Fast Algorithms for Mining Association Rules

Rakesh Agrawal Ramakrishnan Srikant*

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

Abstract

We consider the problem of discovering association rules between items in a large database of sales transactions. We present two new algorithms for solving this problem that are fundamentally different from the known algorithms. Experiments with synthetic as well as real-life data show that these algorithms outperform the known algorithms by factors ranging from three for small problems to more than an order of magnitude for large problems. We also show how the best features of the two proposed algorithms can be combined into a hybrid algorithm, called AprioriHybrid. Scale-up experiments show that AprioriHybrid scales linearly with the number of transactions. AprioriHybrid also has excellent scale-up properties with respect to the transaction size and the number of items in the database.

1 Introduction

Database mining is motivated by the decision support problem faced by most large retail organizations [S+93]. Progress in bar-code technology has made it possible for retail organizations to collect and store massive amounts of sales data, referred to as the *basket* data. A record in such data typically consists of the transaction date and the items bought in the transaction. Successful organizations view such databases as important pieces of the marketing infrastructure [Ass92]. They are interested in instituting information-driven marketing processes, managed by database technology, that enable marketers to develop and implement customized marketing programs and strategies [Ass90].

The problem of mining association rules over basket data was introduced in [AIS93b]. An example of such a rule might be that 98% of customers that purchase tires and auto accessories also get automotive services done. Finding all such rules is valuable for cross-marketing and attached mailing applications. Other applications include catalog design, add-on sales, store layout, and customer segmentation based on buying patterns. The databases involved in these applications are very large. It is imperative, therefore, to have fast algorithms for this task.

*Visiting from the Department of Computer Science, University of Wisconsin, Madison.

The following is a formal statement of the problem [AIS93b]: Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of literals, called items. Let $\mathcal{D}$ be a set of transactions, where each transaction T is a set of items such that $T \subseteq \mathcal{I}$. Associated with each transaction is a unique identifier, called its *TID*. We say that a transaction T *contains* X, a set of some items in $\mathcal{I}$, if $X \subseteq T$. An *association rule* is an implication of the form $X \Longrightarrow Y$, where $X \subset \mathcal{I}$, $Y \subset \mathcal{I}$, and $X \cap Y = \emptyset$. The rule $X \Longrightarrow Y$ holds in the transaction set $\mathcal{D}$ with *confidence* c if $c\%$ of transactions in $\mathcal{D}$ that contain X also contain Y. The rule $X \Longrightarrow Y$ has *support* s in the transaction set $\mathcal{D}$ if $s\%$ of transactions in $\mathcal{D}$ contain $X \cup Y$. Our rules are somewhat more general than in [AIS93b] in that we allow a consequent to have more than one item.

Given a set of transactions $\mathcal{D}$, the problem of mining association rules is to generate all association rules that have support and confidence greater than the user-specified minimum support (called *minsup*) and minimum confidence (called *minconf*) respectively. Our discussion is neutral with respect to the representation of $\mathcal{D}$. For example, $\mathcal{D}$ could be a data file, a relational table, or the result of a relational expression.

An algorithm for finding all association rules, henceforth referred to as the *AIS* algorithm, was presented in [AIS93b]. Another algorithm for this task, called the *SETM* algorithm, has been proposed in [HS93]. In this paper, we present two new algorithms, *Apriori* and *AprioriTid*, that differ fundamentally from these algorithms. We present experimental results, using both synthetic and real-life data, showing that the proposed algorithms always outperform the earlier algorithms. The performance gap is shown to increase with problem size, and ranges from a factor of three for small problems to more than an order of magnitude for large problems. We then discuss how the best features of Apriori and AprioriTid can be combined into a hybrid algorithm, called *AprioriHybrid*. Experiments show that the AprioriHybrid has excellent scale-up properties, opening up the feasibility of mining association rules over very large databases.

The problem of finding association rules falls within the purview of database mining [AIS93a] [ABN92] [HS94] [MKKR92] [S$^+$93] [Tsu90], also called knowledge discovery in databases [HCC92] [Lub89] [PS91b]. Related, but not directly applicable, work includes the induction of classification rules [BFOS84] [Cat91] [FWD93] [HCC92] [Qui93], discovery of causal rules [CH92] [Pea92], learning of logical definitions [MF92] [Qui90], fitting of functions to data [LSBZ87] [Sch90], and clustering [ANB92] [C$^+$88] [Fis87]. The closest work in the machine learning literature is the KID3 algorithm presented in [PS91a]. If used for finding all association rules, this algorithm will make as many passes over the data as the number of combinations of items in the antecedent, which is exponentially large. Related work in the database literature is the work on inferring functional dependencies from data [Bit92] [MR87]. Functional dependencies are rules requiring strict satisfaction. Consequently, having determined a dependency $X \rightarrow A$, the algorithms in [Bit92] [MR87] consider any other dependency of the form $X + Y \rightarrow A$ redundant and do not generate it. The association rules we consider are probabilistic in nature. The presence of a rule $X \rightarrow A$ does not necessarily mean that $X + Y \rightarrow A$ also holds because the latter may not have minimum support. Similarly, the presence of rules $X \rightarrow Y$ and $Y \rightarrow Z$ does not necessarily mean that $X \rightarrow Z$ holds because the latter may not have minimum confidence.

There has been work on quantifying the "usefulness" or "interestingness" of a rule [PS91a]. What is useful or interesting is often application-dependent. The need for a human in the loop and providing tools to allow human guidance of the rule discovery process has been articulated, for example, in [B+93] [KI91] [Tsu90]. We do not discuss these issues in this paper, except to point out that these are necessary features of a rule discovery system that may use our algorithms as the engine of the discovery process.

1.1 Problem Decomposition and Paper Organization

The problem of discovering all association rules can be decomposed into two subproblems [AIS93b]:

1. Find all sets of items (*itemsets*) that have transaction support above minimum support. The *support* for an itemset is the number of transactions that contain the itemset. Itemsets with minimum support are called *large* itemsets, and all others *small* itemsets. In Section 2, we give new algorithms, Apriori and AprioriTid, for solving this problem.

2. Use the large itemsets to generate the desired rules. We give algorithms for this problem in Section 3. The general idea is that if, say, $ABCD$ and AB are large itemsets, then we can determine if the rule $AB \Longrightarrow CD$ holds by computing the ratio *conf* = support($ABCD$)/support(AB). If *conf* $\geq$ *minconf*, then the rule holds. (The rule will surely have minimum support because $ABCD$ is large.)
   Unlike [AIS93b], where rules were limited to only one item in the consequent, we allow multiple items in the consequent. An example of such a rule might be that in 58% of the cases, a person who orders a comforter also orders a flat sheet, a fitted sheet, a pillow case, and a ruffle. The algorithms in Section 3 generate such multi-consequent rules.

In Section 4, we show the relative performance of the proposed Apriori and AprioriTid algorithms against the AIS [AIS93b] and SETM [HS93] algorithms. To make the paper self-contained, we include an overview of the AIS and SETM algorithms in this section. We also describe how the Apriori and AprioriTid algorithms can be combined into a hybrid algorithm, AprioriHybrid, and demonstrate the scale-up properties of this algorithm. We conclude by pointing out some related open problems in Section 5.

2 Discovering Large Itemsets

Algorithms for discovering large itemsets make multiple passes over the data. In the first pass, we count the support of individual items and determine which of them are *large*, i.e. have minimum support. In each subsequent pass, we start with a seed set of itemsets found to be large in the previous pass. We use this seed set for generating new potentially large itemsets, called *candidate* itemsets, and count the actual support for these candidate itemsets during the pass over the data. At the end of the pass, we determine which of the candidate itemsets are actually large, and they become the seed for the next pass. This process continues until no new large itemsets are found.

The Apriori and AprioriTid algorithms we propose differ fundamentally from the AIS [AIS93b] and SETM [HS93] algorithms in terms of which candidate itemsets are counted in a pass and in the way that those candidates are generated. In both the AIS and SETM algorithms (see Sections 4.1 and 4.2 for a review), candidate itemsets are generated on-the-fly during the pass as data is being read. Specifically, after reading a transaction, it is determined which of the itemsets found large in the previous pass are present in the transaction. New candidate itemsets are generated by extending these large itemsets with other items in the transaction. However, as we will see, the disadvantage is that this results in unnecessarily generating and counting too many candidate itemsets that turn out to be small.

The Apriori and AprioriTid algorithms generate the candidate itemsets to be counted in a pass by using only the itemsets found large in the previous pass – without considering the transactions in the database. The basic intuition is that any subset of a large itemset must be large. Therefore, the candidate itemsets having k items can be generated by joining large itemsets having $k{-}1$ items, and deleting those that contain any subset that is not large. This procedure results in generation of a much smaller number of candidate itemsets.

The AprioriTid algorithm has the additional property that the database is not used at all for counting the support of candidate itemsets after the first pass. Rather, an encoding of the candidate itemsets used in the previous pass is employed for this purpose. In later passes, the size of this encoding can become much smaller than the database, thus saving much reading effort. We will explain these points in more detail when we describe the algorithms.

Notation We assume that items in each transaction are kept sorted in their lexicographic order. It is straightforward to adapt these algorithms to the case where the database $\mathcal{D}$ is kept normalized and each database record is a <TID, item> pair, where TID is the identifier of the corresponding transaction.

We call the number of items in an itemset its *size*, and call an itemset of size k a k-itemset. Items within an itemset are kept in lexicographic order. We use the notation $c[1] \cdot c[2] \cdot \ldots \cdot c[k]$

Table 1: Notation

k-itemset	An itemset having k items.
L_k	Set of large k-itemsets (those with minimum support). Each member of this set has two fields: i) itemset and ii) support count.
C_k	Set of candidate k-itemsets (potentially large itemsets). Each member of this set has two fields: i) itemset and ii) support count.
$\overline{C}_k$	Set of candidate k-itemsets when the TIDs of the generating transactions are kept associated with the candidates.

to represent a k-itemset c consisting of items $c[1], c[2], \ldots c[k]$, where $c[1] < c[2] < \ldots < c[k]$. If $c = X \cdot Y$ and Y is an m-itemset, we also call Y an *m-extension* of X. Associated with each itemset is a count field to store the support for this itemset. The count field is initialized to zero when the itemset is first created.

We summarize in Table 1 the notation used in the algorithms. The set $\overline{C}_k$ is used by AprioriTid and will be further discussed when we describe this algorithm.

2.1 Algorithm Apriori

Figure 1 gives the Apriori algorithm. The first pass of the algorithm simply counts item occurrences to determine the large 1-itemsets. A subsequent pass, say pass k, consists of two phases. First, the large itemsets L_{k-1} found in the $(k-1)$th pass are used to generate the candidate itemsets C_k, using the apriori-gen function described in Section 2.1.1. Next, the database is scanned and the support of candidates in C_k is counted. For fast counting, we need to efficiently determine the candidates in C_k that are contained in a given transaction t. Section 2.1.2 describes the subset function used for this purpose. Section 2.1.3 discusses buffer management.

$L_1 = \{$large 1-itemsets$\}$;
for ($k = 2$; $L_{k-1} \neq \emptyset$; k++) **do begin**
    C_k = apriori-gen(L_{k-1}); // New candidates – see Section 2.1.1
    forall transactions $t \in \mathcal{D}$ **do begin**
        C_t = subset(C_k, t); // Candidates contained in t – see Section 2.1.2
        forall candidates $c \in C_t$ **do**
            c.count++;
    end
    $L_k = \{c \in C_k \mid c.\text{count} \geq \text{minsup}\}$
end
Answer $= \bigcup_k L_k$;

Figure 1: Algorithm Apriori

2.1.1 Apriori Candidate Generation

The `apriori-gen` function takes as argument L_{k-1}, the set of all large $(k-1)$-itemsets. It returns a superset of the set of all large k-itemsets. The function works as follows. [1] First, in the *join* step, we join L_{k-1} with L_{k-1}:

insert into C_k
select $p.\text{item}_1$, $p.\text{item}_2$, ..., $p.\text{item}_{k-1}$, $q.\text{item}_{k-1}$
from L_{k-1} p, L_{k-1} q
where $p.\text{item}_1 = q.\text{item}_1, \ldots, p.\text{item}_{k-2} = q.\text{item}_{k-2}, p.\text{item}_{k-1} < q.\text{item}_{k-1}$;

Next, in the *prune* step, we delete all itemsets $c \in C_k$ such that some $(k-1)$-subset of c is not in L_{k-1}:

forall itemsets $c \in C_k$ **do**
  forall $(k-1)$-subsets s of c **do**
    if $(s \notin L_{k-1})$ **then**
      delete c from C_k;

Example Let L_3 be {{1 2 3}, {1 2 4}, {1 3 4}, {1 3 5}, {2 3 4}}. After the join step, C_4 will be {{1 2 3 4}, {1 3 4 5} }. The prune step will delete the itemset {1 3 4 5} because the itemset {1 4 5} is not in L_3. We will then be left with only {1 2 3 4} in C_4.

Contrast this candidate generation with the one used in the AIS and SETM algorithms. In pass k of these algorithms (see Section 4 for details), a database transaction t is read and it is determined which of the large itemsets in L_{k-1} are present in t. Each of these large itemsets l is then extended with all those large items that are present in t and occur later in the lexicographic ordering than any of the items in l. Continuing with the previous example, consider a transaction {1 2 3 4 5}. In the fourth pass, AIS and SETM will generate two candidates, {1 2 3 4} and {1 2 3 5}, by extending the large itemset {1 2 3}. Similarly, an additional three candidate itemsets will be generated by extending the other large itemsets in L_3, leading to a total of 5 candidates for consideration in the fourth pass. Apriori, on the other hand, generates and counts only one itemset, {1 3 4 5}, because it concludes *a priori* that the other combinations cannot possibly have minimum support.

[1] Concurrent to our work, the following two-step candidate generation procedure has been proposed in [MTV94]:

$$C'_k = \{X \cup X' | X, X' \in L_{k-1}, |X \cap X'| = k-2\}$$

$$C_k = \{X \in C'_k | X \text{ contains } k \text{ members of } L_{k-1}\}$$

These two steps are similar to our join and prune steps respectively. However, in general, step 1 would produce a superset of the candidates produced by our join step. For example, if L_2 were {{1 2}, {2, 3}}, then step 1 of [MTV94] will generate the candidate {1 2 3}, whereas our join step will not generate any candidate.

Correctness We need to show that $C_k \supseteq L_k$. Clearly, any subset of a large itemset must also have minimum support. Hence, if we extended each itemset in L_{k-1} with all possible items and then deleted all those whose $(k-1)$-subsets were not in L_{k-1}, we would be left with a superset of the itemsets in L_k.

The join is equivalent to extending L_{k-1} with each item in the database and then deleting those itemsets for which the $(k-1)$-itemset obtained by deleting the $(k-1)$th item is not in L_{k-1}. The condition $p.\text{item}_{k-1} < q.\text{item}_{k-1}$ simply ensures that no duplicates are generated. Thus, after the join step, $C_k \supseteq L_k$. By similar reasoning, the prune step, where we delete from C_k all itemsets whose $(k-1)$-subsets are not in L_{k-1}, also does not delete any itemset that could be in L_k.

Variation: Counting Candidates of Multiple Sizes in One Pass Rather than counting only candidates of size k in the kth pass, we can also count the candidates C'_{k+1}, where C'_{k+1} is generated from C_k, etc. Note that $C'_{k+1} \supseteq C_{k+1}$ since C_{k+1} is generated from L_k. This variation can pay off in the later passes when the cost of counting and keeping in memory additional $C'_{k+1} - C_{k+1}$ candidates becomes less than the cost of scanning the database.

Membership Test The prune step requires testing that all $(k-1)$-subsets of a newly generated k-candidate-itemset are present in L_{k-1}. To make this membership test fast, large itemsets are stored in a hash table.

2.1.2 Subset Function

Candidate itemsets C_k are stored in a *hash-tree*. A node of the hash-tree either contains a list of itemsets (a *leaf* node) or a hash table (an *interior* node). In an interior node, each bucket of the hash table points to another node. The root of the hash-tree is defined to be at depth 1. An interior node at depth d points to nodes at depth $d+1$. Itemsets are stored in the leaves. When we add an itemset c, we start from the root and go down the tree until we reach a leaf. At an interior node at depth d, we decide which branch to follow by applying a hash function to the dth item of the itemset. All nodes are initially created as leaf nodes. When the number of itemsets in a leaf node exceeds a specified threshold, the leaf node is converted to an interior node.

Starting from the root node, the subset function finds all the candidates contained in a transaction t as follows. If we are at a leaf, we find which of the itemsets in the leaf are contained in t and add references to them to the answer set. If we are at an interior node and we have reached it by hashing the item i, we hash on each item that comes after i in t and recursively apply this procedure to the node in the corresponding bucket. For the root node, we hash on every item in t.

To see why the subset function returns the desired set of references, consider what happens at the root node. For any itemset c contained in transaction t, the first item of c must be in t. At the root, by hashing on every item in t, we ensure that we only ignore itemsets that start with an item not in t. Similar arguments apply at lower depths. The only additional factor is that, since the items in any itemset are ordered, if we reach the current node by hashing the item i, we only need to consider the items in t that occur after i.

If k is the size of a candidate itemset in the hash-tree, we can find in O(k) time whether the itemset is contained in a transaction by using a temporary bitmap. Each bit of the bitmap corresponds an item. The bitmap is created once for the data structure, and reinitialized for each transaction. This initialization takes O(size(transaction)) time for each transaction.

2.1.3 Buffer Management

In the candidate generation phase of pass k, we need storage for large itemsets L_{k-1} and the candidate itemsets C_k. In the counting phase, we need storage for C_k and at least one page to buffer the database transactions.

First, assume that L_{k-1} fits in memory but that the set of candidates C_k does not. The apriori-gen function is modified to generate as many candidates of C_k as will fit in the buffer and the database is scanned to count the support of these candidates. Large itemsets resulting from these candidates are written to disk, while those candidates without minimum support are deleted. This procedure is repeated until all of C_k has been counted.

If L_{k-1} does not fit in memory either, we externally sort L_{k-1}. We bring into memory a block of L_{k-1} in which the first $k-2$ items are the same. We now generate candidates using this block. We keep reading blocks of L_{k-1} and generating candidates until the memory fills up, and then make a pass over the data. This procedure is repeated until all of C_k has been counted. Unfortunately, we can no longer prune those candidates whose subsets are not in L_{k-1}, as the whole of L_{k-1} is not available in memory.

2.2 Algorithm AprioriTid

The AprioriTid algorithm, shown in Figure 2, also uses the apriori-gen function (given in Section 2.1.1) to determine the candidate itemsets before the pass begins. The interesting feature of this algorithm is that the database $\mathcal{D}$ is not used for counting support after the first pass. Rather, the set $\overline{C}_k$ is used for this purpose. Each member of the set $\overline{C}_k$ is of the form $< TID, \{X_k\} >$, where each X_k is a potentially large k-itemset present in the transaction with identifier TID. For $k = 1$, $\overline{C}_1$ corresponds to the database $\mathcal{D}$, although conceptually each item i is replaced by the itemset $\{i\}$. For $k > 1$, $\overline{C}_k$ is generated by the algorithm (step 10). The member of $\overline{C}_k$ corresponding to transaction t is $<t.TID, \{c \in C_k | c \text{ contained in } t\}>$. If a transaction does not contain any candidate k-itemset, then $\overline{C}_k$ will not have an entry for this transaction. Thus, the number of entries in $\overline{C}_k$ may be smaller than the number of transactions in the database, especially for large values of k. In addition, for large values of k, each entry may be smaller than the corresponding transaction because very few candidates may be contained in the transaction. However, for small values for k, each entry may be larger than the corresponding transaction because an entry in C_k includes all candidate k-itemsets contained in the transaction. We further explore this trade-off in Section 4.

We establish the correctness of the algorithm in Section 2.2.1. In Section 2.2.2, we give the data structures used to implement the algorithm, and we discuss buffer management in Section 2.2.3.

1) $L_1 = \{$large 1-itemsets$\}$;
2) $\overline{C}_1 =$ database $\mathcal{D}$;
3) **for** ($k = 2$; $L_{k-1} \neq \emptyset$; k++) **do begin**
4) $C_k =$ apriori-gen(L_{k-1}); // New candidates – see Section 2.1.1
5) $\overline{C}_k = \emptyset$;
6)   **forall** entries $t \in \overline{C}_{k-1}$ **do begin**
7)   // determine candidate itemsets in C_k contained in the transaction with identifier t.TID
        $C_t = \{c \in C_k \mid (c - c[k]) \in t.\text{set-of-itemsets} \ \wedge \ (c - c[k-1]) \in t.\text{set-of-itemsets}\}$;
8)   **forall** candidates $c \in C_t$ **do**
9) c.count++;
10)   **if** ($C_t \neq \emptyset$) **then** $\overline{C}_k$ += $< t.\text{TID}, C_t >$;
11)   **end**
12) $L_k = \{c \in C_k \mid c.\text{count} \geq \text{minsup}\}$
13) **end**
14) Answer $= \bigcup_k L_k$;

Figure 2: Algorithm AprioriTid

Example Consider the database in Figure 3 and assume that minimum support is 2 transactions. Calling apriori-gen with L_1 at step 4 gives the candidate itemsets C_2. In steps 6 through 10, we count the support of candidates in C_2 by iterating over the entries in $\overline{C}_1$ and generate $\overline{C}_2$. The first entry in $\overline{C}_1$ is { {1} {3} {4} }, corresponding to transaction 100. The C_t at step 7 corresponding to this entry t is { {1 3} }, because {1 3} is a member of C_2 and both ({1 3} - {1}) and ({1 3} - {3}) are members of t.set-of-itemsets.

Calling apriori-gen with L_2 gives C_3. Making a pass over the data with $\overline{C}_2$ and C_3 generates $\overline{C}_3$. Note that there is no entry in $\overline{C}_3$ for the transactions with TIDs 100 and 400, since they do not contain any of the itemsets in C_3. The candidate {2 3 5} in C_3 turns out to be large and is the only member of L_3. When we generate C_4 using L_3, it turns out to be empty, and we terminate.

2.2.1 Correctness

Rather than using the database transactions, AprioriTid uses the entries in $\overline{C}_k$ to count the support of candidates in C_k. To simplify the proof, we assume that in step 10 of AprioriTid, we always

Database

TID	Items
100	1 3 4
200	2 3 5
300	1 2 3 5
400	2 5

$\overline{C}_1$

TID	Set-of-Itemsets
100	{ {1}, {3}, {4} }
200	{ {2}, {3}, {5} }
300	{ {1}, {2}, {3}, {5} }
400	{ {2}, {5} }

L_1

Itemset	Support
{1}	2
{2}	3
{3}	3
{5}	3

C_2

Itemset
{1 2}
{1 3}
{1 5}
{2 3}
{2 5}
{3 5}

$\overline{C}_2$

TID	Set-of-Itemsets
100	{ {1 3} }
200	{ {2 3}, {2 5}, {3 5} }
300	{ {1 2}, {1 3}, {1 5}, {2 3}, {2 5}, {3 5} }
400	{ {2 5} }

L_2

Itemset	Support
{1 3}	2
{2 3}	2
{2 5}	3
{3 5}	2

C_3

Itemset
{2 3 5}

$\overline{C}_3$

TID	Set-of-Itemsets
200	{ {2 3 5} }
300	{ {2 3 5} }

L_3

Itemset	Support
{2 3 5}	2

Figure 3: Example

add $<t.\text{TID}, C_t>$ to $\overline{C}_k$, rather than adding an entry only when C_t is non-empty. For correctness, we need to establish that the set C_t generated in step 7 in the kth pass is the same as the set of candidate k-itemsets in C_k contained in the transaction with identifier t.TID.

We say that the set $\overline{C}_k$ is *complete* if $\forall t \in \overline{C}_k$, t.set-of-itemsets includes all large k-itemsets contained in the transaction with identifier t.TID. We say that the set $\overline{C}_k$ is *correct* if $\forall t \in \overline{C}_k$, t.set-of-itemsets does not include any k-itemset not contained in the transaction with identifier t.TID. The set L_k is *correct* if it is the same as the set of all large k-itemsets. We say that the set C_t generated in step 7 in the kth pass is *correct* if it is the same as the set of candidate k-itemsets in C_k contained in the transaction with identifier t.TID.

Lemma 1 *$\forall k > 1$, if $\overline{C}_{k-1}$ is correct and complete and L_{k-1} is correct, then the set C_t generated in step 7 in the kth pass is the same as the set of candidate k-itemsets in C_k contained in the transaction with identifier t.TID.*

By simple rewriting, a candidate itemset $c = c[1] \cdot c[2] \cdot \ldots \cdot c[k]$ is present in transaction t.TID if and only if both $c^1 = (c - c[k])$ and $c^2 = (c - c[k-1])$ are in transaction t.TID. Since C_k was obtained by calling apriori-gen(L_{k-1}), all subsets of $c \in C_k$ must be large. So, c^1 and c^2 must be large itemsets. Thus, if $c \in C_k$ is contained in transaction t.TID, c^1 and c^2 must be members of t.set-of-itemsets since $\overline{C}_{k-1}$ is complete Hence c will be a member of C_t. Since $\overline{C}_{k-1}$ is correct, if

c^1 (c^2) is not present in transaction t.TID then c^1 (c^2) is not contained in t.set-of-itemsets. Hence, if $c \in C_k$ is not contained in transaction t.TID, c will not be a member of C_t. □

Lemma 2 *$\forall k > 1$, if L_{k-1} is correct and the set C_t generated in step 7 in the kth pass is the same as the set of candidate k-itemsets in C_k contained in the transaction with identifier t.TID, then the set $\overline{C}_k$ is correct and complete.*

Since the apriori-gen function guarantees that $C_k \supseteq L_k$, the set C_t includes all large k-itemsets contained in t.TID. These are added in step 10 to $\overline{C}_k$ and hence $\overline{C}_k$ is complete. Since C_t only includes itemsets contained in the transaction t.TID, and only itemsets in C_t are added to $\overline{C}_k$, it follows that $\overline{C}_k$ is correct. □

Theorem 1 *$\forall k > 1$, the set C_t generated in step 7 in the kth pass is the same as the set of candidate k-itemsets in C_k contained in the transaction with identifier t.TID.*

We first prove by induction on k that the set $\overline{C}_k$ is correct and complete and L_k correct for all $k \geq 1$. For $k = 1$, this is trivially true since $\overline{C}_1$ corresponds to the database $\mathcal{D}$. By definition, L_1 is also correct. Assume this holds for $k = n$. From Lemma 1, the set C_t generated in step 7 in the $(n+1)$th pass will consist of exactly those itemsets in C_{n+1} contained in the transaction with identifier t.TID. Since the apriori-gen function guarantees that $C_{n+1} \supseteq L_{n+1}$ and C_t is correct, L_{n+1} will be correct. ¿From Lemma 2, the set $\overline{C}_{n+1}$ will be correct and complete.

Since $\overline{C}_k$ is correct and complete and L_k correct for all $k \geq 1$, the theorem follows directly from Lemma 1. □

2.2.2 Data Structures

We assign each candidate itemset a unique number, called its ID. Each set of candidate itemsets C_k is kept in an array indexed by the IDs of the itemsets in C_k. A member of $\overline{C}_k$ is now of the form $<$ TID, {ID} $>$. Each $\overline{C}_k$ is stored in a sequential structure.

The apriori-gen function generates a candidate k-itemset c_k by joining two large $(k-1)$-itemsets. We maintain two additional fields for each candidate itemset: i) *generators* and ii) *extensions*. The generators field of a candidate itemset c_k stores the IDs of the two large $(k-1)$-itemsets whose join generated c_k. The extensions field of an itemset c_k stores the IDs of all the $(k+1)$-candidates that are extensions of c_k. Thus, when a candidate c_k is generated by joining l^1_{k-1} and l^2_{k-1}, we save the IDs of l^1_{k-1} and l^2_{k-1} in the generators field for c_k. At the same time, the ID of c_k is added to the extensions field of l^1_{k-1}.

We now describe how Step 7 of Figure 2 is implemented using the above data structures. Recall that the t.set-of-itemsets field of an entry t in $\overline{C}_{k-1}$ gives the IDs of all $(k-1)$-candidates contained in transaction t.TID. For each such candidate c_{k-1} the extensions field gives T_k, the set of IDs of all the candidate k-itemsets that are extensions of c_{k-1}. For each c_k in T_k, the generators field gives the IDs of the two itemsets that generated c_k. If these itemsets are present in the entry for t.set-of-itemsets, we can conclude that c_k is present in transaction t.TID, and add c_k to C_t.

We actually need to store only l_{k-1}^2 in the generators field, since we reached c_k starting from the ID of l_{k-1}^1 in t. We omitted this optimization in the above description to simplify exposition. Given an ID and the data structures above, we can find the associated candidate itemset in constant time. We can also find in constant time whether or not an ID is present in the t.set-of-itemsets field by using a temporary bitmap. Each bit of the bitmap corresponds to an ID in C_k. This bitmap is created once at the beginning of the pass and is reinitialized for each entry t of $\overline{C}_k$.

2.2.3 Buffer Management

In the kth pass, AprioriTid needs memory for L_{k-1} and C_k during candidate generation. During the counting phase, it needs memory for C_{k-1}, C_k, and a page each for $\overline{C}_{k-1}$ and $\overline{C}_k$. Note that the entries in $\overline{C}_{k-1}$ are needed sequentially and that the entries in $\overline{C}_k$ can be written to disk as they are generated.

At the time of candidate generation, when we join L_{k-1} with itself, we fill up roughly half the buffer with candidates. This allows us to keep the relevant portions of both C_k and C_{k-1} in memory during the counting phase. In addition, we ensure that all candidates with the same first $(k-1)$ items are generated at the same time.

The computation is now effectively partitioned because none of the candidates in memory that turn out to large at the end of the pass will join with any of the candidates not yet generated to derive potentially large itemsets. Hence we can assume that the candidates in memory are the only candidates in C_k and find all large itemsets that are extensions of candidates in C_k by running the algorithm to completion. This may cause further partitioning of the computation downstream. Having thus run the algorithm to completion, we return to L_{k-1}, generate some more candidates in C_k, count them, and so on. Note that the prune step of the apriori-gen function cannot be applied after partitioning because we do not know all the large k-itemsets.

When L_k does not fit in memory, we need to externally sort L_k as in the buffer management scheme used for Apriori.

3 Discovering Rules

The association rules that we consider here are somewhat more general than in [AIS93b] in that we allow a consequent to have more than one item; rules in [AIS93b] were limited to single item consequents. We first give a straightforward generalization of the algorithm in [AIS93b] and then present a faster algorithm.

To generate rules, for every large itemset l, we find all non-empty subsets of l. For every such subset a, we output a rule of the form $a \Longrightarrow (l - a)$ if the ratio of support(l) to support(a) is at least *minconf*. We consider all subsets of l to generate rules with multiple consequents. Since the large itemsets are stored in hash tables, the support counts for the subset itemsets can be found efficiently.

We can improve the above procedure by generating the subsets of a large itemset in a recursive depth-first fashion. For example, given an itemset $ABCD$, we first consider the subset ABC, then AB, etc. Then if a subset a of a large itemset l does not generate a rule, the subsets of a need not be considered for generating rules using l. For example, if $ABC \Longrightarrow D$ does not have enough confidence, we need not check whether $AB \Longrightarrow CD$ holds. We do not miss any rules because the support of any subset $\tilde{a}$ of a must be as great as the support of a. Therefore, the confidence of the rule $\tilde{a} \Longrightarrow (l - \tilde{a})$ cannot be more than the confidence of $a \Longrightarrow (l - a)$. Hence, if a did not yield a rule involving all the items in l with a as the antecedent, neither will $\tilde{a}$. The following algorithm embodies these ideas:

// Simple Algorithm
forall large itemsets l_k, $k \geq 2$ **do**
    call genrules(l_k, l_k);

// The genrules generates all valid rules $\tilde{a} \Longrightarrow (l_k - \tilde{a})$, for all $\tilde{a} \subset a_m$
procedure genrules(l_k: large k-itemset, a_m: large m-itemset)
$A = \{(m{-}1)\text{-itemsets } a_{m-1} \mid a_{m-1} \subset a_m\}$;
forall $a_{m-1} \in A$ **do begin**
    conf = support(l_k)/support(a_{m-1});
    if (*conf* $\geq$ *minconf*) **then begin**
        output the rule $a_{m-1} \Longrightarrow (l_k - a_{m-1})$, with confidence = *conf* and support = support(l_k);
        if $(m - 1 > 1)$ **then**
            call genrules(l_k, a_{m-1}); // to generate rules with subsets of a_{m-1} as the antecedents
    end
end

3.1 A Faster Algorithm

We showed earlier that if $a \Longrightarrow (l - a)$ does not hold, neither does $\tilde{a} \Longrightarrow (l - \tilde{a})$ for any $\tilde{a} \subset a$. By rewriting, it follows that for a rule $(l - c) \Longrightarrow c$ to hold, all rules of the form $(l - \tilde{c}) \Longrightarrow \tilde{c}$ must also hold, where $\tilde{c}$ is a non-empty subset of c. For example, if the rule $AB \Longrightarrow CD$ holds, then the rules $ABC \Longrightarrow D$ and $ABD \Longrightarrow C$ must also hold.

Consider the above property that for a given large itemset, if a rule with consequent c holds then so do rules with consequents that are subsets of c. This is similar to the property that if an itemset is large then so are all its subsets. ¿From a large itemset l, therefore, we first generate all rules with one item in the consequent. We then use the consequents of these rules and the function apriori-gen in Section 2.1.1 to generate all possible consequents with two items that can appear in a rule generated from l, etc. An algorithm using this idea is given below. The rules having one-item consequents in step 2 of this algorithm can be found by using a modified version of the preceding genrules function in which steps 8 and 9 are deleted to avoid the recursive call.

```
 // Faster Algorithm
1)  forall large k-itemsets l_k, k >= 2 do begin
2)     H_1 = { consequents of rules derived from l_k with one item in the consequent };
3)     call ap-genrules(l_k, H_1);
4)  end

procedure ap-genrules(l_k: large k-itemset, H_m: set of m-item consequents)
    if (k > m + 1) then begin
       H_{m+1} = apriori-gen(H_m);
       forall h_{m+1} ∈ H_{m+1} do begin
          conf = support(l_k)/support(l_k - h_{m+1});
          if (conf >= minconf) then
             output the rule (l_k - h_{m+1}) ==> h_{m+1} with confidence = conf and support = support(l_k);
          else
             delete h_{m+1} from H_{m+1};
       end
       call ap-genrules(l_k, H_{m+1});
    end
```

As an example of the advantage of this algorithm, consider a large itemset $ABCDE$. Assume that $ACDE \Longrightarrow B$ and $ABCE \Longrightarrow D$ are the only one-item consequent rules derived from this itemset that have the minimum confidence. If we use the simple algorithm, the recursive call genrules($ABCDE$, $ACDE$) will test if the two-item consequent rules $ACD \Longrightarrow BE$, $ADE \Longrightarrow BC$, $CDE \Longrightarrow BA$, and $ACE \Longrightarrow BD$ hold. The first of these rules cannot hold, because $E \subset BE$, and $ABCD \Longrightarrow E$ does not have minimum confidence. The second and third rules cannot hold for similar reasons. The call genrules($ABCDE$, $ABCE$) will test if the rules $ABC \Longrightarrow DE$, $ABE \Longrightarrow DC$, $BCE \Longrightarrow DA$ and $ACE \Longrightarrow BD$ hold, and will find that the first three of these rules do not hold. In fact, the only two-item consequent rule that can possibly hold is $ACE \Longrightarrow BD$, where B and D are the consequents in the valid one-item consequent rules. This is the only rule that will be tested by the faster algorithm.

4 Performance

To assess the relative performance of the algorithms for discovering large itemsets, we performed several experiments on an IBM RS/6000 530H workstation with a CPU clock rate of 33 MHz, 64

MB of main memory, and running AIX 3.2. The data resided in the AIX file system and was stored on a 2GB SCSI 3.5" drive, with measured sequential throughput of about 2 MB/second.

We first give an overview of the AIS [AIS93b] and SETM [HS93] algorithms against which we compare the performance of the Apriori and AprioriTid algorithms. We then describe the synthetic datasets used in the performance evaluation and show the performance results. Next, we show the performance results for three real-life datasets obtained from a retail and a direct mail company. Finally, we describe how the best performance features of Apriori and AprioriTid can be combined into an AprioriHybrid algorithm and demonstrate its scale-up properties.

4.1 The AIS Algorithm

Figure 4 summarizes the essence of the AIS algorithm (see [AIS93b] for further details). Candidate itemsets are generated and counted on-the-fly as the database is scanned. After reading a transaction, it is determined which of the itemsets that were found to be large in the previous pass are contained in this transaction (step 5). New candidate itemsets are generated by extending these large itemsets with other items in the transaction (step 7). A large itemset l is extended with only those items that are large and occur later in the lexicographic ordering of items than any of the items in l. The candidates generated from a transaction are added to the set of candidate itemsets maintained for the pass, or the counts of the corresponding entries are increased if they were created by an earlier transaction (step 9).

1) $L_1 = \{$large 1-itemsets$\}$;
2) **for** ($k = 2$; $L_{k-1} \neq \emptyset$; k++) **do begin**
3) $C_k = \emptyset$;
4)   **forall** transactions $t \in \mathcal{D}$ **do begin**
5) $L_t = \text{subset}(L_{k-1}, t)$; // Large itemsets contained in t
6)   **forall** large itemsets $l_t \in L_t$ **do begin**
7) C_t = 1-extensions of l_t contained in t; // Candidates contained in t
8)   **forall** candidates $c \in C_t$ **do**
9)   **if** ($c \in C_k$) **then**
               add 1 to the count of c in the corresponding entry in C_k
            else
               add c to C_k with a count of 1;
10)   **end**
11) $L_k = \{c \in C_k \mid c.\text{count} \geq \text{minsup}\}$
12) **end**
13) Answer $= \bigcup_k L_k$;

Figure 4: Algorithm AIS

Data Structures The data structures required for maintaining large and candidate itemsets were not specified in [AIS93b]. We store the large itemsets in a dynamic multi-level hash table to make the subset operation in step 5 fast, using the algorithm described in Section 2.1.2. Candidate itemsets are kept in a hash table associated with the respective large itemsets from which they originate in order to make the membership test in step 9 fast.

Buffer Management When a newly generated candidate itemset causes the buffer to overflow, we discard from memory the corresponding large itemset and all candidate itemsets generated from it. This reclamation procedure is executed as often as necessary during a pass. The large itemsets discarded in a pass are extended in the next pass. This technique is a simplified version of the buffer management scheme presented in [AIS93b].

4.2 The SETM Algorithm

The SETM algorithm [HS93] was motivated by the desire to use SQL to compute large itemsets. Our description of this algorithm in Figure 5 uses the same notation as used for the other algorithms, but is functionally identical to the SETM algorithm presented in [HS93]. $\overline{C}_k$ ($\overline{L}_k$) in Figure 5 represents the set of candidate (large) itemsets in which the TIDs of the generating transactions have been associated with the itemsets. Each member of these sets is of the form $< TID, \text{itemset} >$.

Like AIS, the SETM algorithm also generates candidates on-the-fly based on transactions read from the database. It thus generates and counts every candidate itemset that the AIS algorithm generates. However, to use the standard SQL join operation for candidate generation, SETM separates candidate generation from counting. It saves a copy of the candidate itemset together with the TID of the generating transaction in a sequential structure (step 9). At the end of the pass, the support count of candidate itemsets is determined by sorting (step 12) and aggregating this sequential structure (step 13).

SETM remembers the TIDs of the generating transactions with the candidate itemsets. To avoid needing a subset operation, it uses this information to determine the large itemsets contained in the transaction read (step 6). $\overline{L}_k \subseteq \overline{C}_k$ and is obtained by deleting those candidates that do not have minimum support (step 13). Assuming that the database is sorted in TID order, SETM can easily find the large itemsets contained in a transaction in the next pass by sorting $\overline{L}_k$ on TID (step 15). In fact, it needs to visit every member of $\overline{L}_k$ only once in the TID order, and the candidate generation in steps 5 through 11 can be performed using the relational merge-join operation [HS93].

The disadvantage of this approach is mainly due to the size of candidate sets $\overline{C}_k$. For each candidate itemset, the candidate set now has as many entries as the number of transactions in which the candidate itemset is present. Moreover, when we are ready to count the support for candidate itemsets at the end of the pass, $\overline{C}_k$ is in the wrong order and needs to be sorted on itemsets (step 12). After counting and pruning out small candidate itemsets that do not have minimum support, the resulting set $\overline{L}_k$ needs another sort on TID (step 15) before it can be used for generating candidates in the next pass.

1) L_1 = {large 1-itemsets};
2) $\overline{L}_1$ = {Large 1-itemsets together with the TIDs in which they appear, sorted on TID};
3) **for** ($k = 2$; $L_{k-1} \neq \emptyset$; k++) **do begin**
4) $\quad \overline{C}_k = \emptyset$;
5) $\quad$ **forall** transactions $t \in \mathcal{D}$ **do begin**
6) $\qquad L_t = \{l \in \overline{L}_{k-1} \mid l.\mathrm{TID} = t.\mathrm{TID}\}$; // Large $(k-1)$-itemsets contained in t
7) $\qquad$ **forall** large itemsets $l_t \in L_t$ **do begin**
8) $\qquad\quad C_t$ = 1-extensions of l_t contained in t; // Candidates in t
9) $\qquad\quad \overline{C}_k$ += $\{< t.\mathrm{TID}, c > \mid c \in C_t\}$;
10) $\qquad$ **end**
11) $\quad$ **end**
12) $\quad$ **sort** $\overline{C}_k$ on itemsets;
13) $\quad$ **delete** all itemsets $c \in \overline{C}_k$ for which c.count < minsup giving $\overline{L}_k$;
14) $\quad L_k = \{< l.$itemset, count of l in $\overline{L}_k > \mid l \in \overline{L}_k\}$; // Combined with step 13
15) $\quad$ **sort** $\overline{L}_k$ on TID;
16) **end**
17) Answer = $\bigcup_k L_k$;

Figure 5: Algorithm SETM

.

Buffer Management The performance of the SETM algorithm critically depends on the size of the set $\overline{C}_k$ relative to the size of memory. If $\overline{C}_k$ fits in memory, the two sorting steps can be performed using an in-memory sort. In [HS93], $\overline{C}_k$ was assumed to fit in main memory and buffer management was not discussed.

If $\overline{C}_k$ is too large to fit in memory, we write the entries in $\overline{C}_k$ to disk in FIFO order when the buffer allocated to the candidate itemsets fills up, as these entries are not required until the end of the pass. However, $\overline{C}_k$ now requires two external sorts.

4.3 Generation of Synthetic Data

We generated synthetic transactions to evaluate the performance of the algorithms over a large range of data characteristics. These transactions mimic the transactions in the retailing environment. Our model of the "real" world is that people tend to buy sets of items together. Each such set is potentially a maximal large itemset. An example of such a set might be sheets, pillow case, comforter, and ruffles. However, some people may buy only some of the items from such a set. For instance, some people might buy only sheets and pillow case, and some only sheets. A transaction may contain more than one large itemset. For example, a customer might place an order for a dress and jacket when ordering sheets and pillow cases, where the dress and jacket together form another large itemset. Transaction sizes are typically clustered around a mean and a few transactions have many items. Typical sizes of large itemsets are also clustered around a mean, with a few large itemsets having a large number of items.

To create a dataset, our synthetic data generation program takes the parameters shown in Table 2.

Table 2: Parameters

$\lvert\mathcal{D}\rvert$	Number of transactions
$\lvert T\rvert$	Average size of the Transactions
$\lvert I\rvert$	Average size of the maximal potentially large Itemsets
$\lvert L\rvert$	Number of maximal potentially large itemsets
N	Number of items

We first determine the size of the next transaction. The size is picked from a Poisson distribution with mean μ equal to $|T|$. Note that if each item is chosen with the same probability p, and there are N items, the expected number of items in a transaction is given by a binomial distribution with parameters N and p, and is approximated by a Poisson distribution with mean Np.

We then assign items to the transaction. Each transaction is assigned a series of potentially large itemsets. If the large itemset on hand does not fit in the transaction, the itemset is put in the transaction anyway in half the cases, and the itemset is moved to the next transaction the rest of the cases.

Large itemsets are chosen from a set $\mathcal{T}$ of such itemsets. The number of itemsets in $\mathcal{T}$ is set to $|L|$. There is an inverse relationship between $|L|$ and the average support for potentially large itemsets. An itemset in $\mathcal{T}$ is generated by first picking the size of the itemset from a Poisson distribution with mean μ equal to $|I|$. Items in the first itemset are chosen randomly. To model the phenomenon that large itemsets often have common items, some fraction of items in subsequent itemsets are chosen from the previous itemset generated. We use an exponentially distributed random variable with mean equal to the *correlation level* to decide this fraction for each itemset. The remaining items are picked at random. In the datasets used in the experiments, the correlation level was set to 0.5. We ran some experiments with the correlation level set to 0.25 and 0.75 but did not find much difference in the nature of our performance results.

Each itemset in $\mathcal{T}$ has a weight associated with it, which corresponds to the probability that this itemset will be picked. This weight is picked from an exponential distribution with unit mean, and is then normalized so that the sum of the weights for all the itemsets in $\mathcal{T}$ is 1. The next itemset to be put in the transaction is chosen from $\mathcal{T}$ by tossing an $|L|$-sided weighted coin, where the weight for a side is the probability of picking the associated itemset.

To model the phenomenon that all the items in a large itemset are not always bought together, we assign each itemset in $\mathcal{T}$ a *corruption level* c. When adding an itemset to a transaction, we keep dropping an item from the itemset as long as a uniformly distributed random number between 0 and 1 is less than c. Thus for an itemset of size l, we will add l items to the transaction $1 - c$ of the time, $l-1$ items $c(1-c)$ of the time, $l-2$ items $c^2(1-c)$ of the time, etc. The corruption level for an itemset is fixed and is obtained from a normal distribution with mean 0.5 and variance 0.1.

We generated datasets by setting $N = 1000$ and $|L| = 2000$. We chose 3 values for $|T|$: 5, 10, and 20. We also chose 3 values for $|I|$: 2, 4, and 6. The number of transactions was to set to 100,000 because, as we will see in Section 4.4, SETM could not be run for larger values. However, for our scale-up experiments, we generated datasets with up to 10 million transactions (838MB for $|T| = 20$). Table 3 summarizes the dataset parameter settings. For the same $|T|$ and $|\mathcal{D}|$ values, the size of datasets in megabytes were roughly equal for the different values of $|I|$.

Table 3: Parameter settings (Synthetic datasets)

Name	$\lvert T\rvert$	$\lvert I\rvert$	$\lvert\mathcal{D}\rvert$	Size in Megabytes
T5.I2.D100K	5	2	100K	2.4
T10.I2.D100K	10	2	100K	4.4
T10.I4.D100K	10	4	100K	
T20.I2.D100K	20	2	100K	8.4
T20.I4.D100K	20	4	100K	
T20.I6.D100K	20	6	100K	

4.4 Experiments with Synthetic Data

Figure 6 shows the execution times for the six synthetic datasets given in Table 3 for decreasing values of minimum support. As the minimum support decreases, the execution times of all the algorithms increase because of increases in the total number of candidate and large itemsets.

For SETM, we have only plotted the execution times for the dataset T5.I2.D100K in Figure 6. The execution times for SETM for the two datasets with an average transaction size of 10 are given in Table 4. We did not plot the execution times in Table 4 on the corresponding graphs because they are too large compared to the execution times of the other algorithms. For the three datasets with transaction sizes of 20, SETM took too long to execute and we aborted those runs as the trends were clear. Clearly, Apriori beats SETM by more than an order of magnitude for large datasets.

Table 4: Execution times in seconds for SETM

Dataset	Algorithm	Minimum Support				
		2.0%	1.5%	1.0%	0.75%	0.5%
T10.I2.D100K	SETM	74	161	838	1262	1878
	Apriori	4.4	5.3	11.0	14.5	15.3
T10.I4.D100K	SETM	41	91	659	929	1639
	Apriori	3.8	4.8	11.2	17.4	19.3

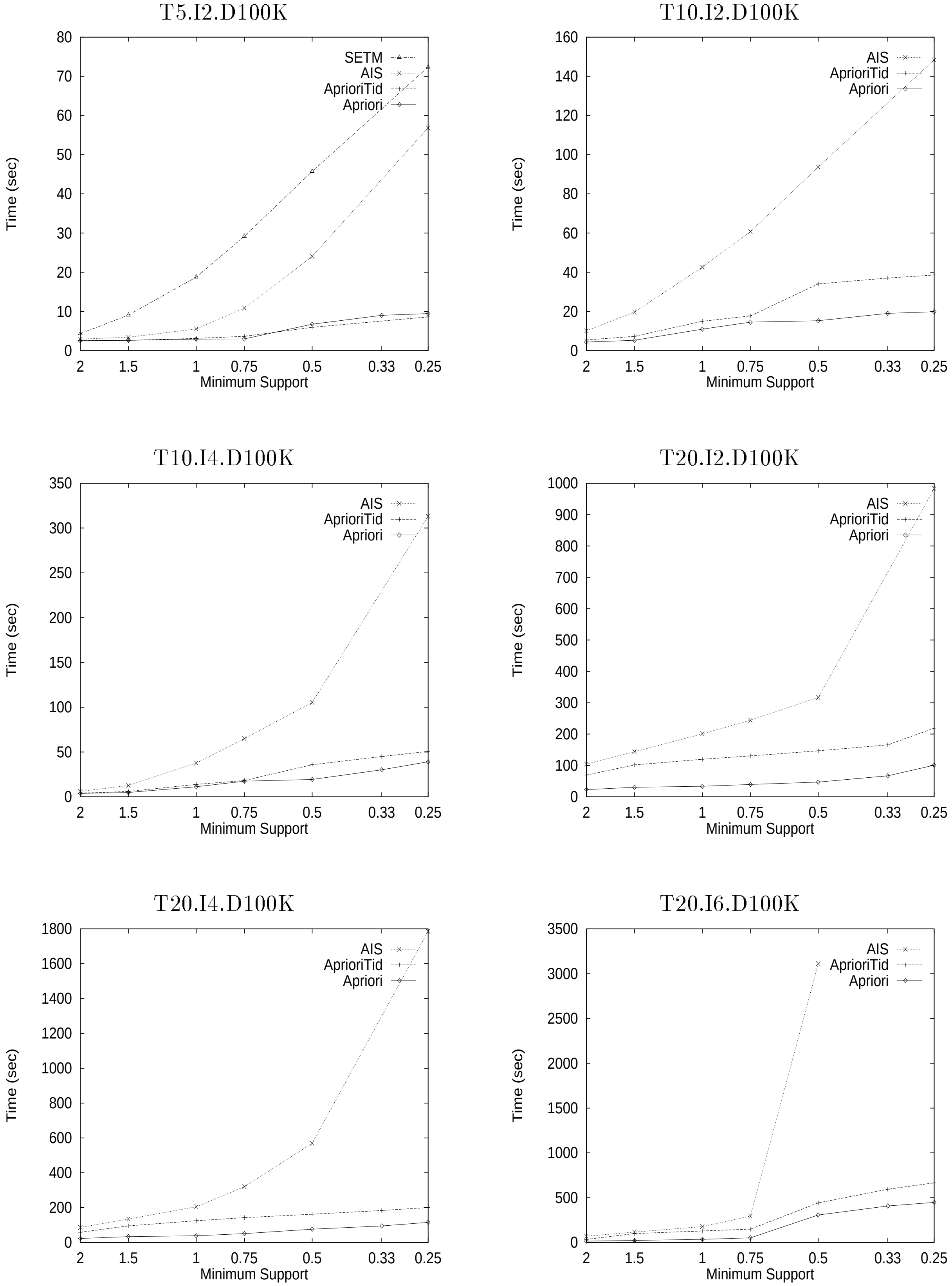

Figure 6: Execution times: Synthetic Data

Apriori beats AIS for all problem sizes, by factors ranging from 2 for high minimum support to more than an order of magnitude for low levels of support. AIS always did considerably better than SETM. For small problems, AprioriTid did about as well as Apriori, but performance degraded to about twice as slow for large problems.

4.5 Explanation of the Relative Performance

To explain these performance trends, we show in Figure 7 the sizes of the large and candidate sets in different passes for the T10.I4.D100K dataset for the minimum support of 0.75%. Note that the Y-axis in this graph has a log scale.

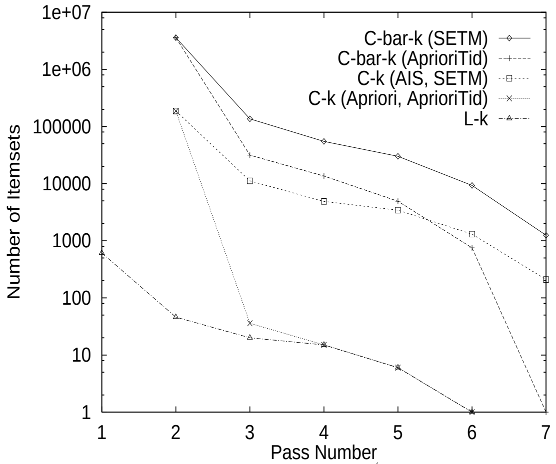

Figure 7: Sizes of the large and candidate sets (T10.I4.D100K, minsup = 0.75%)

The fundamental problem with the SETM algorithm is the size of its $\overline{C}_k$ sets. Recall that the size of the set $\overline{C}_k$ is given by $\sum_{\text{candidate itemsets } c} \text{support-count}(c)$. Thus, the sets $\overline{C}_k$ are roughly S times bigger than the corresponding C_k sets, where S is the average support count of the candidate itemsets. Unless the problem size is very small, the $\overline{C}_k$ sets have to be written to disk, and externally sorted twice, causing the SETM algorithm to perform poorly.[2] This explains the jump in time for SETM in Table 4 when going from 1.5% support to 1.0% support for datasets with transaction size 10. The largest dataset in the scale-up experiments for SETM in [HS93] was still small enough that $\overline{C}_k$ could fit in memory; hence they did not encounter this jump in execution time. Note that for the same minimum support, the support count for candidate itemsets increases linearly with the number of transactions. Thus, as we increase the number of transactions for the same values of $|T|$ and $|I|$, though the size of C_k does not change, the size of $\overline{C}_k$ goes up linearly. Thus, for datasets with more transactions, the performance gap between SETM and the other

[2] The cost of external sorting in SETM can be reduced somewhat as follows. Before writing out entries in $\overline{C}_k$ to disk, we can sort them on itemsets using an internal sorting procedure, and write them as sorted runs. These sorted runs can then be merged to obtain support counts. However, given the poor performance of SETM, we do not expect this optimization to affect the algorithm choice.

algorithms will become even larger.

The problem with AIS is that it generates too many candidates that later turn out to be small, causing it to waste too much effort. Apriori also counts too many small sets in the second pass (recall that C_2 is really a cross-product of L_1 with L_1). However, this wastage decreases dramatically from the third pass onward. Note that for the example in Figure 7, after pass 3, almost every candidate itemset counted by Apriori turns out to be a large set.

AprioriTid also has the problem of SETM that $\overline{C}_k$ tends to be large. However, the apriori candidate generation used by AprioriTid generates significantly fewer candidates than the transaction-based candidate generation used by SETM. As a result, the $\overline{C}_k$ of AprioriTid has fewer entries than that of SETM. AprioriTid is also able to use a single word (ID) to store a candidate rather than requiring as many words as the number of items in the candidate.[3] In addition, unlike SETM, AprioriTid does not have to sort $\overline{C}_k$. Thus, AprioriTid does not suffer as much as SETM from maintaining $\overline{C}_k$.

AprioriTid has the nice feature that it replaces a pass over the original dataset by a pass over the set $\overline{C}_k$. Hence, AprioriTid is very effective in later passes when the size of $\overline{C}_k$ becomes small compared to the size of the database. Thus, we find that AprioriTid beats Apriori when its $\overline{C}_k$ sets can fit in memory and the distribution of the large itemsets has a long tail. When $\overline{C}_k$ doesn't fit in memory, there is a jump in the execution time for AprioriTid, such as when going from 0.75% to 0.5% for datasets with transaction size 10 in Figure 6. In this region, Apriori starts beating AprioriTid.

4.6 Reality Check

To confirm the relative performance trends we observed using synthetic data, we experimented with three real-life datasets: a sales transactions dataset obtained from a retail chain and two customer-order datasets obtained from a mail order company. We present the results of these experiments below.

Retail Sales Data The data from the retail chain consists of the sales transactions from one store over a short period of time. A transaction contains the names of the departments from which a customer bought a product in a visit to the store. There are a total of 63 items, representing departments. There are 46,873 transactions with an average size of 2.47. The size of the dataset is

[3] For SETM to use IDs, it would have to maintain two additional in-memory data structures: a hash table to find out whether a candidate has been generated previously, and a mapping from the IDs to candidates. However, this would destroy the set-oriented nature of the algorithm. Also, once we have the hash table which gives us the IDs of candidates, we might as well count them at the same time and avoid the two external sorts. We experimented with this variant of SETM and found that, while it did better than SETM, it still performed much worse than Apriori or AprioriTid.

very small, only 0.65MB. Some performance results for this dataset were reported in [HS93].

Figure 8 shows the execution times of the four algorithms.[4] The $\overline{C}_k$ sets for both SETM and AprioriTid fit in memory for this dataset. Apriori and AprioriTid are roughly three times as fast as AIS and four times faster than SETM.

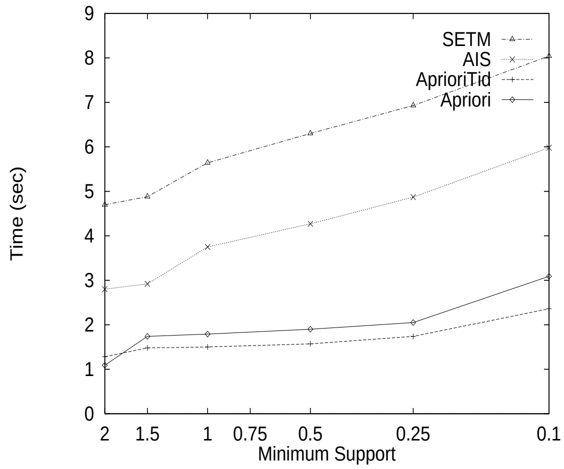

Figure 8: Execution times: Retail sales data

Mail Order data A transaction in the first dataset from the mail order company consists of items ordered by a customer in a single mail order. There are a total of 15836 items. The average size of a transaction is 2.62 items and there are a total of 2.9 million transactions. The size of this dataset is 42 MB. A transaction in the second dataset consists of all the items ordered by a customer from the company in all orders together. Again, there are a total of 15836 items, but the average size of a transaction is now 31 items and there are a total of 213,972 transactions. The size of this dataset is 27 MB. We will refer to these datasets as M.order and M.cust respectively.

The execution times for these two datasets are shown in Figures 9 and 10 respectively. For both datasets, AprioriTid is initially comparable to Apriori but becomes up to twice as slow for lower supports. For M.order, Apriori outperforms AIS by a factor of 2 to 6 and beats SETM by a factor of about 15. For M.cust, Apriori beats AIS by a factor of 3 to 30. SETM had to be aborted (after taking 20 times the time Apriori took to complete) because, even for 2% support, the set $\overline{C}_2$ became larger than the disk capacity.

[4]The execution times for SETM in this figure are a little higher compared to those reported in [HS93]. The timings in [HS93] were obtained on a RS/6000 350 processor, whereas our experiments have been run on a slower RS/6000 530H processor. The execution time for 1% support for AIS is lower than that reported in [AIS93b] because of improvements in the data structures for storing large and candidate itemsets.

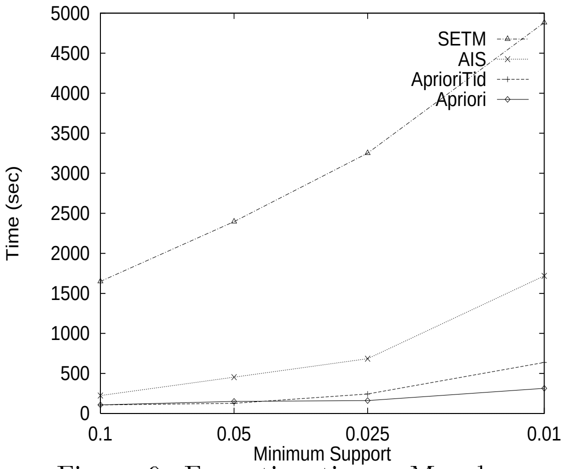

Figure 9: Execution times: M.order

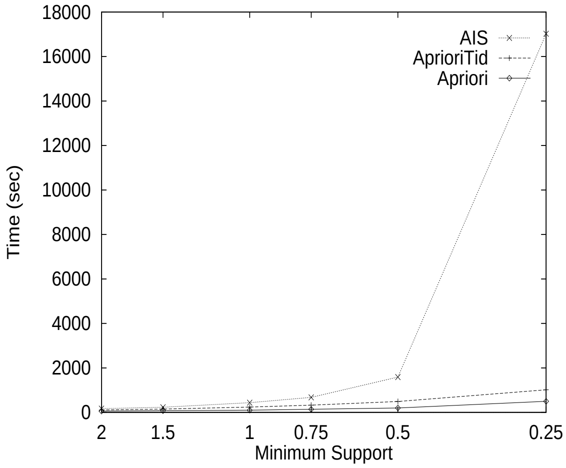

Figure 10: Execution times: M.cust

4.7 Algorithm AprioriHybrid

It is not necessary to use the same algorithm in all the passes over data. Figure 11 shows the execution times for Apriori and AprioriTid for different passes over the dataset T10.I4.D100K. In the earlier passes, Apriori does better than AprioriTid. However, AprioriTid beats Apriori in later passes. We observed similar relative behavior for the other datasets, the reason for which is as follows. Apriori and AprioriTid use the same candidate generation procedure and therefore count the same itemsets. In the later passes, the number of candidate itemsets reduces (see the size of C_k for Apriori and AprioriTid in Figure 7). However, Apriori still examines every transaction in the database. On the other hand, rather than scanning the database, AprioriTid scans $\overline{C}_k$ for obtaining support counts, and the size of $\overline{C}_k$ has become smaller than the size of the database. When the $\overline{C}_k$ sets can fit in memory, we do not even incur the cost of writing them to disk.

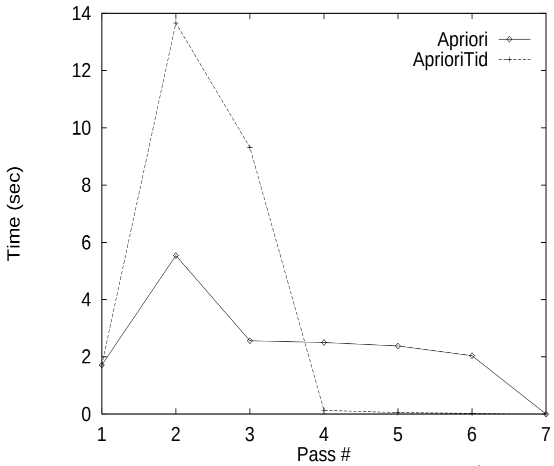

Figure 11: Per pass execution times of Apriori and AprioriTid (T10.I4.D100K, minsup = 0.75%)

Based on these observations, we can design a hybrid algorithm, which we call AprioriHybrid, that uses Apriori in the initial passes and switches to AprioriTid when it expects that the set $\overline{C}_k$ at the end of the pass will fit in memory. We use the following heuristic to estimate if $\overline{C}_k$ would fit in memory in the next pass. At the end of the current pass, we have the counts of the candidates in C_k. From this, we estimate what the size of $\overline{C}_k$ would have been if it had been generated. This size, in words, is $(\sum_{\text{candidates } c \,\in\, C_k} \text{support}(c) + \text{number of transactions})$. If $\overline{C}_k$ in this pass was small enough to fit in memory, and there were fewer large candidates in the current pass than the previous pass, we switch to AprioriTid. The latter condition is added to avoid switching when $\overline{C}_k$ in the current pass fits in memory but $\overline{C}_k$ in the next pass may not.[5]

Switching from Apriori to AprioriTid does involve a cost. Assume that we decide to switch from Apriori to AprioriTid at the end of the kth pass. In the $(k+1)$th pass, after finding the candidate itemsets contained in a transaction, we will also have to add their IDs to $\overline{C}_{k+1}$ (see the description of AprioriTid in Section 2.2). Thus there is an extra cost incurred in this pass relative to just running Apriori. It is only in the $(k+2)$th pass that we actually start running AprioriTid. Thus, if there are no large $(k+1)$-itemsets, or no $(k+2)$-candidates, we will incur the cost of switching without getting any of the savings of using AprioriTid.

Figure 12 shows the performance of AprioriHybrid relative to Apriori and AprioriTid for large datasets. AprioriHybrid performs better than Apriori in almost all cases. For T10.I2.D100K with 1.5% support, AprioriHybrid does a little worse than Apriori since the pass in which the switch occurred was the last pass; AprioriHybrid thus incurred the cost of switching without realizing the benefits. In general, the advantage of AprioriHybrid over Apriori depends on how the size of the $\overline{C}_k$ set decline in the later passes. If $\overline{C}_k$ remains large until nearly the end and then has an abrupt drop, we will not gain much by using AprioriHybrid since we can use AprioriTid only for a short period of time after the switch. This is what happened with the M.cust and T20.I6.D100K datasets. On the other hand, if there is a gradual decline in the size of $\overline{C}_k$, AprioriTid can be used for a while after the switch, and a significant improvement can be obtained in the execution time.

4.8 Scale-up Experiment

Figure 13 shows how AprioriHybrid scales up as the number of transactions is increased from 100,000 to 10 million transactions. We used the combinations (T5.I2), (T10.I4), and (T20.I6) for the average sizes of transactions and itemsets respectively. All other parameters were the same as for the data in Table 3. The sizes of these datasets for 10 million transactions were 239MB, 439MB and 838MB respectively. The minimum support level was set to 0.75%. The execution times are normalized with respect to the times for the 100,000 transaction datasets in the first graph and

[5] Other heuristics are also possible. For example, in a system with multiple disks, it may be faster to switch to AprioriTid as soon as the size of $\overline{C}_k$ is less than the size of the database.

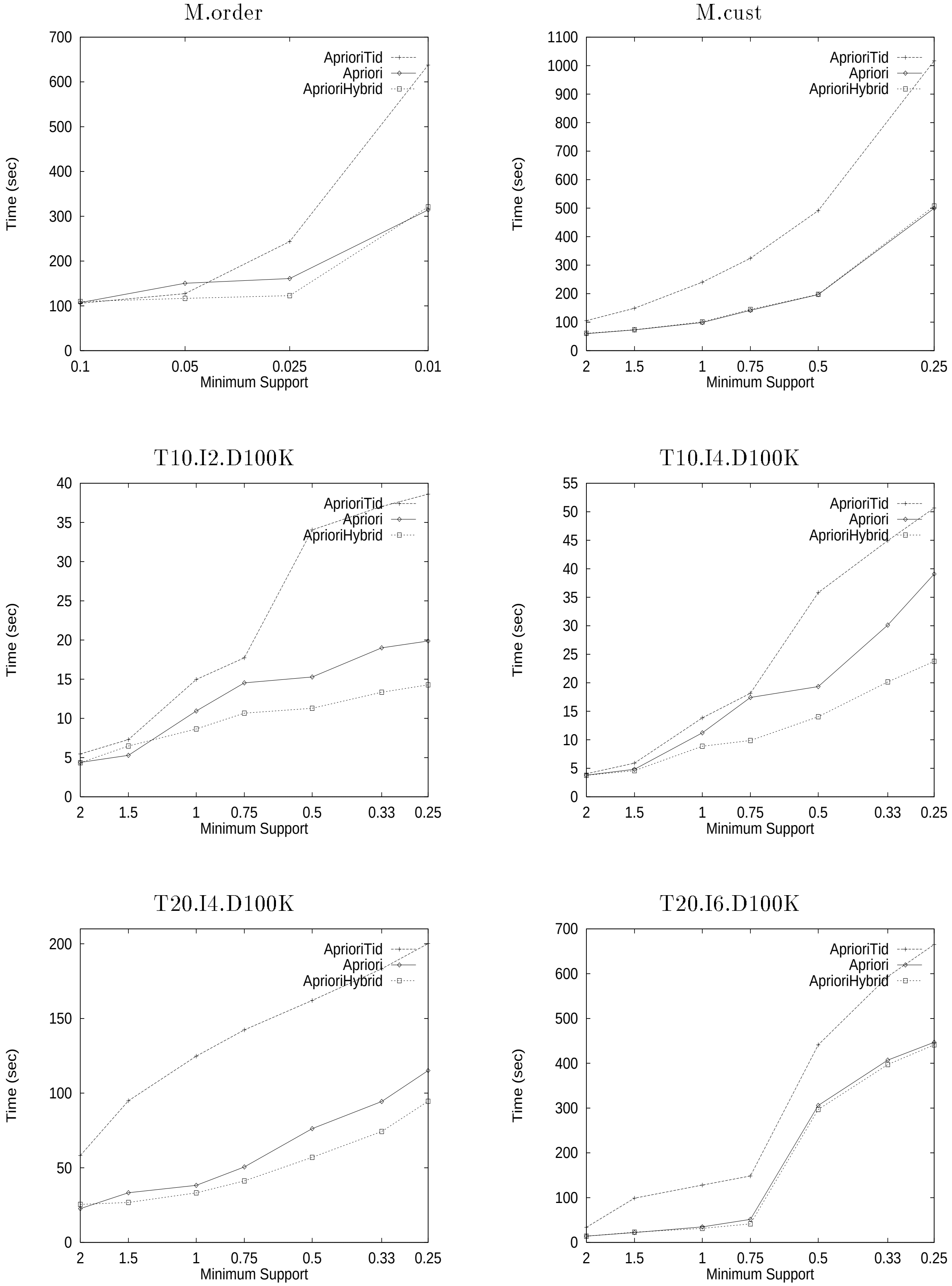

Figure 12: Execution times: AprioriHybrid Algorithm

with respect to the 1 million transaction dataset in the second. As shown, the execution times scale quite linearly.

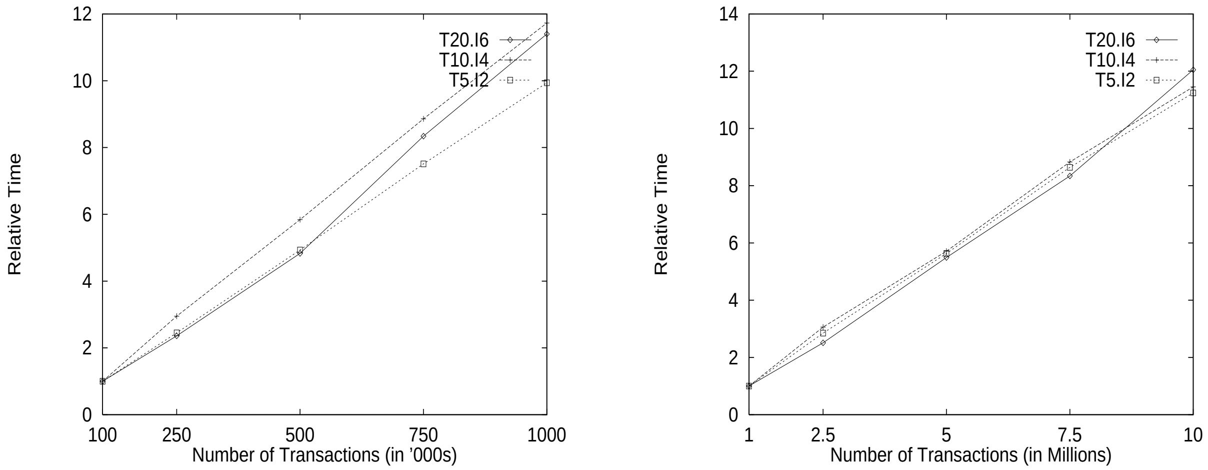

Figure 13: Number of transactions scale-up

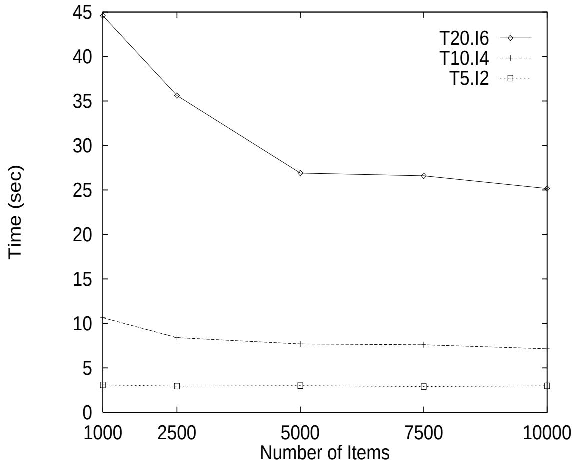

Figure 14: Number of items scale-up

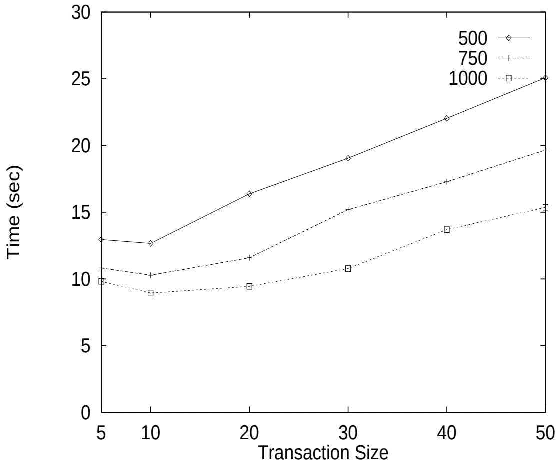

Figure 15: Transaction size scale-up

Next, we examined how AprioriHybrid scaled up with the number of items. We increased the number of items from 1000 to 10,000 for the three parameter settings T5.I2.D100K, T10.I4.D100K and T20.I6.D100K. All other parameters were the same as for the data in Table 3. We ran experiments for a minimum support at 0.75%, and obtained the results shown in Figure 14. The execution times decreased a little since the average support for an item decreased as we increased the number of items. This resulted in fewer large itemsets and, hence, faster execution times.

Finally, we investigated the scale-up as we increased the average transaction size. The aim of this experiment was to see how our data structures scaled with the transaction size, independent of other factors like the physical database size and the number of large itemsets. We kept the physical size of the database roughly constant by keeping the product of the average transaction size and the number of transactions constant. The number of transactions ranged from 200,000 for the database with an average transaction size of 5 to 20,000 for the database with an average transaction size 50. Fixing the minimum support as a percentage would have led to large increases in the number of large itemsets as the transaction size increased, since the probability of a itemset being present in a transaction is roughly proportional to the transaction size. We therefore fixed the minimum support level in terms of the number of transactions. The results are shown in Figure 15. The numbers in the key (e.g. 500) refer to this minimum support. As shown, the execution times increase with the transaction size, but only gradually. The main reason for the increase was that in spite of setting the minimum support in terms of the number of transactions, the number of large itemsets increased with increasing transaction length. A secondary reason was that finding the candidates present in a transaction took a little more time.

5 Conclusions and Future Work

We presented two new algorithms, Apriori and AprioriTid, for discovering all significant association rules between items in a large database of transactions. We compared these algorithms to the previously known algorithms, the AIS [AIS93b] and SETM [HS93] algorithms. We presented experimental results, using both synthetic and real-life data, showing that the proposed algorithms always outperform AIS and SETM. The performance gap increased with the problem size, and ranged from a factor of three for small problems to more than an order of magnitude for large problems.

We showed how the best features of the two proposed algorithms can be combined into a hybrid algorithm, called AprioriHybrid, which then becomes the algorithm of choice for this problem. Scale-up experiments showed that AprioriHybrid scales linearly with the number of transactions. In addition, the execution time decreases a little as the number of items in the database increases. As the average transaction size increases (while keeping the database size constant), the execution time increases only gradually. These experiments demonstrate the feasibility of using AprioriHybrid in real applications involving very large databases.

The algorithms presented in this paper have been implemented on several data repositories, including the AIX file system, DB2/MVS, and DB2/6000. In the future, we plan to extend this work along the following dimensions:

- Multiple taxonomies (*is-a* hierarchies) over items are often available. An example of such a hierarchy is that a dish washer *is a* kitchen appliance *is a* heavy electric appliance, etc. We would like to be able to find association rules that use such hierarchies.

- We did not consider the quantities of the items bought in a transaction, which are useful for some applications. Finding such rules needs further work.

The work reported in this paper has been done in the context of the Quest project at the IBM Almaden Research Center. In Quest, we are exploring the various aspects of the database mining problem. Besides the problem of discovering association rules, some other problems that we have looked into include the enhancement of the database capability with classification queries [AGI+92] and similarity queries over time sequences [AFS93]. We believe that database mining is an important new application area for databases, combining commercial interest with intriguing research questions.

Acknowledgment We wish to thank Mike Carey for his insightful comments and suggestions.

References

[ABN92] Tarek M. Anwar, Howard W. Beck, and Shamkant B. Navathe. Knowledge mining by imprecise querying: A classification-based approach. In *IEEE 8th Int'l Conf. on Data Engineering*, Phoenix, Arizona, February 1992.

[AFS93] Rakesh Agrawal, Christos Faloutsos, and Arun Swami. Efficient similarity search in sequence databases. In *Proc. of the Fourth International Conference on Foundations of Data Organization and Algorithms*, Chicago, October 1993. Also in Lecture Notes in Computer Science 730, Springer Verlag, 1993, 69–84.

[AGI+92] Rakesh Agrawal, Sakti Ghosh, Tomasz Imielinski, Bala Iyer, and Arun Swami. An interval classifier for database mining applications. In *Proc. of the VLDB Conference*, pages 560–573, Vancouver, British Columbia, Canada, August 1992.

[AIS93a] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, December 1993. Special Issue on Learning and Discovery in Knowledge-Based Databases.

[AIS93b] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 207–216, Washington, D.C., May 1993.

[ANB92] Tarek M. Anwar, Shamkant B. Navathe, and Howard W. Beck. Knowledge mining in databases: A unified approach through conceptual clustering. Technical report, Georgia Institute of Technology, May 1992.

[Ass90] David Shepard Associates. The new direct marketing. Business One Irwin, Illinois, 1990.

[Ass92] Direct Marketing Association. Managing database marketing technology for success, 1992.

[B+93] R.J. Brachman et al. Integrated support for data archeology. In *AAAI-93 Workshop on Knowledge Discovery in Databases*, July 1993.

[BFOS84] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Wadsworth, Belmont, 1984.

[Bit92] D. Bitton. Bridging the gap between database theory and practice, 1992.

[C+88] P. Cheeseman et al. AutoClass: A Bayesian classification system. In *5th Int'l Conf. on Machine Learning.* Morgan Kaufman, June 1988.

[Cat91] J. Catlett. Megainduction: A test flight. In *8th Int'l Conf. on Machine Learning*, June 1991.

[CH92] G. Cooper and E. Herskovits. A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 1992.

[Fis87] Douglas H. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2(2), 1987.

[FWD93] Usama Fayyad, Nicholas Weir, and S.G. Djorgovski. Skicat: A machine learning system for automated cataloging of large scale sky surveys. In *10th Int'l Conf. on Machine Learning*, June 1993.

[HCC92] Jiawei Han, Yandong Cai, and Nick Cercone. Knowledge discovery in databases: An attribute oriented approach. In *Proc. of the VLDB Conference*, pages 547–559, Vancouver, British Columbia, Canada, 1992.

[HS93] Maurice Houtsma and Arun Swami. Set-oriented mining of association rules. Research Report RJ 9567, IBM Almaden Research Center, San Jose, California, October 1993.

[HS94] M. Holsheimer and A. Siebes. Data mining: The search for knowledge in databases. Technical Report CS-R9406, CWI, Netherlands, 1994.

[KI91] Ravi Krishnamurthy and Tomasz Imielinski. Practitioner problems in need of database research: Research directions in knowledge discovery. *SIGMOD RECORD*, 20(3):76–78, September 1991.

[LSBZ87] P. Langley, H. Simon, G. Bradshaw, and J. Zytkow. *Scientific Discovery: Computational Explorations of the Creative Process.* MIT Press, 1987.

[Lub89] David J. Lubinsky. Discovery from databases: A review of AI and statistical techniques. In *IJCAI-89 Workshop on Knowledge Discovery in Databases*, pages 204–218, Detroit, August 1989.

[MF92] S. Muggleton and C. Feng. Efficient induction of logic programs. In Steve Muggleton, editor, *Inductive Logic Programming.* Academic Press, 1992.

[MKKR92] R.S. Michalski, L. Kerschberg, K.A. Kaufman, and J.S. Ribeiro. Mining for knowledge in databases: The INLEN architecture, initial implementation, and first results. *Journal of Intelligent Information Systems*, 1:85–113, 1992.

[MR87] Heikki Mannila and Kari-Jouku Raiha. Dependency inference. In *Proc. of the VLDB Conference*, pages 155–158, Brighton, England, 1987.

[MTV94] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Efficient algorithms for discovering association rules. In *KDD-94: AAAI Workshop on Knowledge Discovery in Databases*, July 1994.

[Pea92] J. Pearl. Probabilistic reasoning in intelligent systems: Networks of plausible inference, 1992.

[PS91a] G. Piatestsky-Shapiro. Discovery, analysis, and presentation of strong rules. In G. Piatestsky-Shapiro, editor, *Knowledge Discovery in Databases.* AAAI/MIT Press, 1991.

[PS91b] G. Piatestsky-Shapiro, editor. *Knowledge Discovery in Databases.* AAAI/MIT Press, 1991.

[Qui90] J. Ross Quinlan. Learning logical definitions from examples. *Machine Learning*, 5(3), 1990.

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufman, 1993.

[S+93] M. Stonebraker et al. The DBMS research at crossroads. In *Proc. of the VLDB Conference*, Dublin, August 1993.

[Sch90] C. Schaffer. *Domain-Independent Function Finding.* PhD thesis, Rutgers University, 1990.

[Tsu90] S. Tsur. Data dredging. *IEEE Data Engineering Bulletin*, 13(4):58–63, December 1990.