Fast Vertical Mining Using Boolean Algebra

Hosny M. Ibrahim
Information Technology Department
Faculty of Computer and Information, Assiut University
Assiut, Egypt

M. H. Marghny
Computer Science Department
Faculty of Computer and Information, Assiut University
Assiut, Egypt

Noha M. A. Abdelaziz
Information System Department
Faculty of Computer and Information, Assiut University
Assiut, Egypt

***Abstract*—The vertical association rules mining algorithm is an efficient mining method, which makes use of support sets of frequent itemsets to calculate the support of candidate itemsets. It overcomes the disadvantage of scanning database many times like Apriori algorithm. In vertical mining, frequent itemsets can be represented as a set of bit vectors in memory, which enables for fast computation. The sizes of bit vectors for itemsets are the main space expense of the algorithm that restricts its expansibility. Therefore, in this paper, a proposed algorithm that compresses the bit vectors of frequent itemsets will be presented. The new bit vector schema presented here depends on Boolean algebra rules to compute the intersection of two compressed bit vectors without making any costly decompression operation. The experimental results show that the proposed algorithm, Vertical Boolean Mining (VBM) algorithm is better than both Apriori algorithm and the classical vertical association rule mining algorithm in the mining time and the memory usage.**

Keywords—association rule; bit vector; Boolean algebra; frequent itemset; vertical data format

I. Introduction

Data mining is defined as “The non trivial extraction of implicit, previously unknown and potentially useful information from databases” [1]. Association rules mining is an active research topic in the data mining field, which is the key step in the knowledge discovery process [2, 3]. Mining frequent itemsets (FIs) is the most important task in mining association rules. Furthermore, frequent itemsets detection can be used in other data mining tasks like classification and clustering [4-6]. Therefore a lot of mining frequent itemsets algorithms has been proposed. None of these methods can outperform other methods for all types of datasets with every minimum support [7, 8]. The well-known (FIs) mining algorithms are based on either horizontal or vertical data structures. Some of the horizontal based algorithms are Apriori, AprioriTid and FP-growth. Apriori is a level-wise algorithm that adopts an iterative method to discover frequent itemsets, in which k frequent itemsets is created by joining k-1 frequent itemsets, and then remove itemsets that contain non-frequent items. Non frequent items are detected by scanning the database once for each itemset to calculate its support. This is the most important shortcoming of Apriori algorithm [9]. AprioriTid has been proposed to improve Apriori algorithm’s efficiency by reducing the overhead of I/O by scanning the database only once in the first iteration [9]. FP-growth algorithm mines frequent item sets by scanning the database only two times without candidate generation. It also compresses the data set into a data structure called FP-tree. FP-growth finds all the frequent item sets by searching the FP-tree, recursively [10]. Eclat, BitTableFI and IndexBitTableFI are some examples of vertical based (FIs) mining methods. Eclat uses a structure called Tidset, which store the transaction identifiers for each itemset. The support of an itemset X can be fast derived as the cardinality of the Tidset of the itemset. Thus, the support(X) = |Tidset(X)|. It also proposed the way of computing Tidset(XY) by the intersection operator between Tidset(X) and Tidset(Y). That is, Tidset(XY) = Tidset(X) ∩ Tidset(Y) [11]. In BitTableFI [12] each item occupied |T| bits, called a bit vector, where |T| is the number of transactions in D. The bit vector of a new itemset XY from the two itemsets X and Y could be easily derived by the AND operation on the two bit vectors of X and Y. Because the length of the two bit vectors was the same, the result would be a bit vector with the same length of |T| bits. Dong and Han used the BitTable to mine frequent itemsets based on the level-wise concept in the Apriori algorithm [9]. Their approach was named BitTableFI [12]. Note that in the Apriori algorithm, the supports were computed by re-scanning databases, while in the BitTableFI approach, they were derived by the intersection of bit vectors. The support of an itemset could be found by counting the number of ‘1’ bits in its corresponding bit vector. Later, in Song et al. [13], Index-BitTableFI employed index array to improve the algorithm. From the above discussion, the following points can be concluded: vertical association rule algorithms conquer some disadvantages of horizontal ones, vertical association rule algorithms need memory space too much when the dataset is too large. In order to overcome this issue, in this paper, a proposed algorithm that depends on a simple representation of frequent itemsets, which is, compressing the support sets bitmap of data itemsets that to be sent to memory, so as to save the space required by the algorithm. It contributes to reducing not only the execution time but also the required memory. The rest of this paper is organized as follows; Section II briefly revisits some association rule background information. The difference between vertical and horizontal data formats are listed in Section III. Boolean Algebra rules and theories are given in section IV. In Section V, the new algorithm, VBM algorithm is proposed. Section VI analyzes the performance of the proposed algorithm and conclusion is given in Section VII.

II. Basic Conception

Association rule mining involves detecting items which tend to occur together in transactions and the association rules that relate them [14].

Consider I= {i1, i2,………,im} as a set of items. Let D, the task relevant data, is a set of database transactions where each transaction T is a set of items such that T is a subset of I. Each transaction is associated with an identifier, called TID. Let A be a set of items. A transaction T is said to contain A if and only if $A \subseteq T$. An Association Rule is an implication of the form $A \Rightarrow B$, where $A \subset I$, $B \subset I$, and $A \cap B = \varphi$.

Rule support and confidence are two measures of rule interestingness. They respectively reflect the usefulness and certainty of discovered rules.

The rule $A \Rightarrow B$ holds in the transaction set D with support s, where s is the percentage of transactions in D that contain $A \cup B$ (i.e., the union of sets A and B, or say, both A and B). This is taken to be the probability, $P(A \cup B)$. That is,

$$\text{Support } (A \Rightarrow B) = P(A \cup B). \qquad (1)$$

The rule $A \Rightarrow B$ has confidence c in the transaction set D, where c is the percentage of transactions in D containing A that also contain B. This is taken to be the conditional probability; P(B |A) .That is,

$$\text{Confidence } (A \Rightarrow B) = P(B|A) = \frac{\text{Support}(A \cup B)}{\text{Support}(A)} = \frac{\text{Support\_Count}(A \cup B)}{\text{Support\_Count}(A)} \qquad (2)$$

The definition of a frequent pattern relies on the following considerations. A set of items is referred to as an itemset (pattern). An itemset that contains K items is a K-itemset. The set {X, Y} is a 2-itemset. The occurrence frequency of an itemset is the number of transactions that contain the itemset. This is also known as the frequency or the support count of an itemset [15]. An itemset satisfies minimum support if the occurrence frequency of the itemset is greater than or equal to the minimal support threshold value defined by the user [16]. The number of transaction required for the itemset to satisfy minimum support is therefore referred to as the minimum support count. If an itemset satisfies minimum support, then it is a frequent itemset.

A minimum support threshold and a minimum confidence threshold can be set by users or domain experts. Rules that satisfy both a minimum support threshold (min_support) and minimum confidence threshold (min_confidence) are called strong. The objective of association rule mining is to find rules that satisfy both a minimum support threshold (min_support) and minimum confidence threshold (min_confidence) .Thus the problem of mining association rules can be reduced to that of mining frequent itemsets.

In general, the problems of association rules can be divided into two sub ones [17, 18]:

1) Find out all the frequent itemsets in database D according to the minimum support.

2) Generate association rules from frequent itemsets with the limitations of minimal confidence.

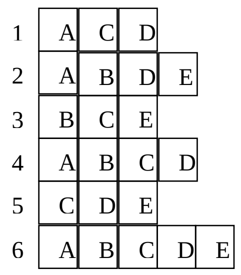
Since the second step is much less costly than the first and the overall performance of mining association rules is determined by the first step, here we are concentrating only on the first step.

III. Vertical Association Rules Mining

Horizontal and vertical data formats are two common kinds of data formats to be adopted in frequent itemsets mining. Horizontal structure is the data distribute way by most association rules mining algorithm, its dataset is made up of a series of transactions, each of them includes transaction's ID TID and relevant transaction's inclusive itemsets. However, vertical structure is that the dataset is made of a series of items; each of the items has its TID-list that is the ID list including all transaction of this item [19]. Table I shows transaction database. Fig. 1 and 2 show respectively horizontal and vertical structures for database in Table 1.

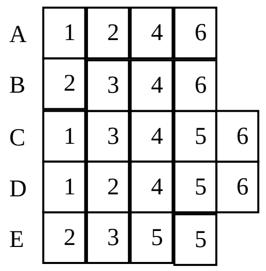
TABLE I. A Transaction Database

TID	Item
1	A, C, D
2	A, B, D, E
3	B, C, E
4	A, B, C, D
5	C, D, E
6	A, B, C, D, E



1	A	C	D		
2	A	B	D	E	
3	B	C	E		
4	A	B	C	D	
5	C	D	E		
6	A	B	C	D	E

Fig. 1. Horizontal structure



A	1	2	4	6	
B	2	3	4	6	
C	1	3	4	5	6
D	1	2	4	5	6
E	2	3	5	5	

Fig. 2. Vertical structure

Algorithms for mining frequent itemsets based on the vertical data format are usually more efficient than those based on the horizontal, because the former often scan the database only once and compute the supports of item sets fast [20].

IV. Boolean Algebra

Boolean algebra which was developed by George Boole in 1854, is an algebraic structure defined by a set of elements, B (i.e. B is defined as a set with only two elements 0 and 1 in two valued Boolean algebra), together with two binary operators, (+) and (•), providing that the following postulates are satisfies[21]:

Note:

- Here is listed only the postulates which are of interest to that work not all Boolean algebra postulates.
- In two valued Boolean algebra, zero and one define the elements of the set B, and variables such as x and y merely represent the elements.

1. (a) The element 0 is an identity with respect to +; that is, x + 0 = 0 + x = x.

1. (b) The element 1 is an identity with respect to •; that is, x • 1 = 1 • x = x.

2. (a) The structure is commutative with respect to +; that is, x + y = y + x.

2. (b) The structure is commutative with respect to •; that is, x • y = y • x.

3. For every element x ∈ B, there exists an element x′ ∈ B (called the complement of x) such that (a) x + x′ = 1 and (b) x • x′ = 0.

Duality principal of Boolean algebra states that: every algebraic expression deducible form the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged, simply interchange OR and AND operators and replace 1's by 0's and vice versa as shown in parts a and b in the above postulates.

Some important theorems that were derived from the above postulates:

1: (a) x + x= x.

(b) x • x= x.

2: (a) x + 1= 1.

(b) x • 0= 0.

3: involution (x′) ′ = x.

4: DeMorgan (a) (x + y) ′ = x′y′.

(b) (x y) ′ = x′ + y′.

V. Vertical Boolean Mining Algorithm (VBM)

This section divided into three subsections the first subsection V.*A.* describes the schema of bitmap used and its compression function. The intersection methods of the compressed vectors are described in V.*B.* Finally the detailed steps of the algorithm are illustrated in the last subsection V.*C.*

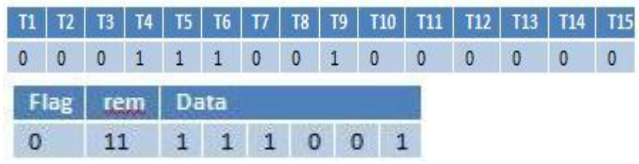
A. Schema of bitmap used and its compression function

This algorithm is based on vertical data format but instead of representing each item with a bit vector of fixed length equal to the total number of transactions, it uses compression function that works as described below.

The primary goal of the compression function is to make each vector starts and ends with consecutive zeros and then it gets rid of these zero bits to compress the vector.

For each bit vector in the bitmap the compression function examines the start and the end of that vector. Three cases could be found:

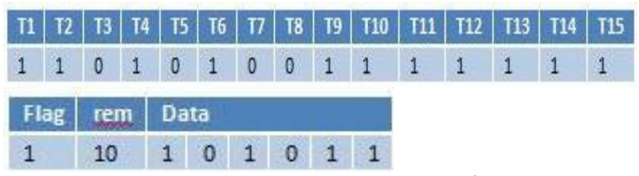
Case1: the vector starts and ends with sequence of zeros.



T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15
0	0	0	1	1	1	0	0	1	0	0	0	0	0	0

Flag	rem	Data					
0	11	1	1	1	0	0	1

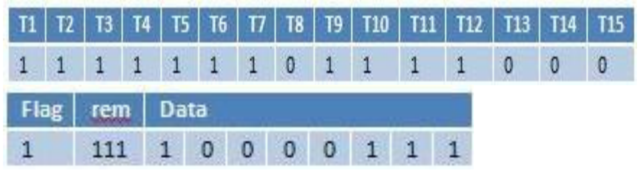
Case2: the vector starts and ends with sequence of ones.



T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15
1	1	0	1	0	1	0	0	1	1	1	1	1	1	1

Flag	rem	Data					
1	10	1	0	1	0	1	1

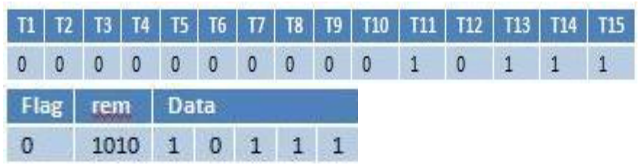
Case3 (a): the vector starts with sequence of ones and ends with zeros.



T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15
1	1	1	1	1	1	1	0	1	1	1	1	0	0	0

Flag	rem	Data							
1	111	1	0	0	0	0	1	1	1

Case3 (b): the vector starts with sequence of zeros and ends with ones.



T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15
0	0	0	0	0	0	0	0	0	0	1	0	1	1	1

Flag	rem	Data				
0	1010	1	0	1	1	1

Fig. 3. Example of transactions before and after compression

In case 1, the compression function does not change the vector form. In case 2, the compression function sets the vector in its complement form to make it starts and ends with sequences of zeros as in case 1. In the third case the compression function counts the number of sequential zeros and the number of sequential ones in the front and tail of the vector. The compression function leaves the vector in its original form if the number of zeros counted is greater than number of ones and puts it in the complement form in the opposite case.

For all three cases given in Fig. 3, the algorithm uses a new data structure for vectors. The vector consists of three elements. The first element, *flag*, Boolean value which indicates either the vector is in the original form (i.e. when *flag*=0) or complement form (i.e. when *flag*=1). The second element, removed (abbreviated as *rem*), binary value representing the number of zeros or ones removed from the beginning of the bit vector.

The third element, *data*, list of bits representing the remaining bits after removing sequences of zero bits or one bits at the front and the tail of the old vector.

B. How to intersect two compressed bit vectors and calculate their support

Fundamental idea: In order to enhance algorithm's operation speed after compressing bitmap, the algorithm makes use of Boolean algebra's rules and postulates to intersect two compressed bit vectors and to calculate their support fast without making any decompression operation for itemsets' bit vectors.

During intersecting two compressed bit vectors according to the schema illustrated above one of the following three cases will be occured:

Case 1: the two vectors are in the original form (i.e. *flag*1=*flag*2=0).

Initially, the decimal equivalents of the "*rem*" values of the two vectors are compared and the larger is used as the "*rem*" value of the resulting vector. Then AND operations are performed for the *data* parts of the two bit vectors. These operations start at position zero for the vector of larger "*rem*". The starting point for the AND operation in the other vector is the difference between the decimal equivalents of vectors' "*rem*" values. If an initial resulting value is 0, then the "*rem*" value of the outcome vector is increased by 1 until the first non-zero resulting value is reached. Next, from the position of non-zero bit, all the resulting bits by the AND operation are kept in the outcome vector's *data* part except the last consecutive zero bits. Finally the resulting vector's *flag* value is set to zero indicating that the result of intersecting two original bit vectors is a vector in the original form. An example is given below to illustrate the intersection operation on case 1. Assume there are two vectors in the original form: {0, 11, {1, 1, 1, 0, 0, 1}} and {0, 111, {1, 1, 0, 1, 0, 1, 1, 1}} and their intersection is to be found. Both vectors are in the original form because their flags equals to 0. Because the "*rem*" value (111) of the second vector which is corresponding to 7 in decimal is larger than that (11) of the first that means 3 in decimal system, the AND operation then begins from position (7-3= 4) of first vector and position (0) of the second, at which the result of 0 and 1 is 0. The resulting "*rem* value" increased by one to be 8. Then, the result of next bits 1 AND 1 is 1 not equals to zero. The rest bits of the second vector are automatically removed because they haven't corresponding bits in the first vector which means that those bits in the first vector were zero bits so that the compression function removed them, and the results are all 0. The resulting vector is then {0, 1000, {1}}.The process is shown in Fig. 4.

Case 2: one vector is in the original form and the other is in the complement form (i.e. *flag*1=0, *flag*2=1 or vice versa).

The result of intersecting two vectors in different forms as in this case is a vector in the original form so the outcome vector's *flag* value is set to zero. The "*rem*" value of the resulting vector initially equals to that of the vector in the original form whether it is the larger or not. Then the decimal equivalents of the "*rem*" values of the two vectors are compared, if the "*rem*" value of the original vector is the larger, the AND operations start at position zero for *data* part of the original vector and from position equals to the difference between original vector's "*rem*" and complement vector's "*rem*" for *data* part of the complement vector. If an initial resulting value is 0, then the "*rem*" value of the outcome vector is increased by 1 until the first non-zero resulting value is reached. But if the complement vector's "*rem*" is the lager, the first complement vector's "*rem*" value minus original vector's "*rem*" value bits of the original vector are added to the *data* part of the resulting vectors as they are, because those bits are corresponding to bits of value 1 that were removed from the complement vector in its original form and according to Boolean algebra postulates element 1 is an identity element with respect to AND operation. Next, AND operations are performed between the bits of the original vector and the complement of bits in the complement vector and the resulting bits are kept in the outcome vector's *data* part except the last continuous zero bits. An example is given below to illustrate the intersection operation on this case. Two vectors are given the first is in the original form: {0, 11, {1, 1, 1, 0, 0, 1}} and the second is in the complement form {1, 111, {1, 0, 0, 0, 0, 1, 1, 1}} as shown by their *flag*s, and their intersection is to be found. Initially the "*rem*" value of the resulting vector will be 3 because the "*rem*" value of the original vector is (11) that means 3 in decimal system. Then the first 4 bits in the original vector will be put in the *data* part of the resulting vector, because those bits in the original vector were actually corresponding to 4 ones in the complement

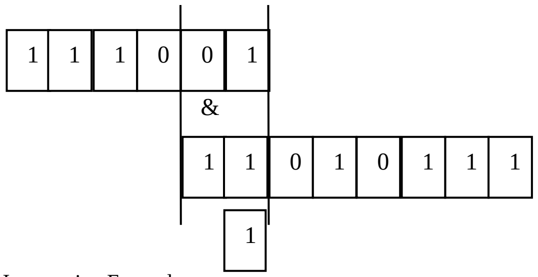


Fig. 4. Intersection Example

vector before compression (i.e. 7-3=4 where 7 is the "*rem*" of complement vector greater than 3, "*rem*" value of original one). Next the AND operation starts at bit number 4 in the original vector and position (0) of the complement one, at which the result of 0 and 1 is 0. The resulting position then moves backward to 8. Then, the result of next bits 1 and 1 is 1 not equals to zero. The rest bits of the second vector are automatically removed because they haven't corresponding bits in the first vector which means that those bits in the first vector were zero bits so that the compression function removed them, and the results are all 0. The resulting vector is then {0, 11, {1,1,1,0,0,1}}.The process is shown in Fig. 5 and the opposite is given in Fig. 6 for vectors {0,11,{1,1,1,0,0,1,1,1}}and {1,10,{1,1,1,0,1,0,1}}with result equals{0,101,{1,0,0,0,1,1}}.

Case 3: the two vectors are in the complement form (i.e. *flag*1=*flag*2=1).

In this case the resulting vector of intersecting two vectors in the complement form is a vector in the complement form, but in some cases this complemented vector may require transforming to the original form according to some conditions that will be described in the algorithm in section V part c.

Depending on Demorgan theory illustrated in section IV, the algorithm follows steps that are exactly the opposite to those that were followed in case 1. Case 3 intersection steps are illustrated here by an example shown in Fig. 7.

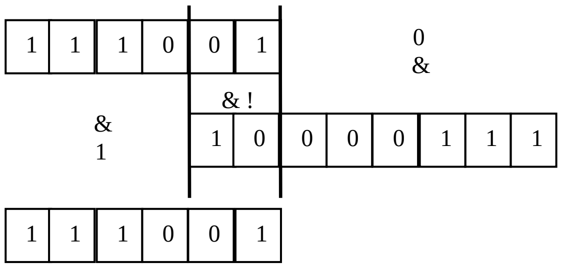


Fig. 5. Intersection Example

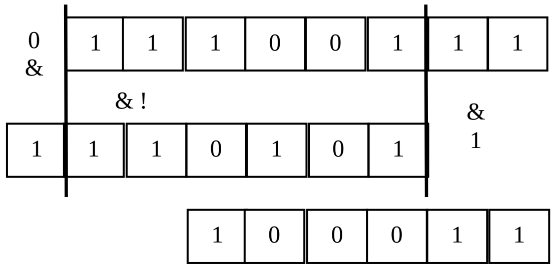


Fig. 6. Intersection Example

Note that: In Fig. 5 and 6 the algorithm automatically puts any bit above sign $\binom{\&}{1}$ in the *data* part of the resulting bit vector and removes any bit below sign $\binom{0}{\&}$ without actually making AND operation to those bits with 0 and 1, depending on Boolean algebra rules illustrated above in order to save execution time.

Fig. 7 shows {1,11,{1,0,1,0,1,0,0,1}} and {1,100,{1,1,0,0,1,0,1}} two compressed bit vectors in the complement form assuming that the original length of the bit vectors before compression is 15 bit.

The result is to be obtained by the following steps:

1) The "rem" value of the resulting vector equals to the smallest "rem" value of the two vectors

As *rem*1=11< *rem*2=100 (i.e. 3 < 4 in decimal system), therefore the resulting "*rem*" value=3.

2) The (rem2-rem1) first bits of the vector with the smallest "rem" value are placed as they are in the data part of the resulting vector according to postulate 1(a) in section IV , because those bits are actually corresponding to zeros in rem2.

As *rem*2-*rem*1= 4-3= 1 bit therefore first bit only of the first vector is to be put in the first bit of the *data* part of the resulting vector as shown in Fig. 7.

3) Since the intersection operation of two original vectors is accomplished through AND operations, therefore the opposite is done here according to Demorgan theory (i.e. (x•y)′=x′+y′) aforementioned in section IV, using An OR operations between each two corresponding bits till reaching the end of one of the bit vectors as shown in Fig. 7.

4) If the bits of one of the vectors finished before the other, the remaining bits of the longer vector will be placed as they are in the data part of the resulting vector, because those bits are actually corresponding to zeros of the shorter vector as discussed in step 2.

5) Finally the resulting vector is equals to {1,11,{1,1,1,0,1,1,0,1}}.

C. How VBM algorithm works

The main steps of the algorithm can be summarized as follows.

First Step: Scan the database once, obtain a compressed bit vector for each data item by the aforementioned method and set up the result in the structure that were described in section V.*A*, and calculate support of each data item to produce frequent 1-itemsets and its related compressed bitmap.

Hint: support of items represented by vectors in the original form is calculated by counting the number of set bits in the *data* part of that vector (i.e. number of "1" bits). But support of items represented by complemented vectors is

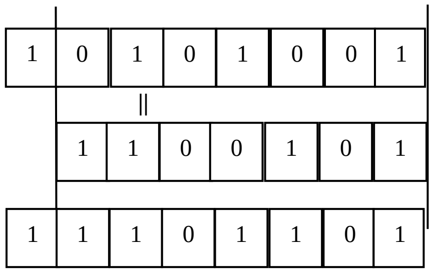


Fig. 7. Intersection Example

equals to the total number of transactions minus the number of set bits in the *data* part of the bit vector.

Second step: In order to get the higher order candidate and frequent (k+1)-itemsets F_{k+1} for each k>1, given a frequent k-itemset F_k, the algorithm uses Depth-first method to join each two itemsets if they have the same first k-1 items (excluding just the last item) and the last item of first k-itemset comes before the last item of the second k-itemset in F_k, and applies a modification of the intersection function, which works on three components of the bit vector consisting of the *flag* part, removed value and the *data* part, so that obtaining higher order frequent (k+1)-itemsets does not require rescanning the database again.

Third step: the intersection function first checks *flag*s of both bit vectors to be intersected to detect which type of intersection needs to be followed as illustrated in section V.*B*, in section V.*B.* cases 1 and 2 are straightforward but case 3 may return vector either in the original or complement form. Case 3 returns vector in the complement form if the *rem* values of both vectors aren't equal to zero and the total of the length of the *data* part of the bit vectors plus the decimal equivalents of their *rem* values (i.e. number of removed zeros) is less than the total number of transactions, because this means that both original vectors were starting and ending with ones so the result will for sure starts and ends with ones, so the resulting vector should be returned as it is (i.e. in the complement form). Case 3 returns vector in the original form (i.e. case 3 will return the complement of the complemented vector section IV Theorem3 involution) if one of the previous conditions aren't satisfied.

Examples on case 3:

a) intersection method returns complement vector:

the result of intersecting {1,11,{1,0,1,0,1,0,0,1}} and {1,100,{1,1,0,0,1,0,1}} equals {1,11,{1,1,1,0,1,1,0,1}}.

b) intersection method returns original vector: the result of intersecting {1,0,{1,1,1,0,1,1,1}} and {1,101,{1,1,0,0,1,0,1,1,1,1}} equals {0,11,{1,0,0,0,1,1,0,1}}.

Forth step: after detecting which type of intersection needs to be followed, the intersection operation is accomplished as illustrated in details in section V.*B.* to obtain the resulting bit vector. Then support count is calculated for the result. If support count>= min_support the result is added to frequent k+1 itemset or removed otherwise.

Finally, this process will be continued until there aren't any longer frequent (k+1)-itemsets, then the algorithm ends.

The proposed VBM approach and the schema of bit vectors used consume less time for computing the intersection among compressed bit vectors and for counting the number of 1 bits in the resulting bit vector due to their shorter lengths so the number of bits to be checked is smaller than in the case of classical vertical association rule mining algorithms.

VI. Experimental Results

All experiments were performed on an Intel Core 2 Duo (2×2 GHz), with 3GBs RAM of memory and running Windows vista and algorithms were coded using java programming language. Three real databases[1] used previously in the evaluation of frequent itemsets mining algorithms [22, 23, 24] are used for the experiments, with their characteristics shown in Table II. Due to the huge amounts of the resulting frequent itemsets the method org.apache.commons.io.FileUtils.contentequals from package commons-io-2.4.jar downloaded from apache library[2] is used to compare the results of the new algorithm with those of the Apriori algorithm and classical vertical association rule mining algorithm without compressed bitmap, to make sure that the results are correct.

Fig. 8 to 10 show the comparison of the execution time of the VBM algorithm, Apriori algorithm and the classical vertical association rules algorithm without compressed bitmap, along with different minimum supports. It could be observed that the VBM algorithm was always faster than the other two in all the results.

Next, experiments were conducted to compare between the VBM total memory usage (in MBs) and the vertical association rules algorithm without compressed bitmap. The VBM algorithm compression percentage is also calculated. The results for the three databases under different min_support values are shown in Table III.

From Fig. 8 to 10 we can see that the mining time of VBM algorithm is far from Apriori algorithm but not faraway from the mining time of vertical association rules algorithm without compressed bitmap. But VBM decreased much in memory used by frequent itemsets bitmap than vertical association rules algorithm without compressed bitmap as illustrated in Table III.

Regarding execution time, the non-parametric wilcoxon significance test has been performed to proof the efficiency of the VBM algorithm for the three datasets. The results of the test are given in Table IV. The VBM algorithm showed significant results when compared to Apriori algorithm and the vertical association rules algorithm as p-value<0.05 in all cases.

The given results show that the strength of the proposed algorithm (VBM) lies in its ability to decrease much in mining time than horizontal association rule mining algorithm and decrease much in memory space than vertical ones. So the proposed algorithm is better than both of them.

TABLE II. Characteristics of Datasets

Dataset	No. of transactions	No. of Items	Average Transaction Length
Chess	3196	75	37
Mushroom	8124	119	23
Connect	67557	129	43

As observed from results the reduction in memory and mining time of the proposed algorithm is significantly affected by the content of dataset. The reduction in memory & time cannot be achieved unless the records in the bitmap starts & ends with sequences of zeros and ones as illustrated in section V.*A.*

[1] http://fimi.cs.helsinki.fi/data
[2] http://commons.apache.org/proper/commons-io/[Accessed 19/7/2014]

TABLE III. MEMORY USAGE OF THE VBM AND VERTICAL ASSOCIATION RULE ALGORITHM

DataSet	**Minimum Support**	**No. Frequent Itemsets**	**Memory Usage of Dataset in Vertical Association Rules Algorithm (MBs)**	**Memory Usage of Dataset in VBM Algorithm (MBs)**	**Compression Percentage**
Chess	65%	111239	42.9	31.8	25.8%
	70%	48731	18.57	13.92	25%
	75%	20993	8	5.91	26%
	80%	8227	3.13	2.26	27.8%
	85%	2669	1.02	0.74	27%
Mushroom	10%	574431	556	350.28	37%
	20%	53583	51.89	34.14	34.2%
	30%	2735	2.6	1.76	32%
	40%	565	0.54	0.35	35.2%
	50%	153	0.15	0.104	30.5
Connect	86%	105047	845	591.5	30%
	90%	27127	218	156.31	28.3%
	94%	4223	34	24.14	29%
	98%	180	1.44	1.045	27.4%

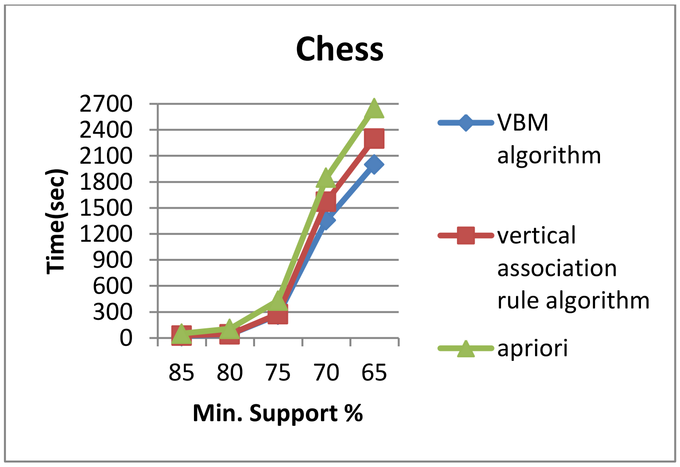


Fig. 8. Execution time of the three algorithms for chess dataset under different min_support values

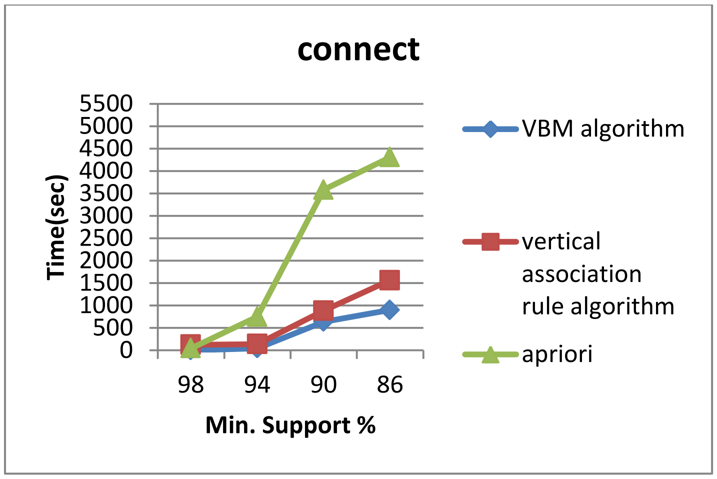


Fig. 9. Execution time of the three algorithms for connect dataset under different min_support values

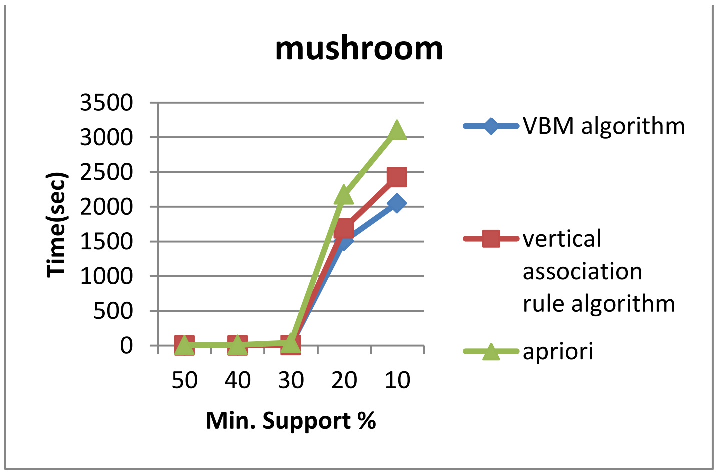


Fig. 10. Execution time of the three algorithms for mushroom dataset under different min_support values

TABLE IV. EXECUTION TIME SIGNIFICANCE TEST

P-value of:	**Chess**	**Mushroom**	**Connect**
VBM vs. Vertical	0.007	0.014	0.002
VBM vs. Apriori	0.001	0.008	0.001

VII. Conclusion

This paper proposes a new algorithm that uses a new data structure for compressed bitmap that allows fast computing of support count. So this algorithm relieves the contradiction between vertical association rules algorithm's run speed and memory space to a certain extent. The contributions could be divided into two parts. First contribution is using new data structure to compress bit vector of transaction list representing each frequent item set in only one database scan. Second In order to enhance algorithm's operation speed after bitmap compression, the algorithm makes use of Boolean algebra theories and postulates to perform bit vectors' intersection operation and calculate support count without need to decode the compressed bit- vectors. Therefore, frequent itemsets is generated quickly. The experimental results indicate that the proposed algorithm is much more efficient than Apriori and the classical vertical algorithm for mining association rules in terms of mining time and memory usage. When the database does not contain consecutive bits of zeros and ones at the start and the end of large number of its transactions, the VBM algorithm may suffer the problem of memory scarcity. So solving this memory problem will be the target addressed in one of our future works. We may use transaction partitioning to solve this mentioned problem or search for other techniques.

References

[1] J. Han and M. Kamber, *Data Mining: concepts and Techniques.* San Francisco, CA : Morgan Kaufmann Publishers, 8-131-20535-5, 2010.

[2] R. Agrawal, T. Imielinski, and A. Swami, "Mining association rules between sets of items in large databases," Proceedings of the ACM SIGMOD International Conference on Management of Data,Washington DC, pp. 207-216. May 1993.

[3] Y. Tong, L. Chen,Y. Cheng, "Mining frequent itemsets over uncertain databases," Proceeding of the VLDB Endowment, Vol. 5(11), pp.1650-1661, Aug. 2012.

[4] J. Han and M. Kamber **,** *Data mining: concepts and techniques*. Morgan Kaufmann Publishers, 1-55860-901-6, 2006.

[5] M. H. Marghny, R. M. Abd El-Aziz and A. I. Taloba**,** "An Effective evolutionary clustering algorithm: hepatitis C case study," Computer Science Department, Egypt, International Journal of Computer Applications, vol. 34, No.6, pp. 0975-8887, 2011.

[6] M. H. Marghny and A. I. Taloba**,** "Outlier detection using improved genetic K-means," International Journal of Computer Applications, vol. 28, No.11, pp. 33-36, 2011.

[7] M. H. Marghny, and A. A. Shakour, "Fast, simple and memory efficient algorithm for mining association rules," International Review on Computers & Software, 2007.

[8] M. H. Margahny and A. A. Shakour, "Scalable algorithm for mining association rules," ICCST, 2006.

[9] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," Proceedings of the 20th International Conference on Very Large Data Bases, Sep. 1994.

[10] J. Han, J. Pei, Y. Yin, "Mining frequent patterns without candidate generation," in Proceedings of the 2000 ACM SIGMOD international conference on Management of data, ACM Press, pp. 1-12, 2000.

[11] MJ. Zaki, S. Parthasarathy, M. Ogihara and W. Li, "New algorithms for fast discovery of association rules," 3rd Int. Conf. Knowl. Disc. Data Min. (KDD), pp. 283-286, 1997.

[12] J. Dong and M. Han**,** "BitTableFI: An efficient mining frequent itemsets algorithm," Knowl.-Based Syst., Vol. 20(4), pp. 329 – 335, 2007.

[13] W. Song, B. Yang and Z. Xu, "Index-BitTableFI: An improved algorithm for mining frequent itemsets," Knowl.-Based Syst., Vol. 21(6), pp. 507-513, 2008.

[14] A. T. Bjorvand, **"**Object Mining: A Practical application of data mining for the construction and maintenance of software components," Proceedings of the Second European Symposium, PKDD-98, pp. 121-129, 1998.

[15] A. Tiwari, R. K. Gupta and D. P. Agrawal, "Cluster based partition approach for mining frequent itemsets," Journal of Computer Science, Vol. 9(6), pp. 191-199, 2009.

[16] M. Houtsma and A. Swami, "Set oriented mining for association rules in relational databases," 11th International conference on Data Engineering, pp. 25-33, 1995.

[17] T. Y. Lin, Hu. Xiaohua and E. Louie, "A fast association rule algorithm based on bitmap and granular computing fuzzy systems," FUZZ '03, Vol. 12(1), pp. 25-28 May 2003.

[18] T. Karthikeyan and N. Ravikumar, **"**A survey on association rule mining," International Journal of Advanced Research in Computer and Communication Engineering, Vol. 3(1), pp. 5223-5227 Jan 2014.

[19] Liu Yang and Mei Qiao, **"**A Bitmap compression algorithm for vertical association rules mining," 2008 International Symposium on Computer Science and Computational Technology (ISCSCT). (IEEE), pp. 101-104, 2008.

[20] V. Bay, H. Tzung-Pei and L. Bac, **"**Dynamic bit vectors: An efficient approach for mining frequent itemset," Scientific Research & Essays, Vol. 6(25), pp. 5358-5368, 2011.

[21] M. Morris Mano & M. D. Ciletti : Digital Design, Chapter (2), Pearson Prentice Hall. 0-13-277420-8. pp. 38-43, 2013.

[22] YU. Xiaomei , H. Wang,"Improvement of Eclat algorithm based on support in frequent itemset mining," journals of computer, Vol. 9(9), pp. 2116-2123, Sep 2014.

[23] Y. Dejnouri, Y. Geraibia, M. Mehdi, A. Bendjoudi and N. Nouali-Taboudjemat, "An efficient measure for evaluating association rules," Proceeding of the 6th international conference of soft computing and pattern recognition (SoCPaR), IEEE explore, pp. 406-410, Aug. 2014.

[24] Z. Khan, F. Haseen, S. T. A. Rizvi and M. ShabbirAlam, " Enhanced BitApriori algorithm: an intelligent approach for mining frequent itemset," Proceeding of the 3rd international conference on frontiersof intelligent computing: Theory and Application (FICTA), Vol. 327, pp. 343-350, Springer, 2015.