Scientific Research and Essays Vol. 6(25), pp. 5358-5368, 30 October, 2011
Available online at http://www.academicjournals.org/SRE
DOI: 10.5897/SRE11.1101
ISSN 1992-2248 ©2011 Academic Journals

Full Length Research Paper

Dynamic bit vectors: An efficient approach for mining frequent itemsets

Bay Vo[1]*, Tzung-Pei Hong[2,3] and Bac Le[4]

[1]Department of Computer Science, Ho Chi Minh City University of Technology, Ho Chi Minh, Vietnam.
[2]Department of Computer Science and Information Engineering, National University of Kaohsiung, Kaohsiung, Taiwan, R.O.C.
[3]Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung, Taiwan, R.O.C.
[4]Department of Computer Science, University of Science, Ho Chi Minh, Vietnam.

Accepted 20 September, 2011

There are two common kinds of data formats to be adopted in data mining. One is horizontal, and the other is vertical. Approaches based on vertical data formats have the advantages of requiring a fewer number of database scans and computing itemset supports fast. One of the vertical data representations, bit vector, has recently been widely used for mining frequent item sets and has caused significant results. The sizes of bit vectors for item sets are, however, always the same, equal to the number of transactions in a database. In this paper, we propose the scheme of dynamic bit vectors to reduce the memory and the computational time for mining frequent item sets from transaction databases. A fast method for computing the intersection of two dynamic bit vectors and an algorithm for mining frequent item sets based on the scheme are presented. The proposed algorithm is also compared with some other approaches and experimental results show that it is quite efficient in both the mining time and the memory usage.

Key words: Data mining, frequent item set, dynamic bit vector, vertical data format.

INTRODUCTION

Mining frequent item sets (FIs) is the most important task in mining association rules. A lot of algorithms for mining FIs have thus been proposed. Some of the famous ones are Apriori (Agrawal and Srikant, 1994), Eclat (Zaki et al., 1997; Zaki and Hsiao, 2005), FP-Growth (Han et al. 2000), FP-Growth* (Grahne and Zhu, 2005), BitTableFI (Dong and Han, 2007), Index-BitTableFI (Song et al., 2008), and so on. They can be divided into two categories according to the database format: horizontal and vertical. The horizontal data format is the same as that presented in a database. The vertical data format is a conversion from the horizontal one, with the transaction identifiers (TIDs) grouped for each item. Most of the vertical-based approaches scan databases once for fast mining FIs. Two representative ways are Tidlist (Zaki et al., 1997; Zaki and Hsiao, 2005) and BitTable (Dong and Han, 2007; Song et al., 2008). The Tidlist maintains a list of TIDs for each interesting item, and the BitTable uses a bit vector to represent the transactions with the item.

The vertical-based approaches above usually store the transformed database in main memory for mining. When the number of transactions is large, it is difficult to store all of them in main memory. Thus, secondary memory is used for helping mining in this situation, causing more execution time. Besides, the bit vectors occupy the same fixed size which depends on the number of transactions in a given database. Much memory and time are thus needed for computing the intersection among bit vectors. In practice, there are usually many bits of '0' in a bit vector. The bit vector of an itemset with many bits of '0' can thus be shortened to reduce space and time.

In this paper, the scheme of dynamic bit vectors (DBVs) is thus designed to solve the above problem. A dynamic bit vector represents the bit vector in bytes after removing

*Corresponding author. E-mail: vdbay@hcmhutech.edu.vn.

the zero bytes at the front and at the tail. Different Item sets thus have different length of vectors. A method for fast computing the intersection of two DBVs is then presented. It uses a look-up table to speed up the counting process. An approach for mining frequent item sets based on the DBV scheme and the intersection method is also proposed. A tree structure called a DBV tree is used to help mine frequent item sets efficiently. Experimental results also show the good performance of the proposed approach in both the mining time and the total memory usage.

RELATED WORK

Mining frequent item sets

Frequent item sets 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 $\sigma(X)$ of an itemset X, $X \subseteq I$, is the number of transactions in D containing X. Itemset X is called frequent if $\sigma(X) \geq minSup$, where $minSup$ is a predefined minimum support threshold. There are many methods proposed for mining FIs from databases. They could be divided into the following three categories:

Generate-and-test approaches

They are mainly based on the Apriori algorithm and use the level-wise approach to discover FIs. Apriori (Agrawal et al., 1994) and BitTableFI (Dong and Han, 2007) are some examples.

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 FIs. Examples include FP-Growth (Han et al., 2000), FP-Growth* (Grahne and Zhu, 2005).

Hybrid approaches

They integrate both the two strategies above to mine FIs. They firstly transform the database into the vertical data format (introduced later) and then use the divide-and-conquer approach to mine FIs. Eclat (Zaki et al., 1997; Zaki and Hsiao, 2005) and Index-BitTableFI (Song et al., 2008) belong to them.

Vertical data format

When mining frequent itemsets, there are two common kinds of data formats to be adopted. One is the horizontal data format, and the other is the vertical data format. The horizontal data format is the same as that presented in a database. The vertical data format is a conversion from the horizontal one, with the transaction identifiers grouped for each item. Algorithms for mining frequent item sets based on the vertical data format are usually more efficient than those based on the horizontal (Dong and Han, 2007; Song et al., 2008; Zaki et al., 1997; Zaki and Hsiao, 2005), because the former often scan the database only once and compute the supports of item sets fast. The disadvantage is that it consumes more memory for storing additional information, like Tidsets (Zaki et al., 1997; Zaki and Hsiao, 2005) or BitTable (Dong and Han, 2007; Song et al., 2008). Some typical algorithms based on the vertical data format are briefly reviewed as follows:

Eclat (Zaki et al., 1997; Zaki and Hsiao, 2005)

It was proposed by Zaki et al.(1997) for mining FIs. An additional structure called Tidset was used, which stored the transaction identifiers for each itemset. The support $\sigma(X)$ of an itemset X can be fast derived as the cardinality of the Tidset of the itemset. Thus, $\sigma(X) = |\text{Tidset}(X)|$. They also proposed the way of computing Tidset(XY) by the intersection operator between Tidset(X) and Tidset(Y). That is, Tidset(XY) = Tidset(X) $\cap$ Tidset(Y). Zaki and Hsiao then proposed the Diffset approach to reduce the storage space and the time needed for computing the supports (Zaki and Hsiao, 2005).

BitTable (Dong and Han, 2007; Song et al., 2008)

It was another way of data compression. 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 (Agrawal et al., 1994). Their approach was named BitTableFI (Dong and Han, 2007). 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. Song et al. then extended BitTableFI and proposed the Index-BitTableFI approach to mine frequent Itemsets (Song et al., 2008). Index-BitTableFI used the "subsumption" concept to save the

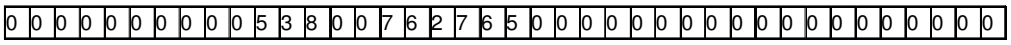

Figure 1. An example of a bit vector with 40 bytes.

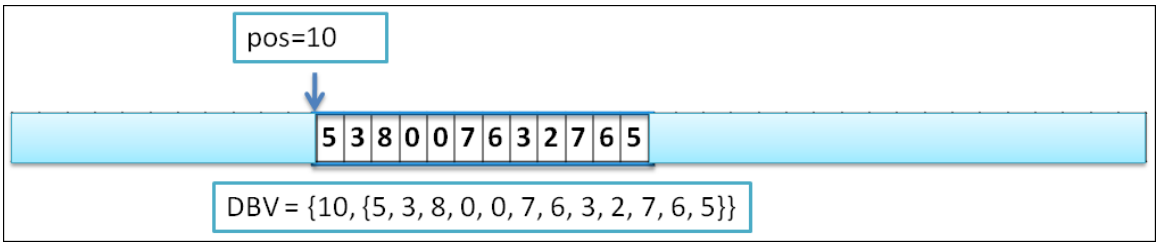

Figure 2. A representation of DBV for the bit vector in Figure 1.

checking time. Initially, items are sorted increasing according to their supports. After that, each item is examined with all the items following it by their bit vectors. If the '1' bits in the bit vector of an item were a subset of those in another item, then the latter item was subsumed by the former. Song et al. showed that Index-BitTableFI had a better performance than BitTableFI.

SCHEME OF DYNAMIC BIT VECTORS

As mentioned previousely, the bit vectors of itemsets occupy the same fixed size which depends on the number of transactions in a given database. Much memory and time are thus needed for computing the intersection among bit vectors. In practice, there are usually many '0' bits in a bit vector. The bit vector of an itemset with many '0' bits can thus be shortened to reduce space and time. In this section, the scheme of dynamic bit vectors (DBVs) is designed to solve the above problem.

The data structure for dynamic bit vectors

In this section, the data structure used to represent a dynamic bit vector is described. Each dynamic bit vector consists of two elements: position (abbreviated as pos) and bit vector. The first element, position, points out the position of the first non-zero byte in the bit vector. Note that for the convenience of programming, the first position in a bit vector is set as position 0. The second element, bit vector, is a list of bytes representing the bit vector in bytes after removing the zero bytes at the front and at the tail. For example, assume a bit vector for an itemset is shown in Figure 1. It has the length of 40 bytes and is represented in the decimal format. The first non-zero byte appears in the tenth position (beginning from the zero position). Figure 2 shows the DBV representation of it.

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

Computing the intersection of two DBVs

As mentioned above, finding the intersection of two bit vectors is an important operation for getting the support of an itemset. It can also be easily achieved as follows for DBVs. Initially, the larger position value in the two DBVs is determined, and the AND operations are performed on the two DBVs from that position. If an initial resulting value is 0, then the position value of the outcome DBV is increased by 1 until the first non-zero resulting value is reached. Next, from the position of non-zero byte, all the resulting bytes by the AND operation are kept unless the last continuous zero bytes. An example is given below to illustrate the intersection operation on two DBVs. Assume there are two 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}} and their intersection is to be found. Because the position value (13) of the second DBV is larger than that (10) of the first, the AND operation then begins from position 13, at which the result of 0 and 4 is 0. The resulting position then moves backward to 14. Again, the result is 0 and 3, which is 0. The position then moves backward to 15. The same process is done until the position is 19, at which the resulting byte is 7 and 6, which is 6 and not equal to zero. The rest bytes of the two DBVs are then performed by the AND operator, and the results are all 0. The resulting DBV is then {19, {6}}. The process is shown in Figure 3.

The pseudo code for computing the intersection of two

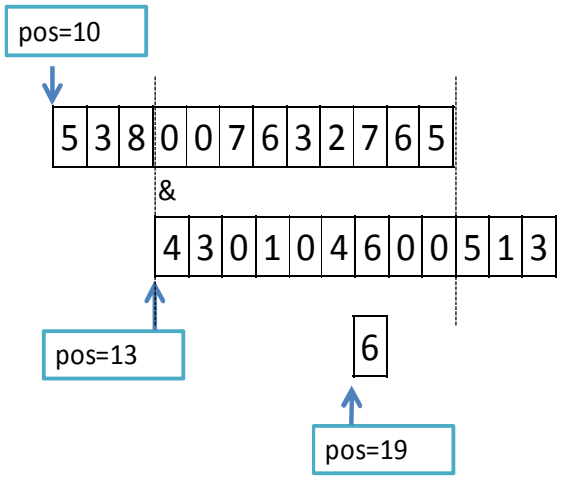

Figure 3. An example for computing the intersection of two DBVs.

```
Input: Two DBVs: {pos1, Bit-vector1} and {pos2, Bit-vector2}.
Output: The resulting DBV: {pos, Bit-vector}.
Method:
1.      pos = Max(pos1, pos2);        // Find the maximal position
2.      i = pos1 < pos2 ? pos2 – pos1 : 0;  //Find the initial byte of Bit-vector1 for intersection
3.      j = pos1 < pos2 ? 0 : pos1 – pos2;  // Find the initial byte of Bit-vector2 for intersection
4.      count = |Bit-vector1| - i < |Bit-vector2| - j ? |Bit-vector1| - i : |Bit-vector2| - j;  // Determine the number of
                                                                                              // bytes for checking
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;}
```

Figure 4. The pseudo code for computing the intersection of two DBVs.

DBVs is presented in Figure 4.

In Figure 4, the maximal position of the two given DBVs is first obtained at Line 1. Then the number of bytes for checking is determined from Lines 2 to 4. The position for the first non-zero byte in the resulting DBV is found from lines 5 to 7. At Line 8, the two variables *i1* and *j1* are the last positions that have to be checked for two given bit vectors. The position for the last non-zero byte in the resulting DBV is found from Lines 9 to 11. The intersection results are actually done at Lines 12 to 14.

Fast computing the support of an itemset from a DBV

The BitTable-based approach may consume more time for computing the intersection among bit vectors and for counting the number of '1' bits in the resulting bit vector than the proposed DBV approach. For example, assume the support of the itemset $X = \{x_1, x_2, \ldots, x_k\}$ is to be calculated. The calculation for Bit-vector(X) = Bit-vector(x_1) $\cap$ Bit-vector(x_2) $\cap$ … $\cap$ Bit-vector(x_k) is done first. After that, Bit-vector(X) is scanned to count the number of '1' bits. The complexity of the counting in the BitTable-based approach is thus O(nk), where n is the number of transactions and k is the length of itemset X. The process can be fast performed on the proposed scheme of DBVs because the length to be checked is shorter.

Additionally, we may use a look-up table with 256 elements to speed up the counting. The table maps each

Table 1. The look-up table used to speed up the counting of '1' bits.

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

Table 2. An example database.

Transactions	Items
1	*A, B, D, E*
2	*B, C, E*
3	*A, B, D, E*
4	*A, B, C, E*
5	*A, B, C, D, E*
6	*B, C, D*

Table 3. The BV and DBV representation of the 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}}

number which can be represented by a byte to the number of '1' bits in the byte. It is shown in Table 1.

With the aid of the look-up table, the number of '1' bits in each byte of a resulting bit vector is known immediately. Therefore, the complexity for the counting of an itemset is O(*m*), where *m* is the number of bits in its DBV. The proposed approach is more efficient than the previous BitTable-based approach.

MINING FREQUENT ITEMSETS BASED ON THE DBV SCHEME

In the section, the approach for mining frequent itemsets based on the DBV scheme is proposed. A tree structure called the DBV tree is used to help mine frequent itemsets efficiently. It is described as follows.

DBV tree

The DBV tree is an extension of a prefix tree with the DBVs stored. In a DBV tree, each node includes two elements, *X* and DBV(*X*), where *X* is an itemset and DBV(*X*) is the dynamic bit vector of *X*. An arc connects node *X* to node *Y* if *X* has the same (|*Y*| - 1) prefix items as *Y*. For example, consider the database as in Table 2. It consists of six transactions.

The bit-vector and the dynamic bit-vector representation of the items in Table 2 is shown in Table 3. Note that the positions are from right to left in the bit-vector representation. The DBV tree constructed from the database in Table 3 is shown in Figure 5.

The first level of the DBV tree in Figure 5 contains single items and their DBVs. Each node *X* at a certain level is combined with the other items to create nodes at higher levels. For example, consider node *A* at the first level. It will be combined with the other items as follows.

(i) A joins B to create a new node AB. Since DBV(A) = {0,{29}} and DBV(B) = {0,{63}}, DBV(AB) = DBV(A) ∩ DBV(B) = {0,{29}} ∩ {0,{63}} = {0,{29and63}} = {0,{29}};
(ii) A joins C to create a new node AC with DBV(AC) = {0, {24}};
(iii) A joins D to create a new node AD with DBV(AD) = {0,{21}};
(iv) A joins E to create a new node AE with DBV(AE) = {0,{29}}.

After that, each child node of *A* will be further processed to create the grandchildren of *A*. This process is repeated recursively until the whole DBV tree is built. Note that only frequent itemsets are stored in a DBV tree.

The DBV-FI algorithm

In the section, the proposed algorithm for constructing a DBV tree and mining FIs from a database is described. It is shown in Figure 6 that presents the algorithm for mining FIs using a DBV tree. It first creates a set nodes *L* of all frequent items and their DBVs, and sorts the nodes in *L* in the increasing supports. The procedure DBV-EXTEND then extends the nodes in *L* to one more level by combining the nodes following them. With each pair (*X*, *Y*), this procedure will compute the intersection of DBV(*X*) and DBV(*Y*) using the algorithm in Figure 4. If the number of '1' bits in the resulting DBV is greater than or equal to *minSup*, then a new node *XY* with DBV(*XY*) is frequent and is added to L_i. After L_i is created, the algorithm will be called recursively to create all child nodes of the nodes in L_i.

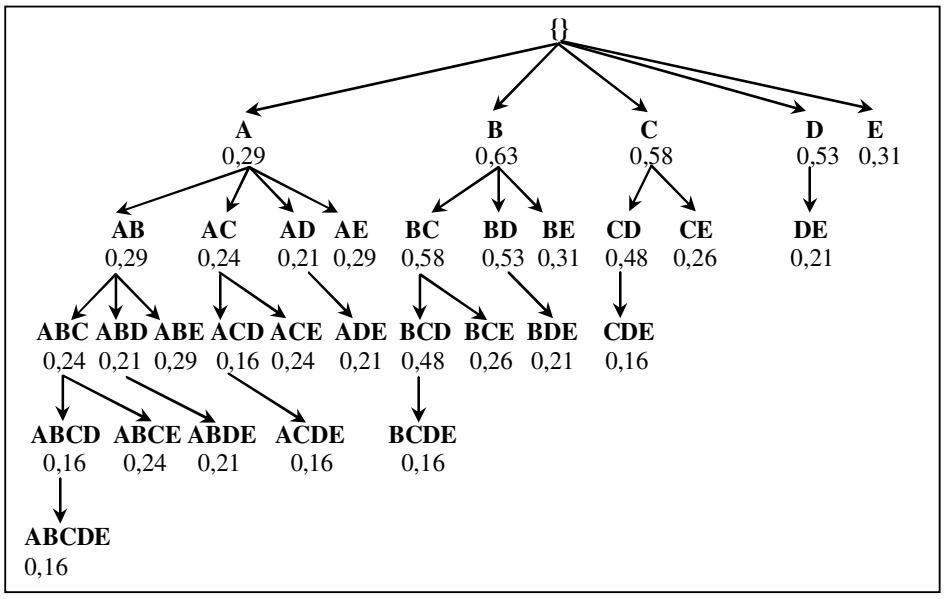

Figure 5. The DBV tree constructed from the database in Table 3.

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*.
Method:

DBV-FI(D, *minSup*)
1. $L = \{ i_{DBV(i)} \mid i \in I \wedge \sigma(i) \geq minSup\}$
2. SORT(L) // *in an increasing order of the supports*
3. DBV_EXTEND(L, *minSup*)

DBV_EXTEND(L, *minSup*)
4. $\forall l_i \in L$:
5. Create the new set L_i by joining l_i with l_j following l_i in L:
6. Compute DBV($l_i \cup l_j$)
7. If the number of '1' bits in DBV($l_i \cup l_j$) $\geq$ *minSup*, then add $l_i \cup l_j$ and DBV($l_i \cup l_j$) into L_i
8. If $|L_i| \geq 2$, then call DBV_EXTEND(L_i, *minSup*)

Figure 6. The proposed algorithm for constructing a DBV tree and mining FIs.

An example

An example is given in this section to illustrate the algorithm above. Consider the previous database in Table 3 and assume the *minSup* value is set at 30%. The DBV tree constructed is shown in Figure 7. The single items are first sorted according to their support increasingly as $L = \{A_{0,29}, C_{0,58}, D_{0,53}, E_{0,31}, B_{0,63}\}$. After that, the procedure DBV-EXTEND(L, *minSup*) is called. Take the processing for node $A_{0,29}$ as an example. It proceeds as follows: (i) $L_i = \varnothing$; (ii) $A_{0,29}$ joins $C_{0,58}$ into a new node $AC_{0,24}$ with $\sigma(AC) = 2 \geq$ minSup, so $AC_{0,24}$ is added into L_i ($\{AC_{0,24}\}$), which keeps the list of child nodes to be processed later. (iii) $A_{0,29}$ joins $D_{0,53}$ into a new node $AD_{0,21}$ with $\sigma(AD) = 3 \geq$ minSup, so $AD_{0,21}$ is added into L_i ($\{AC_{0,24}, AD_{0,21}\}$); (iv) $A_{0,29}$ joins $E_{0,31}$ into a new node $AE_{0,29}$ with $\sigma(AE) = 4 \geq$ minSup, so $AE_{0,29}$ is added into L_i ($\{AC_{0,24}, AD_{0,21}, AE_{0,29}\}$); (v) $A_{0,29}$ joins $B_{0,63}$ into a new

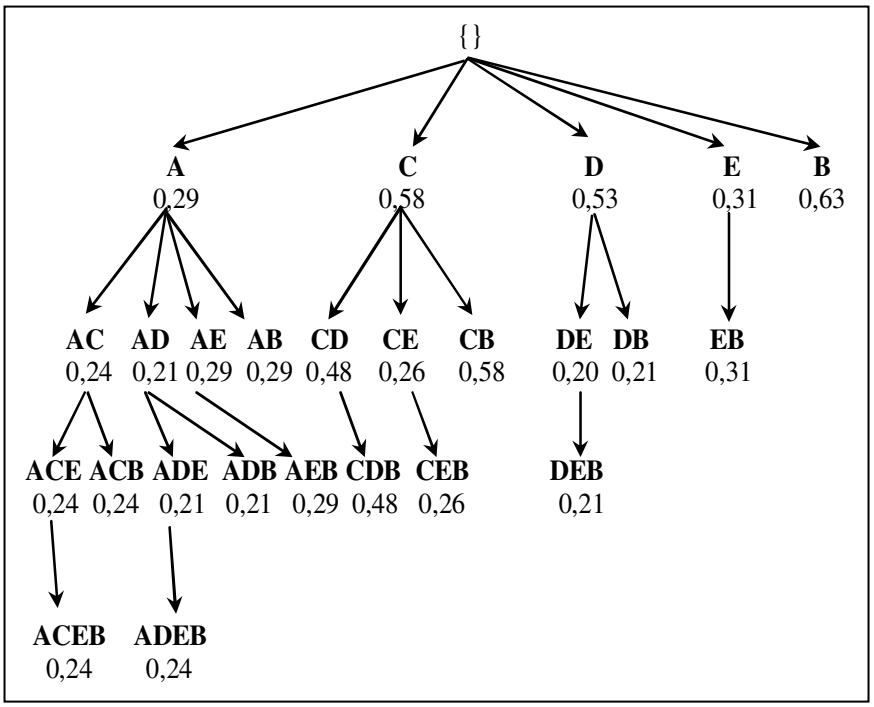

Figure 7. The DBV tree constructed by the DBV-FI algorithm.

Table 4. Features of the databases adopted.

Database	#Trans	#Items
Chess	3196	76
Mushroom	8124	120
Pumsb_star	49046	7117
Connect	67557	130
Accidents	340183	468

node $AB_{0,29}$ with $\sigma(AB) = 4 \geq minSup$, so $AB_{0,29}$ is added into L_i ($\{AC_{0,24}, AD_{0,21}, AE_{0,29}, AB_{0,29}\}$).

After considering node A with all the nodes following it, DBV-EXTEND(L_i, minSup) will be called again to recursively consider the four nodes $\{AC_{0,24}, AD_{0,21}, AE_{0,29}, AB_{0,29}\}$ in L_i. Take the processing for node $AC_{0,24}$ as an example. It proceeds as follows.

(i) $L_i = \varnothing$;

(ii) $AC_{0,24}$ joins $AD_{0,21}$ into a new node $ACD_{0,16}$ with $\sigma(ACD) = 1 < minSup$, so $ACD_{0,16}$ is skipped;

$AC_{0,24}$ joins with $AE_{0,29}$ into a new node $ACE_{0,24}$ with $\sigma(ACE) = 2 \geq minSup$, so $ACE_{0,24}$ is added to L_i ($\{ACE_{0,24}\}$);

$AC_{0,24}$ joins with $AB_{0,29}$ into a new node $ACB_{0,24}$ with $\sigma(ACB) = 2$, so $ACB_{0,24}$ is added into L_i ($\{ACE_{0,24}, ACB_{0,24}\}$);

The same procedure is then done with the other nodes.

The final DBV tree is shown in Figure 7. From the DBV tree, there are 25 frequent itemsets found.

EXPERIMENTAL RESULTS

Experiments were conducted to show the performance of the proposed algorithm. The all algorithms were implemented on a Centrino Core 2 Duo (2×2.53 GHz), with 4GBs RAM of memory and running Windows 7. The Eclat (Tidset-based) (Zaki and Hsiao, 2005) and the Index-BitTableFI (Bit-vector-based, Song et al., 2008) approaches were also executed for comparison. All the algorithms were coded in C# 2008. Five databases from http://fimi.cs.helsinki.fi/data/ (download on April 2005) were used for the experiments, with their features displayed in Table 4. Table 5 shows the number of frequent itemsets from the five databases under different minimum support values.

Experiments were then made to compare the mining time of the proposed approach with Eclat and Index-BitTableFI for different *minSup* values. BitTableFI was not compared because Index-BitTableFI was always faster than BitTableFI (Song et al., 2008). The results for the five databases were shown in Figures 8 to 12.

It could be observed that the DBV-FI algorithm was always faster than the other two in all the results. For example, Figure 8 shows the mining time of Eclat, Index-

Table 5. Number of FIs from the five databases under different *minSup* values.

Database	*minSup* (%)	#FIs
Chess	80	8227
	75	20993
	70	48731
	65	111239
Mushroom	40	565
	30	2735
	20	53583
	10	574431
Pumb_star	55	305
	50	679
	45	1913
	40	27354
Connect	98	180
	94	4223
	90	27127
Accidents	80	149
	70	529
	60	2074
	50	8057

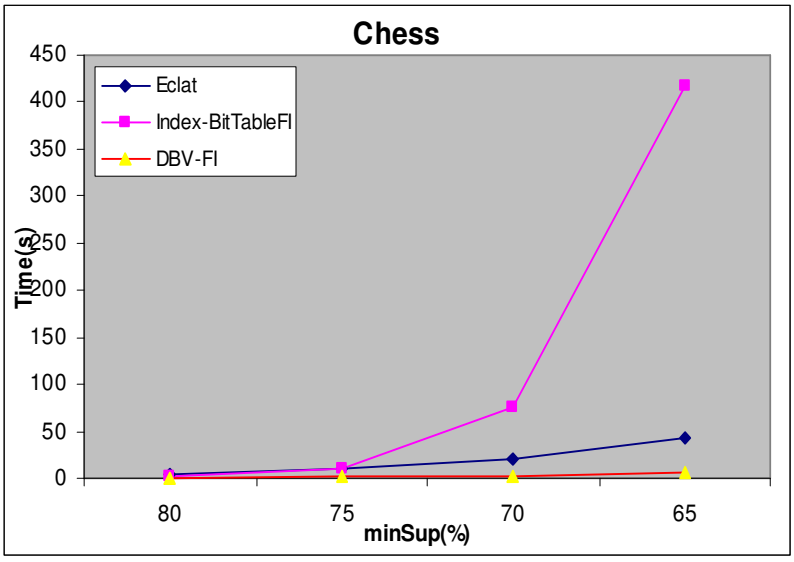

Figure 8. Execution time of the three algorithms for Chess under different *minSup* values.

BitTableFI and DBV-FI for the Chess database. With *minSup* = 65%, the mining time of Eclat is 42.1(s), of Index-BitTableFI is 417.75(s), and of DBV-FI is only 6.63(s). Besides, Index-BitTableFI was faster than Eclat in the two databases of Pumsb_star and Connect, and slower in the rest databases (Chess, Mushroom, Accidents).

Next, experiments were conducted to compare the total memory usage (in MBs) of the three algorithms. The results for the five databases under different *minSup*

Table 6. Memory usage of the three algorithms for the five databases.

Database	*minSup* (%)	Memory usage (in MBs)		
		Eclat	Index-BitTableFI	DBV-FI
Chess	80	42.19	3.87	3.11
	75	102.21	9.89	7.93
	70	224.25	22.90	18.40
	65	480.17	53.69	41.98
Mushroom	40	4.30	0.82	0.53
	30	15.63	3.51	2.45
	20	188.44	56.76	33.32
	10	1102.80	483.91	318.30
Pumb_star	55	17.57	2.81	1.78
	50	36.02	7.33	3.97
	45	90.05	23.82	11.19
	40	656.32	196.89	159.94
Connect	98	45.72	1.68	1.45
	94	1038.19	37.70	34.99
	90	6439.86	41.26	37.22
Accidents	80	83.11	12.82	6.04
	70	266.41	49.35	21.45
	60	909.51	188.29	84.11
	50	2915.39	714.86	326.74

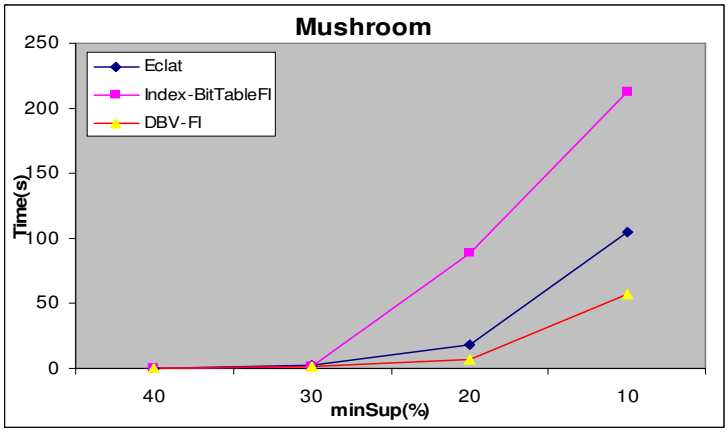

Figure 9. Execution time of the three algorithms for Mushroom under different *minSup* values.

values are shown in Table 6.

It could be seen from Table 6 that Eclat always consumed more memory than the other two and DBV-FI always consumed the smallest memory among the three. For example, consider the Chess database with *minSup* = 65%. The total memory usage for Eclat was 480.17 MBs, for Index-BitTableFI was 53.69 MBs, and for DBV-FI was 41.98 MBs.

Conclusions

In this paper, we have proposed a new method for mining FIs from transaction databases based on the scheme of dynamic bit vectors. The contributions could be divided into the following three parts. Firstly, dynamic bit vectors are used to compress a database in one scan and with shorter length. Secondly, an algorithm for fast computing

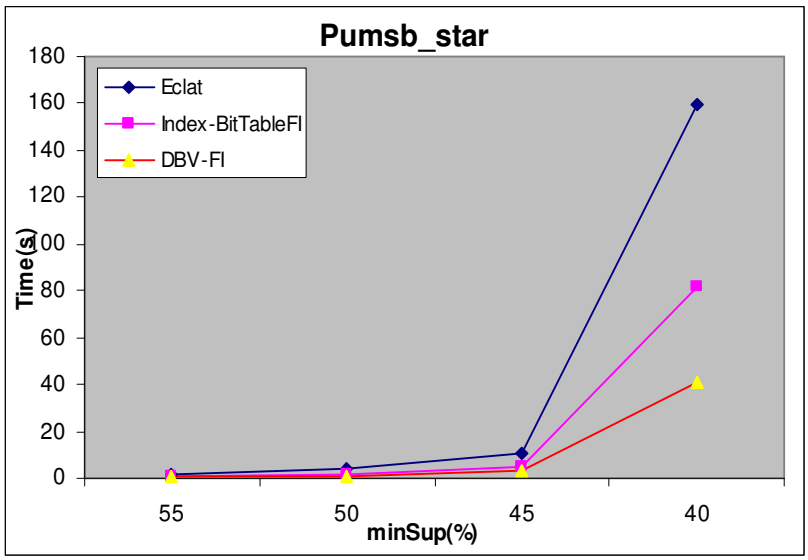

Figure 10. Execution time of the three algorithms for Pumsb_star under different *minSup* values.

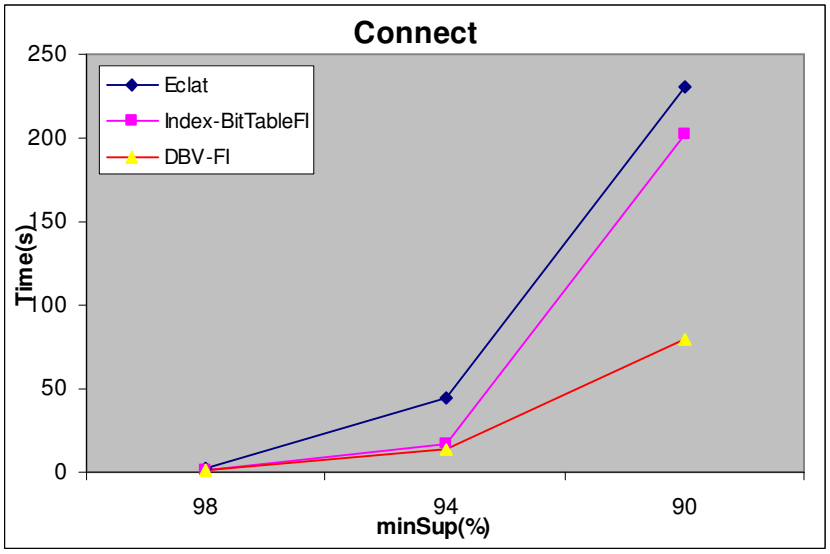

Figure 11. Execution time of the three algorithms for Connect under different *minSup* values.

the intersection between two DBVs and for counting the number of '1' bits is designed. Finally, an algorithm for mining FIs based on a DBV tree is developed. Experimental results also show the efficiency of the proposed approach in both the mining time and the total memory usage. The proposed approach has the following weak point. When the first and the last bytes in a bit vector are non zero, the DBV scheme will not reduce any memory. However, the proposed approach will, in average, be able to save some memory. In the future, we will attempt to use multiple positions to remove zero bytes existing in the middle of bit-vectors. Besides, mining frequent itemsets in incremental databases has been developed in recent years (Bailey and Loekito, 2010; Hong and Wang, 2010; Hong et al., 2009; Li et al., 2006; Lin et al., 2010; Thomas et al., 1997; Valtchev et al.,

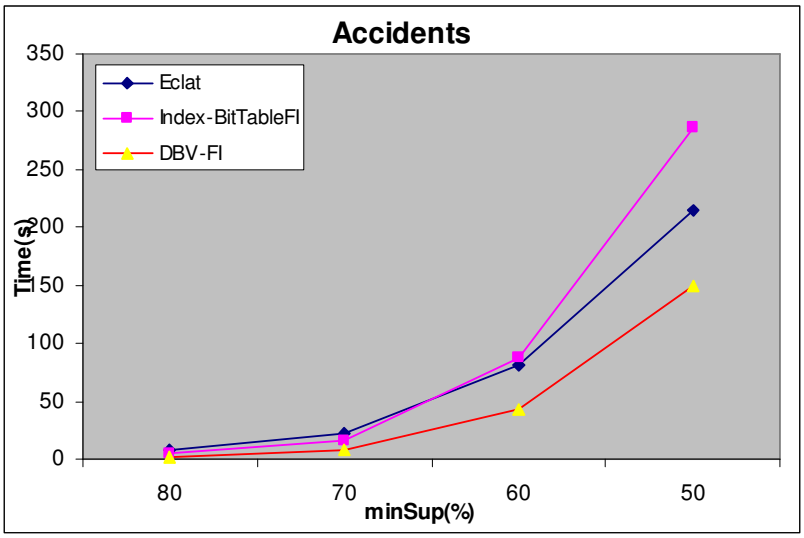

Figure 12. Execution time of the three algorithms for Accidents under different *minSup* values.

2008; Zhang et al., 2009). We will also study to apply the DBV scheme for fast mining frequent itemsets and frequent closed itemsets from this kind of databases.

ACKNOWLEDGEMENT

This work was supported by Vietnam's National Foundation for Science and Technology Development (NAFOSTED), project ID: 102.01-2010.02.

REFERENCES

Agrawal R, Srikant R (1994). Fast algorithms for mining association rules. VLDB'94, pp. 487-499.
Bailey J, Loekito E (2010). Efficient incremental mining of contrast patterns in changing data. Inf. Process. Lett., 110(3): 88-92.
Dong J, Han M (2007). BitTableFI: An efficient mining frequent itemsets algorithm. Knowl.-Based Syst., 20(4): 329 – 335.
Grahne G, Zhu J (2005). Fast algorithms for frequent itemset mining using FP-trees. IEEE Trans. Knowl. Data Eng., 17(10): 1347-1362.
Han J, Pei J, Yin Y (2000). Mining frequent patterns without candidate generation. SIGMODKDD'00, pp 1 – 12.
Hong TP, Wang, CJ (2010). An efficient and effective association-rule maintenance algorithm for record modification. Expert Syst. Appl., 37(1): 618-626
Hong TP, Lin CW, Wu YL (2009). Maintenance of fast updated frequent pattern trees for record deletion. Comput. Stat. Data Anal., 53(7): 2485-2499.
Li X, Deng JH, Tang S (2006). A fast algorithm for maintenance of association rules in incremental databases. ADMA, pp. 56-63.
Lin CW, Hong TP, Lu WH (2010). Efficient modification of fast updated FP-trees based on pre-large concepts. Int. J. Innov. Comput. Inf. Control., 6(11): 5163-5177.
Song W, Yang B, Xu Z (2008). Index-BitTableFI: An improved algorithm for mining frequent itemsets. Knowl.-Based Syst., 21(6): 507-513.
Thomas S, Bodagala S, Alsabti K, Ranka S (1997). An efficient algorithm for the incremental updation of association rules in large databases. SIGKDD'97, pp. 263-266.
Valtchev P, Missaoui R, Godin R (2008). A framework for incremental generation of closed itemsets. Discrete Appl. Math., 156(6): 924-949.
Zhang S, Zhang J, Jin Z (2009). A decremental algorithm of frequent itemset maintenance for mining updated databases. Expert Syst. Appl., 36(8): 10890-10895.
Zaki MJ, Parthasarathy S, Ogihara M, Li W (1997). New algorithms for fast discovery of association rules. 3rd Int. Conf. Knowl. Disc. Data Min. (KDD), pp. 283-286.
Zaki MJ, Hsiao CJ (2005). Efficient algorithms for mining closed itemsets and their lattice structure. IEEE Trans. Knowl. Data Eng., 17(4): 462-478.