Abstract: Discovery of association rules is an important problem in Data Mining. The classical approach is to generate all itemsets that have support (i.e., the fraction of transactions containing the itemset) above a user given threshold. Most existing algorithms aim at reducing the number of scans over the transaction database, i.e., the I/O overhead. We consider the problem of how to calculate efficiently the support, i.e., we try to optimize both I/O and CPU time. A straightforward way is to maintain, for each itemset, the relevant transaction identifiers directly into a list and use a sort-merge algorithm to do the intersection of two itemsets. Instead, we propose bitmap based algorithms. The basic idea is that every couple <transaction - item> is represented by a bit in an index bitmap, and the logical operation AND is used in place of the sort-merge algorithm. We propose two variations of the bitmap based algorithm: the naïve bitmap algorithm (N-BM) and the hierarchical bitmap algorithm (H-BM). We then compare these two novel algorithms with the classical list based algorithm. Our experimental and analytical results demonstrate that the H-BM algorithm can outperform other algorithms by a factor of an order of magnitude. Furthermore, it is less memory demanding and can efficiently exploit existing bitmap index. Finally, we sketch parallel versions of the bitmap algorithms that are very efficient for VLDBs.

keywords: data mining, association rules, bitmap

1. Introduction

Mining for association rules between items in a large database of sale transactions was first introduced in [AIS93]. Association rules describe the relationship between items which are frequently bought together. For instance, a rule “bread => cream” states that if a customer buy bread, she/he will also buy cream. Association rules are further qualified by some statistical
significance, which are detailed in the sequel. This kind of rules is useful for catalog design, store layout, promotions, etc. Association rules were also used to find regularities in other kinds of data such as student enrollment [MTV94], census data [BMU+97], and even semi-structured documents [SSC97]. A SQL-based operator to mine association rules was proposed in [MPC96].

[AS94] gave a formal description of the problem of mining association rules: Let \( I = \{i_1, i_2, \ldots, i_m\} \) 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 is a unique identifier \( TID \). A transaction \( T \) is said to contain \( X \) (a set of items in \( I \)) if \( X \subseteq T \). An association rule has the expression \( X \Rightarrow Y \), where \( X \subseteq I \), \( Y \subseteq I \), and \( X \cap Y = \emptyset \). Two statistical thresholds \textit{Support} and \textit{Confidence} are used to determine whether a rule is useful or not. For example, the rule « bread \Rightarrow cream » having support \( s \) and confidence \( c \) means \( s\% \) of transactions contain both bread and cream, and, if we buy bread, we will also buy cream with the probability \( c \). The problem of mining association rules is to find all association rules whose support and confidence are greater than or equal to user-defined minimum support \((\text{minsup})\) and minimum confidence \((\text{minconf})\).

The number of items contained in an itemset is considered as its \textit{size}. An itemset with size \( k \) is called a \( k \)-itemset. An itemset having a support greater or equal to \( \text{minsup} \) is \textit{frequent}. An itemset with more than 1 item is \textit{large}. [AIS93] decomposed the problem of mining association rules into two steps:

1. Given the minimum support \( \text{minsup} \), find all frequent itemsets;
2. Construct rules from the frequent itemsets found in the step 1. The algorithm to do this looks for every frequent itemset \( l \), finds all its non-empty subsets \( a \), and outputs the rules \( a \Rightarrow (l-a) \) if the ratio of support\((l)\) to support\((a)\) is greater than \( \text{minconf} \).

Thus step 1 becomes the focus problem. The performance of an algorithm is dependent mainly on the number of passes made over the data and the efficiency of those passes. The efficiency of a pass is furthermore determined by the number of candidates generated and by the cost of the calculation of supports. Several algorithms [AS94, SON95, Toi96, BMU+97] attempt to reduce the number of scans over the database and the number of candidates at the same time. Surprisingly, little work has been done on optimizing the evaluation of support.

This paper presents new algorithms to mine association rules based on intensive use of bitmaps to efficiently compute the support of itemsets. While we were exploring association rules with OR and NOT connectors for a specific application, we noticed the constant execution time of all logical operations at the bit level (e.g., AND, OR, NOT, etc.). Additionally, we observed that sorting the data is not required when using bitmaps. Thus, using a bit array to characterize the behavior of an itemset in a set of transactions yields the possibility of efficiently combining itemsets. The basic idea is that every couple <transaction - item> is represented by a bit in an index bitmap (i.e., bit \( i \) in the bit array encodes the presence or absence of the itemset in transaction \( TID \)). The logical operator AND is used in place of the sort-merge algorithm. Up to now only the AND relationship between itemsets has been considered. The bit array encoding makes possible the consideration of any logical operator. For example, sometimes it is possible that a rule “bread \Rightarrow cream \lor cheese” holds, while “bread \Rightarrow cream” does not. [HF95], [SA95] introduced the problem of mining generalized association rules, where a rule can contain categories. Evidently, the relationship between children of an item in the taxonomy is an OR.

\footnote{In other papers, the term \textit{large} is used instead of \textit{frequent}. We prefer \textit{frequent}, because \textit{large} can be confused with the size of itemset.}
The efficiency of the proposed algorithms derives mainly from the efficiency of bitmap manipulation in memory. Thus, we propose two variations of the bitmap based algorithm: the naïve bitmap algorithm (N-BM) and the hierarchical bitmap algorithm (H-BM). The naïve bitmap algorithm manipulates bitmaps in a direct way, while the hierarchical bitmap algorithm uses a bitmap index to the initial bitmap to avoid useless memorization and manipulation of null words. Using the [AS94] benchmark, we compare the proposed algorithms with other well-known ones. From the experimental results, our H-BM can outperform the others with a factor of an order of magnitude.

In fact the basic idea behind our approach has already been exploited in another context: certain modern relational DBMSs do implement bitmap indexes for accelerating join and aggregate computation. Thus, another nice aspect of our algorithms is that they can be implemented in conjunction with bitmap indexes, thereby avoiding the computation of the 1-item bitmaps required by our algorithms.

The rest of this paper is organized as follows: Section 2 reviews the previous work. The Naive Bitmap and the Hierarchy Bitmap algorithms are introduced in Section 3 and Section 4 respectively. Section 5 gives the analytical evaluation of these algorithms. The experimental results are presented in Section 6. Parallel versions are sketched in Section 7. Finally, Section 8 concludes this paper in pointing out some research directions for future work.

2. Previous Work

Since the first introduction of association rules [AIS93], several efficient algorithms have been proposed to mine these rules. One of the main challenge is developing efficient algorithms to handle very large databases. Most of the algorithms proposed so far tried to reduce the I/O overhead. The Apriori algorithm [AS94] performs \( k \) passes over the data to evaluate the support. The Partition algorithm [SON95] scans the database two times at most. The Sampling algorithm of [Toi96] needs only one pass, but generates more candidate itemsets. Recently, [BMU+97] proposed their Dynamic Itemset Counting (DIC) algorithm, which carries out 1.5 passes over the data, and generates less candidate itemsets than the Sampling algorithm [Toi96]. Parallel algorithms are also proposed by [AS96], [PCY95+], while [CHN+95] introduce a distributed algorithm named FDM. In the sequel, we review the Apriori algorithm [AS94] and the Partition algorithm [SON95], which are two good reference algorithms.

2.1 Apriori Algorithm

The Apriori algorithm [AS94] uses a breadth-first way to generate candidate itemsets. The basic intuition is that if an itemset is frequent, so are all its sub-itemsets. Therefore candidate k-itemsets can be generated by joining frequent (k-1)-itemsets and deleting those whose (k-1)-subsets are not frequent. For example, let frequent 3-itemsets be \{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}, \{1, 3, 5\}, \{2, 3, 4\}. After the join, candidate 4-itemsets will be \{1, 2, 3, 4\} and \{1, 3, 4, 5\}. But \{1, 3, 4, 5\} will be pruned because, for example, its sub-3-itemset \{1, 4, 5\} is not frequent. Then the algorithm will only evaluate the support of \{1, 2, 3, 4\}. Almost all existing algorithms [AS96, BMU+97, CHN+96, PCY95, PCY95+, SON95, Toi96] are based on this idea, which states that any subset of a frequent itemset has to be frequent.

To count the support, each itemset is associated with a support counter. When the data is scanned, the support counter of an itemset is incremented by one if it is contained in the current transaction. The first pass of the algorithm simply determines the frequent 1-itemsets. Then, pass \( k \) generates the candidate \( k \)-itemsets by combining two frequent (k-1)-itemsets. Candidate \( k \)-itemsets are evaluated against the database and pruned if not frequent. Thus if the size of the largest itemsets is \( k \), the transaction table is scanned \( k \) times.
2.2 Partition Algorithm

The partition algorithm [SON95] divides a database into several partitions. The partition sizes are chosen such that each partition can be accommodated in main memory. The key idea is that if one itemset is frequent, it must be frequent in at least one of these partitions. The algorithm executes in two passes. In pass 1, the algorithm finds all locally frequent itemsets from every partition. Then, these locally frequent itemsets are merged to generate a set of all potential frequent itemsets. In pass 2, the actual support for these itemsets is computed by scanning the database and globally frequent itemsets are identified. As a result, the Partition algorithm reduces the number of scans over the database to at most two. When the database can fit in main memory, one scan is sufficient. In order to count the support, a list of TIDs (tidlist) is associated to each itemset. The support of an itemset is the ratio of the cardinality of its tidlist and the total number of transactions. The tidlist of a k-itemset is generated by joining the tidlists of its parent (k-1)-itemsets instead of scanning the data. Furthermore, a parallel version can be easily implemented.

We now detail the partition algorithm in the case of a unique partition. This is the basic algorithm we start from. Similarly to the one partition algorithm, our algorithm can be parallelized using several partitions, as shown in the last section. Figure 1 shows the pseudo code of the main background algorithm. Here l.itemcode denotes a set of item codes, l.tidlist a set of TIDs and l.support the support.

```
(1)  L1 = {frequent 1-itemsets} ;
(2)  for (k=2 ; L_{k-1} ≠ ∅ ; k++) do {
(3)    forall itemsets l_1, l_2 ∈ L_{k-1} do {
(4)      if l_1 and l_2 can join then {
(5)        candidate.itemcode = l_1.itemcode ∪ l_2.itemcode ;
(6)        if (cannot prune candidate) then {
(7)          candidate.tidlist = l_1.tidlist ∩ l_2.tidlist ;
(8)          if (candidate.support > minsup) then add candidate into L_k ;
(9)        } // if cannot prune candidate
(10)   } // if can join l_1 and l_2
(11)  } // forall l_1, l_2
(12) } // for k
```

**Figure 1**: The One Partition Algorithm

In [SON95], the tidlists are maintained as arrays, and a sort-merge algorithm is used to do the intersection of tidlists. The authors also suggest that an alternate method can be based on a set of bits, but they do not pursue this idea. As we will compare our bitmap based algorithms with their list based one, the actual implementation of lists is important. Our implementation of lists is shown in Figure 2. Procedure count_support() corresponds to line 7 of Figure 1. This procedure generates the tidlist of a new itemset and calculates its support.

```
procedure count_support(itemset1, itemset2) {
(1)    i = 0 ;   // current position of itemset1.tidlist
(2)    j = 0 ;   // current position of itemset2.tidlist
(3)    while ((i < itemset1.support) and (j < itemset2.support)) do {
(4)        if (itemset1.tidlist[i] == itemset2.tidlist[j]) then {
(5)            add itemset1.tidlist[i] to new_itemset.tidlist;
(6)            new_itemset.support ++ ;
(7)            i++ ; j ++ ;
(8)        }
(9)    } else if (itemset1.tidlist[i] > itemset2.tidlist[j]) then
```

Page 4
3. Naïve Bitmap Algorithm (N-BM)

In this section, we introduce the Naïve BitMap algorithm (N-BM). The core procedure is similar to the One Partition algorithm, but a bit vector is used to determine which transactions contain a given itemset. Merging of itemsets is easily performed through logical AND operation applied to bit vectors.

3.1 Manipulating Bitmap Indexes

The key idea of the algorithm is to use a bitmap index to determine which transactions contain which itemsets. To each itemset, we associate a bit vector, hereafter called a *tidbit*. Bit $i$ is 1 in the *tidbit* if transaction $i$ contains the itemset, and 0 if not. The ordered collection of *tidbits* composes the bitmap. In a bitmap, each column corresponds to a given k-itemset, while each line characterizes a transaction. Let us point out that the bitmap for 1-itemsets is just a classical bitmap index, as implemented in Oracle for example.

As the algorithm intensively manipulates tidbits, the encoding is important. Every 16 bits forms a group. Thus a group is represented by an unsigned short. The total number of groups is the total number of transactions divided by 16. For example, if a database contains 100K transactions, 6250 (100K / 16) unsigned shorts would be needed for one itemset. The integer division of a TID by 16 gives the group number this TID belongs to, and the rest determines which bit to set in this group. For example, if the 25th transaction contains item A, the 9th (25 % 16) bit in the 1st (25 / 16) group of the tidbit of A will be set to 1. The intersection of two tidbits is accomplished by the logical operation AND.

As an example, let A and B be two items with tidlists \{3, 5, 7, 12, 25, 30\} and \{5, 11, 25\} respectively. Dividing by 16 yields that the TIDs \{3, 5, 7, 11, 12\} are in the 0th group, while \{25, 30\} belong to the 1st group. Thus their corresponding tidbits are A.tidbit: \{0th group: 0000 1000 0101 0100, 1st group: 0010 0001 0000 0000\} and B.tidbit: \{0th group: 0000 0100 0001 0000, 1st group: 0000 0001 0000 0000\}. Encoded as shorts, they become in hexadecimal A.tidbit: \{0th group: 0854, 1st group: 2100\} and B.tidbit: \{0th group: 0410, 1st group: 0100\}. The intersection of the two tidbits in hexadecimal is AB.tidbit: \{0th group: 0010, 1st group: 0100\}. That means the 5th (5+16*0) and 25th (9+16*1) transactions contain both A and B.
3.2 The Implemented Algorithm

The support of a given itemset is simply given by the number of 1 in its tidbit divided by the number of transactions (i.e., the size of the tidbit). Thus, it is important to be able to efficiently count the number of 1 in a tidbit. In order to efficiently count the number of 1 in a short, we previously store this information into an array named \textit{nbbit}, i.e., nbbit[0]=0, nbbit[1]=1, nbbit[2]=1 ...etc. Henceforth, counting the number of 1 bits in a short is performed through an indexed access to this array. To optimize the implementation, \textit{max\_group} stores the number of the greatest not null group in each tidbit. Table 1 summarizes the Data Structure. The pseudo code of the N-BM algorithm is given in Figure 3.

<table>
<thead>
<tr>
<th>itemset</th>
<th>Each itemset contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) itemcode : Array of item codes</td>
<td></td>
</tr>
<tr>
<td>(2) max_group (number of the greatest non null group in tidbit)</td>
<td></td>
</tr>
<tr>
<td>(3) tidbit : Array of shorts</td>
<td></td>
</tr>
<tr>
<td>(4) support</td>
<td></td>
</tr>
</tbody>
</table>

Table 1 : Data Structure for N-BM

procedure \textit{count\_support}(itemset1,itemset2) {
(1) new\_itemset.max\_group=min(itemset1.max\_group, itemset2.max\_group) ;
(2) for (i = 0 ; i < new\_itemset.max\_group ; i++) do {
(3) new\_itemset.tidbit[i] = itemset1.tidbit[i] & itemset2.tidbit[i] ;
(4) new\_itemset.support += nbbit[new\_itemset.tidbit[i]] ;
(5) }
}

Figure 3 : The N-BM algorithm.

4. Hierarchical Bitmap Algorithm (H-BM)

The H-BM algorithm is based on the idea of the N-BM algorithm, but a hierarchical compression scheme is applied for the bitmap index. In addition, the compression scheme is clever enough to factorize AND operations, and avoid in many cases accessing the detailed bitmap.

4.1 The Compression Scheme

Running the previous algorithm against a benchmark makes us discover that the bitmaps were often full of 0. For example, let the number of transactions be 100K and the support of an itemset be 2%, then the maximum number of non zero shorts is 2% \times 100K = 2000 (when every group contains only one non zero bit). Thus we have at least 4250 (100K/16 - 2000) zeros in the tidbit. Obviously, it is not very useful to do the AND with zeros.

In order to avoid doing these useless intersections, we construct a second bitmap index over the first bitmap. In the second bitmap, every bit indicates whether a group (short) is empty or not in the first one. Therefore we obtain two bitmaps : the first level bitmap (1-BM) represents the TIDs of an itemset, while the second level bitmap (2-BM) represents the groups (shorts) of the 1-BM. For example, let a tidlist of an itemset A be \{3, 12, 36, 55, 84, 123\}. Thus its 1-BM in hexadecimal is \{0\textsuperscript{th} group : 0804, 1\textsuperscript{st} group : 0000, 2\textsuperscript{nd} group : 0004, 3\textsuperscript{rd} group : 0040, 4\textsuperscript{th} group : 0000, 5\textsuperscript{th} group : 0008, 6\textsuperscript{th} group : 0000, 7\textsuperscript{th} group : 0400\}, and its 2-BM is \{1010 1101\} = \{AD\} (see Figure 4). When calculating the support of two itemsets, we first
do the intersection of their 2-BMs, then we consider only those non zero bits (groups not null).
Suppose the 2-BM of another itemset B in hexadecimal is {76}. Since the result of {AD} AND {76} is {24},
we only do the intersection of the 4th and 6th shorts of 1-BMs of A and B. In total, only 3 AND operations are executed, while with the N-BM, 8 AND operations (8 groups) are needed. In order to save memory, every 1-itemset has both a 1-BM and a 2-BM, while every large itemset keeps only its 2-BM. Here we do not maintain the number of the greatest non null group in 2-BM, because it is rare that an item is not contained within the last 16*8=128 consecutive transactions. There is also no need to memorize the number of the greatest non null group in the 1-BM, because only the non null groups will be visited.

Item A:

<table>
<thead>
<tr>
<th>1-BM groups:</th>
<th>0th</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
<th>5th</th>
<th>6th</th>
<th>7th</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0804</td>
<td>0000</td>
<td>0004</td>
<td>0040</td>
<td>0000</td>
<td>0008</td>
<td>0000</td>
<td>0400</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2-BM bits</th>
<th>0th</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
<th>5th</th>
<th>6th</th>
<th>7th</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 4: Constructing the 2-BM from the 1-BM

4.2 The Implemented Algorithm

For reducing the cost of the intersection of 2-BMs, we perform the AND between 2-BMs on a word basis. In order to efficiently determine which bits of the resulting 2-BM are 1, the 2-BM is scanned by groups of 8 bits during the counting phase. The H-BM algorithm is shown in Figure 5, while the data structures and notations used are described in Table 2.

<table>
<thead>
<tr>
<th>WORD_NUM</th>
<th>Total number of groups in 2-BM in term of word (word = 32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE_NUM</td>
<td>Total number of groups in 2-BM in term of byte (byte = 8 bits)</td>
</tr>
</tbody>
</table>

l-itemset
Each l-itemset has
(1) 1-BM : Array of shorts
(2) 2-BM : Array of bytes
(3) support

k-itemset
Each k-itemset has
(1) itemcode : Array of item codes
(2) 2-BM : Array of bytes
(3) support

Table 2 : Data Structure and Notations for H-BM

procedure count_support(itemset1, itemset2) do {
  (1) for (i := 0 ; i < WORD_NUM ; i++) do {
  (2) new_itemset.2-BM[i] = itemset1.2-BM[i] & itemset2.2-BM[i] ;
  (3) }
  (4) for (i := 0 ; i < BYTE_NUM ; i++) do {
  (5) posi = -1;
  (6) b = new_itemset.2-BM[i];
  (7) for (; (b != 0) ; b >>= 1) do { // shift a bit to the right until b=0
  (8) posi++;
  (9) if (b & 1) do { // if the first bit is 1

5. Analytical Evaluation

This section complements the performance measures presented in Section 6. For conciseness, we will only compare List and H-BM, since H-BM outperforms N-BM in most cases. Anyway, the performance measures will compare the three algorithms. The goal of this analytical evaluation is threefold. First, it allows to study the relative performance of algorithms List and H-BM by varying easily the correlation between items. Second, it compares the main memory consumption of List and H-BM. Main memory consumption determines the maximum number of itemsets that can be treated in parallel. Third, it shows the behavior of each algorithm when it is faced to non traditional item sets such as $\neg(A \wedge B)$. Notation and Parameters used in this section are summarized in Table 3. Using these notations, the support of a 2-item set $AB$ can be expressed as $S_{AB} = \rho_{AB} \times \min(S_A, S_B)$.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>Number of transactions</td>
</tr>
<tr>
<td>$Freq$</td>
<td>Number of frequent 1-itemsets</td>
</tr>
<tr>
<td>$Sup$</td>
<td>Average support of itemsets</td>
</tr>
<tr>
<td>$SizeShort$</td>
<td>Size of a short (16 bits)</td>
</tr>
<tr>
<td>$SizeInt$</td>
<td>Size of an integer (32 bits)</td>
</tr>
<tr>
<td>$SizeWord$</td>
<td>Size of a word (32 bits)</td>
</tr>
<tr>
<td>$SizeByte$</td>
<td>Size of a byte (8 bits)</td>
</tr>
<tr>
<td>$\rho_{AB}$</td>
<td>Correlation between item A and item B</td>
</tr>
<tr>
<td>$S_A, S_B, S_{AB}$</td>
<td>Support of itemsets A, B, AB</td>
</tr>
</tbody>
</table>

Table 3: Notations and Parameters

5.1 Cost Analysis

This section compares the computation cost of List and H-BM in terms of elementary operations. An elementary operation can be a read memory access, a write memory access, a comparison or a logical AND operator. For simplicity, we do not distinguish between the cost of each of them. Indeed, the goal of this evaluation is to capture the global behavior of each algorithm through manageable formulas. Performance measures complement this analysis.

The performance of the List algorithm is determined by the cost of reading the tidlists of itemset1 and itemset2, computing the intersection, writing the resulting list and incrementing the support. In the worst case, the main loop of the List algorithm contains $T \times S_A + T \times S_B$ iterations.

In this loop, each element of both lists is read at least once. Then each element participates in an "==" comparison while computing the intersection. If the "==" condition is satisfied, the

\[ bm = \text{itemset1}.1-\text{BM}[\text{posi} + i \times 8] \& \text{itemset2}.1-\text{BM}[\text{posi} + i \times 8]; \]

\[ \text{new}_{-}\text{itemset}.\text{support} += \text{nbit}[bm]; \]

\[ }\]

\[ }\]

\[ }\]

\[ }\]

\[ }\]

Figure 5 : The H-BM algorithm

---

1 line 10 and line 11 have been split to increase readability

2 In practice (see Figure 2), the same element is accessed several times during the execution. For simplicity, we neglect this fact.
corresponding TID is inserted into the resulting tidlist and the support is incremented. If not, a ">" comparison is required to determine which of the two current TIDs is bigger. It yields for List:

\[
( T*S_A + T*S_B ) \quad \text{// number of memory access to read each element of list(A) and list(B)} \\
+ ( T*S_A + T*S_B ) \quad \text{// number of ">=" comparisons} \\
+ ( T*S_A+T*S_B-T*S_{AB} ) \quad \text{// number of ">=" comparisons} \\
+ T*S_{AB} \quad \text{// write the result into list(AB)} \\
+ T*S_{AB} \quad \text{// increment AB's support}
\]

Thus, in total we obtain: Cost(List) = T * (3S_A + 3S_B + 2S_{AB})

The H-BM algorithm is composed of two loops. The first loop applies the AND operator to each group of A's 2-BM and B's 2-BM. This loop contains \( \frac{T \cdot \text{SizeShort} \cdot \text{SizeInt}}{\text{SizeShort} \cdot \text{SizeInt}} \) iterations. The second loop scans the resulting 2-BM to detect each byte containing a non-zero bit. Thus, this second loop contains \( \frac{T \cdot \text{SizeShort} \cdot \text{SizeByte}}{\text{SizeShort} \cdot \text{SizeByte}} \) iterations. For each non-zero bit, the algorithm accesses the corresponding group of A's 1-BM and B's 1-BM, applies the AND operator to them, accesses the array \( \text{nbbit} \) and increments AB's support from the corresponding value. The probability that a 2-BM bit is 1 is \( \min(1, S_{AB} \cdot \text{SizeByte}) \). Finally, the number of test required to detect all non-zero bits in a byte is \( \text{SizeByte} \cdot \frac{n}{n+1} \) tests where \( n = \text{SizeShort} \cdot \text{SizeByte} \cdot S_{AB} \).

This leads to the following formula for H-BM:

\[
\begin{align*}
\frac{T}{\text{SizeShort} \cdot \text{SizeInt}} \quad \text{// number of iterations in the first loop} \\
(2) \quad \text{// read A's and B's 2-BM group} \\
+ 1 \quad \text{// apply AND to them} \\
+ 1 \quad \text{// write the result into AB's 2-BM} \\
+ \frac{T}{\text{SizeShort} \cdot \text{SizeByte}} \quad \text{// number of iterations in the second loop} \\
( \text{SizeByte} \cdot \frac{n}{n+1} ) \quad \text{// bitwise scan of a byte until all its bits are zero} \\
( 2 ) \quad \text{// test each bit and test whether all non-zero bits have been detected or not} \\
+ \min(1, S_{AB} \cdot \text{SizeByte}) \quad \text{// probability that the current bit is non-zero} \\
( 2 ) \quad \text{// read the corresponding A's 1-BM and B's 1-BM groups} \\
+ 1 \quad \text{// apply AND to them} \\
+ 1 \quad \text{// access the array \( \text{nbbit} \)} \\
+ 1 \quad \text{// increment AB's support}
\end{align*}
\]

Thus, in total we obtain:

\[
\text{Cost(H-BM)} = 4 \cdot \frac{T}{\text{SizeShort} \cdot \text{SizeInt}} + \frac{T}{\text{SizeShort} \cdot \text{SizeByte}} \cdot \frac{\text{SizeByte} \cdot \frac{n}{n+1} \cdot (2+5 \cdot \min(1, S_{AB} \cdot \text{SizeByte}))}{\text{SizeShort} \cdot \text{SizeByte} \cdot S_{AB}}
\]

Let us compare H-BM and List through the following ratio:

\[
\frac{\text{Cost(H-BM)}}{\text{Cost(List)}} = \frac{4}{3} \cdot \frac{T}{\text{SizeShort} \cdot \text{SizeInt}} + \frac{\text{SizeShort} \cdot \text{SizeByte} \cdot S_{AB}}{\text{SizeShort} \cdot \text{SizeByte} \cdot S_{AB} + 1} \cdot \frac{\text{SizeByte} \cdot \frac{n}{n+1} \cdot (2+5 \cdot \min(1, S_{AB} \cdot \text{SizeByte}))}{\text{SizeShort} \cdot \text{SizeByte} \cdot S_{AB} + 1}
\]

\(4\) AB's 1-BM should contain \( T*S_{AB} \) non-zero bits. Thus each byte of AB's 2-BM contains \( n = \frac{T*S_{AB}}{\text{SizeShort} \cdot \text{SizeByte}} \) non-zero bits. We consider here the worst situation for H-BM, that is non-zero bits are uniformly spreaded over all AB's 1-BM group and again uniformly distributed in each AB's 2-BM byte. The algorithm presented in Figure 5 executes \( \text{SizeByte} \cdot \frac{n}{n+1} \) test to detect \( n \) non-zero bits. These \( n \) bits divide a byte in \( n+1 \) intervals, the last of which having not to be scanned since it contains only zero bits. For example, detecting a single non-zero bit in a byte requires a bitwise scan of half of the byte (on average).
Recall that $S_{AB} = \rho_{AB} \cdot \min(S_A, S_B)$, the precedent ratio can be expressed as:

$$
\frac{4 \cdot \text{SizeShort} \cdot \text{SizeByte} \cdot \rho_{AB} \cdot \min(S_A, S_B)}{\text{SizeInt} \cdot \text{SizeShort} \cdot \text{SizeByte} \cdot \rho_{AB} \cdot \min(S_A, S_B) + 1} \cdot (2 + 5 \cdot \min(1, S_{AB} \cdot \text{SizeByte}))
$$

This ratio eliminates parameter $T$, highlighting the influence of parameters $\rho_{AB}$, $S_A$ and $S_B$. Figure 6 plots the ratio with different $S_A$ and $S_B$ by ranging $\rho_{AB}$ from 0.01 to 1. Typical values of $S_A$ and $S_B$ are picked up from the benchmark presented in the following section. H-BM clearly outperforms List for high values of $S_A$ and $S_B$ and/or for low values of $\rho_{AB}$. A low value for $\rho_{AB}$ characterizes a bad correlation between items A and B. H-BM benefits from this bad correlation due to the high selectivity of the 2-BM. List is more insensitive to this factor which impacts only on the cost of building the resulting list. For very low values of $\rho_{AB}$ (e.g., $\rho_{AB} = 0.01$), H-BM is up to ten times faster than List, while for the highest value of $\rho_{AB}$ ($\rho_{AB} = 1$), List is only two times faster than H-BM. High values for $S_A$ and $S_B$ disadvantage List because of the increasing size of the lists to be merged. Finally, note that H-BM behaves very well when $S_A$ and $S_B$ have different values (e.g., $S_A=2\%$ and $S_B=0.25\%$). This is due to the fact that the selectivity of the 2-BM depends on $\min(S_A, S_B)$.

![Figure 6: H-BM versus List](image)

### 5.2 Memory Consumption

This section compares the main memory requirements of H-BM and List. To evaluate the memory consumption, we consider the size of the main memory structures used to execute the most space consuming phase of the mining algorithm, that is finding all frequent 2-itemsets from the frequent 1-itemsets. At this time, the information of both frequent 1-itemsets and candidate 2-itemsets have to be maintained in main memory. The following phases are less space consuming because the number of candidate k-itemsets decreases as k increases.

Let $\text{Freq}$ indicate the number of frequent 1-itemsets, the number of candidate 2-itemsets is $(\text{Freq}-1) + (\text{Freq} - 2) + \ldots + 1 = \frac{\text{Freq} \cdot (\text{Freq}-1)}{2}$. For List, we need to store all TIDs ($T \cdot \text{Sup}$) of frequent 1-itemsets and that of candidate 2-itemsets. For H-BM, only frequent itemsets have...
both 1-BMs and 2-BMs, while candidate 2-itemsets keep only their 2-BMs. The following formulas are expressed in terms of words (i.e., one unit = 32 bits).

We obtain for List :
\[
\text{Cost(List)} = \text{Freq} \times \text{Sup} \times \frac{\text{Freq} \times (\text{Freq} + 1)}{2}
\]

We obtain for H-BM :
\[
\text{Cost(H-BM)} = \text{Freq} \times \text{SizeWord} \times \left( 1 + \frac{1}{\text{SizeShort}} + \frac{\text{Freq} - 1}{2 \times \text{SizeShort}} \right)
\]

Let us compare H-BM and List through the following ratio :
\[
\frac{\text{H-BM}}{\text{List}} = \frac{\frac{\text{Freq} \times \text{Sup} \times (\text{Freq} + 1)}{2}}{\text{Freq} \times \text{SizeWord} \times \frac{\text{Freq} + 1}{2 \times \text{SizeShort}}}
\]

Figure 7 shows this ratio for different values of Freq. Observe that List wins H-BM only when Freq and Sup are both small, while H-BM can be more compact than List up to an order of magnitude. The benefit of H-BM compared to List increases with Sup. This is obviously explained by the fact that the size of the lists augments linearly with Sup. As a conclusion, H-BM is able to treat a high number of itemsets in parallel while saving as much as main memory space as possible. This makes H-BM well adapted to parallel SMP (Symmetric Multi-Processor) architectures where all processors share a same bounded memory space.

5.3 Evaluating Itemsets of the Form \( \neg A \cap B \)
Up to now, we considered only AND relationship between itemsets. As mentioned in [BMS97], sometimes the negative implications are also useful. When the minsup and the support of an itemset are both less than 50%, this itemset and its negation are both frequent. For example, suppose a database containing 1000 records with a minsup of 40%. If the support of item A is 45%, its negation ¬A is also frequent. With List, 1000 integers will be used to store the TIDs of both A and ¬A. However only $2*(1000/32 + 1000/(16*32)) = 68$ integers are sufficient with a bitmap (1-BM + 2-BM).

The formulas introduced in Section 5.1 remain valid for evaluating the cost of each algorithm when they are faced to itemsets of the form (¬A ∩ B), except that $(1 - S_A)$ must be used in place of $S_A$. In this context, N-BM\(^5\) will outperform both H-BM and List. Indeed, the benefit of H-BM compared to N-BM relies on the selectivity provided by the 2-BM. For reasonable values of $S_A$ (e.g., $S_A < 0.05$), $(1 - S_A)$ is closed to 1 so that all entries in ¬A's 2-BM contain a non-zero bit and H-BM becomes worse than N-BM. While the cost of N-BM remains constant, the cost of List raises to: $T * (3(1-S_A) + 3S_B + 2S_{AB})$. Indeed, algorithm List is slow down by the huge size of ¬A's list.

6. Experimental Results

This section reports our experimental results. All algorithms List, N-BM and H-BM were written in C, and were run on a SunOS station with 32 M of main memory and a clock rate of 166 MHz. All experiments were run on synthetic data\(^6\) [AS94].

6.1 Test Data

The generation of the synthetic data is well explained in [AS94]. It is considered as a typical benchmark for mining association rules and has been widely used by several algorithms ([PCY95], [SON95], [Toi96], [BMU+97], [BMS97], etc.). The parameters used are shown in Table 4.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>D</td>
</tr>
<tr>
<td>$</td>
<td>T</td>
</tr>
<tr>
<td>$</td>
<td>I</td>
</tr>
<tr>
<td>$</td>
<td>L</td>
</tr>
<tr>
<td>$N$</td>
<td>Number of items</td>
</tr>
</tbody>
</table>

Table 4: Parameters

Six different databases are used. For all these databases, we set $|D| = 100K$, $N = 1000$ and $|L| = 2000$. The values of $|T|$ are set to 5, 10 and 20. The average size of maximal potentially frequent itemsets is 2, 4 and 6. Table 5 summarizes the data parameter settings.

| Name           | $|T|$ | $|I|$ | Size in Megabytes |
|----------------|-----|-----|------------------|
| T5.12.D100K    | 5   | 2   | 2.4              |
| T10.12.D100K   | 10  | 2   | 4.4              |
| T10.I4.D100K   | 10  | 4   |                  |
| T20.12.D100K   | 20  | 2   | 8.4              |

\(^5\) Cost(N-BM) = 6*T/SizeShort.

Indeed, for each of the (T/SizeShort) 1-BM groups, A's 1-BM and B's 1-BM are accessed, an AND is performed between them, the result is stored in AB's 1-BM, the nbbit array is accessed and finally, AB's support is incremented (6 operations).

\(^6\) http://www.almaden.ibm.com/cs/quest/syndata.html
6.2 Relative Performance

We compare the three algorithms: List, N-BM and H-BM from Figure 8 to Figure 15. The y axis represents the execution time. From Figure 8 to Figure 13, the x axis represents the value of the minimum support, ranging from 2% to 0.25% (these values correspond to the ones selected in [AS94]). In Figure 14 and 15, the x axis represents the average transaction size.

Figure 8 shows that the cost of each algorithm highly depends on the value of minimum support. When minimum support decreases, Freq (the total number of frequent itemsets) increases. Thus more computation is required. The cost of N-BM augments linearly with Freq because the cost to compute each itemset is independent on its support. H-BM and List better resist because the computation of each itemset is cheaper when its support is low. Surprisingly, the difference between H-BM and List augments linearly as minimum support decreases. The analytical evaluation does not show this phenomenon. This is explained by the fact that even if List takes more advantage than H-BM from a low support to evaluate a single itemset, the total difference between both algorithms is amplified by the higher number of itemsets to compute.

Figure 9 shows how each algorithm reacts when we increase $|T|$ from 5 (the value of Figure 8) to 10. Increasing $|T|$ augments the size of the database by the same factor. The immediate effect is to augment the value of Freq and the average support. All algorithms suffer from this higher number of frequent itemsets to compute. The cost of H-BM and List is roughly multiplied by two. N-BM is more robust in this situation because its cost to compute an itemset is independent on its support. The impact of parameter $|T|$ is more precisely captured in Figures 14 and 15. Figure 9 also demonstrates that List wins N-BM only when the minimum support is low. For higher values of the minimum support, List must merge bigger TID lists while N-BM remains constant. In Figure 11, we increase $|T|$ up to 20. In this situation, List becomes the worst algorithm whatever be the value of minimum support.

Figure 10 shows how each algorithm reacts when we increase $|I|$ from 2 (the value of Figure 9) to 4. Since frequent itemsets are larger, each algorithm must perform more iterations. All curves raise roughly in the same proportion (+100 seconds for each algorithm). Thus, this parameter has the same impact on all algorithms. Notice that in this figure the point where List outperforms N-BM appears with a higher minimum support than in Figure 9. The reason is that when $|I|$ increases, the number of frequent itemsets climbs, while the average support goes down, i.e., the size of lists descends. Comparing Figure 11 (T20.I2), Figure 12 (T20.I4) and Figure 13 (T20.I6) leads to the same conclusion.
Figure 8: T5.I2.D100K

Figure 9: T10.I2.D100K

Figure 10: T10.I4.D100K

Figure 11: T20.I2.D100K

Figure 12: T20.I4.D100K

Figure 13: T20.I6.D100K
Figure 14: I4.D100K (minsup = 0.25%)

Figure 15: I4.D100K (minsup = 0.75%)

Figure 14 and Figure 15 show how these algorithms scale up as $|T|$ is increased from 10 to 30 with $|I|=4$ and $|D|=100K$. Two values of the minimum support are chosen, 0.25% (Figure 14) and 0.75% (Figure 15). Once again, H-BM outperforms others. List can outperform N-BM only when the minimum support is low (0.25%) and the transaction size is small. Figures 14 and 15 confirm the result of Figure 9, that is N-BM is more stable than the two others when $|T|$ increases. The reason is that the performance of N-BM depends only on $|D|$. That means given two itemsets, the time used to compute their support is constant ($|D|/16$) regardless of their actual supports. On the contrary, it is the supports of itemsets which impact the behaviors of List and H-BM.

As a conclusion, H-BM is better than both List and N-BM in all situations due to its 2-BM and it is also very stable to $|T|$ and to $|I|$. Depending on the situation, it outperforms List by a factor between 2 and 3 in the experiments. N-BM can also outperform List when transactions become large. For small transactions, List is always better than N-BM.

7. Parallelization

With the development of parallel database systems, notably in the data warehouse environment, data mining requires efficient parallel algorithms. The expected benefit of bitmap algorithms to parallelize the mining of association rules is twofold:

- Bitmap algorithms use only fixed-size structures, namely 1-BM and 2-BM. These fixed-size structures can be easily split into disjoint fragments avoiding any contention when several processors access them.

- The results produced by each phase of the H-BM algorithm are 2-BMs, whose size is rather small (see Section 5). This increases the potential degree of parallelism in Symmetric Multi-Processor architectures (SMP) where memory resources are bounded. This also reduces the communication costs in cluster (i.e., shared disks) and massively parallel (MPP) architectures.

Two phases of the bitmap algorithms can be parallelized: (i) The bitmap construction; (ii) The computation of support. We discuss these two procedures below in the SMP context, i.e., in the case of shared memory and parallel disks. In that case, the database is fragmented on the parallel disks, as with the partition algorithm. Each processor manages one partition.
7.1 Bitmap Construction

We call 1-BM (resp. 2-BM) matrix the collection of 1-BM (resp. 2-BM) vectors associated to all 1-itemsets. The construction of the 1-BM and 2-BM matrices depends on the underlying database organization. Assume the database is a set of tuples of the form \(<\text{TID}_k, \{i_1, i_2, \ldots, i_m\}>\). This organization can be provided by any relational or object oriented DBMS supporting either nested relations or multi-valued attributes. In this case, each processor scans a fragment of the database and fills in the corresponding part of the 1-BM and 2-BM matrix. To avoid contention on the 2-BM matrix, the size of each database fragment must be a multiple of the number of transactions referenced by each 2-BM word, that is SizeWord*SizeShort = 512 transactions.

If the database is organized as first normal form relations (i.e., collections of tuples of the form \(<\text{TID}_k, i_j>>\), contention is likely to occur since tuples with the same TID may be distributed in different database fragments (especially when the TIDs are not sorted). Contention is critical mainly on the 2-BM matrix. To avoid the contention, each processor can build a local copy of the 2-BM matrix. The final 2-BM matrix can be obtained by ORing these local copies. Note that this step can be done in parallel too. The resulting 1-BM and 2-BM matrix are pictured in Figure 16.

![Figure 16: 1-BM and 2-BM matrix](image)

7.2 Computation of Support

Each phase of the H-BM algorithm computes all candidate k-itemsets by joining the frequent (k-1)-itemsets determined during the preceding phase and prunes those not frequent. We discuss below three alternatives to parallelize this algorithm.

- **Vertical Partitioning**: each processor computes the support of one candidate k-itemset at a time by joining two frequent (k-1)-itemsets (hatched columns on Figure 16). This means the processors perform disjoint subsets of the iterations of algorithm H-BM. No synchronization is required between the processors except on the global queue maintaining the list of candidate k-itemsets remaining to be evaluated. To avoid this synchronization, a subset of candidate k-itemsets can be statically assigned to each processor. In this case, load balancing may be difficult to achieve since the time required to compute the support of k-itemsets depends on the correlation between the corresponding frequent (k-1)-itemsets.

- **Horizontal Partitioning**: each processor computes all candidate k-itemsets on a fragment of the 2-BM matrix (dotted rows on Figure 16). To evaluate the supports of these candidate k-itemsets, each processor accesses the corresponding 1-BM fragments and maintains the values of local supports in local counters. This step does not required any synchronization between the processors and the load balancing no longer depends on the correlation between (k-1)-itemsets. At the end of this step, a synchronization is
required to sum up the local counters associated to the same candidate k-itemsets. Horizontal partitioning is more memory space consuming than vertical partitioning since \((\text{Freq(Freq-1)/2})\) 2-BM vectors (one for each candidate k-itemset) are built in parallel and cannot be pruned before the end of each phase.

- **Pipelining**: candidate \((k+1)\)-itemsets can be computed while candidate k-itemsets are under construction. The synchronization between the different stages of the pipeline depends on whether vertical or horizontal partitioning is selected inside each stage. Pipelining increases parallelism at the expense of memory consumption (i.e., intermediate results are produced in parallel at each stage of the pipeline). In addition, pipelining delays the pruning of the non-frequent candidate itemsets under the minimum threshold.

Although more space consuming, horizontal partitioning and pipelining exhibit more parallelism. The small size of 2-BM vectors make them realistic. Hybrid solutions mixing horizontal and vertical partitioning can also be investigated. For example, a parallel algorithm using horizontal partitioning for composing bitmaps and vertical partitioning for computing supports is possible. Anyway, further work is required to evaluate more accurately the tradeoff between all these alternatives. Note that the same alternatives apply to cluster and MPP architectures as well. In this context, the tradeoff between them should integrate the communication cost. Again, the small size of 2-BM vectors decreases the communication cost between processors in all situations.

In general, bitmap-based algorithms are easier to parallelize than list-based algorithms. The main problem comes from the variable - and potentially large - size of the lists of transaction identifiers. For example, this makes horizontal partitioning quite difficult to implement in a fair way. In summary, bitmaps are more convenient than lists due to their structure regularity and to their smaller size (for what concerns 2-BM).

### 8. Conclusion and Future Work

In this paper, we presented two bitmap based algorithms (N-BM and H-BM) for counting the support of frequent itemsets. The proposed H-BM algorithm is not only fast but demands less memory than others. We have shown through analytical and experimental evaluations that H-BM outperforms the other algorithm in most situations. When the database is large and the support is relatively high, even N-BM can outperform List. N-BM is the best algorithm to evaluate rules containing a NOT logical operator. We also verify that bitmap algorithms scale well when the average size of transactions increases. A short discussion shows that they are easy to parallelize in comparison with list-based algorithms. All these factors demonstrate that the proposed bitmap algorithms are very well suited for mining association rules in VLDBs.

Bitmap based algorithms open several new directions of research. One nice aspect is that they are able to deal efficiently with full logic rules, supporting OR and NOT logical operators in addition to the classical AND. OR is very useful for discovering association rules according to a generalization hierarchy. In future, we plan to resolve the problem of generalized association rules using bitmaps. NOT is helpful for rules with exceptions. This should be further explored. More generally, as we are able to compute efficiently the support of any logical rule, the question is how to generate interesting rules to check. We would like to explore genetic algorithms to do so. Furthermore, parallelization of bitmap-based algorithms as sketched in section 7 is a promising area of research. Even specific hardware to efficiently process arrays of bits could find some applications there.

**Acknowledgment**

We would like to thank Luc Bouganim and Fei Sha for their helps and suggestions.
References


[CHN+96] D. W. Cheung, J. Han, V. Ng, A. Fu and Y. Fu. *A Fast Distributed Algorithm for Mining Association Rules*. Int’l Conf. on Parallel and Distributed Information Systems (PDIS), Miami Beach, Florida, USA, 1996.

[HF95] Jianwei Han, Yongjian Fu. *Discovery of Multiple-level Association Rules from Large Databases*. In Proceedings of the 21st International Conference on Very Large Data Bases, Zurich, Switzerland, 1995, pp. 420-431.


