HASH-MINE: A New Framework for Discovery of Frequent Itemsets

Marek Wojciechowski, Maciej Zakrzewicz

Poznan University of Technology
Institute of Computing Science
ul. Piotrowo 3a, 60-965 Poznan, Poland
Marek.Wojciechowski@cs.put.poznan.pl
Maciej.Zakrzewicz@cs.put.poznan.pl

Abstract. Discovery of frequently occurring subsets of items, called itemsets, is the core of many data mining methods. Most of the previous studies adopt Apriori-like algorithms, which iteratively generate candidate itemsets and check their occurrence frequencies in the database. These approaches suffer from serious costs of repeated passes over the analyzed database. To address this problem, we propose a novel method, called HASH-MINE, for reducing database activity of frequent itemset discovery algorithms. The idea of HASH_MINE consists in using hash tables for pruning candidate itemsets. The proposed method requires fewer scans over the source database: the first scan creates hash tables, while the subsequent ones verify discovered itemsets. Its performance improvements have been shown in a series of our experiments.

1 Introduction

Discovery of frequent itemsets is the core of many data mining methods. It has been well studied in the context of association rules introduced in [1]. The problem of mining association rules is usually decomposed into two phases: discovery of frequent itemsets and generation of rules from the discovered frequent itemsets. Since the second step is straightforward, researchers dealing with association rules usually concentrate on efficient algorithms for discovery of frequent itemsets. It has to be noted that frequent itemsets have more applications than only for discovery of association rules. It has been shown that they can be used in discovery of sequential patterns [4] or clustering [7].

Most of the previous studies on frequent itemsets adopt *Apriori*-like algorithms, which iteratively generate candidate itemsets and check their occurrence frequencies in the database. It has been shown that *Apriori* in its original form [3] suffers from serious costs of repeated passes over the analyzed database and from the number of candidates that have to be checked, especially when the frequent itemsets to be discovered are long.

In this paper, we propose a novel method, called *Hash-Mine*, for reducing database activity of frequent itemset discovery algorithms. *Hash-Mine* generates hash tables derived from the original database and uses them for pruning candidate itemsets in some of the iterations. The proposed method requires smaller number of scans over the source database than *Apriori*. Experiments show that our method leads to a significant performance improvement over the classic *Apriori* algorithm. The minimal number of database scans in our approach is two: the first scan creates hash tables, while the second one performs final pruning, however, the best results can be obtained, if we use hash-based pruning starting from the third iteration of the algorithm.

1.1 Background

Frequent itemsets. Let $L=\{l_1, l_2, ..., l_m\}$ be a set of literals, called items. Let a non-empty set of items T be called an *itemset*. Let D be a set of variable length itemsets, where each itemset $T \subseteq L$. We say that an itemset T *supports* an item $x \in L$ if x is in T. We say that an itemset T *supports* an itemset $X \subseteq L$ if T supports every item in the set X. Each itemset has an associated measure of its statistical significance, called *support*. The support of the itemset T in the set D is:

$$support(X, D) = \frac{\left|\{T \in D \mid T \text{ supports } X\}\right|}{|D|}$$

In other words, the itemset X holds in the set D with support s if s is the fraction of itemsets in D supporting X. A *frequent itemset* is an itemset, whose support is above a user-defined threshold.

Example. For the database $D = \{\{A,B,C,D\}, \{A,C,D\}, \{E,F,G\}, \{A,C,D,H\}\}$, and the support threshold value of 0.5 we have the following frequent itemsets: $\{A\}$, $\{C\}$, $\{D\}$, $\{A,C\}$, $\{A,D\}$, $\{C,D\}$, and $\{A,C,D\}$.

Introduction to Apriori. The algorithm called *Apriori* iteratively finds all possible itemsets that have support greater or equal to a given minimum support value (minsup). The first pass of the algorithm counts item occurrences to determine the frequent 1-itemsets (each 1-itemset contains exactly one item). In each of the next passes, the frequent itemsets L_k-1 found in the (k-1)th pass are used to generate the candidate itemsets C_k, using apriori-gen function described below. Then, the database is scanned and the support of candidates in C_k is counted. The output of the first phase of the Apriori algorithm consists of a set of k-itemsets (k=1, 2, ...), that have support greater or equal to a given minimum support value. Figure 1 presents a formal description of the algorithm. We assume that items in each database itemset are kept sorted in their lexicographic order.

```
scan $D$ to find $L_1$;
for ( $k$ = 2; $L_{k-1} \neq 0$; $k$++) do begin
  $C_k$ = apriori_gen ($L_{k-1}$ );
  forall transactions $t \in D$ do begin
        $C_t$ = subset ($C_k$ , $t$);
        forall candidates $c \in C_t$ do
               $c$.count ++;
  end
  $L_k$ = { $c \in C_k$ | $c$.count $\geq$ minsup};
end
Answer = $\cup_k$ $L_k$;
```

Fig. 1. Apriori algorithm

In the algorithm *Apriori*, candidate itemsets C_k are generated from previously found frequent itemsets L_{k-1}, using the *apriori-gen* function. The *apriori-gen* function works in two steps: 1. join step and 2. prune step. First, in the join step, large itemsets from L_{k-1} are joined with other large itemsets from L_{k-1} in the following SQL-like manner:

```
insert into $C_k$
select p.item$_1$, p.item$_2$, ..., p.item$_{k-1}$, q.item$_{k-1}$
from $L_{k-1}$ p, $L_{k-1}$ q
where p.item$_1$ = q.item$_1$
  and p.item$_2$ = q.item$_2$
  ...
  and p.item$_{k-2}$ = q.item$_{k-2}$
  and p.item$_{k-1}$ < q.item$_{k-1}$;
```

Next, in the prune step, each itemset $c \in C_k$ such that some (k-1)-subset of c is not in L_{k-1} is deleted:

```
forall itemsets $c \in C_k$ do
  forall (k-1)-subsets $s$ of $c$ do
    if (s $\notin$ $L_{k-1}$) then delete $c$ from $C_k$;
```

The set of candidate k-itemsets C_k is then returned as a result of the function *apriori-gen*.

A serious problem of practical applications of *Apriori* is its long processing time. The repeated database scanning is its most important drawback.

1.2 Related Work

Many variants of Apriori have been proposed recently to reduce the number of required database scans or the number of candidates to verify. In [9] an algorithm called *Partition* that needs only two scans over the database was proposed. *Partition* divides the database into parts that can be kept in main memory, discovers itemsets in those parts and verifies the results in the final database pass. The *DIC* algorithm [6] begins checking an itemset shortly after all its subsets have been determined frequent, thus potentially reducing the overall number of iterations. The algorithm *DHP* [8] enhances *Apriori* with a hashing scheme that is used in each iteration to prune some candidates before the database pass (the overall number of database scans is the same as in the case of *Apriori*). In [5] an algorithm called *Max-Miner*, significantly different from the previous ones was introduced to reduce the number of processed candidates when patterns (itemsets) to be found are long (the number of candidates checked by *Apriori* grows exponentially with the size of the longest pattern).

2 HASH-MINE Algorithm

Apriori-like algorithms use full database scans for pruning candidate itemsets, which are below the support threshold. *Hash-Mine* prunes candidates by using dynamically generated hash tables, thus reducing the number of database blocks read.

A *hash table* used by *Hash-Mine* is a set of hash signatures generated for each database itemset. The *hash signature* of a set X is an N-bit binary number created, by means of bit-wise OR operation, from the hash keys of all data items contained in X. The *hash key* of the item $x \in X$ is an N-bit binary number defined as follows:

```
hash_key(X) = 2^(X mod n)
```

For example, for the database $D = \{\{0,7,12,13\},\{2,4\},\{10,15,17\}\}$ and N=5, we generate the following hash table $H=\{01101,10100,00001\}$.

The hash signatures have the following property. For any two sets X and Y, we have $X \subseteq Y$ if:

```
hash_signature(X) AND hash_signature(Y) = hash_signature(X)
```

where AND is a bit-wise AND operator. This property is not reversible in general (when we find that the above formula evaluates to *TRUE* we still have to verify the result traditionally).

In order to plan the length N of a hash signature for a given average set size, consider the following analysis. Assuming uniform distribution of items, the probability that representation of the set X sets k bits to '1' in an N-bit hash signature is:

$$P = \frac{\binom{N}{k} f_{k,|X|}}{N^{|X|}}, \text{ where } f_{0,|X|} = 0,\ f_{q,|X|} = q^{|X|} - \sum_{i=1}^{q-1} \binom{q}{i} f_{i,|X|}$$

Example probabilistic expected value of number of bits set to '1' for a 16-bit hash signatures and various set sizes is illustrated in Figure 2. We can observe that e.g. for a set of 10 items, N should be greater than 8 (otherwise we have all bits set to 1 and the signature is unusable since it is always matched).

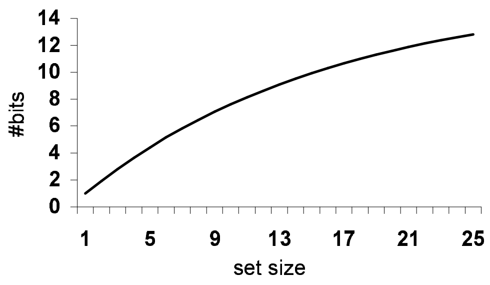

Fig. 2. Number of bitmap signature bits set to '1' for various set sizes (*N*=16)

The probability that a bitmap signature of the length N having k 1's matches another bitmap signature of the length N having m 1's is:

$$\binom{m}{k} / \binom{N}{k}$$

It means that the smaller k, the better pruning is performed during matching bitmap signatures of item sets, in order to check their containment (so we have to verify less item sets).

The *Hash-Mine* algorithm for frequent itemset discovery is presented in Figure 3. A user gives a minimum support value (minsup), and an array (use_hash), which specifies, in what iterations to use a hash-based candidate pruning – *use_hash[i]=1* means to use a hash table in the iteration *i*, instead of a database scan.

```
scan $D$ to generate hash signatures $S$ and to find $L_1$;
for ($k$ = 2; $L_{k-1} \neq 0$; $k$++) do begin
 $C_k$ = apriori_gen ($L_{k-1}$ );
 if use_hash[k]=1 then begin
        forall signatures $t \in S$  do
          forall candidates $c \in C_k$ do
            if $c$ AND $t=c$ then   $c$.count ++;
        end;
 else begin
       forall transactions $t \in D$  do begin
          $C_t$ = subset ($C_k$, $t$);
          forall candidates $c \in C_t$  do $c$.count ++;
       end;
     end;
 $L_k$  = { $c \in C_k$ | $c$.count $\geq$ minsup};
end;
Answer = $\cup_k$  $L_k$;
scan $D$ to verify Answer;
```

Fig. 3. Hash-Mine Algorithm

3 Experimental Results

We performed several experiments on synthetic data to evaluate the performance and scalability of the *Hash-Mine* algorithm and the efficiency of hash-based candidate pruning. The data sets were generated by means of the *GEN* generator from the *Quest* project [2]. The average item set size was 25 items out of 50. The experiments were conducted in a client-server architecture. The database was implemented in *Oracle 8i* DBMS running on a PC with Pentium II 300 MHz processor and 128 MB of main memory. The client was a PC with the same hardware configuration, communicating with the server via *Ethernet* local area network.

Figure 4 shows execution times of two instances of the *Hash-Mine* algorithm: using three and four database passes (counting the final verification path). The execution times of *Hash-Mine* are compared to *Apriori*, which for our test data set needed 7 database scans. The experiments show that our method is c.a. 2 times faster than *Apriori* and scales linearly with the size of the database (same as *Apriori*).

In Figure 5 numbers of candidates that were not pruned out in subsequent iterations are shown for *Apriori* and *Hash-Mine* switching to hash-based pruning in the fourth iteration. The database used in the test consisted of 1000 itemsets and the required minimum support was 20%. Since hash-based pruning leaves some itemsets that are not frequent, *Hash-Mine* may require more iterations than *Apriori* (as shown in our example). Despite this, *Hash-Mine* still outperforms *Apriori* because additional iterations are performed in main memory.

We also experimented with the version of *Hash-Mine* using only two database scans: the initial one and the verification one, but in that case the number of candidates that had to be analyzed due to inaccurate pruning was too large and led to longer execution times than in case of *Apriori*.

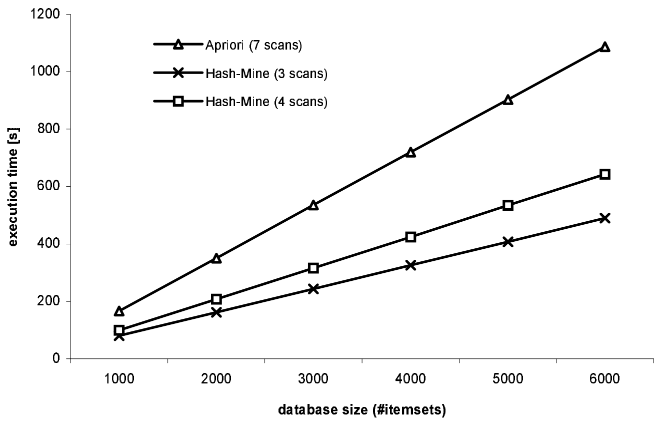

Fig. 4. Execution times comparison

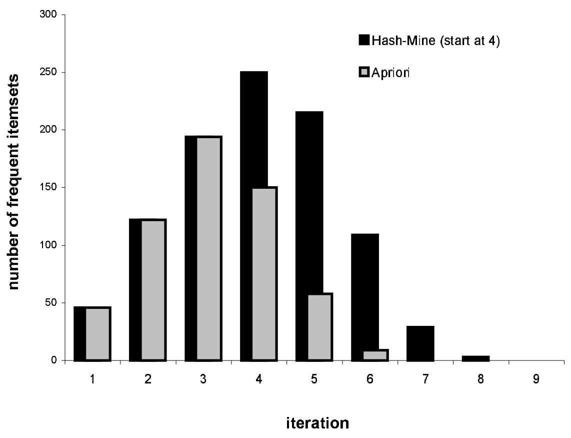

Fig. 5. Numbers of candidates not pruned out

4 Conclusions

In this paper we have shown that dynamically created hash tables can replace costly database scans. Hash tables are extremely small, as compared to the source database, therefore they fit into memory even for very large databases. The *Hash-Mine* method can be used to improve performance of *Apriori*-like data mining algorithms, especially when a number of their iterations is large. Our experimental results show 50% performance increase over the traditional algorithms.

References

1. Agrawal R., Imielinski T., Swami A.: Mining Association Rules Between Sets of Items in Large Databases. Proc. 1993 ACM SIGMOD International Conference on Management of Data, pp. 207-216, Washington DC, USA (1993)
2. Agrawal R., Mehta M., Shafer J., Srikant R., Arning A., Bollinger T.: The Quest Data Mining System. Proc. of the 2nd Int'l Conference on Knowledge Discovery in Databases and Data Mining, Portland, Oregon (1996)
3. Agrawal R., Srikant R.: Fast Algorithms for Mining Association Rules. Proc. 20th Int'l Conf. Very Large Data Bases, pp. 478-499, Santiago, Chile (1994)
4. Agrawal R., Srikant R.: Mining Sequential Patterns. Proc. of the 11th Int'l Conference on Data Engineering (1995)
5. Bayardo R. J.: Efficiently Mining Long Patterns from Databases. Proc. of the 1998 ACM SIGMOD International Conference on Management of Data (1998)
6. Brin S., Motwani R., Ullman J., Tsur S.: Dynamic Itemset Counting and Implication Rules for Market Basket Data. Proc. of the 1997 ACM SIGMOD International Conference on Management of Data (1997)

7. Han E., Karypis G., Kumar V., Mobasher B.: Hypergraph Based Clustering in High-Dimensional Data Sets: A summary of Results. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, Vol.21 No. 1 (1998)
8. Park J.S., Chen M.-S., Yu P. S.: An Effective Hash-Based Algorithm for Mining Association Rules. Proc. 1995 ACM SIGMOD International Conference on Management of Data, San Jose, CA, USA (1995)
9. Savasere A., Omiecinski E., Navathe S.: An Efficient Algorithm for Mining Association Rules in Large Databases. Proc. 21th Int’l Conf. Very Large Data Bases, pp. 432-444, Zurich, Switzerland (1995)