Journal of Computer Science and Cybernetics, V.31, N.1 (2015), 17–30
DOI: 10.15625/1813-9663/31/1/5154

MBIS: AN EFFICIENT METHOD FOR MINING FREQUENT WEIGHTED UTILITY ITEMSETS FROM QUANTITATIVE DATABASES

NGUYEN DUY HAM[1], VO DINH BAY[2], NGUYEN THI HONG MINH[3], TZUNG-PEI HONG[4]

[1]*Department of Math & Informatics, University of People's Security, Ho Chi Minh City, Vietnam*
duyhaman@yahoo.com
[2]*Faculty of Information Technology, Ho Chi Minh City University of Technology, Vietnam*
bayvodinh@gmail.com
[3]*School of Graduate Studies, Vietnam National University, Hanoi, Vietnam*
minhnth@gmail.com
[4]*Department of Computer Science and Information Engineering, National University of Kaohsiung, Kaohsiung, Taiwan, ROC* tphong@nuk.edu.tw

Abstract. In recent years, methods for mining quantitative databases have been proposed. However, the processing time is fairly much, which affects the productivity of intelligent systems that use quantitative databases. This study proposes the multibit segment (MBiS) structure to store and process tidsets to increase the effeciency of mining frequent weighted utility itemsets (FWUIs) from quantitative databases. With this structure, the calculation of the intersection of tidsets between two itemsets becomes more convenient. Based on this structure, the authors define the MBiS-Tree structure and propose an algorithm for mining FWUIs from quantitative databases. Experimental results for a number of databases show that the proposed method outperforms existing methods.
Keywords. Dynamic bit vector frequent itemset, frequent weighted utility itemset, multibit segment, tidset

1. INTRODUCTION

Mining frequent itemsets (FIs) to find relationships among items plays an important role in data mining, especially for associaiton rules [1] and classification based on association rules [2]. Many algorithms have been proposed to deal with this issue, such as Apriori [1], FP-Growth [3], Charm [4], Eclat [5], and dEclat [6]. These approaches use either a horizontal or vertical data format. Apriori and FP-Growth are two typical algorithms that use the horizontal data format. Eclat [7], which is based on IT-Tree is a typical algorithm that uses the vertical data format. Mining FIs using the horizontal data format is time consuming since the data need to be scanned several times and the determination of FIs is fairly complicated. In contrast, the Eclat approach needs to read the data only once to build the tidsets of 1-itemsets. In the mining of subsequent itemsets, Eclat needs to only calculate the intersection of individual tidsets of itemsets. Therefore, mining FIs using the vertical format has received a lot of attention in recent years. A number of algorithms that use the vertical data format for mining frequent itemsets have thus been proposed Zaki *et al.* [5] used a tidlist and stored tidsets in the form of an array, but this representation of tidsets makes the calculation of tidsets timeconsuming

and requires a lot of memory. Dong *et al.* [8] and Song *et al.* [9] used BitTable to store tidsets, with each tidset being a row that contains |T| bits, where |T| is the number of transactions. The bit value is "1" if the transaction ID appears in the tidset; otherwise it is "0". BitTable significantly improves the mining time and memory usage, since the bit array reduces memory and the calculation of the intersection of tidsets is fast due to the usage of the AND bitwise operation. However, BitTable still contains non-significant "0" bits, so memory usage is not optimized and the calculation is not improved much. Vo *et al.* [10] used the dynamic bit vector (DBV), which removes "0" bytes at the start and end of each tidset. DBV considerably improves calculation time. However, DBV does not remove "0" bytes in the middle of each tidset.

Quantitative databases, commonly used in real-world applications, have attributes such as the quantity and profit (price of each item, for example) of each item in the transaction. Besides, people are interested in the profit of each item rather than their presence in each order the same as binary databases. For example, in a supermarket, a goods order includes the quantity and profit, and the sale of a suitcase may occur less frequently than that of fish sauce, but the former gives a much higher profit per unit sold. Therefore, mining FWUIs from a quantitative database is very practical and has thus attracted a lot of research interest [11–21].

This paper optimizes DBV by removing all "0" bits, creating a multi-bit segment (MBiS), for mining frequent weighted utility itemsets (FWUIs) from quantitative databases. MBiS has the following advantages:

(i) It optimizes tidset storage in memory since no "0" bits and the continuous streams of "1" bits are updated in the reading process of the database;

(ii) The calculation of the intersection of two MBiSs are very fast, since only the beginning and end indices of each segment of "1" bits need to be updated;

(iii) The effectiveness of the proposed method in mining FWUIs from quantitative databases is demonstrated using experiments.

The paper is organized as follows: Section 2 is a background and reviews some related work. Section 3 presents the structure of the proposed MBiS, some definitions, and the algorithm for calculating the intersection of two MBiS's. The usage of MBiS in the mining of itemsets from a quantitative database is presented in Section 4. Section 5 shows the results of applying MBiS to some databases. Section 6 gives the conclusions and suggestions for future work.

2. BACKGROUND AND RELATED WORK

2.1. Quantitative databases

A quantitative database$D$is composed of tuples $\langle T,\ I,\ W\rangle$ where $T = \{t_1,\ t_2,\ldots,\ t_m\}$ is a set of quantitative transactions, $I = \{i_1,\ i_2,\ldots,\ i_n\}$ is a set of items and $W = \{w_1,\ w_2,\ \ldots,\ w_n\}$ is a set of weights (profits or benefits) that correspond to the items in set I. Each quantitative transaction t_k has the format $t_k = \{x_{k1},\ x_{k2},\ \ldots,\ x_{kn}\}$, where x_{ki} denotes the quantity of the i-th item in transaction t_k, $k = 1$ to m.

Example 1. Table 1 shows a quantitative database$D$. The set of items $I = \{A,\ B,\ C,\ D,\ E\}$ and there are a total of six quantitative transactions. The set of weights $W = \{0.6,\ 0.1,\ 0.3,\ 0.9,\ 0.2\}$, as shown in Table 2.

In Table 1, transaction $t_1 = \{1, 1, 0, 4, 1\}$ means that there is one of item A, one of item B, four of item D, one of item E, and none of item C in the transaction.

Transaction	A	B	C	D	E
t_1	1	1	0	4	1
t_2	0	1	3	0	1
t_3	2	1	0	3	2
t_4	3	1	1	0	1
t_5	1	2	2	1	3
t_6	0	1	1	1	0

Table 1: Quantitative database

Item	Weight
A	0.6
B	0.1
C	0.3
D	0.9
E	0.2

Table 2: Weights of items in Table 1

Mining FWUIs from a quantitative database requires determining the support of each itemset. Khan *et al.* [15] defined two useful quantities namely transaction weighted utility (*twu*) and weighted utility support (*wus*), as:

$$twu(t_k) = \frac{\sum\limits_{i_j \in S(t_k)} w_j \times x_{k_{i_j}}}{|S(t_k)|} \tag{1}$$

where $twu(t_k)$ is the transaction weighted utility of transaction t_k, w_j is the quantity of item i_j in transaction t_k, w_j is the weight of item i_j, and $S(t_k)$ is the number of items in transaction t_k

Example 2. *twu* of transactions in database D in example 1:

Tid	Formula	*twu*
t_1	(0.6 + 0.1 + 0.94 + 0.2)/4	1.13
t_2	(0.1 + 0.3 3 + 0.2)/3	0.4
t_3	(0.6 2 + 0.1 + 0.9 3+ 0.2 2)/4	1.1
t_4	(0.6 3 + 0.1 + 0.3 + 0.2)/3	0.6
t_5	(0.6 + 0.1 2 +0.3 2 + 0.9 + 0.2 3)/5	0.58
t_6	(0.1+0.3+0.9)/3	0.43
Sum_twu		4.24

Table 3: *twu*values of transactions in Table 1

wus is caculated as:

$$wus(X) = \frac{\sum\limits_{t_k t(X)} twu(t_k)}{\sum\limits_{t_k T} twu(t_k)} \tag{2}$$

Example 3. With item A in database D in example 1, based on the *twu* values in Table 3, $wus(A)$ is calculated as follows:

$$wus(A) = \frac{twu(1) + twu(3) + twu(4) + twu(5)}{twu(1) + twu(2) + twu(3) + twu(4) + twu(5) + twu(6)} = 0.803\cdot$$

An itemset X is frequent if $wus(X) \geq min - wus$ (*min-wus* is a value set by users). The problem of identifying FWUIs from a quantitative database is the problem of identifying the set of all $X's$ such that $X \subseteq I$ and $wus(X) \geq min - wus$.

Note that the FIs determined using the criterion of *min-wus* satisfy the Apriori property, which means that if $X \subset Y$, then $wus(X) \geq wus(Y)$

2.2. Mining FWUIs from quantitative databases

Erwin *et al.* [19] proposed an efficient algorithm for utility mining using the pattern growth approach [12] to overcome the limitations of existing algorithms based on the candidate generate-and-test approach [22]. The authors introduced a compact data representation named Compressed Transaction Utility tree (CTU-tree) and a new algorithm named CTU-Mine for mining high utility itemsets. The CTU-Tree consists of two parts: (i) ItemTable (it contains all high TWUs (hTWUs): Items are sorted in ascending order of their TWU values. (ii) Compressed Transaction Utility Tree: It stores all transactions of high TWU items along with the quantities of transactions in a compressed form. Based on CTU-tree, the authors proposed CTU-Mine algorithm The proposed algorithm not only uses a pattern growth approach, but also eliminates the expensive second phase of scanning the database to remove the spurious high utility itemsets.

Khan *et al.* [15] presented classical and weighted Association Rule Mining The authors then proposed a framework for weighted utility association rule mining (WUARM). This method uses two factors (transactional utilities and item weights) for extracting FWUIs, which are used for WUARM

Vo *et al.* [14] proposed a data structure called MWIT-Tree for mining FWUIs. This tree structure has many nodes, where each node on the tree has itemset X, $t(X)$ and $wus(X)$ (where $t(X)$ is the tidset of X). Based on the tree structure and the Eclat algorithm [5], the authors proposed the MWIT-FWUI algorithm, which scans the database only once, making it more efficient than Apriori-based methods. However, this algorithm uses a linked-list data structure for storing tidsets, which increases runtime and memory usage.

Lin *et al.* [16] proposed a approach for mining FWUIs from transaction deletion in a dynamic database. The authors presented a fast update high utility itemsets for transaction deletion (FUP-HUI-DEL) algorithm for handling transaction deletion in decremental mining. The FUP2 (Fast UPdated) algorithm [21], which was originally designed for association rules, is adopted in the proposed FUP-HUI-DEL algorithm to reduce the time required for re-processing the whole updated database. The two-phase algorithm [20] is applied to the proposed FUP-HUI-DEL algorithm for preserving the downward closure property to reduce the number of candidates. The proposed approach can be concluded as follows: (i) Two-phase algorithm is used to preserve the downward closure property for reducing the number of candidates in high utility mining. (ii) FUP2 is used to reduce the number of scans of the original database in high utility mining. (iii) The proposed FUP-HUI-DEL algorithm can easily handle transaction deletion in dynamic databases.

2.3. Methods for mining FIs using vertical database format

Methods for mining itemsets use either a horizontal or vertical data format The horizontal data format is often used with the Apriori and FP-Growth algorithms. The vertical data format used with the Eclat algorithm is based on IT-Tree. With the vertical format, the database is scanned only once [5]. The main disadvantage of the Eclat algorithm is high memory usage for storing the tidset of itemsets, and the high processing time for determining the intersection of tidsets, particularly for a large database with millions of transactions.

Zaki *et al.* [5] proposed the Eclat algorithm, which calculates the support of itemsets based on tidset, where $tidset(X)$ is the set of all transaction IDs of itemset X in a database and the support of itemset X is $support(X) = |tidset(X)|$. The authors also showed the calculation of the tidset of itemsets from the intersection of tidsets, i.e. $tidset(XY) = tidset(X) \cap tidset(Y)$. Tidsets are represented in a list format called tidlist. This representation is inefficient when the number of

transactions in a database is large, since a lot of time is required to verify and compare the lists.

Dong *et al.* [8] and Song *et al.* [9] used BitTable is a bitlist to store tidsets. When calculating the intersection of itemset$X$and itemset Y to create itemset Z, we have $bitlist(Z) = bitlist(X) \cap bitlist(Y)$. The bitwise AND operation is used to calculate the intersection of two bitlists. Bitlists of X, Y, and Z all have a length of $\left(\frac{T}{8} + 1\right)$ bytes. The algorithm proposed by Dong *et al.* [8] uses BitTable based on the Apriori algorithm [5] to quickly determine the support of an itemset by computing the number of bits different from 0 in the bitlist of the individual itemset instead of rescaning the database, as done for Apriori.

Vo *et al.* [10] proposed DBV which significantly outperforms BitTable in terms of runtime and memory. For DBV, all "0" bytes at the start and end of each bitlist are removed (no transaction is recorded in "0" bytes), making the bitlist of items more compact. In addition, Vo *et al.* proposed a method that uses an array of constants to quickly calculate the support of an itemset by determining the number of "1" bits in each byte with a value of 0 to 255.

3. REPRESENTATION OF MULTI-BIT SEGMENTS

3.1. Structure of MBiS

MBiS consists of several segments of continuous "1" bits in a bit vector. Each segment includes two components:

(i) Start, which is the beginning index of the segment.
(ii) End, which is the end index of the segment.

An example of a bit vector with 96 bits is shown below.

Example 4. Consider the bit vector shown in Table 4, which represents the bits in a bitlist with 96 elements (12 bytes). The MBiS representation of this bitlist is shown in Table 5.

Bit index	1	2	···	15	16	17	···	35	36	37	···	58	59	60	···	80	81	82	···	96
Bit value	0	0	0	0	1	1	···	1	0	0	···	0	1	1	···	1	0	0	···	0

Table 4: Bit vector with 96 elements

[16, 35]	[59, 80]
Segment 1	Segment 2

Table 5: MBiS representation of bit vector in Table 4

In Table 4, the bit vector requires 96 bits (12 bytes), whereas the MBiS representation requires only 4 bytes to store its two segments, reducing memory usage.

3.2. Definitions

Let $MBiS(X)$ and $MBiS(Y)$ be MBiS's of itemset X and itemset Y, respectively for database D.The following definitions are given.

Definition 1: The MBiS of itemset $X$is a set of segments with continuous "1" bits described as follows:

$$MBiS(X) = \{[S_1, e_1], [S_2, e_2], \ldots, [S_k, e_k]\},$$

where $e_i \geq s_i \forall i \in \{1, 2, \ldots k\}$, and $S_i \geq e_{i-1}\ \forall i \in \{2, 3, \ldots k\}$.

Example 5. In Table 4, the derived MBiS contains two segments, [16, 35] and [59, 80].

Definition 1. An element i belongs to $MBiS(X)$ (i.e. $i \in MBiS(X)$) if there exists a value j such that $S_j \geq i \geq e_j$ with S_j, e_j is a segment of $MBiS(X)$.

Example 6. With $MBiS(X) = \{[16, 35], [59, 80]\}$, it is to say that element 30 belongs to $MBiS(X)$ because $16 \geq 30 \geq 35$ ([16, 35] is a segment of $MBiS(X)$).

Definition 2. The intersection of $MBiS(X)$ and $MBiS(Y)$ is denoted as $MBiS(X) \cap MBiS(Y)$. If $\forall i \in MBiS(X) \cap MBiS(X)$, then $i \in MBiS(X)$ and $i \in MBiS(Y)$ and otherwise.

Remark 1. $MBiS(X) \cap MBiS(Y) = MBiS(X \cup Y)$.

Proof. Let $Z = X \cup Y \Rightarrow$ if $i \in MBiS(Z)$ then $i \in MBiS(X)$ and $i \in MBiS(Y) \Rightarrow i \in MBiS(X) \cap MBiS(Y)$. Otherwise, if $i \in MBiS(X) \cap MBiS(Y)$ then $i \in MBiS(X)$ and $i \in MBiS(Y) \Rightarrow i \in MBiS(Z)$. Therefore, $MBiS(X) \cap MBiS(Y) = MBiS(Z) = MBiS(X \cup Y)$. □

Example 7. For $MBiS(X) = \{[5, 20], [35, 50]\}$ and $MBiS(Y) = \{[15, 25], [40, 60]\}$, their intersection is $\{[15, 20], [40, 50]\}$.

Definition 3. For itemset X, the index of "1" bits in $MBiS(X)$ is the transaction ID of X in the database; therefore:

$$wus(X) = \frac{\sum_{i \in MBiS(X)} twu(i)}{Sum\_twu} \tag{3}$$

where $Sum\_twu = \sum_{i=1}^{n} twu[i]$ with n is the number transactions in the database.

Example 8. With item C of database D in example 1, $MBiS(C) = \{[2], [4, 6]\} \Rightarrow tidset(C) = \{2, 4, 5, 6\}$; therefore:

$$wus(C) = (twu(2) + twu(4) + twu(5) + twu(6))/(Sum\_twu) = 0.475.$$

3.3. Algorithm for determining intersection of two *MBiS*'s

The problem of mining FIs using the vertical data format requires the calculation of the bitlist intersection of two individual itemsets to find the final bitlist of the created itemset. The calculation of the intersection of two tidsets is the calculation of the intersection of two individual *MBiS*'s. Since *MBiS* contains several segments of continuous "1" bits, it is only needed to determine the intersections of segments of *MBiS*s when calculating the intersection. This is performed by calculating the start and end of each result segment. The algorithm for calculating the intersection of two *MBiS*'s is described in Figure 1.

The use of the intersection algorithm is illustrated through the following example.

Example 9. $MBiS(X) = \{[6, 20], [25, 43], [43, 60]\}$ and $MBiS(Y) = \{[22, 40], [48, 66]\}$. $Z = MBiS(X) \cap MBiS(Y) = \{[25, 40], [48, 60]\}$. Segment $[25, 43]$ in $MBiS(X)$ and segment $[20, 40]$ in $MBiS(Y)$ are used to create the new segment [25, 40] (max(25, 20) = 25 and min(43, 40) = 40). Similarly, segment $[43, 60]$ in $MBiS(X)$ and segment $[48, 66]$ in $MBiS(Y)$ are used to create the new segment $[48, 60]$.

Algorithm 1. Intersection Algorithm

```
Input:
    - MBiS(X) with n1 segments. Segment i starts with Si and ends
    with Ei
    - MBiS(Y) with m1 segments. Segment j starts with Sj and ends
    with Ej
Output: Z = MBiS(X)∩MBiS(Y)
Method name: INTERSECTION(MBiS(X), MBiS(Y))
1   INTERSECTION(MBiS(X), MBiS(Y))
2      Z = ∅;
3      While (i ≤ n1 && j ≤ m1)
4          Let segment i1 on MBiS(X) and segment j1 on MBiS(Y)
           be the first intersection segments from segment i on
           MBiS(X) and segment j on MBiS(Y), respectively;
5          Start = Max(MBiS(X)[i1].start, MBiS(Y)[j1].start);
6          End = Min(MBiS(X)[i1].end, MBiS(Y)[j1].end);
7          Z = Z ∪{[Start,End]};
8          if (End = MBiS(X)[i1].end)
9             i = i1 + 1;// move to the next segment on MBiS(X)
10            j = j1;
11         else
12            j = j1 + 1;// move to the next segment on MBiS(Y)
13            i = i1;
14     return Z;//Z is the result of MBiS(X)∩MBiS(Y)
```

Figure 1: Intersection algorithm.

The complexity of the intersection algorithm in Figure 1 is only

$O(n_1 + m_1)$, where n_1 and m_1 are the numbers of segments in $MBiS(X)$ and $MByS(Y)$, respectively. To determine the intersection of two itemsets, it is needful only to determine the Max and Min of the start and end of the intersection segments. Therefore, the runtime of the intersection algorithm is lower than those of algorithms that use other structures [2, 7, 8, 11].

4. FAST ALGORITHM FOR MINING FWUIs FROM QUANTITATIVE DATABASES

4.1. *MBiS*-Tree

A data structure, called *MBiS*-Tree, is used to mine FWUIs from a quantitative database Each node on an *MBiS*-Tree includes three components, namely X, $MBiS(X)$, and $wus(X)$, where:

- X is an itemset
- $MBiS(X)$ is the $MBiS$ of X, and
- $wus(X)$ is the wus of$X$.

To connect nodes X and Y to create a new node $Y \cup X$, X and Y must have the same length and the same prefix, with length $|X| - 1$ items, and $wus(X \cup Y) \geq min - wus$ where $min - wus$ is set by users.

4.2. Mining FWUIs from quantitative database using *MBiS*-Tree

To mine FWUIs from a quantitative database, *wus* must be calculated for each itemset using equation (3) Therefore, it is necessary to determine the tidlist of the itemset. This is equal to determining the index set of "1" bits in an *MBiS*. The algorithm for calculating *wus* is shown in Figure 2.

Algorithm 2. *wus* Calculation Algorithm

```
Input: MBiS(X)
Output: wus value of itemset X
Method name: WUS_CALCULATION()
1   WUS_CALCULATION(MBiS(X))
2   for all i ∈ MBiS(X)
3      for all j ∈ segment i of MBiS(X)
4        s = s + twu[j];
5   return s/Sum_twu;
6
```

Figure 2: *wus* calculation algorithm.

With the *MBiS* structure, *wus* is simple to compute. Because the complexity of determining a transaction IDs of an itemset is O(1), the complexity of $WUS\_CALCULATION$ is $O(k)$, where k is the number of transactions of itemset X. On other way, k is thus a total of "1" bits in $MBiS(X)$. The runtime of $WUS\_CALCULATION(MBiS(X))$ thus depends on the number "1" bits in $MBiS(X)$.

Each level in an *MBiS*-Tree has a number of equivalence classes. Each equivalence class has the same parents in the previous level (the equivalence class in the first level, denoted by $[\emptyset]$, contains all FWUIs). Itemsets $X_1X_2 \ldots X_kX_{k+1}$ and $X_1X_2 \ldots X_kX_{k+2}$ have the same equivalence class, denoted by $[X_1X_2 \ldots X_k]$. A k-itemset in an *MBiS*-Tree is then created from two (k-1)-itemsets that have the same equivalence class. These k-itemsets are used to create the equivalence class of the (k+1)-itemsets. An itemset is inserted into an *MBiS*-Tree if its $wus \geq min - wus$, a user-defined threshold. The itemsets in an *MBiS*-Tree are thus FWUIs that satisfy the *min-wus* constraint. This process is shown in Figure 3.

The input of the MBiS_FWUI algorithm is a quantitative database and *min-wus*. A quantitative database includes two tables, as in Example 1. First the *wus* values of 1-itemsets are calculated The next $[\emptyset]$ is equal to all 1-itemsets whose *wus* values satisfy *min-wus* (line 2). On line 3, the *MBiS*-Tree is initialized using the empty set. The input of the FWUI_Mine function is the equivalence class $P([P])$ (line 5). On line 6, the itemsets of the equivalence class $[P]$ that satisfy *min-wus* are added to *MBiS*-Tree. On lines 7 to 10, each l_i is connected with all l_j following $l_i (X = l_i \cup l_j,\ l_i, l_j \in [P])$. On lines 11 and 12, $MBiS(X)$ is determined using the intersection algorithm (Figure 1) and $wus(X)$ is calculated using the *wus* calculation algorithm (Figure 2). Line 14 adds node $\{X, Y,\ wus(X)\}$, which satisfies *min-wus*, into P_i. On the last line, the FWUI_Mine($[P_i]$) function is called recursively.

The MBiS-FWUI algorithm 3 creates an MBiS-Tree whose elements are FWUIs obtained under the constraint of *min-wus*. Below, an example is given to illustrate the proposed idea.

Example 10. Consider database D in Tables 1 and 2 with *min-wus* = 0.4. Table 6 shows the *MBiS* representations of the 1-itemsets belonging to database D in example 1.

For $min - wus = 0.4$ the MBiS-FWUI algorithm builds an MBiS-Tree, as shown in Figure 4

Algorithm 3. MBiS-FWUI Algorithm

```
Input: Quantitative database D and min-wus;
Output: MBiS-Tree containing all FWUIs that satisfy min-wus from D
Method name: MBiS_FWUI();
1   MBiS_FWUI()
2     [∅] = ⟨j∈I| wus(j) ≥ min-wus⟩;
3     MBiS-Tree = ∅;
4     FWUI_Mine([∅]);
5     FWUI_Mine([P])
6           MBiS-Tree = MBiS-Tree ∪ [P];
7           for all li∈[P] do
8              [Pi] = ∅;
9              for all lj ∈ [P], with j > i do
10                   X  = li ∪ lj;
11                   Y  = INTERSECTION(MBiS(li), MBiS(lj));
12                   wus = WUS_CALCULATION(Y);
13                   if (wus ≥ min-wus)
14                        [Pi] = [Pi] ∪ {( X, Y, wus(X)};
15       FWUI_Mine([Pi]);
```

Figure 3: *MBiS-FWUI* algorithm.

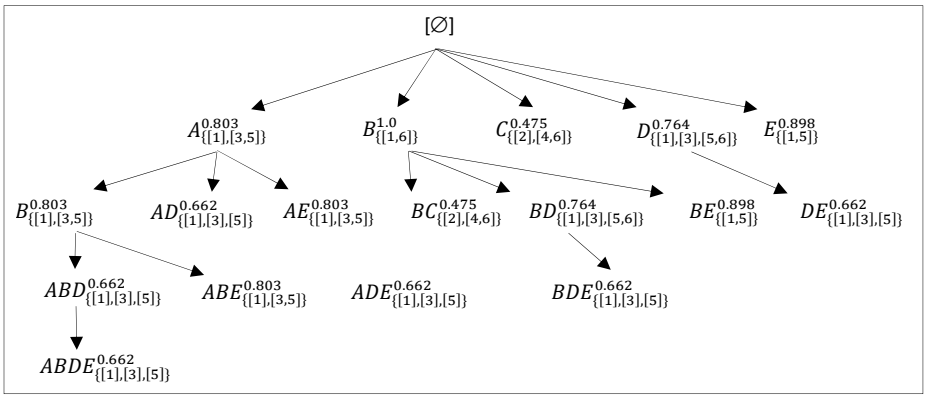

Figure 4: MBiS-Tree built for database D.

Item	Tidset	MBiS
A	1, 3, 4, 5	{[1], [3, 5]}
B	1, 2, 3, 4, 5, 6	{[1, 6]}
C	2, 4, 5, 6	{[2], [4, 6]}
D	1, 3, 5, 6	{[1], [3], [5, 6]}
E	1, 2, 3, 4, 5	{[1, 5]}

Table 6: *MBiS* representations of 1-itemsets in Table 1

All 1-itemsets $\langle A, B, C, D, E\rangle$ that satisfy the *min-wus* threshold are added into the *MBiS*-Tree. Consider the two items A and B $MBiS(A) \cap MBiS(B) = \{[1], [3, 5]\} \cap \{[1, 6]\}$ and thus $MBiS(AB) = \{[1], [3, 5]\}$. Based on the *wus* calculation algorithm, there exists $wus(AB) = wus(A) = 0.803 > min - wus$. Thus, AB is added into the MBiS-Tree.

Next consider items A and C. $MBiS(A) \cap MBiS(C) = \{[1], [3, 5]\} \cap \{[1], [4, 6]\}$, and thus $MBiS(AC) = \{[4, 5]\}$. Because $wus(AC) = 0.278 > min - wus$, AC is not added into the *MBiS*-Tree.

Next consider items A and D. $MBiS(A) \cap MBiS(D) = \{[1], [3, 5]\} \cap \{[1], [3], [5, 6]\}$, and thus $MBiS(AD) = \{[1], [3], [5]\}$. $wus(AD) = 0.662 > min - wus$. AD is thus added into the *MBiS*-Tree.

Similar to the above cases, the other items are considered to construct the *MBiS*-Tree in Figure 4. The common FWUI set with *min-wus* = 0.4 is $\langle A, B, C, D, E, AB, AD, AE, BC, BD, BE, DE, ABD, ABE, ADE, BDE, JABDE\rangle$.

5. EXPERIMENTAL RESULTS

This section compares methods based on *MBiS* and DBV in terms of mining time and memory usage for seven databases, namely RETAIL, MBS-POS, SALE_FACT_1997, SALE_FACT_SYNC, CHESS, ACCIDENTS and CONNECT. SALE_FACT_1997 and SALE_FACT_SYNC are databases from Microsoft Foodmart2000 for Microsoft SQL2000 (SALE_FACT_SYNC is the combination of sales_fact_1997, sales_fact_1998 and sales_fact_dec_1998 in Foodmart2000). Table 7 shows the characteristics of the experimental databases. All the experiments were performed with C# on a personal computer with an Intel Haswell Core i5 1.4-GHZ CPU and 4 GB of RAM running Microsoft Windows 8.1.

Database	Number of items	Number of transtractions	Remark
BMS-POS	1.657	515.597	Modified
RETAIL	16.470	88.162	Modified
SALE_FACT_1997	1.753	20.522	
SALE_FACT_SYN	1.753	58.308	
ACCIDENTS	468	340.183	Modified
CONNECT	130	67.557	Modified
CHESS	76	3.196	Modified

Table 7: Characteristics of databases used in experiments

The databases are modified by adding a random value in the range of 1 to 10 for each item corresponding to its quantity in each transaction, and one more table is created to store the weight values of items (in the range of 1 to 10).

Figures 5-11 show that *MBiS* is effective for sparse databases (RETAIL, BMS-POS, SALE_FACT_1997, and SALE_FACT_SYNC). For example, with *min-wus* = 0.1, the processing times for *MBiS* are 4.7 and 2.73 times lower and memory usage is 5.86 and 1.96 times lower than those for DBV for RETAIL and BMS-POS databases, respectively. For dense databases (ACCIDENTS, CONNECT, and CHESS), DBV is slightly more effecient. For instance, with *min-wus* = 0.9, the processing times for DBV are 1.18 and 1.34 times lower and memory usage is 1.39 and 1.09 times lower than those for *MBiS* for CONNECT and CHESS databases, respectively.

For sparse databases (low number of items in each transaction), the number of "1" bits is low and thus the *MBiS* has small segments, leading to low runtimes for the intersection and *wus* calculation algorithms. In contrast, dense databases include many items in transactions, so the elimination of "0" bits is not effective because the memory required for segment indices is large, and thus the calculation is slow.

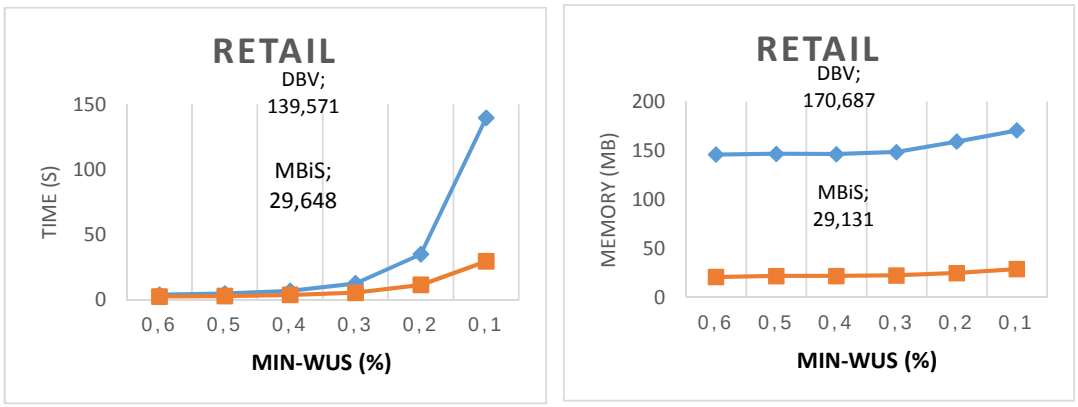

Figure 5: Comparison of runtime (left) and memory usage (right) for RETAIL database.

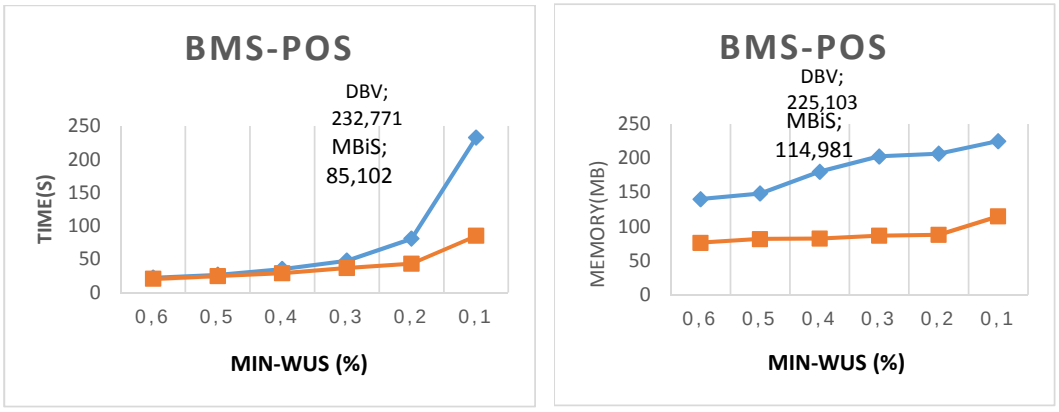

Figure 6: Comparison of runtime (left) and memory usage (right) for BMS-POS database.

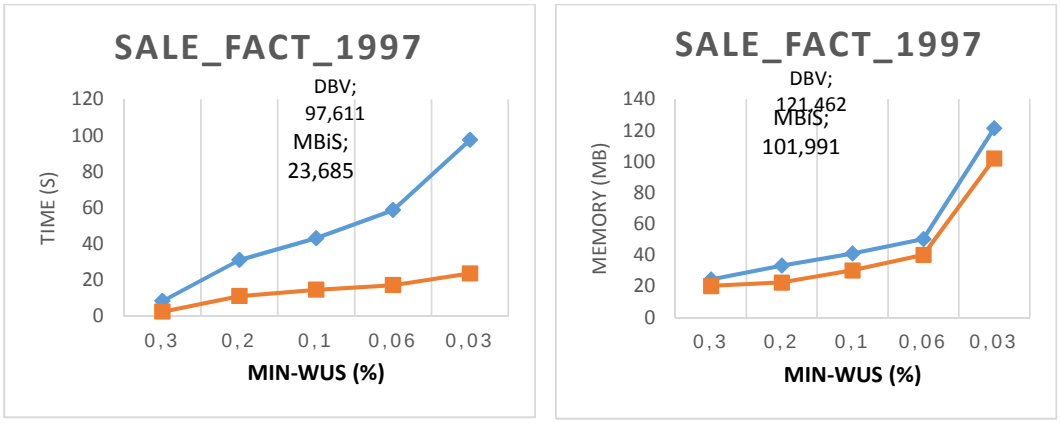

Figure 7: Comparison of runtime (left) and memory usage (right) for SALE_FACT_1997 database.

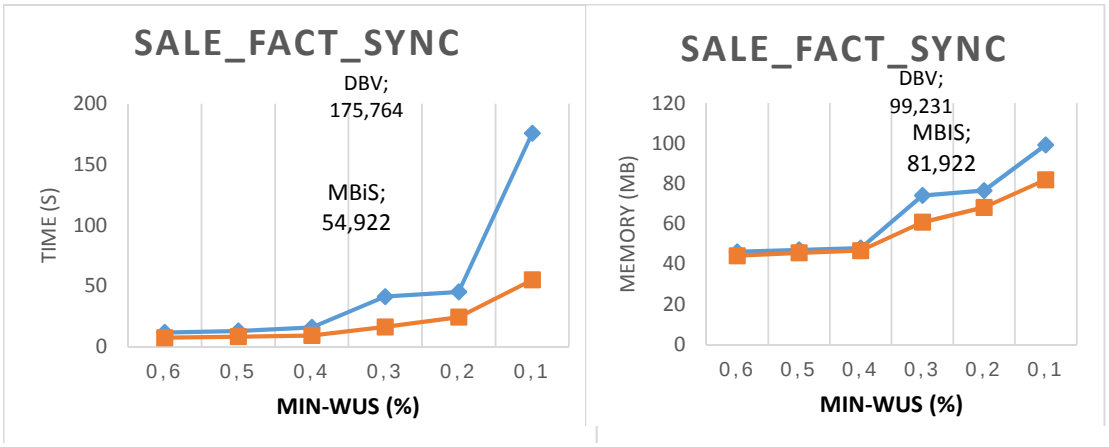

Figure 8: Comparison of runtime (left) and memory usage (right) for SALE_FACT_SYNC database.

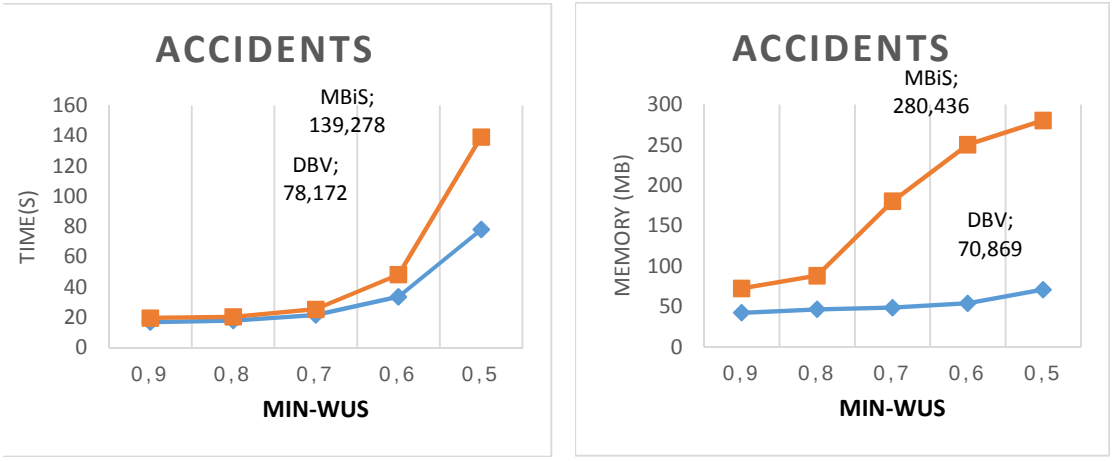

Figure 9: Comparison of runtime (left) and memory usage (right) for ACCIDENTS database.

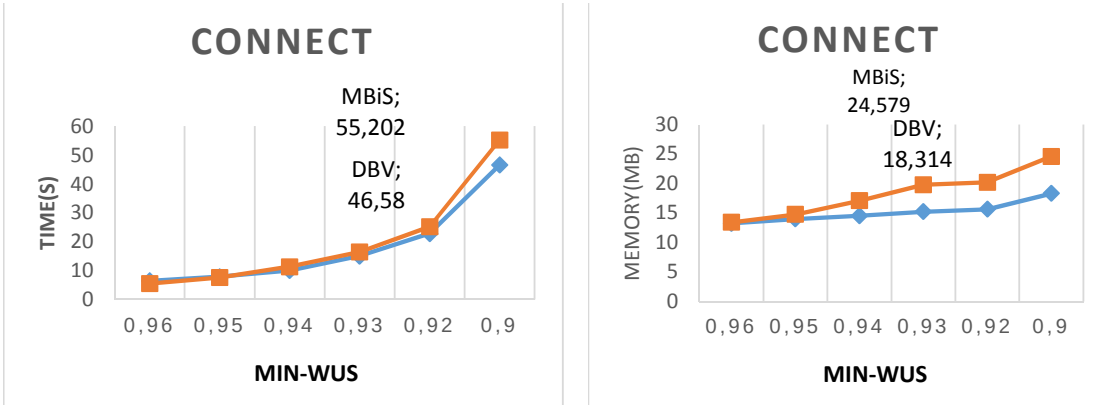

Figure 10: Comparison of runtime (left) and memory usage (right) for CONNECT database.

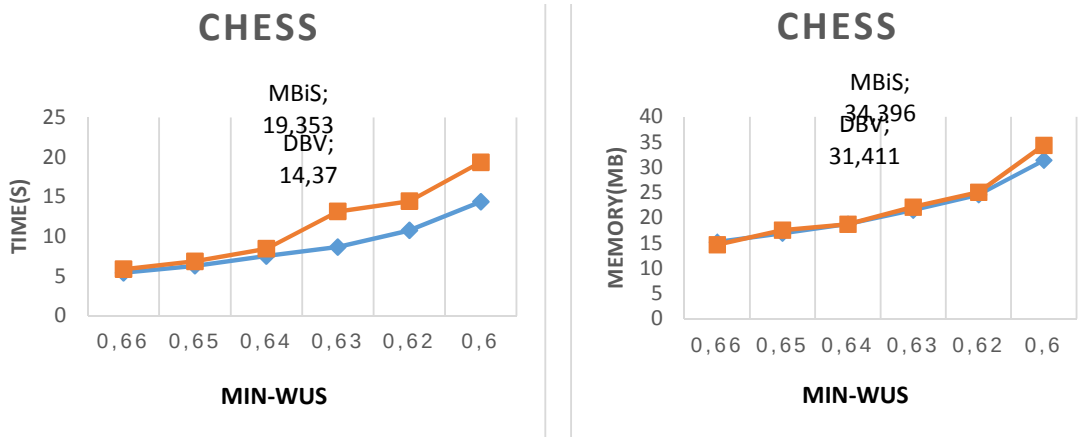

Figure 11: Comparison of runtime (left) and memory usage (right) for CHESS database.

6. CONCLUSION AND FUTURE WORK

This paper proposes an effective approach for storing tidset information when mining FWUIs from quatitative databases. *MBiS* is used to remove all "0" bits; only "1" bits are stored in the bit vector. *MBiS* thus reduces the runtime and memory usage for mining FWUIs from quantitative databases. The experimental results for seven databases show that the proposed approach outperforms an existing approach in terms of memory usage and runtime. When the *min-wus* threshold is small, *MBiS* is especially effective. However, the proposed approach is not effective when mining FWUs from dense databases. In future work, the authors will further study the problem of mining frequent weighted itemsets from weighted and hierarchical databases.

REFERENCES

[1] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215, 1994, pp. 487–499.

[2] B. Vo and B. Le, "A novel classification algorithm based on association rules mining," in *Knowledge Acquisition: Approaches, Algorithms and Applications*. Springer, 2009, pp. 61–75.

[3] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *Proc. ACMSIGMOD Int. Conf. Management of Data (SIGMOD00), Dallas, TX*, 2000, pp. 1–12.

[4] S. Mousa, "Comparison between riss and dcharm for mining gene expression data." *International Journal of Data Mining & Knowledge Management Process*, vol. 3, no. 5, pp. 53–59, 2013.

[5] M. J. Zaki, S. Parthasarathy, M. Ogihara, W. Li *et al.*, "New algorithms for fast discovery of association rules." in *Third International Conference on Knowledge Discovery and Data Mining (KDD)*, vol. 97, 1997, pp. 283–286.

[6] M. J. Zaki and K. Gouda, "Fast vertical mining using diffsets," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2003, pp. 326–335.

[7] M. J. Zaki and C.-J. Hsiao, "Efficient algorithms for mining closed itemsets and their lattice structure," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 17, no. 4, pp. 462–478, 2005.

[8] J. Dong and M. Han, "Bittablefi: An efficient mining frequent itemsets algorithm," *Knowledge-Based Systems*, vol. 20, no. 4, pp. 329–335, 2007.

[9] W. Song, B. Yang, and Z. Xu, "Index-bittablefi: An improved algorithm for mining frequent itemsets," *Knowledge-Based Systems*, vol. 21, no. 6, pp. 507–513, 2008.

[10] B. Vo, T.-P. Hong, and B. Le, "Dbv-miner: A dynamic bit-vector approach for fast mining frequent closed itemsets," *Expert Systems with Applications*, vol. 39, no. 8, pp. 7196–7206, 2012.

[11] G.-C. Lan, T.-P. Hong, and V. S. Tseng, "Discovery of high utility itemsets from on-shelf time periods of products," *Expert Systems with Applications*, vol. 38, no. 5, pp. 5851–5857, 2011.

[12] B. Le, H. Nguyen, T. A. Cao, and B. Vo, "A novel algorithm for mining high utility itemsets," in *Intelligent Information and Database Systems, 2009. ACIIDS 2009. First Asian Conference on.* IEEE, 2009, pp. 13–17.

[13] C.-W. Lin, G.-C. Lan, and T.-P. Hong, "An incremental mining algorithm for high utility itemsets," *Expert Systems with Applications*, vol. 39, no. 8, pp. 7173–7180, 2012.

[14] B. Vo, B. Le, and J. J. Jung, "A tree-based approach for mining frequent weighted utility itemsets," in *Computational Collective Intelligence. Technologies and Applications.* Springer, 2012, pp. 114–123.

[15] M. S. Khan, M. Muyeba, and F. Coenen, "A weighted utility framework for mining association rules," in *Computer Modeling and Simulation, 2008. EMS'08. Second UKSIM European Symposium on.* IEEE, 2008, pp. 87–92.

[16] C.-W. Lin, G.-C. Lan, and T.-P. Hong, "Mining high utility itemsets for transaction deletion in a dynamic database," *Intelligent Data Analysis*, vol. 19, no. 1, pp. 43–55, 2015.

[17] T. Lu, B. Vo, H. T. Nguyen, and T.-P. Hong, "A new method for mining high average utility itemsets," in *Computer Information Systems and Industrial Management.* Springer, 2014, pp. 33–42.

[18] V. S. Tseng, C.-J. Chu, and T. Liang, "Efficient mining of temporal high utility itemsets from data streams," in *UBDM'06, Philadelphia Pennsylvania USA*, 2006.

[19] A. Erwin, R. P. Gopalan, and N. Achuthan, "CTU-Mine: an efficient high utility itemset mining algorithm using the pattern growth approach," in *Computer and Information Technology, 2007. CIT 2007. 7th IEEE International Conference on.* IEEE, 2007, pp. 71–76.

[20] Y. Liu, W.-k. Liao, and A. Choudhary, "A two-phase algorithm for fast discovery of high utility itemsets," in *Advances in Knowledge Discovery and Data Mining.* Springer, 2005, pp. 689–695.

[21] D. W. Cheung, S. D. Lee, and B. Kao, "A general incremental technique for maintaining discovered association rules," in *The International Conference on Database Systems for Advanced Applications.* IEEE., 1997, pp. 185–194.

[22] G. Grahne and J. Zhu, "Fast algorithms for frequent itemset mining using fp-trees," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 17, no. 10, pp. 1347–1362, 2005.

Received on November 02 - 2014
Revised on March 15 - 2015