# An Efficient Hash-Based Method for Discovering the Maximal Frequent Set

*Don-Lin Yang, Ching-Ting Pan and Yeh-Ching Chung*
Department of Information Engineering,
Feng Chia University, Taichung, Taiwan 407
TEL: 886-4-2451-7250x3700
FAX: 886-4-2451-6101
Email: {dlyang, ctpan, ychung}@iecs.fcu.edu.tw

## *Abstract*

The association rule mining can be divided into two steps. The first step is to find out all frequent itemsets, whose occurrences are greater than or equal to the user-specified threshold. The second step is to generate reliable association rules based on all frequent itemsets found in the first step. Identifying all frequent itemsets in a large database dominates the overall performance in the association rule mining. In this paper, we propose an efficient hash-based method, *HMFS*, for discovering the maximal frequent itemsets. The *HMFS* method combines the advantages of both the *DHP* (*Direct Hashing and Pruning*) and the *Pincer-Search* algorithms. The combination leads to two advantages. First, the *HMFS* method, in general, can reduce the number of database scans. Second, the *HMFS* can filter the infrequent candidate itemsets and can use the filtered itemsets to find the maximal frequent itemsets. These two advantages can reduce the overall computing time of finding the maximal frequent itemsets. In addition, the *HMFS* method also provides an efficient mechanism to construct the maximal frequent candidate itemsets to reduce the search space. We have implemented the *HMFS* method along with the *DHP* and the *Pincer-Search* algorithms on a Pentium III 800 MHz PC. The experimental results show that the *HMFS* method has better performance than the *DHP* and the *Pincer-Search* algorithms for most of test cases. In particular, our method has significant improvement over the *DHP* and the *Pincer-Search* algorithms when the size of a database is large and the length of the longest itemset is relatively long.

## 1. Introduction

The data mining refers to extract knowledge from a large database [10]. In recent years, data mining has attracted a growing amount of attention in the database community. One of the most important reasons is the fast growing huge amount of data that is far from the ability of human to analyze. Discovering association rules from a database is an important technique in the data mining area. For example, for the retail business, a database may contain sales transactions. We can analyze customers buying habits through discovering the associations among products, that is, a customer who buys some products will also buy other products in the same transaction. These mined rules are very useful for the retailer to improve the marketing strategies. Other popular applications include mining web contents, mining web path traversal patterns [11], etc.

The process of mining association rules can be decomposed into two steps [13]. The first step is to find out all frequent itemsets, whose occurrences are greater than or equal to the user-specified threshold. The second step is to generate reliable association rules based on all frequent itemsets found in the first step. The cost of the first step is much more expensive than the second step. Therefore, much research focused on developing efficient algorithms for finding frequent itemsets. A well-known *Apriori* algorithm proposed by R. Agrawal and R. Sriank [13] was the first efficient method to find the frequent itemsets. The main contribution of the *Apriori* algorithm is that it utilizes the downward closure property, i.e., any superset of an infrequent itemset must be an infrequent itemset, to efficiently generate candidate itemsets for the next database scan. By scanning a database $k$ times, the *Apriori* algorithm can find all frequent itemsets of a database, where $k$ is the length of the longest frequent itemset in the database.

Many methods based on the *Apriori* algorithm have been proposed in the literature. In general, they can be classified into three categories, reduce the number of candidate itemsets, reduce the number of database scans, and the combination of bottom-up and top-down search.

- Reduce the number of candidate itemsets: Methods in this category try to generate a small number of candidate itemsets efficiently in order to reduce the computational cost. The hash-based algorithm *DHP* (*Direct Hashing and Pruning*) proposed by Park *et al.* [6] is an example. The main contribution of the *DHP* algorithm is that it uses a hash table to filter the huge infrequent candidate itemsets before the next database scan. However, the

*DHP* algorithm needs to perform database scans as many times as the length of the longest frequent itemset in a database.

- Reduce the number of database scans: Scanning a database iteratively is time consuming. Thus, methods in this category try to reduce database scans aiming at reducing disk I/O costs. The *Partition* algorithm proposed by Savasere *et al.* [1] generates all frequent itemsets with two database scans. The *Partition* algorithm divides the database into several blocks such that each block in the database can be fitted into the main memory and can be processed by the *Apriori* algorithm. However, the *Partition* algorithm examines much more candidate itemsets than the *Apriori* algorithm. Brin *et al.* [17] proposed the *DIC* algorithm that also divides the database into several blocks like the *Partition* algorithm. Unlike the *Apriori* algorithm, once some frequent itemsets are obtained, the *DIC* algorithm can generate the candidate itemsets in different blocks and then add them to count for the rest blocks. However, the *DIC* algorithm is very sensitive to the data distribution of a database.
- The combination of bottom-up and top-down search: Methods in this category are also based on the downward closure. They obtain the frequent itemsets in a bottom-up fashion like the *Apriori* algorithm. In the mean time, they use the infrequent itemsets found in the bottom-up direction to split the maximal frequent candidate itemsets in the top-down direction in each round. The advantage is that once the maximal frequent itemsets are obtained, all subsets of the maximal frequent itemsets are also identified. Therefore, all subsets of the maximal frequent itemsets do not need to be examined from the bottom-up direction. Without the top-down pruning, they need to scan database as many times as the length of the longest frequent itemset. However, the improvement is not clear when the length of the longest frequent itemset is relatively short. The *Pincer-Search* algorithm proposed by D. Lin *et al.* [2] and the *MaxMiner* algorithm proposed by R.J. Bayardo [6] are two examples. In these two methods, the generation of the maximal frequent candidate itemsets is not efficient. They may spend a lot of time on finding the maximal frequent itemsets.

In this paper, we propose an efficient hash-based method to generate the maximal frequent itemsets (*HMFS*) in the category of the combination of bottom-up and top-down search. The proposed method combines the advantages of both the *DHP* and the *Pincer-Search* algorithms. Unlike the *DHP* algorithm, the *HMFS* method is very efficient in reducing the number of database scans when the length of the longest frequent itemset is relatively long. Unlike the *Pincer-Search* algorithm, the *HMFS* method can filter the infrequent itemsets with the hash technique from the bottom-up direction and then can use the filtered itemsets to find the maximal frequent itemsets in the top-down direction. In addition, the *HMFS* method also provides an efficient mechanism to construct the maximal frequent candidate itemsets. To evaluate the performance of the *HMFS* method, we have implemented this method along with the *DHP* and the *Pincer-Search* algorithms on a Pentium III 800 MHz PC. The experimental results show that our method has better performance than the *DHP* and the *Pincer-Search* algorithms for most of test cases. In particular, our method has significant improvement over the *DHP* and the *Pincer-Search* algorithms when the size of a database is large and the length of the longest itemset is relatively long.

The rest of the paper is organized as follows. Section 2 introduces the terminology used in this paper. The detail of the proposed *HMFS* method is presented in section 3. The experimental results of the proposed method are presented in section 4.

## 2. Preliminaries

In this section, we introduce some basic definitions of the association rule mining. Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of distinct items.

Definition 1: A database $D$ is a set of transactions, where each transaction $T$ contains a set of items in $I$.

Definition 2: A subset of $I$ is called an *itemset*. An itemset is called a $k$-itemset if it contains $k$ items.

Definition 3: The support of an itemset $X \subseteq I$ for a database, denoted as *support*($X$), is the number of transactions in the database that contain all items in $X$.

Definition 4: An itemset is called a frequent itemset if its support is greater than or equal to some user-specified minimum support. Otherwise, it is an infrequent itemset. The set of all frequent $k$-itemsets is denoted as $L_k$.

Definition 5: Given $L_{k-1}$, the set of all candidate $k$-itemsets, $C_k$, is defined as $L_{k-1} \times L_{k-1} = \{X \cup Y \mid X, Y \in L_{k-1}, \mid X \cap Y \mid = k-2\}$, where $k > 1$.

Definition 6: A frequent itemset is called a maximal frequent itemset if it is not a subset of any other frequent itemsets.

Definition 7: An association rule is defined as $X \Rightarrow Y$, where $X, Y \subseteq I$, $X, Y \neq \varnothing$ and $X \cap Y = \varnothing$.

The task of the association rule mining is to discover all association rules that satisfy the minimum support and the minimum confidence. We now give an example to explain the terms described above. Consider the database shown in Table 1. Assume that $I = \{A, B, C, D, E, F\}$ and there are five transactions in the database $D$. Let the minimum support and the minimum confidence be 40% and 100%, respectively. All frequent itemsets are shown in Table 2. The itemsets whose support smaller than $5 \times 40\% = 2$ are infrequent itemsets. In this example, itemsets $D$, $AB$, and $AE$ are infrequent itemsets. Note that $D$, $AB$, and $AE$ are the short-cut of $\{D\}$, $\{A, B\}$, and $\{A, E\}$, respectively. In this paper, all itemsets are represented by the short-cut notation. We have $L_1 = \{A, B, C, E, F\}$, $L_2 = \{AF, BC, BF, CE, EF, AC, BE, CF\}$, $L_3 = \{ACF, BCE, BCF, BEF, CEF\}$ and $L_4 = \{BCEF\}$. The maximal frequent itemsets are $ACF$ and $BCEF$. $BC \Rightarrow EF$ is one of the association rules that can be derived from the database shown in Table 1. Its confidence is $support(BC \cup EF)/support(BC) = 2/2 \times 100\% = 100\%$. $BC \Rightarrow EF$ is a frequent and reliable association rule.

**Table 1: Database *D*.**

| Transaction | Items |
|---|---|
| 1 | *A, C, D* |
| 2 | *B, C, E, F* |
| 3 | *A, B, C, E, F* |
| 4 | *B, E* |
| 5 | *A, C, F* |

**Table 2: All frequent itemsets.**

| Support | Itemsets |
|---|---|
| 2 | *AF, BC, BF, CE, EF, ACF, BCE, BCF, BEF, CEF, BCEF* |
| 3 | *A, B, E, F, AC, BE, CF* |
| 4 | *C* |

## 3. The *HMFS* Method

Our *HMFS* method combines the advantages of both the *DHP* and *Pincer-Search* algorithms. In the *HMFS* method, it uses the hash technique of the *DHP* algorithm to filter infrequent itemsets in the bottom-up direction. Then it uses a top-down technique that is similar to the *Pincer-Search* algorithm to find the maximal frequent itemsets. The main difference of the top-down techniques between the *HMFS* method and the *Pincer-Search* algorithm is that the *HMFS* method provides a more efficient mechanism to initialize the set of maximal frequent candidate itemsets than that of the *Pincer-Search* algorithm. By combining the advantages of the *DHP* and *Pincer-Search* algorithms, the number of database scans and the search space of items can be reduced. The algorithm of the *HMFS* method is given as follows.

```
Algorithm HMFS()
1. In the first round, scan the database $D$ to count the support of all
   1-itemsets and build a hash table $H_2$;
2. $C_2$ is constructed by $L_1 \times L_1$ and is filtered by $H_2$;
3. call construct_maximal_frequent_candidate_itemsets($C_2$, $H_2$);
4. In the second round, divide $D$ into several blocks;
5. for all blocks $b \in D$ do
6.    Count the supports of itemsets in $C_2$ and $MFCS$;
7.    call process_collision($C_2$, $H_2$) to process the collisions of the hash
      buckets;
8.    Move the maximal frequent itemsets from $MFCS$ to the hash tree;
9. Apply the Pincer-Search algorithm to the rest of rounds;
End_of_algorithm
```

```
Function construct_maximal_frequent_candidate_itemsets($C_2$, $H_2$)
1.  $C_{max} = \{x = x_1x_2x_3 \ldots x_n \mid x_1x_2, x_1x_3, \ldots, x_1x_n \in C_2$, where $n > 2\}$;
2.  $m = 3$; $MFCS = \emptyset$;
3.  for all $x = x_1x_2x_3 \ldots x_n \in C_{max}$ do
4.     Push $x$ into the stack initially;
5.     while the stack is not empty do
6.        Popup an element $x$ from the stack;
7.           while $m \neq n$ do
8.           $k = h_2(x_ix_m)$, for $i = 2, 3, \ldots, m-1$;   // $h_2$ is a hash function
9.           if ($H_2(k)$ < minimum support) then
10.             Split $x_1x_2x_3 \ldots x_n$ into two $(n-1)$-itemsets,
                   $x' = x_1x_2x_3 \ldots x_i \ldots x_{m-1}x_{m+1} \ldots x_n$ and
                   $x'' = x_1x_2x_3 \ldots x_{i-1}x_{i+1} \ldots x_mx_{m+1} \ldots x_n$;
11.          if is_maximal_candidate_itemset($x''$) = true then
                push $x''$ into the stack;
12.          else discard $x''$;
13.          if is_maximal_candidate_itemset($x'$) = true
14.          then continue processing $x'$;
15.               $m$++;
16.          else $x'$ is discarded;
17.          break;
18.         if ($m = n$ and the length of $x > 2$) then $MFCS = MFCS \cup \{x\}$;
19.  return $MFCS$;
End_of_construct_maximal_frequent_candidate_itemsets
```

```
Function is_maximal_candidate_itemset(itemset $x$)
1. for all itemset $s$ in the stack do
2.    if all items in $x$ are also in $s$ then return false;
3.    else return true;
End_of_is_maximal_candidate_itemset
```

```
Function process_collision($C_2$, $H_2$)
1. for all blocks $b \in D$ do
2.    for all $H_2(k) \geq$ minimum support do
3.       for all $c_i \in C_2$ that hashed into $H_2(k)$, where $i = 1, 2, \ldots, n$ do
4.          if $(H_2(k) - \sum_{i=1}^{n}$ support $(c_i)$ + support $(c_j)$
               < minimum support, $\forall j = 1, 2, \ldots, n$ )
5.          then use the infrequent 2-itemset $c_j$ to split the itemsets in
            $MFCS$;
6. Remove the infrequent 2-itemset $c_j$ from $C_2$;
End_of_ process_collision
```

In the algorithm *HMFS*(), lines 1-2 use the hash technique to filter the infrequent itemsets in $C_2$ in the bottom-up direction. Line 3 constructs the set of maximal frequent candidate itemsets *MFCS*. Line 6 counts the supports of itemsets in *MFCS* and $C_2$. Line 7 splits the maximal frequent candidate itemsets if some conditions are satisfied. Line 8 moves the maximal frequent itemsets from *MFCS* to the hash tree. Line 9 performs the *Pincer-Search* algorithm to get the maximal frequent itemsets.

We first explain how the function *construct_maximal_frequent_candidate_itemsets*() works. Line 1 constructs $C_{max}$ with all 2-itemsets that have the same first item in $C_2$. Lines 3-19 generate the set of maximal frequent candidate itemsets, *MFCS*. The generation process is as follows. Assume that an itemset $x$ in $C_{max}$ is denoted as $x_1x_2x_3 \ldots x_n$. Consider the first $m$

items in $x_1x_2x_3....x_n$, for $m = 3, ..., n$, and examine the 2-item subset $x_ix_m$ of $x$, for $i = 2, 3,..., m-1$. If the number of 2-itemsets in the corresponding hash bucket of $x_ix_m$ is smaller than minimum support, i.e., $x_ix_m$ is not in $C_2$, split $x$ into $x' = x_1x_2x_3...x_i...x_{m-1}x_{m+1}...x_n$ and $x'' = x_1x_2x_3...x_{i-1}x_{i+1}...x_mx_{m+1}...x_n$. Itemsets $x'$ and $x''$ are then compared with elements in the stack. We have the following four cases.

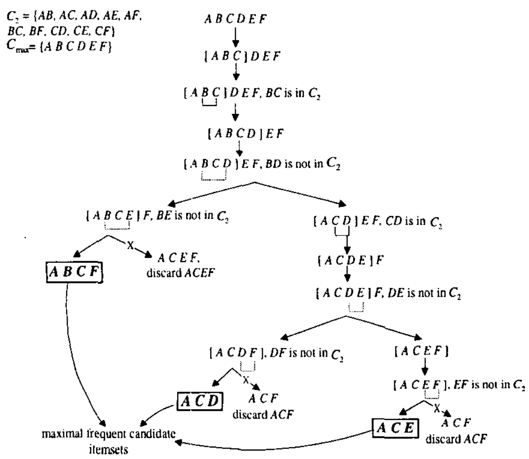
Case 1. All items in $x'$ and $x''$ are also in any element in the stack. Both $x'$ and $x''$ are discarded. The next itemset is popped up from the stack and the generation process continues.

Case 2. Only items in $x'$ are also in any element in the stack. Itemset $x'$ is discarded. The generation process continues to examine $x_{i+1}x_m$ of $x''$.

Case 3. Only items in $x''$ are also in any element in the stack. Itemset $x''$ is discarded. The generation process continues to examine the $x_ix_{m+1}$ of $x'$.

Case 4. Otherwise, itemset $x''$ is pushed into the stack and the generation process continues to examine $x_ix_{m+1}$ of $x'$.

The generation process continues until $m = n$. Then we get a maximal frequent candidate itemset. Once one maximal frequent candidate itemset is generated, the next itemset in the stack is popped up and the generation process is applied until the stack is empty.



$C_2$ = {AB, AC, AD, AE, AF, BC, BF, CD, CE, CF}
$C_{max}$= {A B C D E F}
A B C D E F
[A B C] D E F
[A B C] D E F, BC is in $C_2$
[A B C D] E F
[A B C D] E F, BD is not in $C_2$
[A B C E] F, BE is not in $C_2$
X
A B C F
A C E F, discard ACEF
[A C D] E F, CD is in $C_2$
[A C D E] F
[A C D E] F, DE is not in $C_2$
[A C D F], DF is not in $C_2$
X
A C D
A C F discard ACF
[A C E F]
[A C E F], EF is not in $C_2$
X
A C E
A C F discard ACF
maximal frequent candidate itemsets

**Figure 1: An example of the split process.**

An example of the generation process is shown in Figure 1. Let $C_2$ = {*AB, AC, AD, AE, AF, BC, BF, CD, CE, CF*}. $C_{max}$ is {*ABCDEF*}. Consider the first 3 items *ABC* in *ABCDEF*. Since *BC* is in $C_2$, we examine *ABCD* in *ABCDEF*. Since *BD* is not in $C_2$, *ABCDEF* is split into *ABCEF* and *ACDEF*. Compare *ABCEF* and *ACDEF* with elements in the stack, we have Case 4. *ACDEF* is pushed into the stack and the generation process continues with *ABCEF*. Since *BE* is not in $C_2$, *ABCEF* is split into *ABCF* and *ACEF*. Compare *ABCF* and *ACEF* with elements in the stack, we have Case 2. *ACEF* is discarded. A maximal frequent candidate itemset *ABCF* is obtained. Since the stack is not empty, itemset *ACDEF* is popped up from the stack and the generation process continues in a similar manner. Finally, all maximal frequent candidate itemsets, *ABCF*, *ACD*, and *ACE* are generated from *ABCDEF*.

In *HMFS* method, the collision of the hash buckets cannot be avoided by using the hash technique. The collision may result in an infrequent itemset be used to construct the maximal frequent candidate itemsets. For example, assume that $C_2$ = {*AB, AC, AD, AE, AF, BC, BF, CD, CE, CF*} is given. One of the maximal frequent candidate itemsets of $C_2$ is *ABCF*. Assume that *AC*, a frequent itemset, and *AF*, an infrequent itemset, are hashed into bucket 2. Since *AC* and *AF* are in the same bucket, *AF* cannot be filtered and will be used to construct the maximal frequent candidate itemsets. Function *process_collision*() provides a solution of this problem. In the following, we explain how it works. First, it divides the database into several blocks. In the second round, the supports of elements in $C_2$ and *MFCS* are counted. The number of 2-itemsets hashed into bucket $k$ in $H_2$ is denoted as $H_2(k)$. Assume that there are $n$ 2-itemsets, $c_1, c_2,..., c_n$ in $C_2$, hashed into bucket $k$. An infrequent itemset $c_j$ can be identified by the following equation:

$$H_2(k) - \sum_{i=1}^{n} \text{support}(c_i) + \text{support}(c_j) < \text{minimum support}, \quad \forall j = 1,2,...,n \qquad (1)$$

where the supports of $c_i$ and $c_j$ among the $k$ blocks are denoted as *support*($c_i$) and *support*($c_j$), respectively. In each block scanning, all infrequent itemsets in $C_2$ are identified and are deleted from $C_2$. The identified infrequent itemsets are used to split itemsets in *MFCS* as well.

We now give an example to explain Equation (1). Assume that the number of transactions in a database is 10,000, the minimum support is 0.5%, and $H_2(k)$ is 100. After scanning several blocks in the database, *support*(*AC*) is 70 and *support*(*AF*) is 10. By applying Equation (1), $100-(70+10)+10 = 30 < 10{,}000 \times 0.5\% = 50$. Thus, we can identify *AF* is an infrequent itemset and *AF* can be discarded. The purpose of dividing a database into several blocks is that some infrequent itemsets in $C_2$ may be determined earlier when some blocks are scanned. The maximal frequent candidate itemsets that contain these infrequent itemsets cannot be counted further. Therefore, the division may lead us to identify those maximal frequent candidate itemsets that contain infrequent itemsets earlier and reduce the time of finding the maximal frequent itemsets.

## 4. Experimental results

Table 3: The meanings of all parameters.

| | |
|---|---|
| *D* | Number of transactions |
| *T* | Average size of transactions |
| *I* | Average size of the maximal potentially large itemsets |
| *L* | Number of potentially large itemsets |
| *N* | Number of items |

To evaluate the performance of the proposed method, we have implemented the *HMFS* method in *C* language along with the *DHP* and the *Pincer-Search* algorithms on a Pentium III 800 MHz PC with 512MB of main memory. The program designed by IBM Almaden Research Center is used to generate synthetic databases [5]. This program has been widely used by many researchers [1, 2, 6, 7, 8, 9, 12, 14, 17]. By setting up parameters of the program, we can generate desired databases as benchmarks to evaluate the performance of our method. Table 3 describes all the parameters used in the program. In our experiments, we set $N = 1000$ and $L = 2000$. The number of the hash buckets is 500,000. We designed two tests. In the first test, we compare the relative performance and the number of database scans for the three algorithms on four databases. The results of the first test are shown in Figure 2 and Figure 3.

Figure 2 shows the execution time of these three algorithms for test databases with various minimum supports. In Figure 2, our method is a little slower than the *DHP* algorithm on T10I4D100K when the minimum support is 1%. In this case, the execution time of the *DHP* algorithm and the *HMFS* method are 4 and 6 seconds, respectively. The reason is that the length of the longest itemset is two for T10I4D100K when the minimum support is 1%, i.e., only two database scans are required for T10I4D100K. The *HMFS* method and the *DHP* algorithm all require two database scans. However, the *HMFS* method needs to spend some time on constructing the maximal frequent candidate itemsets based on $C_2$. Therefore, it takes more time than the *DHP* algorithm. For other test cases, the *HMFS* method outperforms the *DHP* and the *Pincer-Search* algorithms. The summary reasons are given as follows.

1. In contrast with the *DHP* algorithm, the *HMFS* method finds the frequent itemsets not only in the bottom-up direction but also in the top-down direction. The execution time is improved since the number of database scans is reduced. The number of database scans is shown in Figure 3. The number of database scans required by the *DHP* algorithm is the length of the longest frequent itemset. In general, the number of database scans of the *HMFS* method is half of that of the *DHP* algorithm when the minimum support = 0.25% and 0.5%.
2. In contrast with the *Pincer-Search* algorithm, the *HMFS* method still has better performance than the *Pincer-Search* algorithm even though the number of database scans required by the *HMFS* method is the same as the *Pincer-Search* algorithm. There are two reasons. First, the *HMFS* method uses the hash table to filter a huge number of infrequent 2-itemsets in the $C_2$ instead of actually counting the supports of all 2-itemsets. Second, it constructs the maximal frequent candidate itemsets by using the hash technique instead of the combination of all distinct 1-itemsets in a database. The search space is reduced substantially.

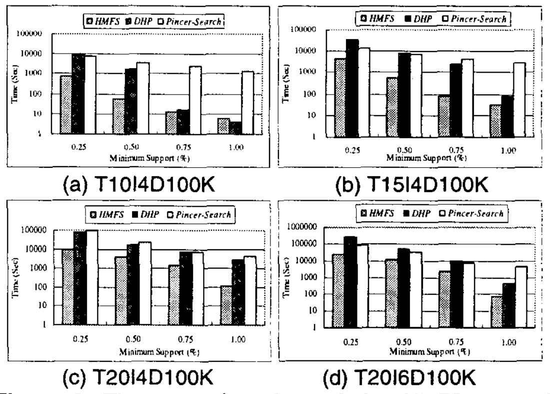


**Figure 2: The execution time of the *HMFS* method, *DHP*, and *Pincer-Search* algorithms on various test databases with increasing minimum supports.**

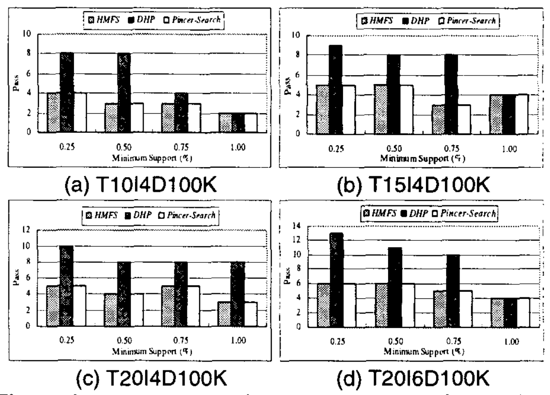
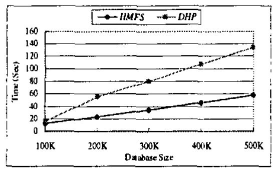
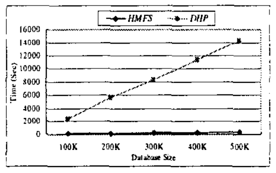


**Figure 3: The number of database scans of the *HMFS* method, *DHP*, and *Pincer-Search* algorithms on various test databases with increasing minimum supports.**

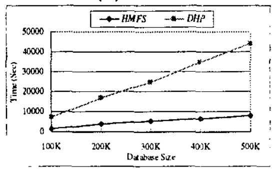
In the second test, we evaluate the performance of the *HMFS* method and the *DHP* algorithm on the test databases with various database sizes. The results of the second test are shown in Figure 4. The performance of the *Pincer-Search* algorithm is not included since it takes too much time to get the execution results for the test databases. In Figure 4, the number of transactions in the test databases is set from 100K to 500K and the minimum support is 0.75%. From Figure 4, we can see that both the execution time of *HMFS* and *DHP* increases when the number of transactions increases. However, the execution time of the *DHP* algorithm is near linear to the size of test databases. The *HMFS* method is not so sensitive to the size of a database compared to the *DHP* algorithm. Therefore, our *HMFS* method performs much better when the database size is larger.



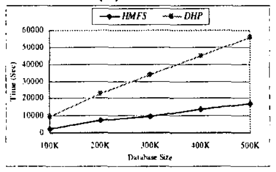
(a) T10I4



(b) T15I4



(c) T20I4



(d) T20I6

**Figure 4: The execution time of the *HMFS* method and the *DHP* algorithm on the test databases with various database sizes. (Minimum Support = 0.75%)**

## 5. Conclusions

In this paper, we have proposed an efficient hash-based method, *HMFS*, for discovering the maximal frequent itemsets. The method combines the advantages of the *DHP* and the *Pincer-Search* algorithms. The combination leads to two advantages. First, the *HMFS* method, in general, can reduce the number of database scans. Second, the *HMFS* method can filter infrequent itemsets and use the filtered itemsets to find the maximal frequent itemsets faster. In addition, an efficient mechanism to construct the maximal frequent candidate itemsets is provided. To evaluate the performance of our method, we have implemented the proposed method along with the *DHP* and the *Pincer-Search* algorithms on a Pentium III 800 MHz PC. The experiments were conducted on various benchmark databases. The experimental results show that our method has better performance than the *DHP* and the *Pincer-Search* algorithms for most of test cases. In particular, our method has significant improvement over the *DHP* and the *Pincer-Search* algorithms when the size of a database is large and the length of the longest itemset is relatively long.

## References

[1] A. Savasere, E. Omiecinski, and S. Navathe, "An Efficient Algorithm for Mining Association Rules in Large Databases", *In Proceedings of 21st VLDB*, pp. 432-444, 1995.

[2] D. Lin and Z. M. Kedem, "Pincer-Search: A New Algorithm for Discovering the Maximum Frequent Set", *In Proceedings of VI Intl. Conference on Extending Database Technology*, 1998.

[3] Eui-Hong Han, George Karypis and Vipin Kumar, "Scalable Parallel Data Mining for Association Rules", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 12, No. 3, MAY/JUNE 2000.

[4] H. Toivonen, "Sampling Large Databases for Association Rules", *VLDB*, pp. 134-145, 1996.

[5] IBM Quest Data Mining Project, "Quest Synthetic Data Generation Code", "http"//www. almaden. ibm. com/cs/quest/syndata. html", 1996

[6] J. S. Park, M. S. Chen, and P. S. Yu, "An Effective Hash Based Algorithm for Mining Association Rules", *Proceedings of the ACM SIGMOD*, pp. 175-186, 1995.

[7] M. Houtsma and A. Swami, "Set-Oriented Mining of Association Rules in Relational Databases," *11th Int'l Conference on Data Engineer*, 1995.

[8] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, "New Algorithms for Fast Discovery of Association Rules", *3rd Int'l Conference on Knowledge Discovery & Data Mining (KDD)*, Newport, CA, August 1997.

[9] Mohammed J. Zaki, "Scalable Algorithm for Association Mining", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 12, No. 3, MAY/JUNE 2000.

[10] M. S. Chen, J. Han, and P. S. Yu, "Data Mining: An Overview from a Database Perspective", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 8, No. 6, December 1996.

[11] M. S. Chen, J. S. Park, and P. S. Yu, "Efficient Data Mining for Path Traversal Patterns", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 10, No. 2, 1998, pp. 209-220.

[12] R. Agrawal, T. Imilienski, and A. Swami, "Mining Association Rules between Sets of Items in Large Databases", *In Proceedings of the ACM SIGMOD Int'l Conference on Management of Data*, pp. 207-216, May 1993.

[13] R. Agrawal and R. Srikant, "Fast Algorithm for Mining Association Rules in Large Databases", *In Proceedings of 1994 Int'l Conference on VLDB*, pp. 487-499, Santiago, Chile, Sep. 1994.

[14] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Inkeri Verkamo, "Fast Discovery of Association Rules," *Advances in Knowledge Discovery and Data Mining*, U. Fayyad and *et al.*, eds., pp. 307-328, Menlo Park, Calif.: AAAI Press, 1996.

[15] R. Agrawal and J. Shafer, "Parallel Mining of Association Rules," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 8, No. 6, pp. 962-969, Dec. 1996.

[16] R. J. Bayardo Jr., "Efficiently Mining Long Patterns from Databases", *In Proceedings of the ACM SIGMOD Conference on Management of Data*, pp. 85-93, Seattle, Washington, June 1998.

[17] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur, "Dynamic Itemset Counting and Implication Rules for Market Basket Data", *1997 ACM SIGMOD Conference on Management of Data*, pp. 255-264, 1997.