Expert Systems with Applications 39 (2012) 7196–7206

Contents lists available at SciVerse ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

DBV-Miner: A Dynamic Bit-Vector approach for fast mining frequent closed itemsets

Bay Vo [a,*], Tzung-Pei Hong [b,c], Bac Le [d]

[a] Department of Computer Science, Ho Chi Minh City University of Technology, Ho Chi Minh, Viet Nam
[b] Department of Computer Science and Information Engineering, National University of Kaohsiung, Kaohsiung, Taiwan, ROC
[c] Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung, Taiwan, ROC
[d] Department of Computer Science, University of Science, Ho Chi Minh, Viet Nam

ARTICLE INFO

Keywords:
BitTable
Data mining
Dynamic Bit-Vector
Frequent closed itemsets
Vertical data format

ABSTRACT

Frequent closed itemsets (FCI) play an important role in pruning redundant rules fast. Therefore, a lot of algorithms for mining FCI have been developed. Algorithms based on vertical data formats have some advantages in that they require scan databases once and compute the support of itemsets fast. Recent years, BitTable (Dong & Han, 2007) and IndexBitTable (Song, Yang, & Xu, 2008) approaches have been applied for mining frequent itemsets and results are significant. However, they always use a fixed size of Bit-Vector for each item (equal to number of transactions in a database). It leads to consume more memory for storage Bit-Vectors and the time for computing the intersection among Bit-Vectors. Besides, they only apply for mining frequent itemsets, algorithm for mining FCI based on BitTable is not proposed. This paper introduces a new method for mining FCI from transaction databases. Firstly, Dynamic Bit-Vector (DBV) approach will be presented and algorithms for fast computing the intersection between two DBVs are also proposed. Lookup table is used for fast computing the support (number of bits 1 in a DBV) of itemsets. Next, subsumption concept for memory and computing time saving will be discussed. Finally, an algorithm based on DBV and subsumption concept for mining frequent closed itemsets fast is proposed. We compare our method with CHARM, and recognize that the proposed algorithm is more efficient than CHARM in both the mining time and the memory usage.

1. Introduction

Mining frequent itemsets (FI) is a one of the most important task in mining association rules. However, mining association rules from FI will generate a lot of redundant rules (Bastide, Pasquier, Taouil, Stumme, & Lakhal, 2000; Pasquier, Bastide, Taouil, & Lakhal, 1999a, 1999b; Zaki, 2000, 2004). Whereas, mining association rules from FCI will overcome the disadvantage of this problem. Therefore, a lot of algorithms for mining FCI have been proposed such as Close (Pasquier et al., 1999b), A-Close (Pasquier et al., 1999a), Closet (Pei, Han, & Mao, 2000), Closet+ (Wang, Han, & Pei, 2003), CHARM (Zaki & Hsiao, 2005), etc. They can be divided into two categories according to the database format: horizontal and vertical. Vertical-based approaches often scan the database once and base on the divide-and-conquer strategy for fast mining FI/FCI. We can list some of them.

(i) Tidlist (Zaki & Hsiao, 2005; Zaki, Parthasarathy, Ogihara, & Li, 1997): the database is transformed into item × Tidset form. A new itemset from two itemsets $X \times \text{Tidset}(X)$ and $Y \times \text{Tidset}(Y)$ is $(X \cup Y) \times (\text{Tidset}(X) \cap \text{Tidset}(Y))$. The support of itemset X is the cardinality of Tidset(X).

(ii) BitTable (Dong & Han, 2007; Song, Yang, & Xu, 2008): the database is transformed into item × Bit-Vector form. A new itemset from two itemsets $X \times \text{Bit-Vector}(X)$ and $Y \times \text{Bit-Vector}(Y)$ is $(X \cup Y) \times (\text{Bit-Vector}(X) \cap \text{Bit-Vector}(Y))$, where $\text{Bit-Vector}(X) \cap \text{Bit-Vector}(Y)$ is computed by using AND operation of each bit in Bit-Vectors. The support of an itemset X is the number of bits 1 in Bit-Vector(X).

All above approaches must store the whole database in main memory. When the number of transactions is large, it is difficult to store them in main memory and therefore, we must mine in secondary memory and lead to consume more time.

In this paper, we propose a new approach for mining FCI based on Dynamic Bit-Vector. Our approach has some advantages: (i) Fast computing the intersection of two DBVs and the support of itemsets; (ii) It overcomes the disadvantage of CHARM in that it checks

* Corresponding author. Tel.: +84 089744186.
E-mail addresses: vdbay@hcmhutech.edu.vn (B. Vo), tphong@nuk.edu.tw (T.-P. Hong), lhbac@fit.hcmus.edu.vn (B. Le).

0957-4174/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
doi:10.1016/j.eswa.2012.01.062

whether a generated itemset is closed or not immediately whenever this itemset is created, and it need not use hash table to check non-closed itemsets; (iii) The subsumption concept is used for pruning search space. With subsumption concept, itemsets that have the same transaction identifiers will be subsumed into an itemset. Hence, we can save the memory for storage and the time for computing the intersection of DBVs.

The rest of this paper is follows: Section 2 presents the related work. Section 3 introduces Dynamic Bit-Vector, some definitions are also presented in this section, and an algorithm for getting the intersection between two DBVs is proposed. Some theorems are developed in Section 4 and based on them, the method for mining FCI will be proposed. Section 5 presents experimental results. We conclude our work in Section 6.

2. Related work

2.1. Mining frequent closed itemsets

Frequent itemsets play an important role in the mining process. A frequent itemset can be formally defined as follows. Let D be a transaction database and I be the set of items in D. The support of an itemset X ($X \subseteq I$), denoted $\sigma(X)$, is the number of transactions in D containing X. X is called a frequent itemset if $\sigma(X) \geqslant minSup$, where $minSup$ is a predefined minimum support threshold.

Frequent closed itemsets are a variant of frequent itemsets for reducing rule numbers. Formally, an itemset X is called a frequent closed itemset if it is frequent, and it does not exist any frequent itemset Y such that $X \subset Y$ and $\sigma(X) = \sigma(Y)$. There are many methods proposed for mining FCI from databases. They could be divided into the following four categories (Lee, Wang, Weng, Chen, & Wu, 2008; Yahia, Hamrouni, & Nguifo, 2006):

(i) Generate-and-test approaches: they are mainly based on the Apriori algorithm and use the level-wise approach to discover FCI. Some examples are Close (Pasquier et al., 1999b) and A-Close (Pasquier et al., 1999a).
(ii) Divide-and-conquer approaches: they adopt the divide-and-conquer strategy and use compact data structures extended from the frequent-pattern (FP) tree to mine FCI. Examples include Closet (Pei et al., 2000), Closet+ (Wang et al., 2003) and FPClose (Grahne & Zhu, 2005).
(iii) Hybrid approaches: they integrate both two strategies above to mine FCI. They firstly transform the database into the vertical data format. These approaches develop properties and use the hash table to prune non-closed itemsets. CHARM (Zaki & Hsiao, 2005) and CloseMiner (Singh, Singh, & Mahanta, 2005) also belong to them.
(iv) Hybrid approaches without duplication: these approaches differ from the hybrid ones in not using the subsumption-checking technique, such that FCI need not be stored in the main memory. They don't use the hash-table technique as CHARM (Zaki & Hsiao, 2005). DCI-Close (Lucchese, Orlando, & Perego, 2006), LCM (Uno, Asai, Uchida, & Arimura, 2004) and PGMiner (Moonestinghe, Fodeh, & Tan, 2006) also belong to this category.

2.2. Vertical data format

In mining FI and FCI, if we are interesting in data format, there are two main kinds: horizontal and vertical data formats. Algorithms for mining frequent itemsets based on the vertical data format are usually more efficient than that on the horizontal data format (Dong & Han, 2007; Song et al., 2008; Zaki, 2000, 2004; Zaki & Hsiao, 2005; Zaki et al., 1997), because they often scan the database once and compute fast the support of itemsets. The disadvantage is that it consumes more memory for storing Tidsets (Zaki, 2000, 2004; Zaki & Hsiao, 2005; Zaki et al., 1997), Bit-Vector (Dong & Han, 2007; Song et al., 2008).

Some typical algorithms based on the vertical data format are summarized as follows:

(i) Eclat (Zaki et al., 1997): proposed by Zaki *et al.* in 1997. Authors used Galois connection to propose an efficient algorithm for mining frequent itemsets. We can see that the support of an itemset is cardinality of Tidset. Thus, we can compute the support of X fast by using Tidset(X), $\sigma(X) = |\text{Tidset}(X)|$. Author also proposed the way of computing Tidset(XY) by using the intersection between Tidset(X) and Tidset(Y), i.e., $\text{Tidset}(XY) = \text{Tidset}(X) \cap \text{Tidset}(Y)$. Some improvements of Eclat also proposed in Zaki and Hsiao (2005). In this paper, the divide-and-conquer strategy was used to mine fast frequent itemsets.
(ii) BitTable-FI (Dong & Han, 2007): Another way of data compressing is that it represents the database in form of BitTable, each item occupies in $|T|$ bits, where $|T|$ is number of transactions in D. When we create a new itemset XY from two itemsets X and Y, Bit-Vector of XY will be computed by using Bit-Vector of items in XY (by getting the result of AND operation between 2 bytes in Bit-Vectors). Because the cardinalities of two Bit-Vectors are the same, the result will be a Bit-Vector with the length of $|T|/8 + 1$ bytes. The algorithm for mining frequent itemsets in Dong and Han (2007) is based on Apriori (Agrawal & Srikant, 1994). The different point is in computing the support, the Apriori algorithm computes the support by re-scanning the database, while BitTable-FI only computes the intersection in Bit-Vectors. The support of outcome itemset can be computed faster by computing the number of bits 1 in Bit-Vector. Another improvement of BitTable-FI was proposed in Song et al. (2008). Authors base on the "subsume" concept to subsume items and propose Index-BitTableFI for mining frequent itemsets. The way of subsuming is: initially, items are sorted by increasing order according to their supports. For considering each item i with all successive items (in sorted order), if the Bit-Vector of item ith is subset of the Bit-Vector of item jth, then the item jth belongs to the "subsume" set of ith. Mining frequent itemsets based on the "subsume" concept has improved significantly the time of mining frequent itemsets in comparison with BitTable-FI.

3. Dynamic Bit-Vector

As mentioned above, Bit-Vector of each itemset always occupies the fixed size corresponding to the number of transactions in the given database, so it consumes more memory and the time of computing the intersection between Bit-Vectors. In practice, the Bit-Vector of an itemset that contains many bits 0 can be removed to reduce the space and time. Thus, we develop the Dynamic Bit-Vector approach to solve above problem.

3.1. DBV data structure

Each DBV has two elements:

- pos: the position of the first non-zero byte in Bit-Vector.
- Bit-Vector: a list of bytes in Bit-Vector after removing 0 bytes from head and tail.

Example 1. Given a Bit-Vector (in decimal format) as shown in Fig. 1.

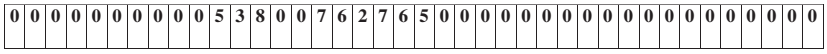

Fig. 1. An example of a bit vector with 40 bytes.

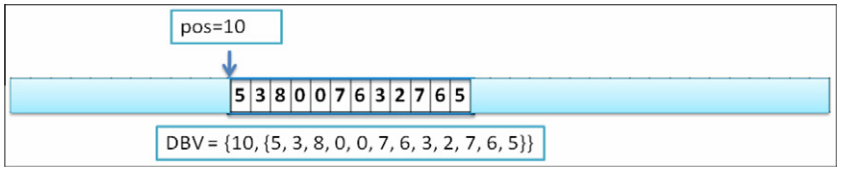

Fig. 2. A representation of DBV for the bit vector in Fig. 1.

For the example above, BitTable needs 40 bytes to store (Fig. 1), while DBV only consumes 14 bytes (12 bytes for the bit vector and 2 bytes for the position as in Fig. 2). The DBV scheme thus needs less memory than the original BitTable approach.

3.2. Algorithm for getting the intersection between two DBVs

The idea of the algorithm: Initially, from the greater position value in two position values of two DBVs, performing AND operation between 2 bytes from this position, if the result value is 0, then the pos value is increased by 1 until reaching the first non-zero value. Next, from this position, keep performing AND operation for each byte of Bit-Vectors until values of the rest bytes is 0, we have the outcome DBV.

Getting the intersection between two DBVs will be performed by the following example.

Example 2. Assume that we want to compute the intersection between 2 DBVs {10,{5,3,8,0,0,7,6,3,2,7,6,5}} and {13,{4,3,0,1,0,4,6,0,0,5,1,3}} (Fig. 3). Because 13 > 10, it implies pos = 13. At position 13, we have 0&4 = 0, so pos = 14. Similarly, 0&3 = 0, 7&0 = 0, 6&1 = 0, 3&0 = 0, 2&4 = 0, so pos = 19. Next, because of 7&6 = 6 ≠ 0, outcome Bit-Vector will be 6. Because the rest bytes are 0, the outcome DBV will be {19,{6}}.

From the example above, the pseudo code for getting the intersection of two DBVs is presented as follows.

Input: Two DBVs {pos_1, Bit-Vector$_1$} and {pos_2, Bit-Vector$_2$}.
Output: DBV = {pos, Bit-Vector}.
Method: See Fig. 4.

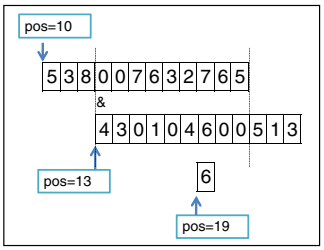

Fig. 3. An example for getting the intersection between two DBVs.

Fig. 4 presents a pseudo code for computing the intersection between two DBVs. First of all, it finds a maximal position between two positions (line 1). Then, it finds the positions (i and j) that have the intersection of two Bit-vectors (lines 2 and 3). From these positions, using AND operator to find the first byte that is non-zero (lines 5–7). Besides, the pos variable will increase by 1 if the result of AND operator is zero (finding the first position that is non-zero for the DBV result) and the count variable decrease by 1 (the number of bytes of the intersection decreases by 1). $i1$ and $j1$ are the last positions that can be had the intersection between two Bit-vectors (line 8). From these positions, using the AND operator to find the last position that is non-zero (lines 9–11). The rest of bytes belong to the intersection between two Bit-vectors (lines 12–14).

3.3. Some definitions and corollaries

Definition 3.1. Given two DBVs, DBV$_1$ = {pos_1,Bit-Vector$_1$} and DBV$_2$ = {pos_2,Bit-Vector$_2$}, DBV$_1$ is called a subset of DBV$_2$, denote DBV$_1$ ⊆ DBV$_2$, if and only if DBV$_1$ ∩ DBV$_2$ = DBV$_1$.

```
1.  pos = Max(pos1, pos2); // Find the maximal position
2.  i = pos1 < pos2 ? pos2 – pos1 : 0; //The first byte of Bit-Vector1 has the intersection with Bit-Vector2
3.  j = pos1 < pos2 ? 0 : pos1 – pos2;//The first byte of Bit-Vector2 has the intersection with Bit-Vector1
4.  count = |Bit-Vector1| - i < |Bit-Vector2| - j ? |Bit-Vector1| - i : |Bit-Vector2| - j; //Number of bytes in the intersection
5.  while count > 0 AND Bit-Vector1[i] & Bit-Vector2[j] = 0 do //Find the first non-zero byte
6.      i = i + 1; j = j + 1;
7.      pos = pos + 1; count = count – 1;
8.  i1 = i + count - 1; j1 = j + count - 1;
9.  while count > 0 AND Bit-Vector1[i1] & Bit-Vector2[j1] = 0 do //Find the last non-zero byte
10.     i1 = i1 - 1; j1 = j1 - 1;
11.     count = count – 1;
12. for k = 0 to count -1 do //Find the intersection
13.     Bit-Vector[k] = Bit-Vector1[i] & Bit-Vector2[j];
14.     i = i + 1; j = j + 1;
```

Fig. 4. The pseudo code for getting the intersection between two DBVs.

Definition 3.2. Given two DBVs, $DBV_1 = \{pos_1, Bit\text{-}Vector_1\}$ and $DBV_2 = \{pos_2, Bit\text{-}Vector_2\}$, DBV_1 is called equal to DBV_2, denoted $DBV_1 = DBV_2$, if $pos_1 = pos_2$, $|Bit\text{-}Vector_1| = |Bit\text{-}Vector_2|$, and $\forall i \in [0..|Bit\text{-}Vector_1| - 1]$: $Bit\text{-}Vector_1[i] = Bit\text{-}Vector_2[i]$.

Definition 3.3. Given DBV = {pos, Bit-Vector}, DBV is called $\varnothing$ if Bit-Vector is $\varnothing$.

Corollary 3.1. Given two DBVs, $DBV_1 = \{pos_1, Bit\text{-}Vector_1\}$ and $DBV_2 = \{pos_2, Bit\text{-}Vector_2\}$. If one of two following cases satisfy, then $DBV_1 \cap DBV_2 = \varnothing$.

(i) $pos_1 + |Bit\text{-}Vector_1| < pos_2$ or,
(ii) $pos_2 + |Bit\text{-}Vector_2| < pos_1$.

Proof

(i) $pos_1 + |Bit\text{-}Vector_1| < pos_2$ implies that the count variable (in the Fig. 3) will be 0. Therefore, Bit-Vector is $\varnothing$ or $DBV_1 \cap DBV_2 = \varnothing$.
(ii) Similar to (i).

Corollary 3.2. *Consider the algorithm in Fig. 4, if count* $\times$ *8 < minSup, then the itemset that contains DBV is not a frequent itemset.*

Proof. The number of bits 1 in Bit-Vector is the support of an itemset. Besides, the value of count variable (in line 4) is always larger than or equal to that of |Bit-Vector|, i.e., |Bit-Vector| $\leqslant$ count. Therefore, number of bits 1 $\leqslant$ |Bit-Vector| $\times$ 8 $\leqslant$ count $\times$ 8 < *minSup* or the itemset corresponding to the Bit-Vector will be not frequent. □

In fact, if the count variable in line 12 satisfies count $\times$ 8 < *minSup*, then the itemset that contains DBV is not a frequent itemset.

Based on Corollary 3.2, the algorithm in Fig. 4 can be re-written as in Fig. 5.

The algorithm of Fig. 5 differs from the algorithm of Fig. 4 in line 4 (lines 4, and 5 of Fig. 5), and line 14 of Fig. 5. They check whether the count variable satisfies the corollary 3.2 or not. If it satisfies, the algorithm will return NULL value for the DBV result (it means that Bit-Vector is NULL).

3.4. A method for fast computing the itemset support

A limit of BitTable-based approach is that it consumes more time for computing the intersection among Bit-Vectors and counting the number of bits 1 of a Bit-Vector. Algorithms in Dong and Han (2007) and Song et al. (2008) count the number of bits 1 of an itemset after computing the intersection among Bit-Vectors of items in this itemset. For example: Assume that we need compute the support of itemset $X = \{x_1, x_2, \ldots, x_k\}$, we must compute Bit-Vector(X) = Bit-Vector(x_1) $\cap$ Bit-Vector(x_2) $\cap \cdots \cap$ Bit-Vector(x_k) first. After that, we scan Bit-Vector(X) to count the number of bits 1. The complexity of support counting is O(nk), where n is number of transactions and k is the length of X.

This paper proposes a new method for fast computing the support of a new itemset. When computing the intersection between 2 bytes to create the result byte, we use a lookup table to get the number of bits 1 in this byte. With this method, when the Bit-Vector is computed, we will have number of bits 1 of Bit-Vector immediately. Therefore, the complexity for the support counting of an itemset is O(m), where m is the number of bits in DBV. Because $m \leqslant n$, O(m) $\ll$ O(nk). If we only consider the time for counting the number of bits 1 in the result Bit-Vector, then the complexity of BitTable approach is O(n) and of our approach is only O(1).

Because each element in Bit-Vector is one byte, we can use a lookup table (as in Table 1) with 256 elements, the ith element contains the number of bits 1 of the value i.

Table 1
Lookup table for look up the number of bits 1.

Value	0	1	2	3	4	5	...	...	255
Binary value	00000000	00000001	00000010	00000011	00000100	00000101			11111111
#bit 1	0	1	1	2	1	2	...	...	8

```
1.  pos = Max(pos1, pos2); // Find the maximal position
2.  i = pos1 < pos2 ? pos2 – pos1 : 0; //The first byte of Bit-Vector1 has the intersection with Bit-Vector2
3.  j = pos1 < pos2 ? 0 : pos1 – pos2;//The first byte of Bit-Vector2 has the intersection with Bit-Vector1
4.  count = |Bit-Vector1|-i <|Bit-Vector2|-j ?|Bit-Vector1|-i :|Bit-Vector2|-j; //Number of bytes in the intersection
5.  if count × 8 < minSup then return NULL; // Using corollary 3.2
6.  while count > 0   AND   Bit-Vector1[i] & Bit-Vector2[j] = 0 do // Find the first non-zero byte
7.      i = i + 1; j = j + 1;
8.      pos = pos + 1; count = count – 1;
9.  i1 = i + count - 1; j1 = j + count - 1;
10. while count > 0   AND   Bit-Vector1[i1] & Bit-Vector2[j1] = 0 do // Find the last non-zero byte
11.     i1 = i1 - 1; j1 = j1 - 1;
12.     count = count – 1;
13. if count × 8 < minSup then return NULL; // using corollary 3.2
14. for k = 0 to count -1 do // Find the intersection
15.     Bit-Vector[k] = Bit-Vector1[i] & Bit-Vector2[j];
16.     i = i + 1; j = j + 1;
```

Fig. 5. The modified pseudo code for getting the intersection between two DBVs.

4. Mining frequent closed itemsets

4.1. DBV-tree

Each node in DBV-tree includes two elements X and DBV(X), where X is an itemset and DBV(X) is Dynamic Bit-Vector containing X. Arc connects between two nodes X and Y must satisfy condition that X, Y have the same length and the same $|X| - 1$ prefix items.

Example 3. Consider an example database as shown in Table 2.

Bit-Vector and DBV of each items in Table 2 will be as shown in Table 3.

Fig. 6 illustrates the method for creating DBV-tree of the database in Table 3 with 5 first items { A,B,C,D,E}: The first level of DBV-tree contains single items and their DBV. To create higher levels, we join each child node with all its following nodes.

For example: consider node A in the level 1 of DBV-tree in Fig. 6.

- A joins B to create a new node AB, DBV(A) = {0,{29}} and DBV(B) = {0,{63}}, so DBV(AB) = DBV(A) ∩ DBV(B) = {0,{29}} ∩ {0,{63}} = {0,{29&63}} = {0,{29}}.
- A joins C to create a new node AC, DBV(C) = {0,{58}}, so DBV(AC) = {0,{24}}.
- A joins D to create a new node AD, DBV(D) = {0,{53}}, so DBV(AD) = {0,{21}}.
- A joins E to create a new node AE, DBV(E) = {0,{31}}, so DBV(AE) = {0,{29}}.

After that, each child node of A will join with all its successive nodes to create the grandchildren of A. This process will be repeated recursively to create DBV-tree.

Table 2
An example database.

Transactions	Bought items
1	A,B,D,E,G
2	B,C,E,F
3	A,B,D,E,G
4	A,B,C,E,F,G
5	A,B,C,D,E,F,G
6	B,C,D,F,G,H

Table 3
Bit-Vector and DBV of each items in Table 2.

Items	Transactions	Bit-Vector	DBV
A	1, 3, 4, 5	011101	{0,29}
B	1, 2, 3, 4, 5, 6	111111	{0,63}
C	2, 4, 5, 6	111010	{0,58}
D	1, 3, 5, 6	110101	{0,53}
E	1, 2, 3, 4, 5	011111	{0,31}
F	2, 4, 5, 6	111010	{0,58}
G	1, 3, 4, 5, 6	111101	{0,61}
H	6	100000	{0,32}

4.2. Subsumption concept

Definition 4.1. Itemset X is subsumed by Y if and only if $\sigma(X) = \sigma(XY)$.

The subsumption concept in our definition is more general than from the definitions in Song et al. (2008) and Zaki and Hsiao (2005). In Song et al. (2008), authors used subsumption concept for subsuming single items, and in Zaki and Hsiao (2005), authors used subsumption concept for checking non-closed itemsets. The subsumption concept in Definition 4.1 can be used for subsuming itemsets and eliminating non-closed itemsets from DBV-tree.

Theorem 4.1. *Given two DBVs, DBV(X) and DBV(Y), if DBV(X) ⊆ DBV(Y) then X is subsumed by Y.*

Proof. We have Bit-Vector(XY) = Bit-Vector(X) ∩ Bit-Vector(Y) = Bit-Vector(X). It implies that number of bits 1 of Bit-Vector(X) is equal to that of Bit-Vector(XY) or $\sigma(X) = \sigma(XY)$. □

Theorem 4.2. *Some properties of the subsumption concept:*

(i) *If X is subsumed by Y, then X is not a closed itemset.*
(ii) *If X is subsumed by Y, and Y is subsumed by X, then X and Y are not closed itemsets.*

Proof

(i) X is subsumed by Y, i.e., $\sigma(X) = \sigma(XY)$. According to the definition of frequent closed itemset in section 2.1, X is not a closed itemset.
(ii) If X is subsumed by Y, and Y is subsumed by X, then $\sigma(X) = \sigma(XY) = \sigma(Y)$. It implies that both X and Y is not closed. □

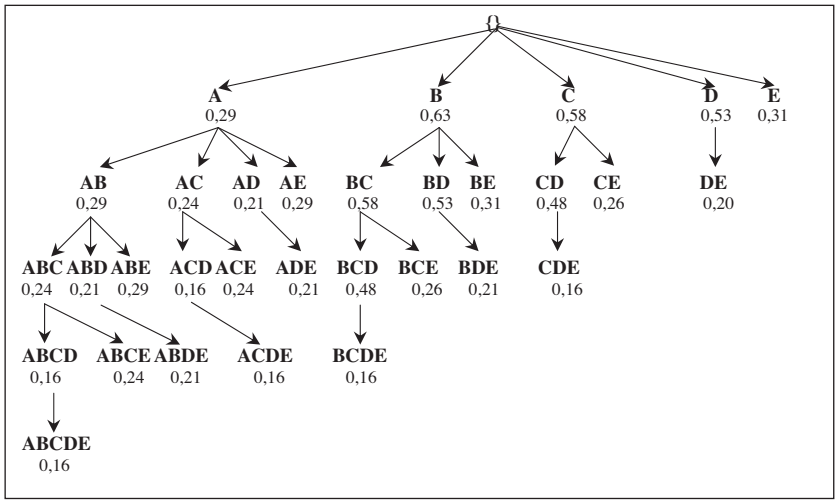

Fig. 6. DBV-tree for mining itemsets from the database in Table 3 with 5 first items {A,B,C,D,E}.

Input: A database D with a set of items I and a support threshold *minSup*.

Output: The DBV tree containing all FIs that satisfy *minSup*.

DBV-FCI(D, *minSup*)

1. $L = \{ i_{DBV(i)} \mid i \in I \wedge \sigma(i) \geq minSup \}$
2. Sort items in L_r increasingly by their supports, and subsume them together
3. DBV_EXTEND(L, *minSup*)

DBV_EXTEND(L, *minSup*)

4. $\forall X_{DBV(X)} \in L$:
5. Create the new set L_i by joining $X_{DBV(X)}$ with each node $Y_{DBV(X)}$ following $X_{DBV(X)}$ with $Y \not\subset X$, there are two cases:
6. If L_i has child node $Z_{DBV(Z)}$ such that Z is subsumed by XY then
7. Replace Z by ZXY.
8. Insert $XY_{DBV(XY)}$ into the child nodes of L_i (sort according to the support) if $DBV(Z) \subset DBV(XY)$, where $DBV(XY) = DBV(X) \cap DBV(Y)$.
9. Otherwise, insert $XY_{DBV(XY)}$ into the child node of L_i (sort according to the support) if $\sigma(XY) \geq minSup$ if XY is not subsumed by any successive node of $X_{DBV(X)}$ or of parent($X_{DBV(X)}$).
10. If $|L_i| \geq 2$, then call DBV_EXTEND(L_i, *minSup*)

Fig. 7. Algorithm for mining FCI using DBV-tree.

Remarks. According to the Theorem 4.2.

(i) If X is subsumed by Y, then X is not closed. Therefore, we can subsume X with Y to be XY.
(ii) If X is subsumed by Y, and Y is subsumed by X, then X will be subsumed by Y into XY and Y will be subsumed by X into YX. It means that there exist two nodes with the same itemsets XY and YX. It is necessary to remove one of them.

4.3. DBV-Miner – an algorithm for mining FCI using DBV-tree

Fig. 7 presents an algorithm for mining frequent closed itemsets from transaction databases. Firstly, it creates a set of nodes in the first level of DBV-tree, each node contains a single item such that its support satisfies *minSup* (line 1). After that, it sorts nodes in the first level increasing according to their supports, the subsumption concept is used in this step to subsume items (line 2). This work differs from Index-BitTableFI in that it will replace itemset X by itemset XY if DBV(X) = DBV(XY) according to the Theorems 4.1 and 4.2.i. Besides, by Theorem 4.2.ii, if DBV(X) = DBV(XY), then Y will be deleted from the tree. Finally, the algorithm will call the procedure DBV-EXTEND (line 3). This procedure will consider each node $X_{DBV(X)}$ in L with all nodes following it to create all child nodes of X (lines 4-9). With each node $Y_{DBV(Y)}$ following it such that $Y \not\subset X$, the algorithm will check the conditional in line 6, if true, then replace Z by ZXY (by the Theorem 4.2) and insert XY as a child node of L_i. Otherwise, if XY is not subsumed by any successive node of $X_{DBV(X)}$ in L, then by Lemma 4.1 (below), XY is inserted as a child node of L_i (line 9). Finally, if number of nodes in L_i is larger than 1, then call recursively the procedure DBV-EXTEND to create all child nodes of L_i (line 10).

Lemma 4.1. When node $X_{DBV(X)}$ joins with node $Y_{DBV(Y)}$ to create a new node $XY_{DBV(XY)}$, if in L_r exists node $Z_{DBV(Z)}$ such that X is a successive node of Z, and XY is subsumed by Z then XY will not be closed.

Proof. When the algorithm joins node Z with all successive nodes in L_r, then Z has been joined with X and Y. There are two cases:

(i) If $XY \subseteq Z$, then $\sigma(XY) = \sigma(XYZ) = \sigma(Z)$ or XY is not closed.
(ii) $XY \not\subset Z$, it means that $X \not\subset Z$ or $Y \not\subset Z$. According to the algorithm, there is at least one child node Z' of Z such that $XY \subseteq Z'$. It implies that XY is not closed. □

4.4. An example

Consider the database in Table 2 with *minSup* = 30%, we have:

- Firstly, assume that single items of the database are sorted increasingly by their support, $L = \{A_{0,29}, C_{0,58}, D_{0,53}, F_{0,58}, E_{0,31}, G_{0,61}, B_{0,63}\}$ (H is pruned since its support < *minSup*).
- Consider node A: A is subsumed by $\{G, E, B\}$, so A is replaced into $AEGB$.
- Similarly, C is replaced into CFB, and F is pruned (F has the same tidset with C), D is replaced into DGB, E is replaced into EB, and G is replaced into GB.
- Finally, the children nodes of L_r are $\{AEGB_{0,29}, CFB_{0,58}, DGB_{0,53}, EB_{0,31}, GB_{0,61}, B_{0,63}\}$ as in Fig. 8.

After subsuming items in the first level, the algorithm will call procedure DBV-EXTEND.

- Consider node $AEGB_{0,29}$:
  - $L_i = \varnothing$.
  - $AEGB_{0,29}$ joins $CFB_{0,58}$ into a new node $AEGBCF_{0,24}$, $\sigma(AEGBCF) = 2 \geqslant minSup$. $AEGBCF_{0,24}$ is inserted into the list of child nodes of L_i ($\{AEGBCF_{0,24}\}$).
  - $AEGB_{0,29}$ joins $DGB_{0,53}$ into a new node $AEGBD_{0,21}$, $\sigma(AEGBD) = 3 \geqslant minSup$. $AEGBD_{0,21}$ is inserted into the list of child nodes of L_i ($\{AEGBCF_{0,24}, AEGBD_{0,21}\}$).
  - Consider node $EB_{0,31}$, because $EB \subset AEGB$, it is skipped.
  - Similarly to node $GB_{0,61}$ and node $B_{0,63}$.
  - After considering node all $AEGB_{0,29}$ with all nodes following it, we have $L_i = (\{AEGBCF_{0,24}, AEGBD_{0,21}\}$. Because $|L_i| \geqslant 2$, DBV-EXTEND is called recursively with $L_r = L_i$.
    - Node $AEGBCF_{0,24}$ joins with node $AEGBD_{0,21}$, we have DBV($AEGBCFD$) = {0,{16}}, the support of $AEGBCFD$ is 1 < *minSup* ⇒ it is skipped.

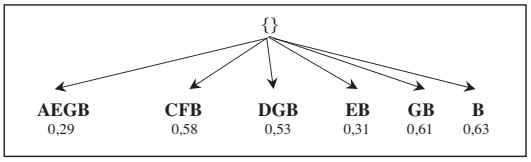

Fig. 8. The level 1 of DBV-tree for mining FCI.

- Consider node $CFB_{0,58}$:
  - $L_i = \varnothing$.
  - $CFB_{0,58}$ joins with node $DGB_{0,53}$ into a new node $CFBDG_{0,48}$, $\sigma(CFBDG) = 2 \geqslant minSup$. $CFBDG_{0,48}$ is inserted into the list of child nodes of L_i ($\{CFBDG_{0,48}\}$).
  - $CFB_{0,58}$ joins with node $EB_{0,31}$ into a new node $CFBE_{0,26}$, $\sigma(CFBE) = 3 \geqslant minSup$. $CFBE_{0,26}$ is inserted into the list of child nodes of L_i ($\{CFBDG_{0,48}, CFBE_{0,26}\}$).
  - $CFB_{0,58}$joins with node $GB_{0,61}$ into a new node $CFBG_{0,56}$, $\sigma(CFBG) = 3 \geqslant minSup$. $CFBG_{0,56}$ is inserted into the list of child nodes of L_i ($\{CFBDG_{0,48}, CFBE_{0,26}, CFBG_{0,56}\}$).
  - Because $|L_i| \geqslant 2$, DBV-EXTEND is called recursively with $L_r = L_i = (\{CFBDG_{0,48}, CFBE_{0,26}, CFBG_{0,56}\})$.
    - o $CFBDG_{0,48}$ joins with $CFBE_{0,26}$, we have $DBV(AEGBCFD) = \{0,\{16\}\}$, the support of $AEGBCFD$ is 1 < $minSup \Rightarrow$ it is skipped.
    - o $CFBDG_{0,48}$ joins with $CFBG_{0,56}$, because $CFBG \subset CFBDG$, it is skipped.
    - o $CFBE_{0,26}$ joins with $CFBG_{0,56}$, into a new node $CFBEG_{0,24}$, because $CFBEG_{0,24}$ is subsumed by node $AEGB_{0,29}$, it is skipped.

Fig. 9 illustrates the process of mining FCI from the database in Table 2 with $minSup = 30\%$ (mining frequent closed itemsets that contain in at least two transactions). From Fig. 9, we have all FCI that satisfy $minSup$ are {*AEGB*:4, *CFB*:4, *DGB*:4, *EB*:5, *GB*:5, *B*:6, *AEGBCF*:2, *AEGBD*:3, *CFBDG*:2, *CFBE*:3, *CFBG*:3}.

According to the Lemma 4.1, if XY is not closed, then we can prune it away DBV-tree. For example, consider the Fig. 9, $DGBE_{0,21}$ is subsumed by node $AEGB_{0,29}$ ($DBV(AEGBD) = \{0,\{21\}\}$, it implies that $\sigma(DGBE) = \sigma(AEGBD)$), $DGBE$ is not added into DBV-tree. Similarly, EBG and $CFBEG$ are subsumed by $AEGB$, they are not added into DBV-tree.

By Lemma 4.1, we need not use hash table to check non-closed itemset as CHARM. Besides, an itemset will be pruned immediately when it is created if it satisfies the Lemma 4.1.

Table 4
Features of the databases adopted.

Database	#Trans	#Items
Chess	3196	76
Mushroom	8124	120
Pumsb	49,046	7117
Pumsb*	49,046	7117
Connect	67,557	130
Accidents	340,183	468

Table 5
Number of FCI of six databases under different *minSup* values.

Database	*minSup* (%)	#FCI
Chess	75	11,525
	70	23,892
	65	49,034
	60	98,392
Mushroom	40	140
	30	427
	20	1197
	10	4095
Pumsb	96	54
	94	216
	92	610
	90	1465
Pumsb*	55	116
	50	248
	45	713
	40	2610
Connect	98	135
	94	1237
	90	3486
	86	7087
Accidents	80	149
	70	529
	60	2074
	50	8057

5. Experimental results

Experiments were conducted to show the performance of the proposed algorithms. They were implemented on a Centrino Core 2 Duo (2×2.53 GHz), with 4 GBs RAM of memory and running Windows 7. The algorithms were coded in C# 2008. Six databases from http://fimi.cs.helsinki.fi/data/ (download on April 2005) were used for the experiments, with their features displayed in Table 4.

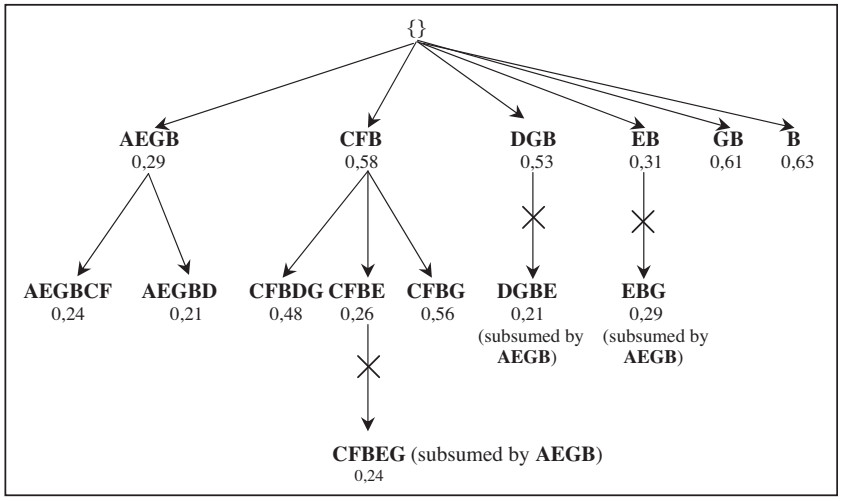

Fig. 9. Results of the algorithm DBV-Miner with $minSup = 30\%$.

Table 5 shows the number of frequent closed itemsets of six databases above under different *minSup* values.

5.1. Comparing of mining time

Experiments were then made to compare the mining time of CHARM (Zaki & Hsiao, 2005), and DBV-Miner for different *minSup* values. The results for the six databases were shown in Figs. 10–15.

Fig. 10 shows the mining time of CHARM (Zaki & Hsiao, 2005), and DBV-Miner in Chess database. The results show that our approach is efficient than CHARM. For example, with *minSup* = 60%, the mining time of CHARM is 19.83(s), and of DBV-Miner is 4.83(s). The scale is $\frac{4.83}{19.83} \times 100\% = 24.4\%$.

In this database, the distance of two approaches is very large. For example, with *minSup* = 86%, the mining time of CHARM is 40.03 (s), while of DBV-Miner is 13.29 (s).

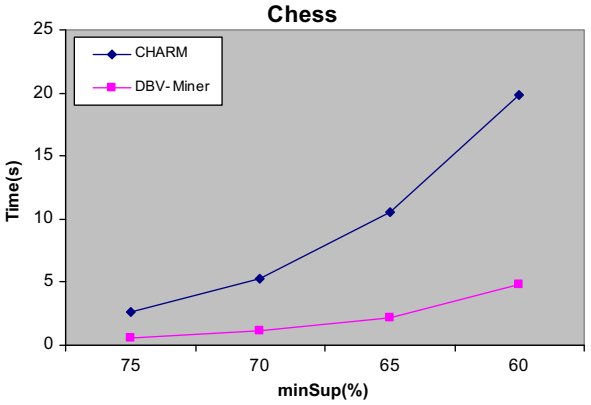

Fig. 10. Execution time of the two algorithms for Chess under different *minSup* values.

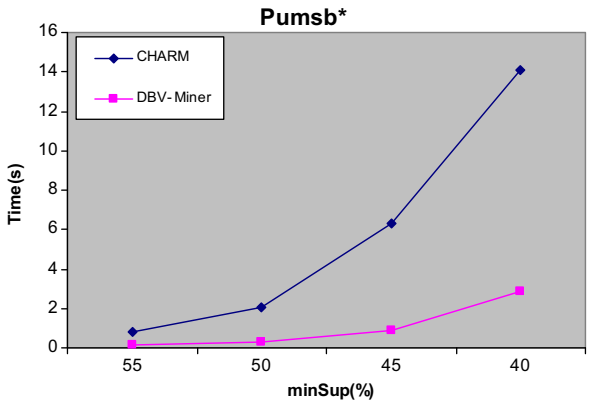

Fig. 13. Execution time of the two algorithms for Pumsb* under different *minSup* values.

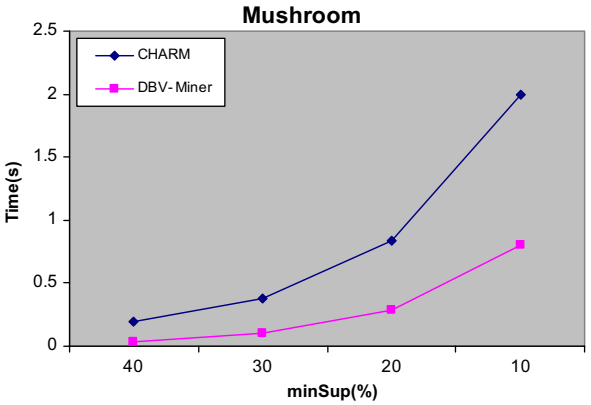

Fig. 11. Execution time of the two algorithms for Mushroom under different *minSup* values.

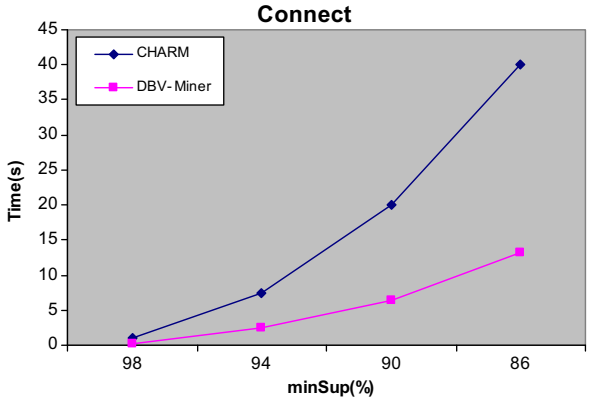

Fig. 14. Execution time of the two algorithms for Connect under different *minSup* values.

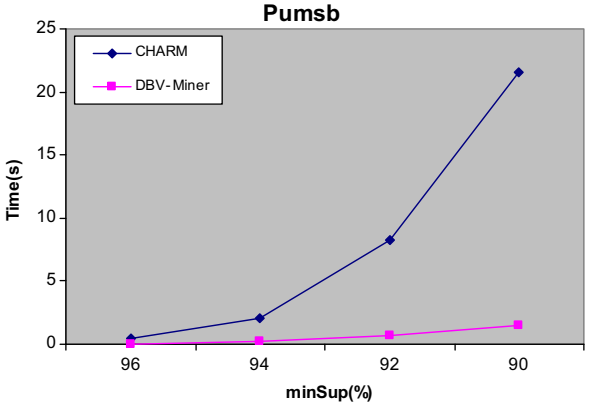

Fig. 12. Execution time of the two algorithms for Pumsb under different *minSup* values.

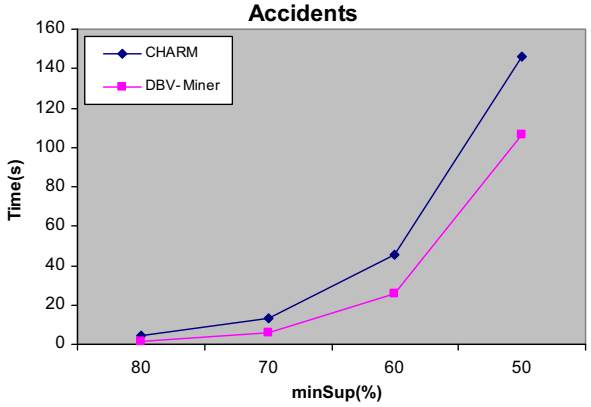

Fig. 15. Execution time of the two algorithms for Accidents under different *minSup* values.

To show the efficient of DBV approach with BitTable, we replace DBV in DBV-tree by BitTable (Dong & Han, 2007; Song et al., 2008). The results for the six databases were shown in Figs. 16–21.

Fig. 16 shows the mining time of BitTable-based, and DBV-based in Chess database. The results show that DBV-based is efficient than BitTable-based. For example, with *minSup* = 60%, the mining time of CHARM is 5.65 (s), and of DBV-Miner is 4.83 (s).

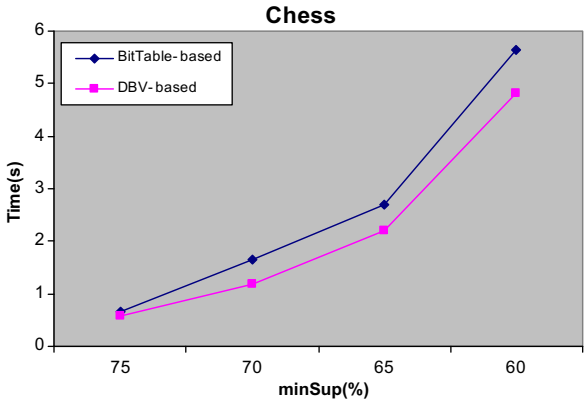

Fig. 16. Execution time of BitTable-based and DBV-based in the proposed algorithm for Chess under different *minSup* values.

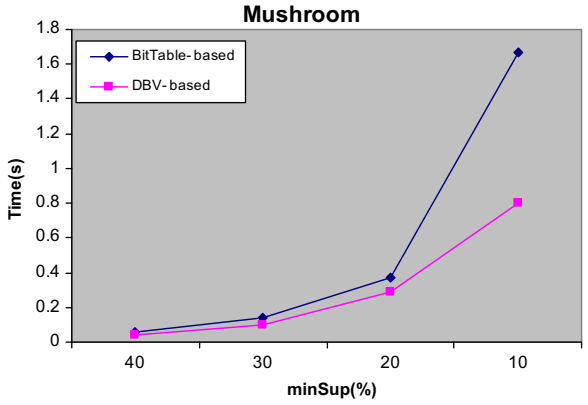

Fig. 17. Execution time of BitTable-based and DBV-based in the proposed algorithm for Mushroom under different *minSup* values.

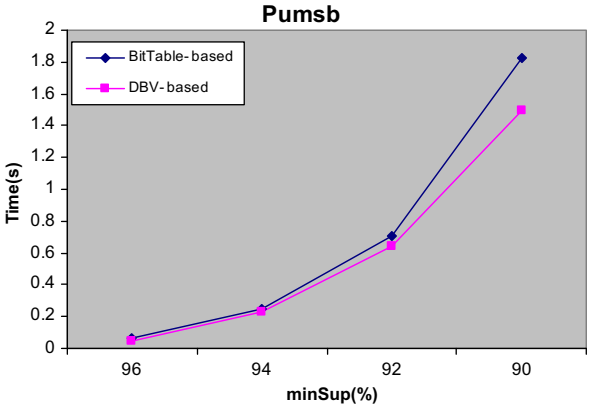

Fig. 18. Execution time of BitTable-based and DBV-based in the proposed algorithm for Pumsb under different *minSup* values.

5.2. Comparing of memory usage

Next, experiments were conducted to compare the total memory used (MBs) for Tidset/Bit-Vector of the following two algorithms: CHARM (Zaki & Hsiao, 2005), DBV-Miner. The results for six databases under different *minSup* values are shown in Figs. 22–27.

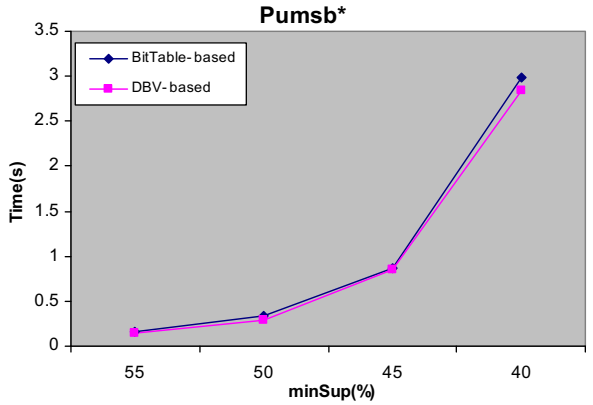

Fig. 19. Execution time of BitTable-based and DBV-based in the proposed algorithm for Pumsb* under different *minSup* values.

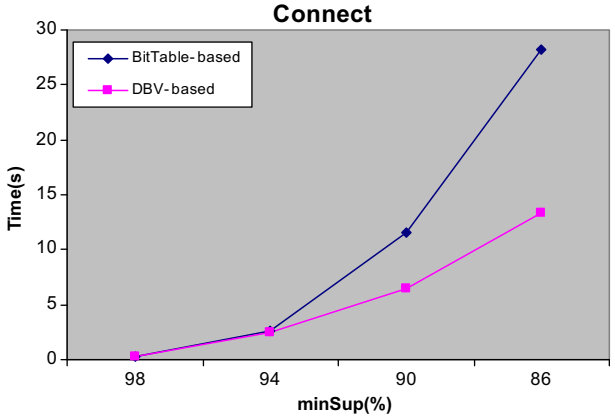

Fig. 20. Execution time of BitTable-based and DBV-based in the proposed algorithm for Connect under different *minSup* values.

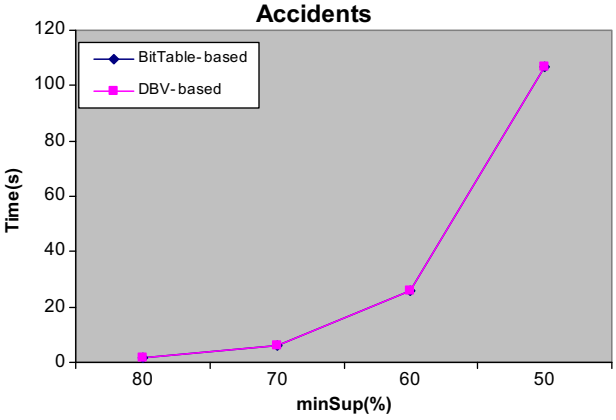

Fig. 21. Execution time of BitTable-based and DBV-based in the proposed algorithm for Accidents under different *minSup* values.

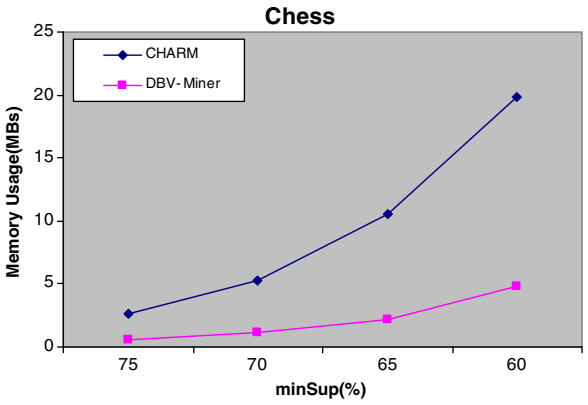

Fig. 22. Memory used (MBs) of the two algorithms for Chess under different *minSup* values.

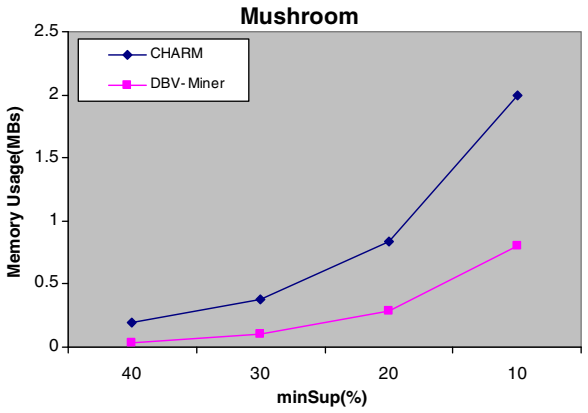

Fig. 23. Memory used (MBs) of the two algorithms for Mushroom under different *minSup* values.

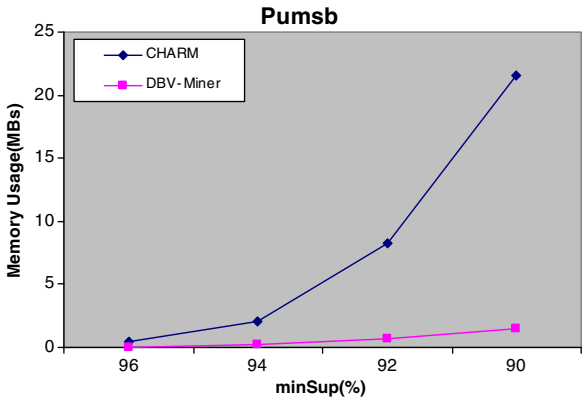

Fig. 24. Memory used (MBs) of the two algorithms for Pumsb under different *minSup* values.

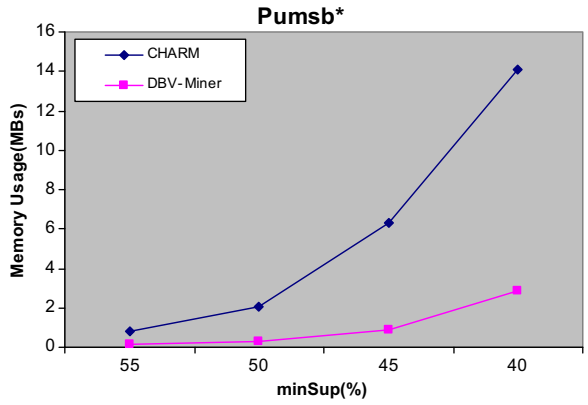

Fig. 25. Memory used (MBs) of the two algorithms for Pumsb* under different *minSup* values.

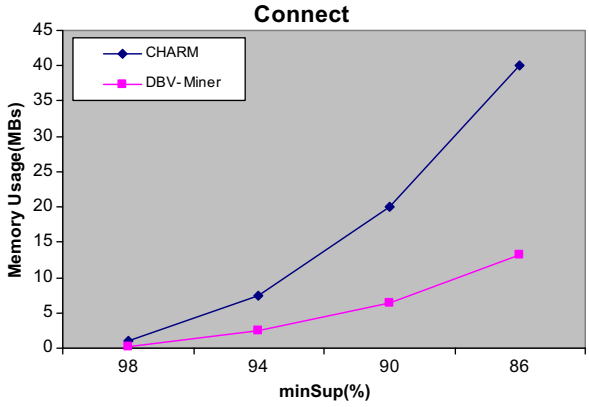

Fig. 26. Memory used (MBs) of the two algorithms for Connect under different *minSup* values.

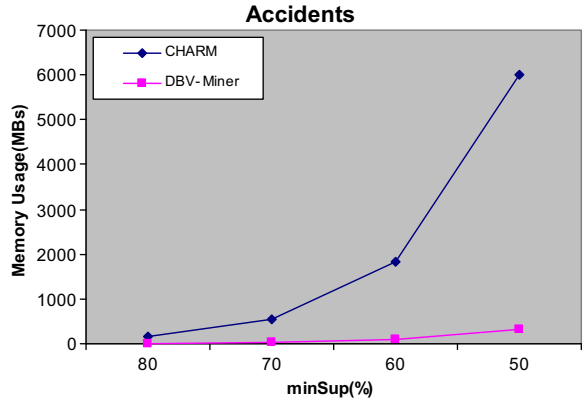

Fig. 27. Memory used (MBs) of the two algorithms for Accidents under different *minSup* values.

Figs. 22–27 show the total memory used of six databases under different *minSup*, we can see that the memory used of CHARM always consumes more than that of DBV-Miner. For example, consider the Chess database with *minSup* = 60%, the total memory used for Tidset is 802.6 MBs, and for DBV (DBV-Miner) is 31.12 MBs.

6. Conclusion and future work

In this paper, we have proposed a new method for mining FCI from transaction databases. This method has some advantages: Firstly, it uses Dynamic Bit-Vector for compress the database with one scan in the whole mining process. Next, an algorithm for fast mining FCI has been proposed. This algorithm uses DBV for storing the database. Advantages of this method are based on the intersection between two DBVs for fast computing the support, and checking non-closed itemsets by using Lemma 4.1. Experimental results show the efficient of this method in both the mining time and memory usage.

Mining frequent itemsets in incremental databases has been developed in recent years (Hong, Lin, & Wu, 2009; Hong & Wang, 2010; Hong, Wu, & Wang, 2009; Lin, Hong, & Lu, 2009; Zhang, Zhang, & Jin, 2009). We can see that DBV can be applied for fast

mining frequent itemsets (FI) and frequent closed itemsets (FCI) from this kind of databases. Based on DBV, we can mine all FI/FCI when transactions are inserted, updated or deleted. Besides, mining association rules from lattice is more efficient than from FI/FCI (Vo & Le, 2009, 2011a, 2011b). Therefore, we will develop an algorithm for building frequent closed itemsets lattice based on DBV.

References

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules. In *VLDB'94* (pp. 487–499).

Bastide, Y., Pasquier, N., Taouil, R., Stumme, & G., Lakhal, L. (2000). Mining minimal non-redundant association rules using frequent closed itemsets. In *First international conference on computational logic* (pp. 972–986).

Dong, J., & Han, M. (2007). BitTable-FI: An efficient mining frequent itemsets algorithm. *Knowledge Based Systems, 20*(4), 329–335.

Grahne, G., & Zhu, J. (2005). Fast algorithms for frequent itemset mining using fp-trees. *IEEE Transactions on Knowledge and Data Engineering, 17*(10), 1347–1362.

Hong, T.-P., Lin, C.-W., & Wu, Y.-L. (2009). Maintenance of fast updated frequent pattern trees for record deletion. *Computational Statistics and Data Analysis, 53*(7), 2485–2499.

Hong, T.-P., & Wang, C.-J. (2010). An efficient and effective association-rule maintenance algorithm for record modification. *Expert Systems with Applications, 37*(1), 618–626.

Hong, T.-P., Wu, Y.-Y., & Wang, S.-L. (2009). An effective mining approach for up-to-date patterns. *Expert Systems with Applications, 36*(6), 9747–9752.

Lee, A. J. T., Wang, C. S., Weng, W. Y., Chen, J. A., & Wu, H. W. (2008). An efficient algorithm for mining closed inter-transaction itemsets. *Data and Knowledge Engineering, 66*(1), 68–91.

Lin, C.-W., Hong, T.-P., & Lu, W.-H. (2009). The Pre-FUFP algorithm for incremental mining. *Expert Systems with Applications, 36*(5), 9498–9505.

Lucchese, B., Orlando, S., & Perego, R. (2006). Fast and memory efficient mining of frequent closed itemsets. *IEEE Transaction on Knowledge and Data Engineering, 18*(1), 21–36.

Moonestinghe, H. D. K., Fodeh, S., & Tan, P. N. (2006). Frequent closed itemsets mining using prefix graphs with an efficient flow-based pruning strategy. In *Proceedings of 6th ICDM, Hong Kong* (pp. 426–435).

Pasquier, N., Bastide, Y., Taouil, R., & Lakhal, L. (1999a). Discovering frequent closed itemsets for association rules. In *Proceedings of the 5th international conference on database theory. LNCS* (pp. 398–416). Jerusalem, Israel: Springer.

Pasquier, N., Bastide, Y., Taouil, R., & Lakhal, L. (1999b). Efficient mining of association rules using closed itemset lattices. *Information Systems, 24*(1), 25–46.

Pei, J., Han, J., & Mao, R. (2000). CLOSET: An efficient algorithm for mining frequent closed itemsets. In *Proceedings of the 5th ACM-SIGMOD workshop on research issues in data mining and knowledge discovery, Dallas, Texas, USA* (pp. 11–20).

Singh, N. G., Singh, S. R., & Mahanta, A.K. (2005). CloseMiner: Discovering frequent closed itemsets using frequent closed tidsets. In *Proceedings of the 5th ICDM, Washington DC, USA* (pp. 633–636).

Song, W., Yang, B., & Xu, Z. (2008). Index-BitTableFI: An improved algorithm for mining frequent itemsets. *Knowledge Based Systems, 21*(6), 507–513.

Uno, T., Asai, T., Uchida, Y., & Arimura, H. (2004). An efficient algorithm for enumerating closed patterns in transaction databases. In *Proceedings of the 7th international conference on discovery science. LNCS* (pp. 16–31). Padova, Italy: Springer.

Vo, B., & Le, B. (2009). Mining traditional association rules using frequent itemsets lattice. In *The 39th international conference on computers and industrial engineering, Troyes, France, July 6–8* (pp. 1401-1406). IEEE.

Vo, B., & Le, B. (2011a). Interestingness measures for mining association rules: Combination between lattice and hash tables. *Expert Systems with Applications, 38*(9), 11630–11640.

Vo, B., & Le, B. (2011b). Mining minimal non-redundant association rules using frequent itemsets lattice. *Journal of Intelligent Systems Technology and Applications, 10*(1), 92–206.

Wang, J., Han, J., & Pei, J. (2003). CLOSET+: Searching for the best strategies for mining frequent closed itemsets. In *ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 236–245).

Yahia, S. B., Hamrouni, T., & Nguifo, E. M. (2006). Frequent closed itemset based algorithms: a thorough structural and analytical survey. *ACM SIGKDD Explorations Newsletter, 8*(1), 93–104.

Zaki, M. J. (2000). Generating non-redundant association rules. In *Proceedings of the 6th ACM SIGKDD international conference on knowledge discovery and data mining, Massachusetts, Boston, USA* (pp. 34–43).

Zaki, M. J. (2004). Mining non-redundant association rules. *Data Mining and Knowledge Discovery, 9*(3), 223–248.

Zaki, M. J., Parthasarathy, S., Ogihara, M., & Li, W. (1997). New algorithms for fast discovery of association rules. In: *Third international conference on knowledge discovery and data mining (KDD)* (pp. 283–286).

Zaki, M. J., & Hsiao, C. J. (2005). Efficient algorithms for mining closed itemsets and their lattice structure. *IEEE Transactions on Knowledge and Data Engineering, 17*(4), 462–478.

Zhang, S., Zhang, J., & Jin, Z. (2009). A decremental algorithm of frequent itemset maintenance for mining updated databases. *Expert Systems with Applications, 36*(8), 10890–10895.