itemset are also small. Therefore, if at some stage it is found that itemset *ADE* is small, then none of the itemsets which are extensions of *ADE*, i.e., *ADEF*, *ADEFG*, etc., need be tested for minimum support.

All existing algorithms for mining association rules are variants of the following general approach: initially support for all itemsets of length 1 (1-itemsets) are tested by scanning the entire 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 entire database. Many of these itemsets may turn out to be small, and hence discarded. The remaining itemsets are extended by 1 and tested for support. This process is repeated until no more large itemsets are found. 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. The itemsets that do not have the minimum support are discarded and the remaining itemsets are designated large k-itemsets.

Therefore, only extensions of those itemsets that are found to be large are considered in subsequent passes. This process is stopped when in some iteration n, no large itemsets are generated. The algorithm, in this case, makes n database scans.

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 a relational database environment. In [2], two new algorithms called *Apriori* and *AprioriTid* were proposed. These algorithms achieved significant improvements over the previous algorithms and were specifically applicable to large databases. In [2], the rule generation process was 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 using only the large itemsets from the previous pass. It is performed by joining the large itemset with itself. The resulting set is further pruned to exclude any itemset whose subset is not contained in the previous large itemsets. This technique produces a much smaller candidate set than the former technique.

To count the supports for the candidate itemsets, 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, bitmaps are generated for transactions as well as the candidate itemsets. To determine whether a candidate itemset is contained in a transaction, the corresponding bitmaps are compared. A hash tree structure is used to restrict the set of candidate itemsets compared so that subset testing is optimized. 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 of the transactions can be much smaller than the actual database. However, in initial passes the size of the encoding can be larger than the database. A hybrid algorithm is 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 straight-forward to adapt the algorithm to the case where the transactions are kept normalized in $\langle TID, item \rangle$ form. We also assume that the TIDs 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. The items in an itemset are also kept sorted in lexicographic order.

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

Notation	Meaning
c_k^p	A local candidate k–itemset in partition p
l_k^p	A local large k–itemset in partition p
$s^p(l)$	Support for an itemset l within partition p
C_k^p	Set of local candidate k–itemsets in partition p
L_k^p	Set of local large k–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
$s^G(l)$	Support for a global itemset l

Table 1: Notation used in this paper

local support for an itemset as the fraction of transactions containing that itemset in a partition. We define a *local large* itemset as an itemset whose local support in a partition is at least the user defined minimum support[2]. In other words, a local large itemset is large only in the context of a partition (i.e., consider the entire database as consisting of only that partition). A *local candidate* itemset is an itemset that is being tested for minimum support within a given partition. A local large itemset may or may not be large in the context of the entire database.

We define the *global* support, *global large* itemset, and *global candidate* itemset as above except they are in the context of the entire database $\mathcal{D}$. Clearly, 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 $c[1]$, $c[2]$, $\ldots$, $c[k]$.

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, $L_2^i, L_3^i, \ldots, L_l^i$ 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. The set of global candidate itemsets of length j is computed as $C_j^G = \cup_{i=1,\ldots,n} L_j^i$. In phase II, the algorithm sets up counters for each global candidate itemset and counts their support in all n partitions. It outputs itemsets that have the minimum global support along with their support.

The algorithm reads the entire database once during phase I and once during phase II.

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. Since the global candidate set is the union of all local large itemsets and the algorithm counts the support for every global candidate

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

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

Figure 1: Algorithm Partition

itemset, all large itemsets are found. We illustrate the proof as follows.

Let L be the set of actual large itemsets in the database. Since the global large itemsets are generated by counting the support for every itemset in C^G, it is sufficient to show that $C^G \supseteq L$.

Assume there exists an itemset l that is actually large but does not appear in C^G, i.e., $l \in L$ but $l \notin C^G$. But $C^G = \cup_{i=1,\ldots,n} L_j^i$. Hence $l \notin L^i$ for $i = 1, 2, \ldots, n$.

Let tx^i be the number of transactions in partition p^i. Let $tx^i(l)$ be the number of transaction containing the itemset l in partition p^i.

Since $l \notin L^i$ for $i = 1, 2, \ldots, n$, it must be true that

$$tx^1(l)/tx^1 < minSup \wedge tx^2(l)/tx^2 < minSup \wedge \ldots \wedge tx^n(l)/tx^n < minSup. \tag{1}$$

or

$$tx^1(l) < minSup \times tx^1 \wedge tx^2(l) < minSup \times tx^2 \wedge \ldots \wedge tx^n(l) < minSup \times tx^n. \tag{2}$$

This is equivalent to

$$tx^1(l) + tx^2(l) + \ldots + tx^n(l) < minSup \times (tx^1 + tx^2 + \ldots + tx^n). \tag{3}$$

But $tx^1 + tx^2 + \ldots + tx^n$ is the total number of transactions in the database and $tx^1(l) + tx^2(l) + \ldots + tx^n(l)$ is the total number of transactions containing l in the database. Therefore *support(l)* $< minSup$ and so $l \notin L$. But this is a contradiction.

```
    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 $l_1[1] = l_2[1] \wedge l_1[2] = l_2[2] \wedge \ldots \wedge l_1[k-1] < l_2[k-1]$ 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 $|c$.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

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 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. We represent the tidlist for an itemset l by l.tidlist. 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 $\{1\ 2\ 4\}$.

```
    procedure gen_final_counts($C^G$: global candidate set, $p$: database partition)
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 $\cap c$[2].tidlist $\cap \ldots \cap$ $c$[k].tidlist
6)          $c$.count = $c$.count + | $templist$ |
7)      end
8)  end
```

Figure 3: Procedure gen_final_counts

Correctness It has been shown in [2] that the candidate generation process correctly produces all potential large candidate itemsets. Therefore, it is sufficient to show that steps 8–9 correctly generate the support for an itemset in a partition. This can be shown by induction. By definition the tidlists of all 1-itemsets contain only the transactions that contain those itemsets. Assume that it is correct for some iteration $k - 1$. Steps 3 to 6 generate a candidate itemset by extending a large $(k-1)$-itemset, l_1, with an item from another large $(k-1)$-itemset, l_2. Hence the candidate k-itemset contains only the items in l_1 and l_2. The transactions that contain this k-itemset are ones that contain l_1 *and* ones that contain l_2, i. e., intersection of tidlists of l_1 and l_2.

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

Correctness As shown earlier, steps 5–6 generate the support for an itemset in the given partition. 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.

Number of Partitions	Size of largest Local Large Set	Average size of Local Large set	Size of Global Candidate Set
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.

3.4 Size of the Global Candidate Set

Our algorithm is based on the premise that the size of the global candidate set is considerably smaller than the set of all possible itemsets. But in practice, how large is this set?

It is not possible to answer this question with certainty as the number of local and global large itemsets depends on many factors such as the data characteristics and the specified minimum support. We try to give only an intuitive answer here. The global candidate set is equal to the union of all locally large itemsets. Hence its size is bounded by n times the size of the largest such set, 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 %. The number of actual large itemsets for the entire database was found to be 88. 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, a large fraction of the itemsets found large in each partition are common to all partitions. For example, consider the case where number of partitions = 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]The database used was T10.I4.100K described in Section 4.1

	Partitioning by reading Sequential blocks		Partitioning by reading Random blocks	
Number of partitions	Total number of large itemsets	Size of global candidate set	Total number of large itemsets	Size of global candidate set
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

3.4.1 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 or localized changes in data, can lead to the generation of a large number of local large sets which may not have global support. This is also true for any localized changes in data. For example, due to sever weather conditions, there may be an abnormally high sales of certain items. However, these items may not be bought during the rest of the year. If our partition comprises of data from only this period, then certain itemsets will have high support for that partition, but will be rejected later 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 within partitions. 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.

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 hashtable for performing pruning efficiently.

For computation of the intersection, the tidlists are maintained as arrays and sort–merge join algorithm is used. Recall that the *TID*s appear in ascending order in the database. Hence the tidlists are in the sort order initially and all resulting tidlists are automatically generated in the sort order. The intersection operation in this case involves only the cost of traversing the two lists once.

An alternate implementation uses fixed length bit fields in place of tidlists. The bits are set by hashing on the *TID* of the transaction. Intersection is accomplished by the logical AND of the corresponding bit fields. To determine the support, a fast algorithm to count the number of bits set in the bit field is implemented using a lookup table.

3.5.2 Generating the Global Candidate Set

To generate the global candidate set, the local large itemsets are merged with 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. We describe how we detect such itemsets below.

Associated with each global candidate itemset, are two counts: called its *cumulative support*, s^G, and *partition count*, p^G.

These counts are generated for an itemset l^i_j while it is being merged as follows:

1. If the itemset is already in C^G_j, then its cumulative support, s^G is set to $s^G + s^i$, where s^i is the local support for the itemset. The partition count, p^G is set to $p^G + 1$.

2. If the itemset is not present, then we perform $C^G_j = C^G_j \cup \{l^i_j\}$. The cumulative support is set to s^i and the partition count, p^G is set to 1.

Once the merge step is complete and before beginning phase II, the itemsets are pruned according to the following rules:

1. If the partition count for an itemset is equal to the number of partitions, the itemset must have been reported as large in every partition. In this case, the cumulative count for the itemset gives its actual global support. The itemset is removed from the global candidate set and included in the global large set and its global support is set to its cumulative support.

2. If the cumulative support plus the partition count for an itemset in C^G is less than the sum of minimum supports for the partitions in which the itemset was large and the number of partitions, then the itemset cannot possibly have global support. This itemset is discarded. Every extension of this itemset is also discarded.

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 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 hashtable to facilitate efficient lookup.

After processing all partitions, the final counts are compared with the minimum support and the large itemsets are identified. 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 the tidlists that are being joined in main memory. This leads to thrashing and degrading the 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 the 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. The partition size is then estimated based on the available main memory size and the average length of the transactions.

Assume for example, the number of items is equal to 1,000 and the average length of the transactions is 20. Let the size of available main memory be 10 Mbytes and we need 4 bytes to store each TID. Initially, consider only 1-itemsets. The maximum size of a partition such that we can accommodate all 1-itemsets can be estimated as follows: since the average length of transactions is 20, each TID can appear in the tidlists of 20 1-itemsets on an average. Therefore, the maximum size of the partition is 10 MB / 4 $\times$ 20 = 125,000 transactions. This partition size corresponds to an average length of tidlists equal to 10 MB / 4 $\times$ 1,000 = 2,500 TIDs for 1,000 itemsets (of length 1). Since the tidlists for subsequent iterations are generated by joining the tidlists of the earlier iteration, the average length of tidlists can be at most 2,500 for all subsequent iterations. Therefore, at least 1,000 itemsets can be accommodated in memory at any time. This number is a loose lower bound as the tidlists may be much smaller in later iterations in which case more than 1,000 itemsets can reside in main memory.

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 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 the Apriori algorithm. The synthetic data generation procedure is described in detail in [2]. In the following section, we describe the 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 $|I|$. Items in the first itemset are chosen randomly from the set of items. The items in

```
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

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

Table 4: Parameters

subsequent 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. The correlation level was set to 0.5. Each itemset in $\mathcal{T}$ has an associated weight that determines the probability that this itemset will be picked. The weight is picked from according to an exponential distribution with mean set to 1. The weights are normalized such that the sum of all weights equals 1. 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.

The parameters used in the generation of the synthetic data are shown in Table 4.

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

Name	$\|T\|$	$\|I\|$	$\|D\|$	Size in Megabytes
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

complete buffer and disk management, we did not include disk I/O times in the execution times to keep the comparison uniform. We expect the overheads due to disk I/O to be comparable for both algorithms once the buffer management algorithms are implemented.

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.

As expected Partition–1 performed better than partition–10 in all cases. The reason is that the Partition–10 tests support for more itemsets which have only local support and 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 minimum 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 86 % (707 seconds for Apriori vs. 97 for Partition for T20.I6.100K). This is an improvement by a factor of more than 7.

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 which is orders of magnitude larger.

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 transactions for inclusion to determine their counts. Apriori utilizes a hashtree data structure to reduce the number of itemsets that must be tested for inclusion in a transaction. For fast comparison, bit fields are used. 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.

Dataset	Algorithm	Minimum Support		
		0.5%	0.33%	0.25%
T10.I4.100K	Apriori	9	19	26
	Partition-1	8	10	11
T20.I2.100K	Apriori	28	56	67
	Partition-1	30	36	50
T20.I4.100K	Apriori	58	90	124
	Partition-1	34	39	43
T20.I6.100K	Apriori	456	633	707
	Partition-1	65	90	97

Table 6: Execution times in seconds for Apriori and Partition

If the database contains, say, 1,000 distinct items, then each bit field is 128 bytes long. This requires 0.01 $\times$ 1,000 $\times$ 1 million $\times$ 32, or 320 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. 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 $\times$ 10 / 1,000, or about 10,000. So the overall cost is about 1,000 $\times$ 10,000, or about 10 million basic integer compare operations. There are no additional costs. This is orders of magnitude improvement compared to Apriori algorithm. However, if the number of partitions is more than 1, this value can be much larger. For the purpose of illustration, assume the number of partitions is 10. Assume that 2,000 itemsets are tested for support in each partition during phase I. The average length of a tidlist for this case is one tenth of 10,000 or 1,000. So the total number of comparisons is 10 $\times$ 2,000 $\times$ 1,000 = 10 million in phase I. Let the size of the global candidate set be 2,500. Assume on an average 3 joins are performed for each of these itemsets[4]. So, 3 $\times$ 2,500 $\times$ 10,000 = 75 million comparisons are performed during phase II. The total is still less than 100 million comparisons. The above example assumes a very simple scenario. 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.

We have compared the actual number of comparisons performed by Partition and Apriori algorithms for some different support levels for T10.I4.1M (all parameters same as T10.I4.100K, except the number of transactions is 1 million). The number of partitions was set to 10 for the Partition algorithm. The results are shown in Table 7.

[4]Recall that for generating global support, as many joins as the length of the itemset needs to be performed.

Dataset	Algorithm	Minimum Support			
		0.75%	0.5%	0.33%	0.25%
T10.I4.1M	Apriori	30,355,136	76,348,944	330,498,976	492,271,536
	Partition	2,507,984	13,600,744	99,290,682	148,917,473

Table 7: Number of comparison operations

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[5]. 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 of a 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. These improvements varied between 0 — 71 % (a factor of 3.5). At very high support levels for some of the datasets, Partition made as many page read requests as Apriori. These represent the degenerate cases, where the support level is set so high that no itemsets are found to have the required support. In these 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 300 for 10M transactions for the Partition algorithm. We have also shown the execution times of Apriori. The execution times are normalized with respect to the execution times taken by the Partition algorithm for 100,000 transactions dataset in the first graph, and with respect to 1 million transaction dataset in the second graph. The minimum supports were set at 0.75 % and 0.25 %. As can be seen from the figure, both algorithms scale linearly.

We also studied the performance of the algorithm for average transaction size scale-up. For this experiment, we varied the transaction length from 5 to 50. 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 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

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

8. As we have seen earlier, Apriori performs better compared to Partition when the processing requirements are low such as when minimum support is set high. This trend is also shown in Figure 8 when the minimum support is 750. This is in line with our expectations. When the minimum support is lowered to 250, the Partition algorithm outperformed Apriori. This also confirms the trend exhibited in earlier performance studies. As the problem difficulty increases, the Partition algorithm performs much better than Apriori.

5 Parallelization

Parallel database systems have been shown to be viable means of delivering the performance required in supporting very large databases [5]. Many commercial parallel databases are available to the users today. These are beginning to replace mainframe computers for very large database and OLTP tasks. Eventually such databases may need to augment their functionality with database mining capabilities. The algorithm we have proposed in this paper is ideally suited for parallelization.

Shared nothing architectures have been shown to be highly suitable for parallel database systems [4]. It contains a number of processing nodes each with its own primary memory and a set of local disks. As a realistic implementation of a OLTP system for collecting data, we assume that the transactions are directed to the node with the least load. Alternatively, it can be assumed that the data is written on all disks in a round robin fashion. So, we assume the data is distributed over all the disks in approximately equal size chunks.

Recall that the partitions are processed entirely independently in both phase I and II of the Partition algorithm indicating that the processing can be essentially done in parallel. The parallel algorithm executes in four phases. During phase I, all the processing nodes independently generate the large itemsets for their local data. Two different cases are possible: (a) local memory at each processing node can accommodate the local data completely; and (b) local data is too large to load into the local memory all at once. In the first case, local data is processed completely as one partition and all local large items are generated. In the second case, the local data is in turn partitioned into a number of smaller chunks and the large itemsets for each chunk is determined. Since the size of local data at each processing node is approximately equal, this phase will take approximately equal time at all nodes achieving load balance. Additionally, only locally available data is processed. Hence, there is no communication cost for synchronization or data transfer among the nodes.

During phase II, the large itemsets at each node is exchanged with all other nodes. After the end of this phase all processing nodes will have exactly the same set of large itemsets. This set is the candidate set for the next phase.

In phase III, at each node support for each itemset in the candidate set with respect to the local data is measured. This step can be performed similar to phase I. Note that once again, during this phase there is no need for communication or synchronization among nodes.

In the last phase, the local counts at each node is sent to all other nodes. The global support is the sum of all local supports. To eliminate duplication of efforts, each node may be assigned a certain number of itemsets. Each node sends the local count for an itemset to the node to which that itemset is assigned.

6 Conclusions

We have described an algorithm which is not only fast but also efficient for discovering association rules in large databases. An important (and surprising) contribution of our approach is that we achieve *both* CPU *and* I/O improvements over Apriori, one of the best previous algorithms. We have also presented experimental results for the same synthetic data as used in [2]. The Partition algorithm outperformed Apriori for large problem sizes by upto a factor of seven at the same time reduced the number of disk reads by a factor of eight.

An interesting feature of the algorithm is that it scans the database at most twice, whereas in Apriori this is not known in advance and may be quite large. Hence the reduction in I/O overhead is potentially huge.

We demonstrated that the algorithm scales linearly with the number of transactions. We also described how the inherent parallelism in the algorithm can be exploited for implementation on a parallel machine. These factors demonstrate that the Partition algorithm is especially suited for very large databases in a high data and resource contention environment such as an OLTP system.

In future, we plan to extend this work by implementing 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] A. Bhide, F. Bancilhon, and D. J. Dewitt. An analysis of three transaction processing architectures. In *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 339–350, Los Angeles, CA, August 29 - September 1 1988.

[5] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[6] 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.

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

[8] M. Houtsma and A. Swami. Set-oriented mining of association rules. Technical Report RJ 9567, IBM, October 1993.

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

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

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

[12] 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.

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

[14] 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.

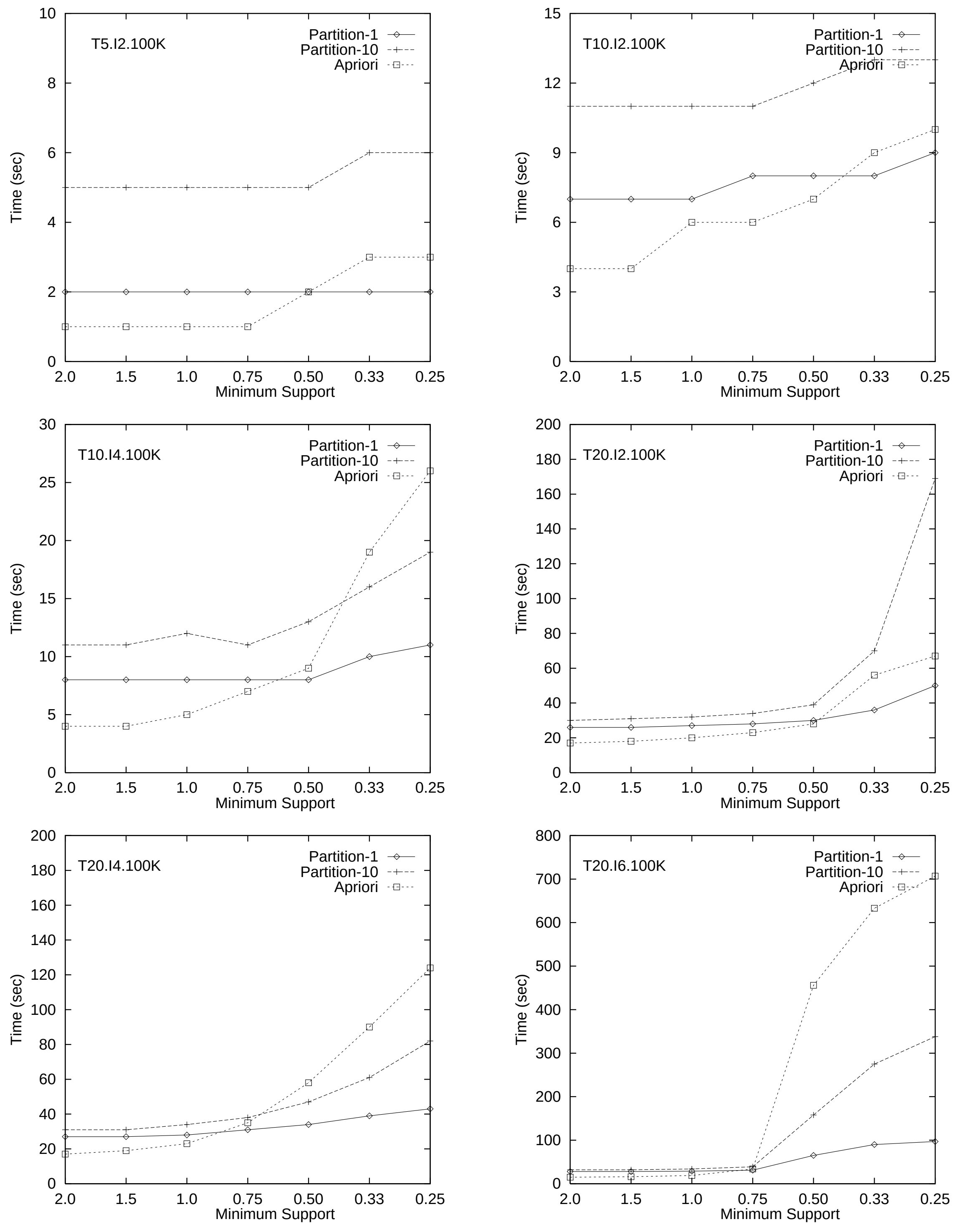

Figure 5: Execution Times

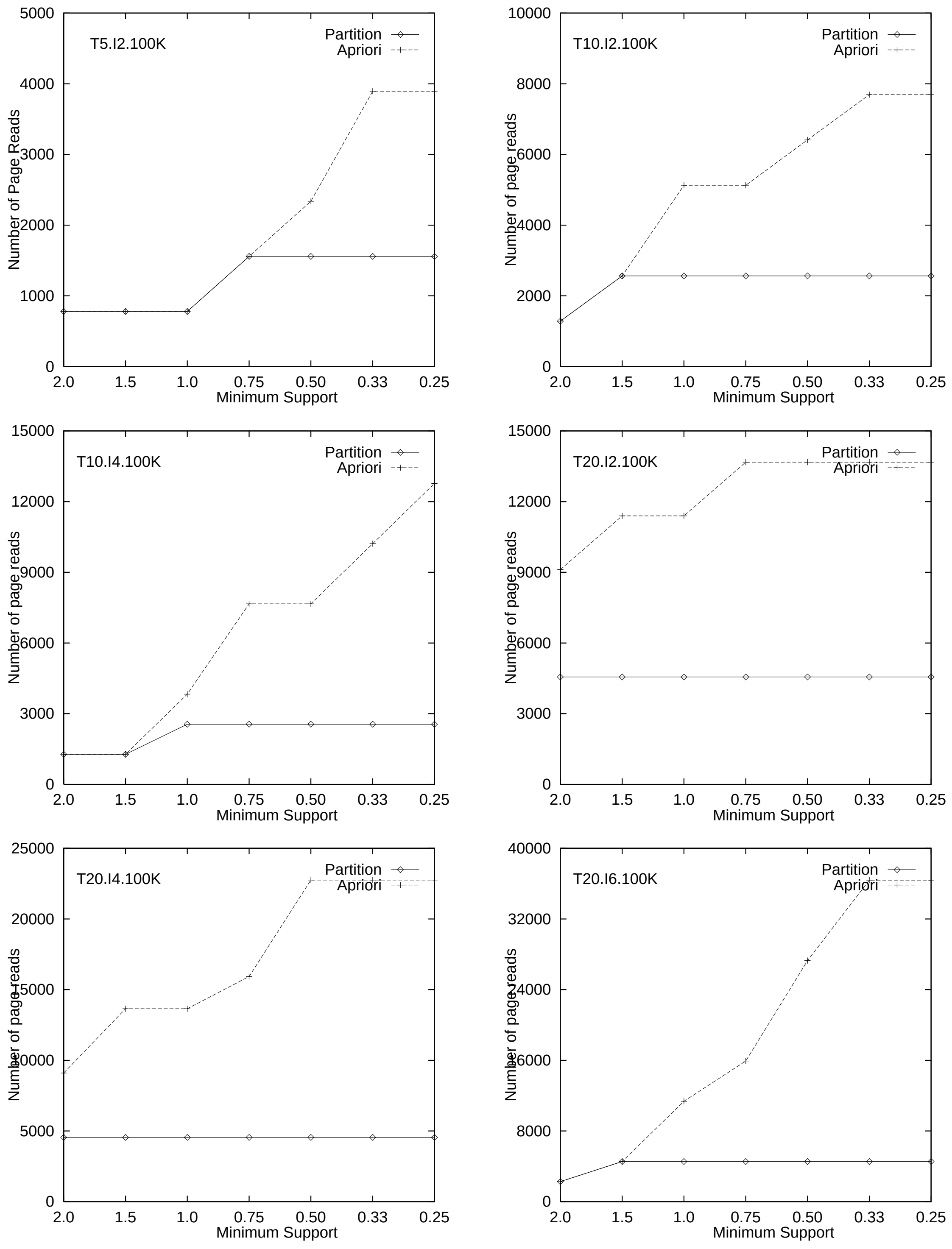

Figure 6: Number of database read requests

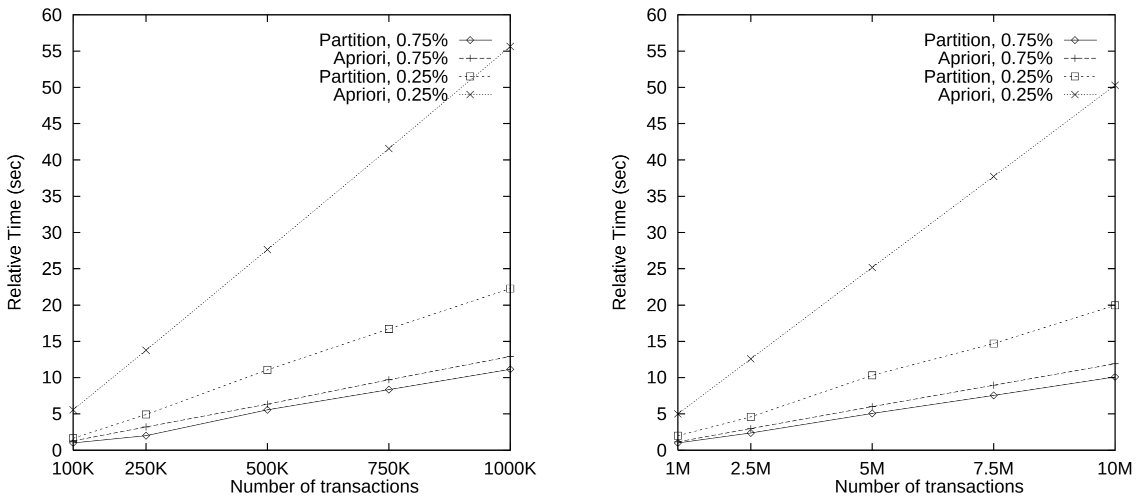

Figure 7: Number of transactions scale-up

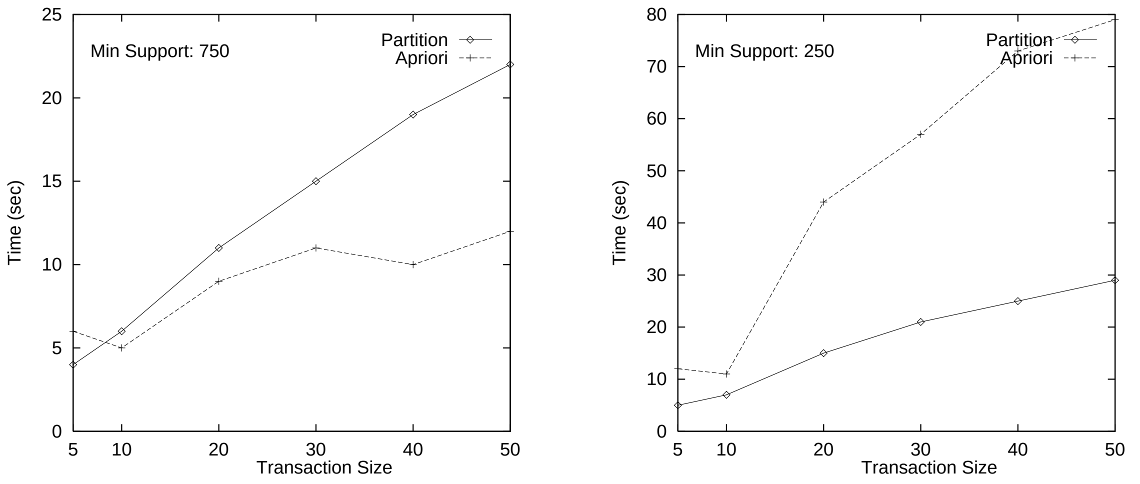

Figure 8: Average transaction size scale-up