Fast Algorithms For Mining Association Rules In Datamining

P. Usharani

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. Empirical evaluation shows 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
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. 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 [6].

The problem of mining association rules over basket data was introduced in [4]. An example of such a rule might be that 96% of customers that purchase Visiting from the Department of Computer Science, University of Wisconsin, Madison. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment. 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. The following is a formal statement of the problem [4]: Let \( I = \{ i_1, i_2, \ldots \}; \) be a set of literals, called items. Let \( D \) be a set of transactions, where each transaction \( T \) is a set of items such that \( T \subseteq I \). Associated with each transaction \( T \) is a unique identifier, called its TID. We say that a transaction \( T \) contains \( X \), a set of some items in \( I \), if \( X \subseteq T \). An association rule is an implication of the form

\[ X \rightarrow Y, \]  
where \( X \subseteq I, Y \subseteq I \), and \( X \setminus Y = \emptyset \). The rule \( X \rightarrow Y \) holds in the transaction set \( D \) with confidence \( c \) if \( c \% \) of transactions in \( D \) that contain

\[ X \]  
also contain \( Y \). The rule \( X \rightarrow Y \) has support \( s \)
in the transaction set \( D \) if \( s \% \) of transactions in \( D \) contain \( X \). Our rules are somewhat more general than in [4] in that we allow a consequent to have more than one item. Given a set of transactions \( D \), 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 \( D \). For example, \( D \) could be a data le, 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 [4]. Another algorithm for this task, called the SETM algorithm, has been proposed in [13]. In this paper, we present two new algorithms, Apriori and AprioriTid, that differ fundamentally from these algorithms. We present experimental results 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 Apriori-Hybrid 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 [3] [12], also called knowledge discovery in databases [21]. Related, but not directly applicable, work includes the induction of classification rules [8] [11] [22], discovery of causal rules [19], learning of logical definitions [18], mining of functions to data [15], and clustering [9] [10]. The closest work in the machine learning literature is the KDD algorithm presented in [20]. 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 [16]. Functional dependencies are rules requiring strict satisfaction. Consequently, having determined a dependency \( X \rightarrow A \), the algorithms in [16] consider any other dependency of the form \( X \rightarrow Y \) 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 \( X \rightarrow Y \) 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 [20]. 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 [7] [14]. 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 [4]:

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 sup-port 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. Here is a straightforward algorithm for this task. For every large itemset \( L \), nd all non-empty subsets of \( L \). For every such subset \( a \), output a rule of the form \( \{a\} \Rightarrow \{\} \). If the ratio of support(\( L \)) to support(\( a \)) is at least minconf. We need to consider all subsets of \( L \) to generate rules with multiple consequents. Due to lack of space, we do not discuss this subproblem further, but refer the reader to [5] for a fast algorithm.

In Section 3, we show the relative performance of the proposed Apriori and AprioriTid algorithms against the AIS [4] and SETM [13] 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 4.

2 Discovering Large Itemsets

Algorithms for discovering large itemsets make multiple passes over the data. In the rst 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 sup-port 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 con-tinues until no new large itemsets are found. The Apriori and AprioriTid algorithms we propose differ fundamentally from the AIS [4] and SETM [13] 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, 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 \( k \) 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 rst 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 \( D \) is kept normalized and each database record is a \( < \text{TID}, \text{item} > \) pair, where \( \text{TID} \) is the identi er 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 \( c[1] c[2] \ldots c[k] \) 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 \( c = X Y \) and \( Y \) is an \( m \)-itemset, we also call \( Y \) an \( m \)-extension of \( X \). Associated with each itemset is a count eld to store the support for this itemset. The count eld is initialized to zero when the itemset is rst created. We summarize in Table 1 the notation used in the algorithms. The set \( C^0 \) 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 rst 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 \( L^k \) 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
given transaction t. Section 2.1.2 describes the subset function used for this purpose. See [5] for a discussion of buffer management.

1) \( L^1 = \) large 1-itemsets

2) for ( \( k = 2 \); \( L^{k-1} \) = \( \ldots \); \( k = + \) ) do begin

3) \( C^k = \) apriori-gen(\( L^{k-1} \); // New candidates

4) for all transactions t 2 D do begin

5) \( C_t = \) subset\( (C^k, t) \); // Candidates contained in t

6) for all candidates c 2 \( C_t \) do

7) c:count++;

8) end

9) \( L^k = fc \ 2 \ C^k \) j c:count \( \geq \) \( \minsup\)

10) end

11) 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 \( L^k \) with \( L^{k-1} \):

\[
\text{insert into } C^k
\]

select \( p \cdot \text{item}^1, p \cdot \text{item}^2, \ldots, p \cdot \text{item}^{k-1}, q \cdot \text{item}^{k-1} \) from \( L^k \) p, \( L^{k-1} \) q

where \( p \cdot \text{item}^1 = q \cdot \text{item}^1, \ldots, p \cdot \text{item}^{k-1} = q \cdot \text{item}^{k-1} \); \( p \cdot \text{item}^k < q \cdot \text{item}^k \);

Next, in the prune step, we delete all itemsets \( c 2 C^k \) such that some \((k, 1)\)-subset of \( c \) is not in \( L^{k-1} \):

\(^1\)Concurrent to our work, the following two-step candidate generation procedure has been proposed in [17]:

\[
C^0_k = fX [ \{q\}X; X^0 2 L_k,1]; \ jX \ \setminus \ X^0 = k , 2g
\]

\[
C_k = fX 2 C^0_k jX \text{contains } k \text{ members of } L_k,1g
\]

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.

forall itemsets \( c 2 C^k \) do for all \((k,1)\)-subsets \( s \) of \( c \)

if \( (s \ 62L_k,1) \) then delete \( c \) from \( C^k \);

Example Let \( L3 \) be \( f11 \ 2 \ 3g, f1 2 \ 4g, f13 \ 4g, f13 \ 5g, f23 \ 4g, f23 \ 5g \). After the join step, \( C4 \) will be \( f11 \ 2 \ 3 \ 4g, f11 \ 3 \ 4 \ 5g \). The prune step will delete the itemset \( f13 \ 4 \ 5g \) because the itemset \( f1 \ 4 \ 5g \) is not in \( L3 \). We will then be left with only \( f11 \ 2 \ 3 \ 4g \) in \( C4 \). Contrast this candidate generation with the one used in the AIS and SETM algorithms. In pass \( k \) of these algorithms, 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 \( f11 \ 2 \ 3 \ 4 \ 5g \). In the fourth pass, AIS and SETM will generate two candidates, \( f11 \ 2 \ 3 \ 4g \) and \( f12 \ 3 \ 5g \), by extending the large itemset \( f1 \ 2 \ 3 \ 4g \). Similarly, an additional three candidate itemsets will be generated by extending the other large itemsets in \( L3 \), leading to a total of \( 5 \) candidates for consideration in the fourth pass. Apriori, on the other hand, generates and counts only one itemset, \( f11 \ 3 \ 4 \ 5g \), because it concludes a priori that the other combinations cannot possibly have minimum support. Correctness We need to show that \( Ck \ L_k \). 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 \( p \cdot \text{item}^{k-1} < q \cdot \text{item}^{k-1} \) simply ensures that no duplicates are generated. Thus, after the join step, \( C^k \ L_k \). 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.
cost of scanning the database.

### 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 needed 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 $d$th 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 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 root 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 $i$.

### 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 $D$ is not used for counting support after the pass begins. Rather, the set $C^k$ is used for this purpose. Each member of the set $C^k$ is of the form $<\text{TID}; fXkg>$, where each $X^k$ is a potentially large $k$-itemset present in the transaction with identi er TID. For $k = 1$, $C_1$ corresponds to the database $D$, although conceptually each item $i$ is replaced by the itemset $f_i$. For $k > 1$, $C^k$ is generated by the algorithm (step 10). The member of $C_k$ corresponding to transaction $t$ is $<t:TID, fc 2 C^j, cg contained in tg>$. If a transaction $t$ does we generate $C_4$ using $L_3$, it turns out to be empty, and we terminate. Not contain any candidate $k$-itemset, then $C^k$ will not have an entry for this transaction. Thus, the number of entries in $C^k$ 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. In Section 2.2.1, we give the data structures used to implement the algorithm. See [5] for a proof of correctness and a discussion of buffer management.

### Database

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
<th>TID</th>
<th>Set-of-Itemsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1 3 4</td>
<td>100</td>
<td>f11g, f3g, f4g g</td>
</tr>
<tr>
<td>200</td>
<td>2 3 5</td>
<td>200</td>
<td>f2g, f3g, f5g g</td>
</tr>
<tr>
<td>300</td>
<td>1 2 3 5</td>
<td>300</td>
<td>f11g, f2g, f3g, f5g g</td>
</tr>
<tr>
<td>400</td>
<td>2 5</td>
<td>400</td>
<td>f2g, f5g g</td>
</tr>
</tbody>
</table>

### Itemset Support

<table>
<thead>
<tr>
<th>Itemset</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1g</td>
<td>2</td>
</tr>
<tr>
<td>f2g</td>
<td>3</td>
</tr>
<tr>
<td>f3g</td>
<td>3</td>
</tr>
<tr>
<td>f5g</td>
<td>3</td>
</tr>
</tbody>
</table>

### Set of Itemsets

<table>
<thead>
<tr>
<th>TID</th>
<th>Set-of-Itemsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>f11g, f3g, f4g g</td>
</tr>
<tr>
<td>200</td>
<td>f2g, f3g, f5g g</td>
</tr>
<tr>
<td>300</td>
<td>f11g, f2g, f3g, f5g g</td>
</tr>
<tr>
<td>400</td>
<td>f2g, f5g g</td>
</tr>
</tbody>
</table>

### Count C1

<table>
<thead>
<tr>
<th>Itemset</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1g</td>
<td>2</td>
</tr>
<tr>
<td>f2g</td>
<td>3</td>
</tr>
<tr>
<td>f3g</td>
<td>3</td>
</tr>
</tbody>
</table>

### Count C2

<table>
<thead>
<tr>
<th>Itemset</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1g</td>
<td>2</td>
</tr>
<tr>
<td>f2g</td>
<td>3</td>
</tr>
<tr>
<td>f3g</td>
<td>2</td>
</tr>
</tbody>
</table>

### Count L1

<table>
<thead>
<tr>
<th>TID</th>
<th>Set-of-Itemsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>f11g, f3g, f4g g</td>
</tr>
<tr>
<td>200</td>
<td>f2g, f3g, f5g g</td>
</tr>
<tr>
<td>300</td>
<td>f11g, f2g, f3g, f5g g</td>
</tr>
<tr>
<td>400</td>
<td>f2g, f5g g</td>
</tr>
</tbody>
</table>

### Count L2

<table>
<thead>
<tr>
<th>TID</th>
<th>Set-of-Itemsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>f11g, f3g, f4g g</td>
</tr>
<tr>
<td>200</td>
<td>f2g, f3g, f5g g</td>
</tr>
<tr>
<td>300</td>
<td>f11g, f2g, f3g, f5g g</td>
</tr>
<tr>
<td>400</td>
<td>f2g, f5g g</td>
</tr>
</tbody>
</table>

1) $L_1^c$ = large 1-itemsets;
2) $C^1$ = database $D$;
3) for $(k = 2;) L^{k-1} = 6 ;k++)$ do begin
4) $C^k = \text{apriori-gen}(L^{k-1});$ // New candidates
5) $C_k = \text{;};$
6) forall entries $t 2 C^{k-1}$ do begin
7) // determine candidate itemsets in $C^k$ contained in the transaction with identi er $TID$
8) forall candidates $c 2 C^k$ do
9) c:count++;
10) if $(C^k = ;)$ then $C^k += <t:TID; C^i>$;
11) end
12) $L^k = fc 2 C^k j c:count \text{ minsuppg}$
13) end
14) Answer = $\text{Sk} L^k$;

IJSRT©2013
WWW.IJSTR.ORG
Example Consider the database in Figure 3 and assume that minimum support is 2 transactions. Calling apriori-gen with L1 at step 4 gives the candidate itemsets C2. In steps 6 through 10, we count the support of candidates in C2 by iterating over the entries in C1 and generate C2. The rest entry in C1 is f1 f1 g f3 g f4 g, corresponding to transaction 100. The C1 at step 7 corresponding to this entry t is f1 f3 g, because f1 3 g is a member of C2 and both (f1 3 g - f1 g) and (f1 3 g - f3 g) are members of t.set-of-items. Calling apriori-gen with L2 gives C3. Making a pass over the data with C2 and C3 generates C3. Note that there is no entry in C3 for the transactions with TIDs 100 and 400, since they do not contain any of the itemsets in C3. The candidate f2 3 5 g in C3 turns out to be large and is the only member of L3. When

\[
\begin{align*}
C3 & \quad \text{Set-of-Itemsets} \\
TID & \quad \text{f1 f2 3 5 g} \\
200 & \quad \text{2} \\
300 & \quad \text{2}
\end{align*}
\]

\[
\begin{align*}
\text{Itemsets} & \quad \text{Support} \\
f2 3 5 g & \quad \text{2}
\end{align*}
\]

Figure 2: Algorithm AprioriTid

3 Performance

To assess the relative performance of the algorithms for discovering large sets, 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 le system and was stored on a 2GB SCSI 3.5" drive, with measured sequential throughput of about 2 MB/second. We rst give an overview of the AIS [4] and SETM [13] 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. 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.

3.1 The AIS Algorithm

Candidate itemsets are generated and counted on-the- y 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. New candidate itemsets are generated by extending these large itemsets with other items in the transaction. A large itemset I is extended with only those items that are large and occur later in the lexicographic ordering of items than any of the items in I. 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. See [4] for further details of the AIS algorithm.

3.2 The SETM Algorithm

The SETM algorithm [13] was motivated by the desire to use SQL to compute large itemsets. Like AIS, the SETM algorithm also generates candidates on-the- y 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. At the end of the pass, the support count of candidate itemsets is determined by sorting and aggregating this sequential structure. 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. Lk C and is obtained by deleting those candidates that do not have minimum support. Assuming that the database is sorted in TID order, SETM can easily nd the large itemsets contained in a transaction in the next pass by sorting Lk on TID. In fact, it needs to visit every member of Lk only once in the TID order, and the candidate generation can be performed using the relational merge-join operation [13]. The disadvantage of this approach is mainly due to the size of candidate sets Ck. 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

Figure 3: Example

2.2.1 Data Structures

We assign each candidate itemset a unique number, called its ID. Each set of candidate itemsets Ck is kept in an array indexed by the IDs of the itemsets in Ck. A member of Ck is now of the form < TID; f1Dg>. Each Ck 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 elds for each candidate itemset: i) generators and ii) extensions. The generators eld of a candidate itemset c{k} stores the IDs of the two large (k - 1)-itemsets whose join generated ck. The extensions eld of an itemset c{k} stores the IDs of all the (k + 1)-candidiates that are extensions of c{k}. Thus, when a candidate c{k} is generated by joining Ik1 and Ik2, we save the IDs of Ik1 and Ik2 in the generators eld for c{k}. At the same time, the ID of c{k} is added to the extensions eld of Ik1. We now describe how Step 7 of Figure 2 is implemented using the above data structures. Recall that the t.set-of-itemsets eld of an entry t in Ck - 1 gives the IDs of all (k - 1)-candidates contained in transaction t.TID. For each such candidate c{k} - 1 the extensions eld 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 eld 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 ck to Ct.
for candidate itemsets at the end of the pass, \( C^k \) is in the wrong order and needs to be sorted on itemsets. After counting and pruning out small candidate itemsets that do not have minimum support, the resulting set \( L^k \) needs another sort on TID before it can be used for generating candidates in the next pass.

### 3.3 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 rugs. 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**

<table>
<thead>
<tr>
<th>( T )</th>
<th>Number of transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T )</td>
<td>Average size of the transactions</td>
</tr>
<tr>
<td>( L_f )</td>
<td>Average size of the maximal potentially large itemsets</td>
</tr>
<tr>
<td>( L_j )</td>
<td>Number of maximal potentially large itemsets</td>
</tr>
<tr>
<td>( N )</td>
<td>Number of items</td>
</tr>
</tbody>
</table>

We first determine the size of the next transaction. The size is picked from a Poisson distribution with mean 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 \( t \) 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 \( T \) of such itemsets. The number of itemsets in \( T \) is set to \( |L_j| \). There is an inverse relationship between \( |L_j| \) and the average support for potentially large itemsets. An itemset in \( T \) is generated by rst picking the size of the itemset from a Poisson distribution with mean equal to \( |L_j| \). Items in the rst 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 nd much difference in the nature of our performance results. Each itemset in \( T \) 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 \( T \) is 1. The next itemset to be put in the transaction is chosen from \( T \) 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 \( T \) 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, \( c \) of the time, \( l \) is added 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 \( |L| \); 2, 4, and 6. The number of transactions was set to 100,000 because, as we will see in Section 3.4, 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 \( |L| \) values, the size of datasets in megabytes were roughly equal for the different values of \( |L| \).

**Table 3: Parameter settings**

<table>
<thead>
<tr>
<th>Name</th>
<th>( T )</th>
<th>( L )</th>
<th>( D )</th>
<th>Size in Megabytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>T5.I2.D100K</td>
<td>5</td>
<td>2</td>
<td>100K</td>
<td>2.4</td>
</tr>
<tr>
<td>T10.I2.D100K</td>
<td>10</td>
<td>2</td>
<td>100K</td>
<td>4.4</td>
</tr>
<tr>
<td>T10.I4.D100K</td>
<td>10</td>
<td>4</td>
<td>100K</td>
<td>6.4</td>
</tr>
<tr>
<td>T20.I2.D100K</td>
<td>20</td>
<td>2</td>
<td>100K</td>
<td>8.4</td>
</tr>
<tr>
<td>T20.I4.D100K</td>
<td>20</td>
<td>4</td>
<td>100K</td>
<td></td>
</tr>
<tr>
<td>T20.I6.D100K</td>
<td>20</td>
<td>6</td>
<td>100K</td>
<td></td>
</tr>
</tbody>
</table>

### 3.4 Relative Performance

Figure 4 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 4. 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.
Figure 4: Execution times
The fundamental problem with the SETM algorithm is the size of its \( C^k \) sets. Recall that the size of the set \( C^k \) is given by

\[ \text{support-count}(c): \text{candidate itemsets} \]

Thus, the sets \( C^k \) 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 \( C^k \) sets have to be written to disk, and externally sorted twice, causing the SETM algorithm to perform poorly. 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 [13] was still small enough that \( C^k \) 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 \( |j| \), though the size of \( C^k \) does not change, the size of \( C^k \) goes up linearly. Thus, for datasets with more transactions, the performance gap between SETM and the other 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. AprioriTid also performs significantly fewer candidates than the transaction-based candidate generation used by SETM. As a result, the \( C^k \) 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. In addition, unlike SETM, AprioriTid does not have to sort \( C^k \). Thus, AprioriTid does not suffer as much as SETM from maintaining \( C^k \). AprioriTid has the nice feature that it replaces a pass over the original dataset by a pass over the set \( C^k \). Hence, AprioriTid is very effective in later passes when the size of \( C^k \) becomes small compared to the \( \text{cost of external sorting in SETM can be reduced somewhat as follows. Before writing out entries in } C_k \text{ to disk, we can sort them on itemsets using an internal sorting procedure, and write them as sorted runs. These sorted runs can then be merged to obtain support counts. However, given the poor performance of SETM, we do not expect this optimization to affect the algorithm choice.} \)

Thus, we need that AprioriTid beats Apriori when its \( C^k \) sets can fit in memory and the distribution of the large itemsets has a long tail. When \( C^k \) 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 4. In this region, Apriori starts beating AprioriTid.

### 3.5 Explanation of the Relative Performance

To explain these performance trends, we show in Figure 5 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 5: Sizes of the large and candidate sets (T10.I4.D100K, minsup = 0.75%)](image)

The fundamental problem with the SETM algorithm is the size of its \( C^k \) sets. Recall that the size of the set \( C^k \) is given by

\[ \text{support-count}(c): \text{candidate itemsets} \]

Thus, the sets \( C^k \) 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 \( C^k \) sets have to be written to disk, and externally sorted twice, causing the SETM algorithm to perform poorly. 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 [13] was still small enough that \( C^k \) 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 \( |j| \), though the size of \( C^k \) does not change, the size of \( C^k \) goes up linearly. Thus, for datasets with more transactions, the performance gap between SETM and the other 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. AprioriTid also performs significantly fewer candidates than the transaction-based candidate generation used by SETM. As a result, the \( C^k \) 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. In addition, unlike SETM, AprioriTid does not have to sort \( C^k \). Thus, AprioriTid does not suffer as much as SETM from maintaining \( C^k \). AprioriTid has the nice feature that it replaces a pass over the original dataset by a pass over the set \( C^k \). Hence, AprioriTid is very effective in later passes when the size of \( C^k \) becomes small compared to the \( \text{cost of external sorting in SETM can be reduced somewhat as follows. Before writing out entries in } C_k \text{ to disk, we can sort them on itemsets using an internal sorting procedure, and write them as sorted runs. These sorted runs can then be merged to obtain support counts. However, given the poor performance of SETM, we do not expect this optimization to affect the algorithm choice.} \)

Thus, we need that AprioriTid beats Apriori when its \( C^k \) sets can fit in memory and the distribution of the large itemsets has a long tail. When \( C^k \) 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 4. In this region, Apriori starts beating AprioriTid.
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 5). However, Apriori still examines every transaction in the database. On the other hand, rather than scanning the database, AprioriTid scans $C^k$ for obtaining support counts, and the size of $C^k$ has become smaller than the size of the database. When the $C^k$ sets can fit in memory, we do not even incur the cost of writing them to disk.

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 $C^k$ at the end of the pass will fit in memory. We use the following heuristic to estimate if $C^k$ 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 $C^k$ would have been if it had been generated. This size, in words, is $(P_{candidates} \times 2^{C_k \text{support}}} + \text{number of transactions})$. If $C^k$ 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 $C^k$ in the current pass fits in memory but $C^k$ in the next pass may not. Switching from Apriori to AprioriTid does involve a cost. Assume that we decide to switch from Apriori to AprioriTid at the end of the $k$th pass. In the $(k + 1)$th pass, after reading the candidate itemsets contained in a transaction, we will also have to add their IDs to $C^{k+1}$ (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 still incur the cost of switching without getting any of the savings of using AprioriTid. Figure 7 shows the performance of AprioriHybrid relative to Apriori and AprioriTid for three 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 $C^k$ set declines in the later passes. $C^k$ 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 T20.I6.D100K dataset. On the other hand, if there is a gradual decline in the size of $C^k$, AprioriTid can be used for a while after the switch, and a significant improvement can be obtained in the execution time.

3.7 Scale-up Experiment

Figure 8 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 rest.
Figure 7: Execution times: AprioriHybrid

Figure 8: Number of transactions 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 T5.12.D100K, T10.14.D100K, and T20.16.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 9. 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...
4 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 [4] and SETM [13] algorithms. We presented experimental results, 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. We have also tested these algorithms against real customer data, the details of which can be found in [5]. 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, 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 [2] and similarity queries over time sequences [1]. 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


