Lê Kim Thư

HASH-BASED APPROACH TO DATA MINING

MINOR THESIS – BACHELOR OF SCIENCE – REGULAR TRAINING
Faculty: Information Technology
Lê Kim Thu

HASH-BASED APPROACH
TO DATA MINING

MINOR THESIS – BACHELOR OF SCIENCE –
REGULAR TRAINING

Faculty: Information technology

Supervisor: Dr. Nguyễn Hùng Sơn
Asso.Prof.Dr. Hà Quang Thụy
ACKNOWLEDGEMENT

First of all, I want to express my deep gratitude to my supervisors, Dr. Nguyen Hung Son and Asso. Prof. Dr. Ha Quang Thuy, who have helped me a lot during my working process.

From the bottom of my heart, I’d like to thanks to all teachers and officer staffs of College of Technology for providing us favorable conditions for learning and researching, in particular, all teachers in Department of Information systems for valuable advices in professional knowledge.

Last but not the least, I’m thankful to my family, my friends, especially my mother who always encourages and helps me to complete this thesis.

Ha Noi, 5/2007,
Student: Le Kim Thu.
ABSTRACT

Using computer, people can collect data in many types. Thus, many applications to revealing valuable information have been considered. One of the most important matters is “to shorten run time” when database become bigger and bigger. Furthermore, we look for algorithms only using minimum required resources but are doing well when database become very large.

My thesis, with the subject “hash-based approach to data mining” focuses on the hash-based method to improve performance of finding association rules in the transaction databases and use the PHS (perfect hashing and data shrinking) algorithm to build a system, which helps directors of shops/stores to have a detailed view about his business. The soft gains an acceptable result when runs over a quite large database.
List of tables

Table 1: Transaction database ................................................................. 8
Table 2: Candidate 1-itemsets ................................................................. 16
Table 3: Large 1-itemsets ............................................................... 16
Table 4: Hash table for 2-itemsets ........................................................... 22
Table 5: Scan and count i-itemsets .......................................................... 26
Table 6: Frequent 1-itemsets .............................................................. 26
Table 7: Candidate 2-itemsets in second pass ....................................... 26
Table 8: Hash table of the second pass .................................................. 26
Table 9: Lookup table of the second pass ............................................... 27
Table 10: Candidate itemsets for the third pass ...................................... 27
Table 11: Large itemsets of the second pass .......................................... 27
Table 12: Hash table of the third pass .................................................... 27
Table 13: Lookup table of the third pass ................................................ 27
List of figures

Figure 1: An example to get frequent itemsets.......................................................... 9
Figure 2: Example of hash table for PHP algorithm .............................................. 21
Figure 3: Execution time of Apriori – DHP .............................................................. 28
Figure 4: Execution time of Apriori – PHP ............................................................. 29
Figure 5: Execution time of PHS – DHP ................................................................. 29
Figure 6: Association generated by software......................................................... 34
List of abbreviate words

AIS : Artificial immune system
C_k : Candidate itemsets k elements
DB : Database
DHP : Direct Hashing and Pruning
H_k : Hash table of k-itemsets
L_k : Large itemsets k elements
PHP : Perfect Hashing and DB Pruning
PHS : Perfect Hashing and data Shrinking
SETM : Set-oriented mining
TxIyDz : Database which has average size of transaction is x, average size of the maximal potentially large itemsets is y and number of transactions is z.
TABLE OF CONTENTS

Abstract ........................................... i
List of tables ..................................... ii
List of figures ................................... iii
List of abbreviate words ....................... iv
FOREWORD ................................................................. 1
CHAPTER 1: Introduction................................. 3
  1.1 Overview of finding association rules ........................................ 3
    1.1.1 Problem description ..................................................... 4
    1.1.2 Problem solution ......................................................... 5
  1.2 Some algorithms in the early state ........................................... 5
    1.2.1 AIS algorithm ............................................................. 6
    1.2.2 SETM algorithm ........................................................... 6
    1.2.3 Apriori algorithm ......................................................... 6
  1.3 Shortcoming problems ......................................................... 10
CHAPTER 2: Algorithms using hash-based approach to find association rules ............................................. 11
  2.1 DHP algorithm (direct hashing and pruning) .............................. 12
    2.1.1 Algorithm description .................................................. 13
    2.1.2 Pseudo-code ............................................................... 14
    2.1.3 Example ................................................................. 16
  2.2 PHP algorithm (perfect hashing and pruning) ............................ 18
    2.2.1 Brief description of algorithm ...................................... 19
    2.2.2 Pseudo-code ............................................................. 20
    2.2.3 Example ................................................................. 21
  2.3 PHS algorithm (Perfect hashing and database shrinking) .............. 22
Problem of searching for association rules and sequential patterns in transaction database in particular become more and more important in many real-life applications of data mining. For the recent time, many research works have been investigated to develop new and to improve the existing solution for this problem, [2-13]. Form Apriori – an early developed, well known algorithm – which has been used for many realistic applications, a lot of improving algorithms were proposed with higher accuracy. Some of our works in the finding association rules and sequential patterns focus on shorten running time. [4-11,13]

In most cases, the databases needed to process are extremely large, therefore we must have some ways to cope with difficulties. Then, the mining algorithms are more scalable. One of trends to face with this problem is using a hash function to divide the original set into subsets. By this action, we will not waste too much time doing useless thing.

Our thesis with the subject “Hash-based approach to data mining” will present DHP, PHP and PHS - some efficient algorithms - to find out association rules and sequential patterns in large databases. We concentrate mostly on those solutions which are based on hashing technique. One of the proposed algorithms, PHS algorithm due to its best performance in the trio will be chosen for using in a real-life application to evaluate the practicability over realistic data.

The thesis is organized as follows:

**Chapter 1: Introduction**

Provide the fundamental concepts and definitions related to the problem of finding association rules. This chapter also presents some basic algorithms (AIS, SETM and Apriori), which have been developed at the beginning of this subject.
Chapter 2: *Algorithms using hash-based approach to find association rules*

Describes some algorithms could be used to improve the accuracy of the whole process. In this chapter, I present three algorithms: Direct Hashing and Pruning (DHP), Perfect Hashing and Pruning (PHP) and Perfect Hashing and Shrinking (PHS). In these algorithms, the PHS is considered to be the best solution, however all of these gained a much better result compare to Apriori algorithm.

Chapter 3: *Experiments*

Included in my work, I intend to build a small system, which uses one of the algorithms that were mentioned in chapter 2. I choose PHS for this position because of its outperforming result. The main feature of system is finding the association rules.

Conclusion

Summarizes the content of the work and brings out some trends to continue in the future.
CHAPTER 1: Introduction

1.1 Overview of finding association rules

It is said that, we are being flooded in the data. However, all data are in the form of strings, characters (text, document) or as numbers which are really difficult for us to understand. In these forms, it seems to be unmeaning but after processing them by a specific program, we can reveal their important roles. That’s the reason why there is more and more scientists concern on searching for useful information (models, patterns) that is hidden in the database. Through the work of data mining, we can discover knowledge – the combination of information, events, fundamental rules and their relationship, the entire thing are apprehended, found out and learned from initial data. Therefore, data mining grows quickly, step by step plays a key role in our lives now. Each application has other requirements, correlate with other methods for the particular databases. In this research work, we limit our concentration on transaction databases (which are available in transaction storage systems) and our goal is find out the hidden association rules.

Association rules are the rules which represent the interesting, important relationship or the interrelate relations, correlated patterns contained in our database.

The finding of association rules have many applications in a lot of areas in life, such as: in commerce, analyzing the business data, customer data to decide whether or not invest, lend…, detecting special pattern relate to cheat, rig… One of the important applications is consumer market analysis, this will analyze the routine while customers choose commodities, take out the correlation between the
productions usually appear together in a transaction. Base on the rules were gained, we have the suitable methods to improve profits. For instances, in a supermarket management system, productions which are often bought together in the sale will be put next to each other, so the customers would be easy to find and remember which they intend to buy. With e-commerce, online transaction, the net order selling system, we can store and control customer by an ID, so each time they login, from found rules we’ll have mechanism to display on the screen exactly the items often looked for by them. This action is really easy if we have had rules, but could bring us the customer pleasant and it’s maybe one of many reasons they think about us next time…

1.1.1 Problem description

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of elements, called items. Let $D$ be a set of transactions, where each transaction $T$ is a set of items such that $T \subseteq \mathbb{Z}$. Note that the quantities of items bought in a transaction are not considered, that is each item is a binary variable represent if an item was bought. Each transaction is associated with an identifier, called TID. Let $X$ is a set of items. A transaction $T$ is said to contain $X$ if and only if $X \subseteq T$.

An association rule is an implication of the form $X \rightarrow Y$, where $X \subseteq I$, $Y \subseteq I$ and $X \cap Y = \emptyset$.

The rule $X \rightarrow Y$ holds in the transaction set $D$ with confidence $c$ if $c\%$ of transactions in $D$ that contain $X$ also contain $Y$.

The rule $X \rightarrow Y$ has support $s$ in the transaction set $D$ if $s\%$ of transactions in $D$ contains $X \cup Y$.

From a given database $D$, our goal is to discover association rules among rules inferred from essential database which have supported and confidence greater than the user-specified minimum support (called $\text{minsup}$) and minimum confidence (called $\text{minconf}$) respectively [2-13].
1.1.2 Problem solution

In order to solve the problem, they decomposed the process into two sub-processes [2-13]:

- **Discover the frequent itemsets:** Find all sets of items (called *itemsets*) that have transaction support above *minsup*. The *support* of an itemsets is the amount of transactions which contain the itemsets. These itemsets are called *frequent* (large) itemsets and the remainders are called *non-frequent* itemsets or *small* itemsets. Number of elements in an itemsets is considered as its size (so an itemsets have k items is called k-itemsets). To determine whether a k-itemsets is a frequent itemsets, we have to examine its support equal or greater *minsup*. Due to great amounts of k-itemsets, it’s expected that we could save many time by examining a subset of all k-itemsets (C_k) – called candidate itemsets – this set must contain all the frequent itemsets (L_k) in the database.

- **Use the frequent itemsets to generate the association rules:** Here is a straightforward algorithm for this task. For every frequent itemsets l, find all non-empty subsets of l, with each subset a, output a rule of the form a → (l \ a) if the value of support (l) divide by support (a) at least *minconf*.

In the second step, there’re few algorithms to improve performance [11], in this research, I confine myself to the first process of the two – try to discover all frequent itemsets as fast as possible.

**Input:** Transaction database, *minsup.*

**Output:** Frequent itemsets in given database with given *minsup.*

1.2 Some algorithms in the early state

It’s expected that found association rules will have many useful, worthy properties. So, the work of discovery these rules have been developed prematurely, some of them could be listed here as AIS, SETM, Apriori [8,9,11…]
1.2.1 AIS algorithm

AIS [9,11] stand for Artificial Immune System. In this algorithm, candidate itemsets are generated and counted on-the-fly as the database is scanned. First, we read a transaction then determined the itemsets which are contained in this transaction and appeared in the list of frequent itemsets in previous pass. New candidate itemsets will be generated by extending these frequent itemsets \((l)\) with other items - which have to be frequent and occur later in the lexicographic ordering of items than any of the items in \(l\) - in the transaction. After generated candidates, we add them to the set of candidate itemsets of the pass, or go to next transaction if they were created by an earlier transaction. More details are presented in [9].

1.2.2 SETM algorithm

This algorithm was motivated by the desire to use SQL to compute large itemsets. Similar to AIS, SETM (Set-oriented mining) [8,11] also generated and counted candidate itemsets just after a transaction have read. However, SETM uses the join operator in SQL to generate candidate itemsets. It’s also separates generating process from counting process. A copy of itemsets in \(C_k\) have been associated with the TID of its transaction. They are put in a sequential structure. At the end of each phase, we sort the support of candidate itemsets and choose the right sets. This algorithm requires too much arrangement that why it is quite slow.

1.2.3 Apriori algorithm

Different from AIS and SETM algorithms, Apriori [11] was proposed by Agrawal in 1994, in his research, he gave a new way to make candidate itemsets. According to this algorithm, we’ll no longer generate candidate itemsets on-the-fly. We make multiple passes to discovering frequent itemsets over the data. First, we count the support of each items and take out items have minimum support, called \(large\). In each later pass, we use the itemsets which is frequent in previous pass as the core to combine new potentially frequent itemsets, and count the support for these candidates. At the end of the pass, we pick out the candidate itemsets which are really frequent. And then, we repeat this work.
In a pass, the Apriori algorithms generate the candidate itemsets by using the large itemsets found in the previous pass. This is based on a really simple property that any subset of a large itemsets must be large. So, one candidate is really large if its have no subset which is not large.

We assume that items in transactions were stored in lexicographic order. The algorithm could be considered as the iteration of two steps:

Algorithm description:
First: Generate candidate itemsets – C_k
Here, we define an operator: Join.
We use notation x[1],…, x[k] to represent a k-itemsets X consists of k items: x[1], … , x[k] where x[1] < x[2] < … < x[k]

Given two k-itemsets X and Y which k-1 first elements are the same. And x[k] < y[k]
The result of operator join X.Y is a new (k+1)-itemsets consist of items: x[1],… , x[k], y[k].

We generate C_k by joining L_{k-1} with itself.

Second: Prune the C_k to retrieve L_k

It is easy to find that all sets appearing in L_k is also contained in C_k (from the above property). Therefore, to gain the large itemsets we scan and count itemsets in C_k and remove elements which do not contain any (k-1)-itemsets which is not belong to L_{k-1}. After that we have the set of large k-itemsets L_k.

Pseudo-code:

\[ L_1 = \{ \text{frequent 1-itemsets} \}; \]

\[ \text{for } (k = 2; L_{k-1} \neq \emptyset; k++) \text{ do begin} \]
\[ C_k = \text{apriori\_gen} (L_{k-1}); \quad \text{//New candidates} \]
\[ \text{forall transactions } t \in D \text{ do begin} \]
\[ C_t = \text{subset} (C_k, t); \quad \text{//Candidates contained in t} \]
\[ \text{forall candidates } c \in C_t \text{ do} \]

---

---

---
c.count++;

end

\[ L_k = \{ c \in C_k \mid c.\text{count} \geq \text{minsup} \} \]

end

Answer = \cup_k L_k;

**Example:**

Example 1: Consider the database in table 1 and assume that the minsup is 3. Find all of the large itemsets of the database.

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>ABCD</td>
</tr>
<tr>
<td>200</td>
<td>ABCDF</td>
</tr>
<tr>
<td>300</td>
<td>BCDE</td>
</tr>
<tr>
<td>400</td>
<td>ABCDF</td>
</tr>
<tr>
<td>500</td>
<td>ABEF</td>
</tr>
</tbody>
</table>
Hash-Based Approach to Data Mining

Figure 1: An example to get frequent itemsets.

First: Scan the database and pick out frequent items (which appear at least 3 times in transactions) – \(L_1 = \{\{A\};\{B\};\{C\};\{D\};\{F\}\}\)

Second: Join \(L_1\) with \(L_1\) to generate \(C_2\):
\(C_2 = \{\{AB\};\{AC\};\{AD\};\{AF\};\{BC\};\{BD\};\{BF\};\{CD\};\{DF\}\}. 
At this pass, $C_2 = C_1 \times C_1$, after that we scan database and count to build $L_2$.

$L_2 = \{\{AB\};\{AC\};\{AD\};\{AF\};\{BC\};\{BD\};\{BF\};\{CD\}\}$

Third: this pass is quite more complicated than the previous. We must find pair of elements in $L_2$ which have the same first element to join. So $C_3$ is not $C_2 \times C_2$ any more. First, we perform (*): join in $C_2$ with $C_2$, and then prune it (**). As you see that, $\{ACF\}$ contain $\{CF\}$ which is not belong to $L_2$ so $\{ACF\}$ could not be large, similarly, $\{ADF\}$ contain $\{DF\}$ not in $L_2$ so $\{ADF\}$ is small and so on…. after calculating, we get $C_3$ as the figure below. Now we do the same as previous step and get $L_3$.

After that: iterate exactly what we have done until there is no element in $L_k$ (in this case, $k = 5$)

And we obtain the result: Frequent itemsets of the transaction database are:

$L = L_1 \cup L_2 \cup L_3 \cup L_4$

$= \{\{A\};\{B\};\{C\};\{D\};\{F\}\} \cup \{\{AB\};\{AC\};\{AD\};\{AF\};\{BC\};\{BD\};\{BF\};\{CD\}\}$

$\cup \{\{ABC\};\{ABD\};\{ABF\};\{ACD\};\{BCD\}\} \cup \{\{ABCD\}\}$.

### 1.3 Shortcoming problems

Comparing with AIS and SETM, Apriori algorithm is much better (for more detail, see [11]). But there are still some bottlenecks have not been removed yet. One of the easy-to-find disadvantages is requirement to scan database many times. This issue is insignificant when we work with a small database, however, the data of transactions – which we concern – with a quick increasing we have to face with an extremely large database. The idea of reading data repeatedly is very costly and will affect to the accuracy of algorithm. Therefore, many other approaches have been proposed, many algorithms were developed [4,5,6,7,10,11,13] to reach the goal: improve performance of the process. We’ll care about one of these, a trend that was expanded by a lot of scientists: “Hash-based approach to discover association rules”, it is said to be a good method to implement over different types of data in variant size.
CHAPTER 2: Algorithms using hash-based approach to find association rules

Before going into the detail of algorithms, I’d like to give you a brief view of hashing. In term of data structure and algorithm, hash-method often used an array structure to store database. If the database is too large, we can apply multi-level. By this deed, we are able to access database directly by using a key element instead of linear search.

As mentioned above, our databases are growing quickly day after day while the storage devices are not. So, reading data a lot of times brings us a big difficult to process data when their size exceed limitation provided by hardware devices. Notice that hash method is not only useful to access directly to data, but also help us divide the original database into parts, each part fit in a limited space. That’s the reason why it is used in such situation like ours. We intend to use hash functions to hash itemsets into another buckets of a hash table and we could reduce the size and even reduce the total task. Here, I am going to present three algorithms, which provide good results when they were tested with real data: DHP (direct hashing and pruning), PHP (perfect hashing and pruning) and PHS (Perfect hashing and data shrinking) [4,5,6,12].
2.1 DHP algorithm (direct hashing and pruning)

It is easy to recognize that with the Apriori algorithm we generated a too large candidate itemsets (compare with real large itemsets) because we joined $L_k$ with $L_k$ without removing any set, while this result contained a lot of “bad” sets. It’s really slow in some first passes. In these passes, if we generate a big $C_k$, this’ll lead us to have to do many works to scan and count data. Cause of, at these passes, size of itemsets is small, and it’s contained in many different transactions. In some published researches, it’s proved that the initial candidate set generation, especially candidate for 2-itemsets, is the main problem to increase algorithms performance [6]. Therefore, our desire is an algorithm which could reduce the wasted time and required resource on generating and examining wrong itemsets.

We realize that, the way we create the candidate itemsets will affect to the number of sets, consequently affecting to algorithm performance. In order to make number of candidates smaller, we tend to choose sets with high ability to be a frequent itemsets. In the other side, if we keep searching $(k+1)$-itemsets to count in transactions that do not contained any frequent $k$-itemsets we’ll misspent time to do wasteful works. From these two comments, we think of doing something to solve these problems: first, depend on the appearance of $k$-itemsets to reduce candidates, after that we trim to minimize the size of the database.

From the above ideas, an algorithm has been proposed by a group of IBM Thomas J.Watson Research Center, that is DHP (direct hashing and pruning) [6]. Name of the algorithm echoed its content, included of two parts: one uses a hash function to limit the chosen sets and the other prunes wasteful items and transactions to make database becomes smaller, so it could be more efficient to work with.

DHP algorithm was established on some constraints: if $X$ is a large $(k+1)$-itemsets, then when we remove one of it’s $(k+1)$ elements, the result is a large $k$-itemsets. Consequently, a $(k+1)$-itemsets must contain at least $(k+1)$ large $k$-itemsets to be a large $(k+1)$-itemsets. Secondly, if a transaction doesn’t contain any large itemsets, or if an item is not contained in any large itemsets in one pass, then there after, keep these elements will bring us nothing but superfluous
calculations. There are two sub processes in the algorithm according to the algorithm’s name: hashing and pruning. The DHP algorithm employed a hash mechanism to filter out unuseful itemsets; while counting for support of a candidate k-itemsets to determine whether it’s large, we also gather information for candidate (k+1)-itemsets generation. All possible (k+1)-itemsets in a truncated transaction will be hashed over a hash function into buckets of a hash table. Each bucket has an entry that represent the numbers of itemsets were hashed into it. After this work have finished we decide which itemsets will be retained and which will be cut off. By this step, we reduce size of Ck and would gain Lk faster. But it is not all the things, when we have had Lk, we scan and remove all the transactions which haven’t any large itemsets and remove all the items is not belong to any large itemsets from database. These steps will be repeated progressively until cannot detect any nonempty Lk.

### 2.1.1 Algorithm description

**Hashing** process is divided into 3 parts are as follow:

**Part 1:** With a given support, we scan database and count how many times it appears, build hash table for 2-itemsets. (Called H2; hash table for k-itemsets is called Hk) and choose items which support at least equal to minsup to add into L1

**Part 2:** From the hash table we’ll obtain the set of candidates. These candidates will be examined to generate Lk. When we’ve finished making Lk, database will be trimmed to remove unuseful items and hash table for next pass will be built.

**Part 3:** Do the same thing as in part 2, except building hash table.

Why we separate into part 2 and part 3? The answer: as we known from the beginning of this part, the significantly difference of candidates and really large itemsets in some first pass, after that, the difference is not too much. Whereas, to create a hash table we must do some extra work, it’s not a smart idea if the fraction is higher than one threshold. Therefore, the process contained two different parts, one is used at first, and the other is used when the difference of the
candidates and the large itemsets is not much (this threshold depends on the manager)

**Pruning** task consists of transactions pruning and items pruning:

As I showed, one transaction contained a large (k+1) k-itemsets if it has at least (k+1) large k-itemsets. It’s mean that we are able to cut off the transactions which don’t have enough (k+1) large k-itemsets.

In addition, we found that, if an item belongs to a (k+1) frequent (k+1)-itemsets, then, it’s contained in at least k frequent k-itemsets (k+1 minus 1). Thus, we’ll count and trim the items which have the number of appearance in sets of L_k less than k.

### 2.1.2 Pseudo-code

**Main program:**

/* Part 1 */

s = a minimum support;

Set all the buckets of H_2 to zero; //hash table

**for all** transaction t ∈ D **do begin**

insert and count 1-items occurrences in a hash tree;

**for all** 2-subsets x of t **do**

H_2[h_2(x)] ++;

end

L_1 = {c|c.count >= s, c is in a leaf node of the hash tree};

/* Part 2 */

k = 2

D_k = D; //DB for large k-itemsets

**while** (|{x|H_k[x] >= s}| >= LARGE) {

//make a hash table

gen_candidate(L_{k-1}, H_k, C_k);

set all the buckets of H_{k+1} to rezo;

D_{k+1} = \emptyset;
for all transactions $t \in D_k$ do begin
    count_support($t$, $C_k$, $t'$); // $t' \subseteq t$
    if ($|t'| > k$) then $D_{k+1} = D_{k+1} \cup \{t'\}$
end

$L_k = \{ c \in C_k | c.count \geq s \}$;
$k++$;

/* Part 3 */
gen_candidate ($L_{k-1}$, $H_k$, $C_k$);
while ($|C_k| > 0$) {
    $D_{k+1} = \emptyset$;

    for all transactions $t \in D_k$ do begin
        count_support($t$, $C_k$, $t'$); // $t' \subseteq t$
        If ($|t'| > k$) then $D_{k+1} = D_{k+1} \cup \{t'\}$
    end

    $L_k = \{ c \in C_k | c.count \geq s \}$;
    if ($|D_{k+1}| = 0$) then break;
    $C_{k+1} = Apriori_gen(L_k)$;
    $k++$;
}

Answer = $\cup_k L_k$;

Sub procedures:
Procedure gen_candidate ($L_{k-1}$, $H_k$, $C_k$)
    $C_k = \emptyset$;
    for all $c = c_{p[1]} \ldots c_{p[k-2]}c_{p[k-1]}c_{q[k-1]}$, $c_p, c_q \in L_{k-1}$, $c_p \cap c_q = \emptyset$ do
        if ($H_k[h_k(c)] \geq s$) then
            $C_k = C_k \cup \{c\}$;
    end Procedure
Procedure count_support(t, C_k, k, t)
    for all c such that c ∈ C_k and c (= t_{i1}…t_{ik}) ∈ t do
        begin
            c.count++;
            for (j = 1; j<= k; j++) a[ij]++;
        end
    for (i = 0; j = 0; i < |t|; i++)
        if (a[i] >= k) then do begin t'_j = t_i; j++; end
end Procedure

Procedure make_hasht(t’, H_k, k, H_{k+1}, t”)
    for all (k+1)-subsets x (= t'_{i1}, … , t'_{ik}) of t’ do
        if (for all k-subsets y of x, H_k[h_k(y)] >= s) then do
            begin
                H_{k+1}[h_{k+1}(x)]++;
                for (j = 1; j<= k+1; j++) a[ij]++;
            end
        for (j = 1; j <= k+1; j++)
            if (a[i] > 0) then do begin t”_j = t’_i; j++; end
end Procedure

2.1.3 Example
Example 2: With the database as in table 1 and minsup = 4

At the beginning: From the original database, we scan, count support for items and making hash table. After that, keep and add large item into L_1.

Table 1: Transaction database  Table 2: Candidate 1-itemsets

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
<th>Items</th>
<th>Sup.</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>ABCD</td>
<td>{A}</td>
<td>4</td>
</tr>
<tr>
<td>200</td>
<td>ABCDF</td>
<td>{B}</td>
<td>5</td>
</tr>
<tr>
<td>300</td>
<td>BCDE</td>
<td>{C}</td>
<td>4</td>
</tr>
<tr>
<td>400</td>
<td>ABCDF</td>
<td>{D}</td>
<td>4</td>
</tr>
<tr>
<td>500</td>
<td>ABF</td>
<td>{E}</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{F}</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3: Large 1-itemsets

<table>
<thead>
<tr>
<th>Items</th>
<th>Sup.</th>
</tr>
</thead>
<tbody>
<tr>
<td>{A}</td>
<td>4</td>
</tr>
<tr>
<td>{B}</td>
<td>5</td>
</tr>
<tr>
<td>{C}</td>
<td>4</td>
</tr>
<tr>
<td>{D}</td>
<td>4</td>
</tr>
</tbody>
</table>

16
Hash-Based Approach to Data Mining

(We write an transaction in form of: <TID, {set of items contained in it}>)

Making hash table with hash function:

\[ H(x, y) = [(x’s \, order \times 13 + y’s \, order) \mod 10] \]

Generate possible 2-itemsets in transactions:

100: \{ {AB}; {AC}; {AD}; {BC}; {BD}; {CD} \}
200: \{ {AB}; {AC}; {AD}; {AF}; {BC}; {BD}; {BF}; {CD}; {CF}; {DF} \}
300: \{ {BC}; {BD}; {BE}; {CD}; {CE}; {DE} \}
400: \{ {AB}; {AC}; {AD}; {AF}; {BC}; {BD}; {BF}; {CD}; {CF}; {DF} \}
500: \{ {AB}; {AE}; {AF}; {BE}; {BF}; {EF} \}

These sets are hashed in to hash table and count:

Bucket 0: BC (4) 
Bucket 1: BD (4) 
Bucket 2: BF (3); EF (1) 
Bucket 3: CD (4); EF (1) 
Bucket 4: CE (1) 
Bucket 5: AB (4); CF (2) 
Bucket 6: AC (3) 
Bucket 7: AD (3); DE (1) 
Bucket 8: AE (1) DE (1) 
Bucket 9: AF (3) 

Determine if the value of entry of a bucket satisfied the \textit{minsup} or not. In this case, the result we have is (true, true, true, true, false, true, false, true, false). Use this result to filter out result of \( L_1 \times L_1 \), we have: \( C_2 = \{ {BC}, {BD}, {CD}, {AB}, {AD} \} \) (notice that the candidate set does not contain \{AC\} since bucket 6 have value 3 less than \textit{minsup}). From \( C_2 \) we scan to have \( L_2 = C_2 \setminus \{ AD \} \).

\textit{Pruning process}: first, remove transactions after that remove items.

- Remove transaction 500 because it has only one large 2-itemsets.
- Items pruning:
  + in transaction 100: appear(A)=1, appear(C)=appear(D)=2, appear(B)=3
  Requirement is appear(x) \( \geq 2 \) (since we are considering 2-itemsets) so A is deleted.

  Similarly, we delete A, F in transaction 200 and 400; E in transaction 300

The new database for next pass:
$D_3 = \{<100,\{BCD\}>;<200,\{BCD\}>;<300,\{BCD\}>;<400,\{BCD\}>\}$

In the next pass, all the remaining transaction contain only 3 items and they are the same, then all of them will be hashed in to one bucket in hash table and we have result immediately: $C_3 = \{BCD\}$ and $L_3 = \{BCD\}$.

In the pruning step, all transactions are removed, so all the process have finished swiftly.

In the end, we have the set of large itemsets is:

$L = L_1 \cup L_2 \cup L_3 = \{\{A\};\{B\};\{C\};\{D\};\{AB\};\{BC\};\{BD\};\{CD\};\{BCD\}\}$

These results were generated quite quickly compare to Apriori algorithm as size of candidate itemsets was reduced by hash method. However, there is still some weakness due to the collision in hash table: the processes to generate candidate itemsets may be omit to filter out itemsets when they laid together in a bucket to make the entry value is greater than $\minsup$. We will consider this problem in the next part.

### 2.2 PHP algorithm (perfect hashing and pruning)

As we mentioned in the last session, the DHP algorithm is much better performance compare to Apriori, it’s not only reduced the number of scanning the whole database but also decrease size of database, so we could finished our job faster. But its effect is relying on the hash table. As we saw in the example 2, there are some buckets, which contain more than one sets, each set have $support$ less than our requirement, but the total number – the value presents how many times itemsets have been hashed into it – is over the minimum, so these sets are not removed and we have to do some extra works to examine them. In the example above, when we generate $C_2$, we included $\{AD\}$ in the result because the bucket contain $\{AD\}$ 3 times and $\{DE\}$ 1 times, the value of the entry for the bucket is 4, equal to $\minsup$, we removed $\{DE\}$ since it contains $E$ - not a large item and keep $\{AD\}$ despite of $\text{sup}(AD) = 3$. This itemsets is removed after we scan to choose the final sets.
This foible makes us think of a mechanism which will reduce the collision in our hash table. Let imagine that if each bucket couldn’t be hashed by two or more itemsets, so the entry value represents the number of sets which were hashed into a bucket. It is also the number that itemsets was generated. So, the decide will more precise.

The PHP (perfect hashing and database pruning) algorithm [12] was proposed to solve problem of DHP, based on principle as we have already analyzed. This algorithm is a developed version of DHP algorithm, not like DHP, the hash function of PHP was designed to match two distinct itemsets into two different place in the hash table. Therefore, we won’t need to concern on problem of multi-sets in a bucket. To avoid making hash table too large, we only insert into hash table the itemsets which all the subsets (k-1) items are frequent. So the table should fit in memory.

2.2.1 Brief description of algorithm

First: create a table with number of buckets equal to the number of distinct items. And hash items in transactions of database into corresponding buckets. From the value associated with each bucket, which represents the number this element appeared, we find out the large itemsets from initial data. Next, remove all items which are not large items in the database. The process will be repeated until no more frequent itemsets could be found. The main different feature we must remember is that, the hash tables only allow exactly the same itemsets to be hashed into one of their bucket. To ensure that, we use a sub function, whose mission is adding a new bucket if the value of itemsets has not yet appeared in table and assigned 1 for this entry, in the other case, when there is a bucket contained the itemsets, it’s entry value will be increase by 1. In the pruning step, all tasks are similar to DHP algorithm that is: delete transaction containing no large itemsets. After that, in a transaction, remove items which are not appeared in at least k frequent k-itemsets. In addition, a pruning task also is done when generating hash table: if a candidate (k+1)-itemsets hasn’t got enough k large sub k-itemsets then it will not be hashed.
2.2.2 Pseudo-code

\[ F_1 = \emptyset; \]

//Make \( H_1 \)

\textbf{for each} transaction \( t \in D \) \textbf{do}

\hspace{1em} \textbf{for each} item \( x \) in \( t \) \textbf{do} \( H_1.add(x); \)

//Find \( L_1 \)

\textbf{for each} itemsets \( y \) in \( H_1 \) \textbf{do}

\hspace{1em} \text{if} \( H_1.count(y) \) \text{then} \( L_1 = L_1 \cup y; \)

//Remove the hash values without the minimum support

\( H_1.prune(minsup); \)

\( D_1 = D; \)

//Find \( L_k \), \( k \geq 2 \) and prune \( DB - D_k \)

\( k = 2; \)

\textbf{repeat}

\hspace{1em} \( D_k = \emptyset; \)

\hspace{2em} \( L_k = \emptyset; \)

\hspace{1em} \textbf{for each} transaction \( t \in D_{k-1} \) \textbf{do begin}

\hspace{2em} \text{if} \( \forall w | w \not\in L_{k-1} \) \text{then} \text{skip} \text{t;}

\hspace{2em} \text{else begin}

\hspace{3em} \textit{items} = \emptyset;

\hspace{3em} \textbf{for each} \( k\)-itemsets \( y \) in \( t \) \textbf{do}

\hspace{4em} \text{if} \( \exists z | ((z = k-1 \text{ subset of } y) \land (\neg H_{k-1}.countt(z))) \)

\hspace{4em} \text{then begin}

\hspace{5em} \( H_k.add(y); \)

\hspace{5em} \textit{items} = \textit{items} \cup y;

\hspace{4em} \text{end}

\hspace{3em} \text{end}

\textbf{end}

\textbf{end}

\textbf{for each} itemsets \( y \) in \( H_k \) \textbf{do}

\hspace{1em} \text{if} \( H_k.count(y) \) \text{then} \( L_k = L_k \cup y; \)
Hash-Based Approach to Data Mining

\begin{align*}
H_k,\text{prune}(\text{minsup}); \\
k++; \\
\text{until } L_{k-1} = \varnothing; \\
\text{Answer } = \cup_k L_k ;
\end{align*}

2.2.3 Example

Example 3: (similar to example 2, using PHP algorithm)

<table>
<thead>
<tr>
<th>Transaction database</th>
<th>Hash function: $h(x) = (x$’s order) mod 6</th>
<th>Hash table for 1-itemsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>TID</td>
<td>Items</td>
<td>B</td>
</tr>
<tr>
<td>100</td>
<td>ABCD</td>
<td>A</td>
</tr>
<tr>
<td>200</td>
<td>ABCDF</td>
<td>F</td>
</tr>
<tr>
<td>300</td>
<td>BCDE</td>
<td>F</td>
</tr>
<tr>
<td>400</td>
<td>ABCDF</td>
<td>F</td>
</tr>
<tr>
<td>500</td>
<td>ABEF</td>
<td>F</td>
</tr>
</tbody>
</table>

Figure 2: Example of hash table for PHP algorithm

$\rightarrow L_1 = \{\{A\};\{B\};\{C\};\{D\}\}$

$\rightarrow$ Database after pruned:

\[D_2 = \{<100;\{ABCD\}>,<200;\{ABCD\}>,<300;\{BCD\}>,<400;\{ABCD\}>,<500;\{AB\}>\}

Hash function for 2-itemsets:

\[H(x, y) = [(x$’s order) * (y$’s order)]] \text{ mod (4*4)}

Possible 2-itemsets from transactions in $D_2$:

100, 200, 400: \{AB\}, \{AC\}, \{AD\}, \{BC\}, \{BD\}, \{CD\}

300: \{BC\}, \{BD\}, \{CD\}

500: \{AB\}
Generate hash table:

\[ L_2 = \{\{AB\},\{BC\},\{BD\},\{CD\}\} \]

Database after pruned:

\[ D_3 = \{<100;\{BCD\}>, <200;\{BCD\}>, <300;\{BCD\}>, <400;\{BCD\}>\} \]

Do the same thing as the previous example and we have the same result.

In this example, we do not create \{AD\} since its entry value is 3, less than 4. So our work was better.

Besides the improvement of the algorithm, it’s can be easily seen that, with the way it works, there are still some disadvantages. From the cause that our hash tables do not have fixed size, it’s will be expanded each time to be added, moreover, to guarantee that each itemsets will be hashed in to distinct place we must have a strategy to build up the hash function. On the other hand, the more elements in an itemsets, the more parameters we need to control our function; the more parameters, the more complicated it is. Not included, in the worst case, the hash table we need maybe very large, that will have many infection to the process in general.

### 2.3 PHS algorithm (Perfect hashing and database shrinking)

As we knew from the previous sessions, from Apriori – one of the earliest algorithms of this field, DHP was much more effective than Apriori due to ability to curtail database, but it’s still the disadvantage while it hasn’t solved collisions positive that means, more than one itemsets are hashed into a bucket. Therefore we found a new way, using the perfect hash function in the PHP algorithm to eliminate the above issue. Despite that benefit, this algorithm has potential to need a really complex function to uses as the perfect hash function. Assuming
that if we are checking for 10-itemsets, each element have 10 candidate values, in order to assure that each possible itemsets (in the worst situation) will be hashed in to other location (although most of these sets will be removed by us) we have to create a function that space supply enough to demand. It’s believed that the function will have 10 parameters and have approximately $10^{10}$ values. This is surely leading us to an extremely complicated function.

Aim to restrict this weakness, we think of a method which could reduces complex level of the hash function. The simplest work we can do is have the number of function parameters limited.

PHS [5] was proposed with the trend of using the perfect hash function, like previous version, it uses perfect hash function to provide separate location for each hashed itemsets. In addition, a mechanism has been used to fix the length of sets at two. Therefore, the algorithm is not only eliminating the collision problem but also easy for us to create hash function, as well as easy to implement in spite of extra work we must do.

Now we will discuss a bit more about this method:

As, I talked briefly above, similar to PHP but PHS algorithm fix only two elements each sets, to execute this difference, we do some extra things, that would transform two essential k-itemsets into a single (k+1)-itemsets each iteration. This job is implemented by the operator $\text{Join}$ which is defined as below:

Supposed that we have two itemsets, which of them have k elements:

$S_1 = p_1 p_2 \ldots p_k$ and $S_2 = q_1 q_2 \ldots . q_k$

These sets satisfied condition: $p_2 = q_1, \ldots , p_k = q_{k-1}$ and $p_1 < q_k$ (as in other algorithm, we assume that transactions are kept items in lexical order. ). The result of joining these sets is a set S that have k+1 elements: $p_1, p_2, \ldots , p_k, q_k$ and S is written as $(S_1, S_2)$.

2.3.1 Algorithm description

At the beginning of this algorithm, we do the same thing as the other, that means, scanning and counting the support, filtering out the items with right support and dropping the remaining – have not enough support - from the
essential database. After having had sets of frequent 1-itemsets, we join and hash these sets into hash table, as we did in PHP (not exceed 1 set per bucket and count the total times the bucket was hashed, stores this value by an entry associated with the bucket.) When this work has been done, we have the table contained candidate 2-itemsets, and the number of their appearance. Then from the counted values, we obtain frequent 2-itemsets. From now on, our task will be different from former algorithms. Now, we have had frequent 2-itemsets, we consider these sets as items in a new system, each 2-itemsets is unique corresponded to a word in the new system, and the correspondent of all sets must be stored in a table (called lookup table), this table will be needed when we want to extract frequent itemsets. It does also truncate the items which is not a frequent 2-itemsets from database.

At this time, we are having a database in the new system (each items is a large 2-itemsets in the old) and we must generate some candidates 3-itemsets in the old by join two 2-itemsets or by join operator two word in new system. So sets of 3-itemsets in the basic database will be in form of 2-itemsets under considering database. If this entire works have finished, then we would have finished iteration of the algorithm.

Next, we reiterate these works to the new items in new system and keep continuing until there are no join could be done.

By the process go ahead, the database shrinks, since all the non frequent items will be trimmed during the processing. So the size of database in the next iteration is much smaller than in the previous.

After finishing the repeat, we must retrace our step to extract large itemsets. This is the time we need help of lookup tables that we were create when iterate. This stage quite simple, its job is doing the reverse. From the lookup tables, we replace recursively the items in a lookup table by two correspondent items in previous pass lookup table. After all lookup tables have been extracted, we have the final result.
2.3.2 Pseudo-code

Derive large 1-itemsets in the same way as Apriori ($L_1$)
s: the minimum support
$H_k$: hash table for candidate $k$-itemsets
$LUT_k$: lookup table for storing coding relationship at $k$-th iteration

$k=2$

$D_k = D$

//get large itemsets $L_k$ ($k \geq 2$)

**While** ($D_k \neq \emptyset$) **do**

\[
D = D_k;
\]

//Construct $H_k$ for candidate $k$-itemsets

**for each** transaction $t \in D_k$ **do**

**for all** 2-itemsets $x$ of $t$ do

$H_k$.add($x$);

//record the relationship if the itemsets has support $\geq$ minsup

**for each** itemsets $y$ in $H_k$ **do**

if $H_k$.count($y$) **then** record the coding relationship on $LUT_k$;

//shrink database and get the next candidate itemsets

$D_k = \emptyset$

**for each** transaction $t \in D$ **do**

concatenate joinable pairs of 2-itemsets in $t$ and then transform
them into new 2-itemsets by looking them up in $LUT_k$

$D_k = D_k \cup \{t\}$;

end

$k++$

end

**Answer** = decode ($LUT_k$);

2.3.3 Example

Example 4: same as in example 2, work with the PHS algorithm.

Transaction database
Now we hash these pairs to a hash table to determine which is large.

Table 5: Scan and count i-itemsets

<table>
<thead>
<tr>
<th>Items</th>
<th>Sup</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 6: Frequent 1-itemsets

<table>
<thead>
<tr>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>D</td>
</tr>
</tbody>
</table>

Table 7: Candidate 2-itemsets in second pass

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>(AB)(AC)(AD)(BC)(BD)(CD)</td>
</tr>
<tr>
<td>200</td>
<td>(AB)(AC)(AD)(BC)(BD)(CD)</td>
</tr>
<tr>
<td>300</td>
<td>(BC)(BD)(CD)</td>
</tr>
<tr>
<td>400</td>
<td>(AB)(AC)(AD)(BC)(BD)(CD)</td>
</tr>
<tr>
<td>500</td>
<td>(AB)</td>
</tr>
</tbody>
</table>

Table 8: Frequent 1-itemsets

<table>
<thead>
<tr>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>D</td>
</tr>
</tbody>
</table>

Now we hash these pairs to a hash table to determine which is large.

Table 5: Scan and count i-itemsets

<table>
<thead>
<tr>
<th>Items</th>
<th>Sup</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 6: Frequent 1-itemsets

<table>
<thead>
<tr>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>D</td>
</tr>
</tbody>
</table>

Table 7: Candidate 2-itemsets in second pass

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>(AB)(AC)(AD)(BC)(BD)(CD)</td>
</tr>
<tr>
<td>200</td>
<td>(AB)(AC)(AD)(BC)(BD)(CD)</td>
</tr>
<tr>
<td>300</td>
<td>(BC)(BD)(CD)</td>
</tr>
<tr>
<td>400</td>
<td>(AB)(AC)(AD)(BC)(BD)(CD)</td>
</tr>
<tr>
<td>500</td>
<td>(AB)</td>
</tr>
</tbody>
</table>

From the hashed table, we choose sets satisfied minsup and put it into the lookup table to use in the inverse phase. And so on…
Now we have only one item so we cannot execute the operator Join anymore. The work will stop here.

The frequent itemsets we have during our process are:

\[
\{\{A\} \{B\} \{C\} \{D\}\} \cup \{\{a\} \{b\} \{c\} \{d\}\} \cup \{x\}
\]

To have value of \(x\), \(a\), \(b\), \(c\), \(d\), we start at the bottom of the algorithm, using values which were stored in the down process to decode this items. In this example, from lookup tables, we have:

\[
\{x\} = \{bd\}
\]

\[
\{a\} = \{AB\}; \{b\} = \{BC\}; \{c\} = \{BD\} \text{ and } \{d\} = \{CD\}
\]

<table>
<thead>
<tr>
<th>Encoding</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>(AB)</td>
<td>(BC)</td>
<td>(BD)</td>
<td>(CD)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>(ab)(ac)(bd)</td>
</tr>
<tr>
<td>200</td>
<td>(ab)(ac)(bd)</td>
</tr>
<tr>
<td>300</td>
<td>(bd)</td>
</tr>
<tr>
<td>400</td>
<td>(ab)(ac)(bd)</td>
</tr>
<tr>
<td>500</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>(AB)(BC)(BD)(CD)</td>
</tr>
<tr>
<td>200</td>
<td>(AB)(BC)(BD)(CD)</td>
</tr>
<tr>
<td>300</td>
<td>(BC)(BD)(CD)</td>
</tr>
<tr>
<td>400</td>
<td>(AB)(AC)(BC)(CD)</td>
</tr>
<tr>
<td>500</td>
<td>(AB)</td>
</tr>
</tbody>
</table>

Table 9: Lookup table of the second pass

Table 10: candidate itemsets for the third

Table 11: large itemsets of the second

Table 12: Hash table of the third pass

Table 13: Lookup table of the third pass

<table>
<thead>
<tr>
<th>Encoding</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>(bd)</td>
</tr>
</tbody>
</table>
\{x\} = \{BCD\}

So, the final result is:

\[ L = \{\{A\}, \{B\}, \{C\}, \{D\}, \{AB\}, \{BC\}, \{BD\}, \{BCD\}\} \]

From this example, we could see that the process to find out frequent itemsets hidden in the database was much simpler than the previous one. In theoretical, it was ameliorated both on the side of complexity and running time since it was not only eliminates collision but also fix the length of candidate itemsets to simplify the task of making hash function.

### 2.4 Summarize of chapter

Via a lot of test over real databases, with a large amount of data [4-12,14], they were proved that three algorithms, which were presented, brought a very comprehensive result compare to the former (for instance: Apriori).

Apriori – DHP: used database has: T15.I4.D100

![Figure 3: Execution time of Apriori and DHP](image)

Apriori – PHP: database include of 11512 transactions, 5000 items
Hash-Based Approach to Data Mining

Figure 4: Execution time of Apriori and PHP.

DHP – PHS: Used database: T5I4D200K

Figure 5: Execution time of PHS and DHP

It is easy to understand after we had studied and analyzed carefully these algorithms, due to the main feature that they significantly reduced the number of total scanning database by using hash mechanism. In addition, trimming redundant items and transactions during their process, so the database becomes smaller; this is one more important reason to make the algorithms better.

In the three ones was studied, the first algorithm – DHP – still exist problem of wrong place when multi itemsets are hashed in the same bucket of hash table,
so the entry value for the bucket may not reflect the right large one, thus, we must scan one more time to have the right one. This problem was solved in the second algorithm state here – PHP – by using hash table with maximum one itemsets per bucket. But the other problem had emerged, that is the complexity in building up the perfect hash function when the number of parameter is increasing and their domain is large. The last one – PHS - gave a new way to extirpate this problem: work with only sets contained 2 items in it. And it make this idea become true by lookup table in each of pass. In short, these algorithms are steps of all the improvement process to find the best one, they provided quite good result and is possible to be used.
CHAPTER 3: Experiments

From chapter 2, we all have known about three algorithms following the hash-based approach. Now I’m going to choose one of these and use it in a simple system established by me, which function is assisting managements in finding association rules that would be using in some other part of a more complex system. Therefore, at first, we will preliminarily evaluate these algorithms and after that chose an algorithm. Finally, we will start to design and build up the system.

3.1 Choosing algorithm

As I mentioned before, our three candidates are DHP, PHP and PHS algorithms. Basically, since summarizing of chapter 2, we saw that PHS may be a good choice for our request: it requires not much complicated process in a competitive running time. Therefore, it is chosen to be used in the system.

Now, the only problem we must care is the hash function of the algorithm. [5] proposed two types: direct hash and partial hash. The direct one is simple and fast, but needed a mount of memories; the other is a bit more complex but could fit in smaller memory.

Direct hash method, really simple, will associates each bucket of the hash table with unique code word – these code words are joined from two large itemsets in the prior pass, and the hash table will have buckets for all of candidates. So we can choose the function:
Hash-Based Approach to Data Mining

\[ \text{hash}(X, Y) = \left( C^n_2 - C^{n-X's\ index}_2 \right) + (X's\ index - Y's\ index) - 1 \]

In this formula, \( n \) is the number of new items - code words - in the preceding pass, \( X \) and \( Y \) are the two new items, \( X \)'s order and \( Y \)'s order are the position of \( X \) and \( Y \) in code table in the lexicographic order. By this function, we ensure that we can inverse the join, i.e. from one items of a pass we correspond with two items of the previous one; this ability is consequence of the unique feature of the hash function.

The second type, partial hash is much more complex compare to the first type. Because it uses two-level perfect hashing (for more detail, see [5]). Because the direct hash method is easier to understand, simpler to be coded, and still provided quite good accuracy. I chose the method of direct hash to implement.

3.2 Implement

Now, we will consider the system. That is easy to see that, nowadays, the commerce is more and more developed, more and more stores, shops, markets and supermarkets have been built to satisfy customer need. So the manager must investigate market to finding laws that would useful for their competition. One of things they can do: start at the sale data; find out the products which could be the best-selling in a period or some products usually bought together… This will help them predict tendency of customers, determine which commodities they should import and how they should put on the sale shelf…. The development the software which manages and finds these rules like this was developed for such a long time in many countries all over the world. However, due to its cost price and the small scale of our retail store, these applications still very few in our country. So, our system will try to do some of these tasks. In it the main task we aim at is find out the association rules from customer transactions. In addition, at least, it has ability of a management system, that is enable user enter new transaction, and may be have some other functions such as find the goods that have smoothly sold, find the percents of a product in a period of time….
To build the system, we must have database contains transactions. Each transaction has two fields: transaction ID (TID) and product ID (PID). The system will be written in C# language.

There is a menu, with basic function: Input transactions, find best-selling products, analyses a product support, find top x product have high accuracy go together….

Input function provides a form for user enter a new transaction (this data may be from another part of total system). The PID in this new transaction will be sort in the lexicographic order and put into database.

The function to find best-selling product… will take some parameter from user, for instance: minsup, minconf, numbers of rules we consider…

Experimental environment:
- Hard ware: HP computer, 3.4GHz, 1GB Ram.
- Database: simulation data from a shop, which controlled transaction by barcode reader.
- Software: include some module as I represented above.

Result: we consider the result of software in situation:
- First: when the database is small (enough for us find association rules by ourselves). This case is check the confident of the software if the are working correctly.
- Second: when the database is large, this will take some sub-case. And based on the runtime of the software, we’d have a more precise evaluate about the accuracy of soft.
  i. we are going to use the transaction database as follow:

With: minsup = 2, minconf = 0.6.

We calculate and find the association rules are:
(pate->bread, bread->ice, bread->beer, beer -> bread, yogurts -> ice-cream, ice-cream -> yogurts, yogurts ->beer, ice-cream -> beer)
And:(bread, pate -> beer; bread, beer -> pate; pa -> bread, beer; pa, beer -> bread)
The result generated by our software is:

<table>
<thead>
<tr>
<th>TID</th>
<th>Products</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bread, yogurts, pate, ice-cream, beer</td>
</tr>
<tr>
<td>1</td>
<td>Meat, bread, beer</td>
</tr>
<tr>
<td>2</td>
<td>Bread, pate, beer</td>
</tr>
<tr>
<td>3</td>
<td>Yogurts, ice-cream, beer</td>
</tr>
<tr>
<td>4</td>
<td>Bread, ice-cream</td>
</tr>
</tbody>
</table>

The second case still collecting data. (The whole process will be completed after a few days.)
CONCLUSION

Finding large itemsets – find out sets of items, which have frequency appear together higher than a given number – is a very important part in the process of finding association rules. It works with a large amount of data so the problem of optimizing the process and reducing data scanning will influence the effect of this step in particular and influences all the process in general a lot. The more data could be ignored, the more running time we save. While database expands day after day, and becomes very colossal, we try to make the size of transactions which are needed to scan in iteration smaller and fit our work in limited resources.

This thesis concerns about the problem above and investigates some related algorithms, these algorithms use hash-based approach to reduce the implement size of database and quickly process. And based on these ideas, establish a small system to find association rules in transaction database.

Contribution of the thesis:

- To evaluate the important of finding associations rules and specify the main cost of the process finding them.
- To present, illustrate and analyze the strength and weakness of some algorithms using hash-based approach.
- To build up a system to manage a small soft, find interesting rules related to customer routines. This soft used PHS algorithms and provided good result.
Because of my ability, my system design experience and the short time, my established system is in small scale and still has some mistakes. However, I found that this is an up-and-coming trend to apply in real-life, especially in our trade, for the time being. In the future, I am going to develop this system.

There are some directions:

- To study and experiment to improve the performance of the algorithm
- To consider some more complicated problems, which care for the time factor in transactions.
- To add some more functions to make the system not only suitable for traditional commerce but also be able to use for e-commerce.
REFERENCES

**Vietnamese**


**English**


[15] [www.citeulike.org](http://www.citeulike.org)

[16] [www-users.cs.umn.edu](http://www-users.cs.umn.edu)

[17] [www.wikipedia.org](http://www.wikipedia.org)