An Effective Hash Based Algorithm for Mining Association Rules

Jong Soo Park*, Ming-Syan Chen and Philip S. Yu

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598
{c1park, mschen, psyu}@watson.ibm.com

Abstract

In this paper, we examine the issue of mining association rules among items in a large database of sales transactions. The mining of association rules can be mapped into the problem of discovering large itemsets where a large itemset is a group of items which appear in a sufficient number of transactions. The problem of discovering large itemsets can be solved by constructing a candidate set of itemsets first and then, identifying, within this candidate set, those itemsets that meet the large itemset requirement. Generally this is done iteratively for each large k-itemset in increasing order of k where a large k-itemset is a large itemset with k items. To determine large itemsets from a huge number of candidate large itemsets in early iterations is usually the dominating factor for the overall data mining performance. To address this issue, we propose in this paper an effective algorithm for the candidate set generation. It is a hash based algorithm and is especially effective for the generation of candidate set for large 2-itemsets. Explicitly, the number of candidate 2-itemsets generated by the proposed algorithm is, in orders of magnitude, smaller than that by previous methods, thus resolving the performance bottleneck. Note that the generation of smaller candidate sets enables us to effectively trim the transaction database size at a much earlier stage of the iterations, i.e., right after the generation of large 2-itemsets, thereby reducing the computational cost for later iterations significantly. Extensive simulation study is conducted to evaluate performance of the proposed algorithm.

Index Terms — Data mining, association rules, hashing, performance analysis.

---

*Visiting from the department of computer science, Sungshin Women's University and partially supported by KOSEF, Korea.

1 Introduction

Recently, database mining has attracted a growing amount of attention in database communities due to its wide applicability in retail industry to improving marketing strategy. As pointed out in [5], the progress in bar-code technology has made it possible for retail organizations to collect and store massive amounts of sales data. Catalog companies can also collect sales data from the orders they receive. A record in such data typically consists of the transaction date, the items bought in that transaction, and possibly also customer-id if such a transaction is made via the use of a credit card or any kind of customer card. It is noted that analysis of past transaction data can provide very valuable information on customer buying behavior, and thus improve the quality of business decisions (such as what to put on sale, which merchandises to be placed on shelves together, how to customize marketing programs, to name a few). It is essential to collect a sufficient amount of sales data (say, over last 30 days) before we can draw any meaningful conclusion from them. As a result, the amount of these sales data tends to be huge. It is therefore important to devise efficient algorithms to conduct mining on these data. The requirement to process large amount of data distinguishes data mining in the database context from its study in the AI context.

One of the most important data mining problems is mining association rules. For example, given a database of sales transactions, it would be interesting to discover all associations among items such that the presence of some items in a transaction will imply the presence of other items in the same transaction. The problem of mining association rules in the context of database was first explored in [3]. In this pioneering work, it is shown that mining association rules can be decomposed into two subproblems. First, we need to identify all sets of items (itemsets) that are contained in a sufficient number of transactions above the minimum (support) requirement. These itemsets are referred to as *large itemsets*. Once all large itemsets are obtained, the desired association rules can be generated in a straightforward manner. Subsequent work in the literature followed this approach and focused on the large itemset generations.

Various algorithms have been proposed [3, 5, 8] to discover the large itemsets. Generally speaking, these algorithms first construct a candidate set of large itemsets based on some heuristics, and then discover the subset that indeed contains large itemsets. This process can be done iteratively in the sense that the large itemsets discovered in one iteration will be used as the basis to generate the candidate set for the next iteration. For example, in [5], at the k^{th} iteration, all large itemsets containing k items, referred to as large k-itemsets, are generated. In the next iteration, to construct a candidate set of large $(k+1)$-itemsets, a heuristic is used to expand some large k-itemsets into a $(k+1)$-itemset, if certain constraints are satisfied[1].

[1]A detailed description of the algorithm used in [5] is given in Section 2.

The heuristic to construct the candidate set of large itemsets is crucial to performance. Clearly, in order to be efficient, the heuristic should only generate candidates with high likelihood of being large itemsets because for each candidate, we need to count its appearances in all transactions. The larger the candidate set, the more processing cost required to discover the large itemsets. As previously reported in [5] and also attested by our experiments, the processing in the initial iterations in fact dominates the total execution cost. A performance study was provided in [5] to compare various algorithms of generating large itemsets. It was shown that for these algorithms the candidate set generated during an early iteration is generally, in orders of magnitude, larger than the set of large itemsets it really contains. *Therefore, the initial candidate set generation, especially for the large 2-itemsets, is the key issue to improve the performance of data mining.*

Another performance related issue is on the amount of data that has to be scanned during large itemset discovery. A straightforward implementation would require one pass over the database of all transactions for each iteration. Note that as k increases, not only is there a smaller number of large k-itemsets, but also there are fewer transactions containing any large k-itemsets. Reducing the number of transactions to be scanned and trimming the number of items in each transaction can improve the data mining efficiency in later stages. In [5], two alternative approaches are considered: Apriori and AprriorTid. In the Apriori algorithm, each iteration requires one pass over the database. In the AprioriTid algorithm, the database is not scanned after the first pass. Rather, the transaction id and candidate large k-itemsets present in each transaction are generated in each iteration. This is used to determine the large $(k+1)$-itemsets present in each transaction during the next iteration. It was found that in the initial stages, Apriori is more efficient than AprioriTID, since there are too many candidate k-itemsets to be tracked during the early stages of the process. However, the reverse is true for later stages. A hybrid algorithm of the two algorithms was also proposed in [5], and shown to lead to better performance in general. The challenge to operate the hybrid algorithm is to determine the switch-over point.

In this paper, we shall propose an algorithm DHP (standing for direct hashing and pruning) for efficient large itemset generation. Specifically, DHP proposed has two major features: one is efficient generation for large itemsets and the other is effective reduction on transaction database size. As will be seen later, by utilizing a hash technique, DHP is very efficient for the generation of candidate large itemsets, in particular for the large 2-itemsets, where the number of candidate large itemsets generated by DHP is, in orders of magnitude, smaller than that by previous methods, thus greatly improving the performance bottleneck of the whole process. In addition, DHP employs effective pruning techniques to progressively reduce the transaction database size. As observed in prior work [5], during the early iterations, tracking the candidate k-itemsets in each transaction is ineffective since, the cardinality of such k-itemsets is very large. Note that the generation of smaller candidate sets by DHP enables us to effectively trim the transaction database at a much earlier stage of the iterations, i.e., right after the generation of large 2-itemsets, thereby reducing the computational cost for later iterations significantly. It will be seen that by exploiting some features of association rules, not only the number of transactions, but also the number of items in each transaction can be substantially reduced. Extensive experiments are conducted to evaluate the performance of DHP. As shown by our experiments, with a slightly higher cost in the first iteration due to the generation of a hash table, DHP incurs significantly smaller execution times than Apriori[2] in later iterations, not only in the second iteration when a hash table is used by DHP to facilitate the generation of candidate 2-itemsets, but also in later iterations when the same procedure for large itemset generation is employed by both algorithms, showing the advantage of effective database trimming by DHP. Sensitivity analysis for various parameters is conducted[3].

We mention in passing that the discovery of association rules is also studied in the AI context [11] and there are various other aspects of data mining explored in the literature. Classification is an approach of trying to develop rules to group data tuples together based on certain common characteristics. This has been explored both in the AI domain [12] and in the context of databases [2, 6, 7, 10]. Another source of data mining is on ordered data, such as stock market and point of sales data. Interesting aspects to explore include searching for similar sequences [1], e.g. stocks with similar movement in stock prices, and sequential patterns [4], e.g. grocery items bought over a set of visits in sequence.

This paper is organized as follows. A detailed problem description is given in Section 2. The algorithm DHP proposed for generating large itemsets is described in Section 3. Performance results are presented in Section 4. Section 5 contains the summary.

2 Problem Description

Let $\mathcal{I}=\{i_1, i_2, ..., i_m\}$ be a set of literals, called items. Let D be a set of transactions, where each transaction T is a set of items such that $T \subseteq \mathcal{I}$. Note that the quantities of items bought in a transaction are not considered, meaning that each item is a binary variable representing if an item was bought. Each transaction is associated with an identifier, called TID. Let X be a set of items.

[2]A comprehensive study on various algorithms to determine large itemsets is presented in [5], where the Apriori algorithm is shown to provide the best performance during the initial iterations. Hence, Apriori is used as the base algorithm to compare with DHP in this study.

[3]It should be noted that in [5], the hybrid algorithm has the option of switching from Apriori to another algorithm AprioriTID after early passes for better performance. For ease of presentation of this paper, such an option is not adopted here. Nevertheless, the benefit of AprioriTID in later passes is complementary to the focus of DHP on initial passes.

Database D

TID	Items
100	A C D
200	B C E
300	A B C E
400	B E

Figure 1: An example transaction database for data mining

A transaction T is said to contain X if and only if $X \subseteq T$. An association rule is an implication of the form $X \Longrightarrow Y$, where $X \subset \mathcal{I}$, $Y \subset \mathcal{I}$ and $X \bigcap Y = \phi$. The rule $X \Longrightarrow Y$ holds in the transaction set D with *confidence* c if $c\%$ of transactions in D that contain X also contain Y. The rule $X \Longrightarrow Y$ has *support* s in the transaction set D if $s\%$ of transactions in D contain $X \bigcup Y$.

As mentioned before, the problem of mining association rules is composed of the following two steps:

1. Discover the large itemsets, i.e., all sets of itemsets that have transaction support above a pre-determined minimum support s.

2. Use the large itemsets to generate the association rules for the database.

The overall performance of mining association rules is fact determined by the first step. After the large itemsets are identified, the corresponding association rules can be derived in a straightforward manner. In this paper, we shall develop an algorithm to deal with the first step, i.e., discovering large itemsets from the transaction database. Readers interested in more details for the second step are referred to [5]. As a preliminary, we shall describe the method used in the prior work Apriori for discovering the large itemsets from a transaction database given in Figure 1[4]. As pointed out earlier, a comprehensive study on various algorithms to determine large itemsets is presented in [5], where the Apriori algorithm is shown to provide the best performance during the initial iterations. Hence, Apriori is used as the base algorithm to compare with DHP. In Apriori [5], in each iteration (or each pass) it constructs a candidate set of large itemsets, counts the number of occurrences of each candidate itemset, and then determine large itemsets based on a pre-determined minimum support. In the first iteration, Apriori simply scans all the transactions to count the number of occurrences for each item. The set of candidate 1-itemsets, C_1, obtained is shown in Figure 2. Assuming that the minimum transaction support required is 2, the set of large 1-itemsets, L_1, composed of candidate 1-itemsets with the minimum support required, can then be determined.

[4]This example database is extracted from [5].

Scan D $\longrightarrow$

C_1

Itemset	Sup.
{A}	2
{B}	3
{C}	3
{D}	1
{E}	3

L_1

Itemset	Sup.
{A}	2
{B}	3
{C}	3
{E}	3

C_2

Itemset
{A B}
{A C}
{A E}
{B C}
{B E}
{C E}

Scan D $\longrightarrow$

C_2

Itemset	Sup.
{A B}	1
{A C}	2
{A E}	1
{B C}	2
{B E}	3
{C E}	2

L_2

Itemset	Sup.
{A C}	2
{B C}	2
{B E}	3
{C E}	2

C_3

Itemset
{B C E}

Scan D $\longrightarrow$

C_3

Itemset	Sup.
{B C E}	2

L_3

Itemset	Sup.
{B C E}	2

Figure 2: Generation of candidate itemsets and large itemsets

To discover the set of large 2-itemsets, in view of the fact that any subset of a large itemset must also have minimum support, Apriori uses $L_1 * L_1$ to generate a candidate set of itemsets C_2 using the apriori candidate generation, where $*$ is an operation for concatenation. C_2 consists of $\binom{|L_1|}{2}$ 2-itemsets. Note that when $|L_1|$ is large, $\binom{|L_1|}{2}$ becomes an extremely large number. Next, the four transactions in D are scanned and the support of each candidate itemset in C_2 is counted. The middle table of the second row in Figure 2 represents the result from such counting in C_2. A hash tree is usually used for a fast counting process [9][5]. The set of large 2-itemsets, L_2, is therefore determined based on the support of each candidate 2-itemset in C_2.

The set of candidate itemsets, C_3, is generated from L_2 as follows. From L_2, two large 2-itemsets with the same first item, such as $\{BC\}$ and $\{BE\}$, are identified first. Then, Apriori tests whether the 2-itemset $\{CE\}$, which consists of their second items, constitutes a large 2-itemset or not. Since $\{CE\}$ is a large itemset by itself, we know that all the subsets of $\{BCE\}$ are large and then $\{BCE\}$ becomes a candidate 3-itemset. There is no other candidate 3-itemset from L_2. Apriori then scans all the transactions and discovers the large 3-itemsets L_3 in Figure 2. Since there is no candidate 4-itemset to be constituted from L_3, Apriori ends the process of discovering large itemsets. As it can be seen above, it is important to generate as small a candidate set of itemsets as possible

[5]The use of a hash tree to count the support of each candidate itemset is a common feature for the two algorithms (i.e., Apriori and DHP) discussed in this paper, and should not be confused with the hash technique used by DHP to generate candidate itemsets.

because the support of each itemset in C_i has to be counted during the scan of the entire database. As we shall see later, by exploiting this feature, algorithm DHP proposed is able to generate large itemsets efficiently.

3 Direct Hashing with Efficient Pruning for Fast Data Mining

In this section, we shall propose an algorithm DHP (standing for direct hashing and pruning) for efficient large itemset generation. The proposed algorithm utilizes a hash method for candidate itemset generation during the initial iterations and employs pruning techniques to progressively reduce the transaction database size. The overall algorithm of DHP is described in Section 3.1. Database pruning techniques are given in Section 3.2.

3.1 Algorithm DHP

The algorithm DHP proposed has two major features: one is efficient generation for large itemsets and the other is effective reduction on transaction database size. We shall describe in this subsection the main flow of algorithm DHP, and explain its first feature, i.e., efficient generation for large itemsets. The feature of reducing database size will be described in Section 3.2. As illustrated in Section 2, in each pass we use the set of large itemsets, L_i, to form the set of candidate large itemsets C_{i+1} by joining L_i with L_i on $(i-1)$ (denoted by $L_i * L_i$) common items for the next pass. We then scan the database and count the support of each itemset in C_{i+1} so as to determine L_{i+1}. As a result, in general, the more itemsets in C_i, the higher the processing cost of determining L_i will be. It is noted that given a large database, the initial extraction of useful information from the database is usually the most costly part. Specifically, as previously reported in [5] and also attested by our experiments given in Section 4, the processing cost of the first two iterations (i.e., obtaining L_1 and L_2) in fact dominates the total processing cost. This can be explained by the reason that, for a given minimum support, we usually have a very large L_1, which in turn results in a huge number of itemsets in C_2 to process. Note that $|C_2|= \binom{|L_1|}{2}$ in Apriori. The step of determining L_2 from C_2 by scanning the whole database and testing each transaction against a hash tree built by C_2 is hence very expensive. By constructing a significantly smaller C_2, DHP can also generate a much smaller D_3 to derive C_3. Without the smaller C_2, the database can not be effectively trimmed. (This is one of the reasons that Apriori-TID in [5] is not effective in trimming the database until later stages.) After this step, the size of L_i decreases rapidly as i increases. A smaller L_i leads to a smaller C_{i+1}, and thus a smaller corresponding processing cost. In view of the above, algorithm DHP is so designed that it will reduce the number of itemsets to be explored in C_i in initial iterations significantly. The corresponding processing cost to determine L_i from C_i is therefore reduced.

In essence, algorithm DHP presented in Figure 3 uses the technique of hashing to filter out un-

```
/* Part 1 */
s = a minimum support;
set all the buckets of H_2 to zero;                    /* hash table */
forall transaction t ∈ D do begin
    insert and count 1-items occurrences in a hash tree;
    forall 2-subsets x of t do
        H_2[h_2(x)] + +;
end
L_1 = {c|c.count ≥ s, c exists in the leaf node of the hash tree};

/* Part 2 */
k = 2;
D_k = D;                          /* database 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 zero;
    D_{k+1} = φ;
    forall transactions t ∈ D_k do begin
        count_support(t, C_k, k, t̂);          /* t̂ ⊆ t */
        if (|t̂| > k) then do begin
            make_hasht(t̂, H_k, k, H_{k+1}, ẗ);
            if (|ẗ| > k) then D_{k+1} = D_{k+1} ∪ {ẗ};
        end
    end
    L_k = {c ∈ C_k|c.count ≥ s};
    k + +;
}

/* Part 3 */
gen_candidate(L_{k-1}, H_k, C_k);
while (|C_k| > 0) {
    D_{k+1} = φ;
    forall transactions t ∈ D_k do begin
        count_support(t, C_k, k, t̂);          /* t̂ ⊆ t */
        if (|t̂| > k) then D_{k+1} = D_{k+1} ∪ {t̂};
    end
    L_k = {c ∈ C_k|c.count ≥ s};
    if (|D_{k+1}| = 0) then break;
    C_{k+1}=apriori_gen(L_k);          /* refer to [5] */
    k + +;
}
```

Figure 3: Main program of algorithm DHP

Procedure gen_candidate($L_{k-1},\ H_k,\ C_k$)
    $C_k = \phi$;
    forall $c = c_p[1] \cdot \ldots \cdot c_p[k-2] \cdot c_p[k-1] \cdot c_q[k-1], c_p, c_q \in L_{k-1}, |c_p \cap c_q| = k-2$ **do**
        if $(H_k[h_k(c)] \geq s)$ **then**
            $C_k = C_k \cup \{c\}$;   /* insert c into a hash tree */
end Procedure

Procedure count_support($t,\ C_k,\ k,\ \hat{t}$)   /* explained in Section 3.2 */
    forall c such that $c \in C_k$ and $c\ (= t_{i_1} \cdots t_{i_k}) \in t$ **do begin**
        $c.count++$;
        for $(j = 1;\ j \leq k;\ j++)\ a[i_j]++$;
    end
    for $(i = 0,\ j = 0;\ i < |t|;\ i++)$
        if $(a[i] \geq k)$ **then do begin** $\hat{t_j} = t_i;\ j++$; **end**
end Procedure

Procedure make_hasht($\hat{t},\ H_k,\ k,\ H_{k+1},\ \ddot{t}$)
    forall $(k+1)$-subsets $x\ (= \hat{t}_{i_1} \cdots \hat{t}_{i_k})$ of $\hat{t}$ **do**
        if (for all k-subsets y of x, $H_k[h_k(y)] \geq s$) **then do begin**
            $H_{k+1}[h_{k+1}(x)]++$;
        for $(j = 1;\ j \leq k+1;\ j++)\ a[i_j]++$;
        end
    for $(i = 0,\ j = 0;\ i < |\hat{t}|;\ i++)$
        if $(a[i] > 0)$ **then do begin** $\ddot{t_j} = \hat{t}_i;\ j++$; **end**
end Procedure

Figure 4: Subprocedures for algorithm DHP

necessary itemsets for next candidate itemset generation. When the support of candidate k-itemsets is counted by scanning the database, DHP accumulates information about candidate $(k+1)$-itemsets in advance in such a way that all possible $(k+1)$-itemsets of each transaction after some pruning are hashed to a hash table. Each bucket in the hash table consists of a number to represent how many itemsets have been hashed to this bucket thus far. We note that based on the resulting hash table, a bit vector can be constructed, where the value of one bit is set to be one if the number in the corresponding entry of the hash table is greater than or equal to s. As can be seen later, such a bit vector can be used to greatly reduce the number of itemsets in C_i. This implementation detail is omitted in Figure 3.

Figure 3 gives the algorithmic form of DHP, which, for ease of presentation, is divided into 3 parts. Part 1 gets a set of large 1-itemsets and makes a hash table (i.e., H_2) for 2-itemsets. Part 2 generates the set of candidate itemsets C_k based on the hash table (i.e., H_k) generated in the previous pass, determines the set of large k-itemsets L_k, reduces the size of database for the next large itemsets (as will be explained in Section 3.2 later), and makes a hash table for candidate large $(k+1)$-itemsets. (Note that in Part 1 and Part 2, the step of building a hash table to be used by the next pass is a unique feature of DHP.) Part 3 is basically same as Part 2 except that it does not employ a hash table. Note that DHP is particularly powerful to determine large itemsets in early stages, thus improving the performance bottleneck. The size of C_k decreases significantly in later stages, thus rendering little justification its further filtering. This is the very reason that we shall use Part 2 for early iterations, and use Part 3 for later iterations when the number of hash buckets with a count larger than or equal to s (i.e., $|\{x|H_k[x] \geq s\}|$ in Part 2 of Figure 3) is less than a pre-defined threshold $LARGE$. We note that in Part 3 Procedure apriori_gen to generate C_{k+1} from L_k is essentially the same as the method used by algorithm Apriori in [5] in determining candidate itemsets, and we hence omit the details on it. Part 3 is included into DHP only for the completeness of our method.

After the setting by Part 1, Part 2 consists of two phases. The first phase is to generate a set of candidate k-itemsets C_k based on the hash table H_k, which is described by Procedure gen_candidate in Figure 4. Same as Apriori, DHP also generates a k-itemset by L_{k-1}. However, DHP is unique in that it employs the bit vector, which is built in the previous pass, to test the validity of each k-itemset. Instead of including all k-itemsets from $L_{k-1} * L_{k-1}$ into C_k, DHP adds a k-itemset into C_k only if that k-itemset passes the hash filtering, i.e., that k-itemset is hashed into a hash entry whose value is larger than or equal to s. As can be seen later, such hash filtering can drastically reduce the size of C_k. Every k-itemset that passes the hash filtering is included into C_k and stored into a hash tree [5], [9]. The hash tree built by C_k is then probed by each transaction later (i.e., in Part 2) when the database is scanned and the minimum support of each candidate itemset is counted. The second phase of Part 2 is to count the support of candidate itemsets and to reduce the size of each transaction, as described by Procedure count_support in Figure 4. Same as in [5], a subset function is used to determine all the candidate itemsets contained in each transaction. As transactions in the database (which is reduced after D_2) are scanned one by one, k-subset of each transaction are obtained and used to count the support of itemsets in C_k. The methods for trimming a transaction and reducing the number of transactions are described in detail in Section 3.2.

An example of generating candidate itemsets by DHP is given in Figure 5. For the candidate set of large 1-itemsets, i.e., C_1={ $\{A\}$,$\{B\}$,$\{C\}$,$\{D\}$,$\{E\}$}, all transactions of the database are scanned to count the support of these 1-items. In this step, a hash tree for C_1 is built on the fly for the purpose of efficient counting. DHP tests whether or not each item exists already in the hash tree. If yes, it increases the count of this item by one. Otherwise, it inserts the item with a count equal to one into the hash tree. For each transaction, after occurrences of all the 1-subsets are counted, all the 2-subsets of this transaction are generated and hashed into a hash table H_2 in such a way that when a 2-subset is hashed to bucket i, the value of bucket i is increased by one. Figure 5 shows a hash table H_2 for a given database. After the database is scanned, each bucket of the hash table has the number of 2-itemsets hashed to the bucket. Given the hash table in Figure 5 and a minimum support equal to 2, we obtain a resulting bit vector $\langle 1,0,1,0,1,0,1 \rangle$. Using this bit vector to filter out 2-itemsets from $L_1 * L_1$, we have C_2= { $\{AC\}$,$\{BC\}$,$\{BE\}$,$\{CE\}$}, instead of C_2= { $\{AB\}$, $\{AC\}$, $\{AE\}$, $\{BC\}$,$\{BE\}$,$\{CE\}$} resulted by Apriori as shown in Figure 2.

3.2 Reducing the Size of Transaction Database

DHP reduces the database size progressively by not only trimming each individual transaction size but also pruning the number of transactions in the database. Note that as observed in [5] on mining association rules, any subset of a large itemset must be a large itemset by itself. That is, {B,C,D} $\in L_3$ implies {B,C} $\in L_2$, {B,D} $\in L_2$, and {C,D} $\in L_2$. This fact suggests that a transaction be used to determine the set of large $(k+1)$-itemsets only if it consists of $(k+1)$ large k-itemsets in the previous pass. In view of this, when k-subsets of each transaction are counted toward candidate k-itemsets, we will be able to know if this transaction meets the necessary condition of containing large $(k+1)$-itemsets. This in turn means that if we have the number of candidate itemsets close to that of large itemsets when counting k-subsets, we can efficiently trim transactions and reduce the number of transactions by eliminating items which are found useless for later large itemset generation.

We now take a closer look at how the transaction size is trimmed by DHP. If a transaction

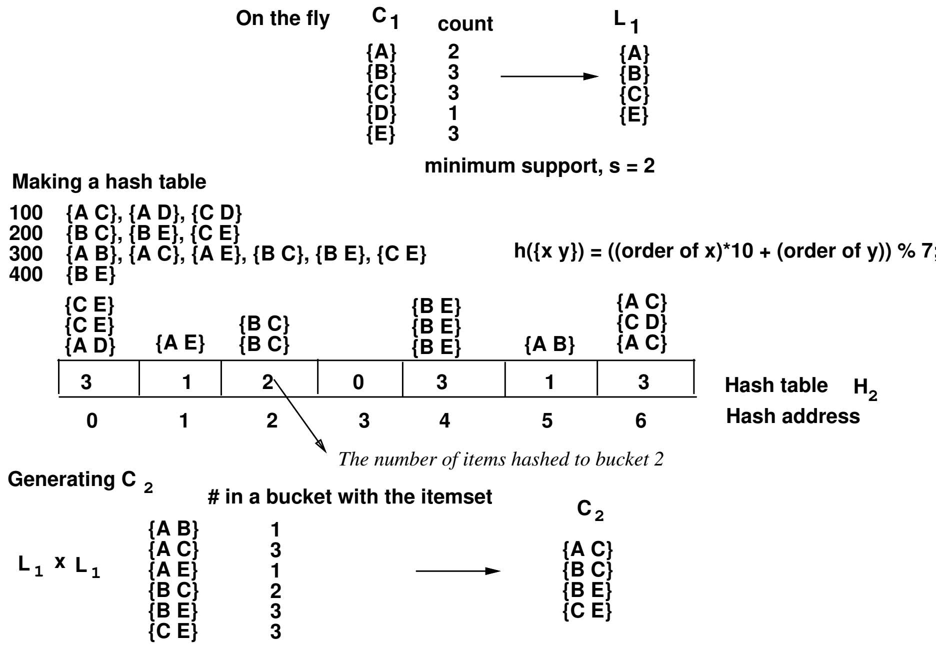

Figure 5: Example of a hash table and generation of C_2

contains some large $(k+1)$-itemsets, any item contained in these $(k+1)$-itemsets will appear in at least k of the candidate k-itemsets in C_k. As a result, an item in transaction t can be trimmed if it does not appear in at least k of the candidate k-itemsets in t. This concept is used in Procedure count_support to trim the transaction size. Certainly, the above is only a necessary condition, not a sufficient condition, for an item to appear in a candidate $(k+1)$-itemset. In Procedure make_hasht, we further check that each item in a transaction is indeed covered by a $(k+1)$-itemset (of the transaction) with all $(k+1)$ of its k-itemsets contained in C_k.

An example to trim and reduce transactions is given in Figure 6. Note that the support of a k-itemset is increased as long as it is a subset of transaction t and also a member of C_k. As described by Procedure count_support, $a[i]$ is used to keep the occurrence frequency of the i-th item of transaction t. When a k-subset containing the i-th item is a member of C_k, we increase $a[i]$ by one according to the index of each item in the k-subset (e.g., in transaction 100, $a[0]$ corresponds to A, $a[1]$ corresponds to C, and $a[2]$ corresponds to D). Then, in Procedure make_hasht, before hashing of a $(k+1)$-subset of transaction $\hat{t}$, we test all the k-subsets of $\hat{t}$ by checking the values of the corresponding buckets on the hash table H_k. To reduce the transaction size, we then check each item $\hat{t}_i$ in $\hat{t}$ to see if item $\hat{t}_i$ is indeed included in one of the $(k+1)$-itemsets eligible for hashing from $\hat{t}$. Item $\hat{t}_i$ is discarded if it does not meet such a requirement.

For example, in Figure 6, transaction 100 has only a single candidate itemset AC. Then, oc-

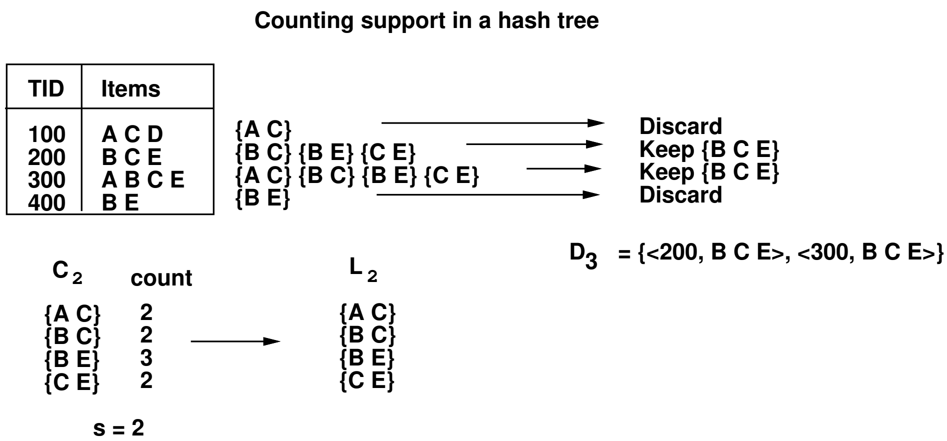

Figure 6: Example of L_2 and D_3

currence frequencies of all the items are: $a[0] = 1$, $a[1] = 1$, and $a[2] = 0$. Since all the values of $a[i]$ are less than 2, this transaction is deemed not useful for generating large 3-itemsets and thus discarded. On the other hand, transaction 300 in Figure 6 has four candidate 2-items and the occurrence frequencies of items are $a[0]$=1 (corresponding to A), $a[1]$=2 (corresponding to B), $a[2]$=2 (corresponding to C), and $a[3]$=2 (corresponding to E). Thus, it keeps three items B, C, E and discards item A.

For another example, if transaction $t = ABCDEF$ and five 2-subsets, (AC, AE, AF, CD, EF), exist in a hash tree built by C_2, we get values of array $a[i]$ as $a[0] = 3$, $a[2] = 2$, $a[3] = 1$, $a[4] = 2$, and $a[5] = 2$ according to the occurrences of each item. For large 3-itemsets, four items, A, C, E, and F, have a count larger than 2. Thus, we keep the items A, C, E, and F as transaction $\hat{t}$ in Procedure count_support and discard items, B and D since they are useless in later passes. Clearly, not all items in $\hat{t}$ contribute to the later large itemset generations. C in fact does not belong to any large 3-itemset since only AC and CD, but not AD, are large 2-itemsets. It can be seen from Procedure make_hasht that spurious items like C are removed from $\ddot{t}$ in the reduced database D_3 for next large itemsets. Consequently, during the transaction scan, many transactions are either trimmed or removed, and only transactions which consist of necessary items for later large itemset generation are kept in D_{k+1}, thus progressively reducing the transaction database size. The fact that D_k decreases significantly along the pass number k is the very reason that DHP achieves a shorter execution time than Apriori even in later iterations when the same procedure for large itemset generation is used by both algorithms. Figure 6 shows an example of L_2 and D_3.

4 Experimental Results

To assess the performance of DHP, we conducted several experiments on large itemset generations by using an RS/6000 workstation with model 560. As will be shown later, the techniques of using a hash table and progressively reducing the database size enable DHP to generate large itemsets efficiently. The methods used to generate synthetic data are described in Section 4.1. The effect of the hash table size used by DHP is discussed in Section 4.2. Comparison of DHP and Apriori algorithms is given in Section 4.3. Results on some scale-up experiments are presented in Section 4.4.

4.1 Generation of Synthetic Data

The method used by this study to generate synthetic transactions is similar to the one used in [5] with some modifications noted below. Table 1 summarizes the meaning of various parameters used in our experiments. Each transaction consists of a series of potentially large itemsets, where those itemsets are chosen from a set of such itemsets W. $|W|$ is set to 2000. The size of each potentially large itemset in W is determined from a Poisson distribution with mean equal to $|I|$. Itemsets in W are generated as follows. Items in the first itemset are chosen randomly from N items. In order to have common items in the subsequent S_q itemsets, in each of these S_q itemsets, some fraction of items are chosen from the first itemset generated, and the other items are picked randomly. Such a fraction, called the correlation level, is chosen from an exponentially distribution with mean equal to 0.5. The number of subsequent itemsets that are correlated to the first itemset, i.e., S_q, is chosen to be a number between 4 and 6 in this study. After the first $(S_q + 1)$ itemsets are generated, the generation process resumes a new cycle. That is, items in the next itemset are chosen randomly, and the subsequent S_q itemsets are so determined as to be correlated to that itemset in the way described above. The generation process repeats until $|W|$ itemsets are generated. It is noted that in [5], the generation of one itemset is only dependent on the previous itemset. Clearly, a larger S_q incurs a larger degree of "similarity" among transactions generated. Here, a larger S_q is used so as to have more large itemset generation scenarios to observe in later iterations. As such, we are also able to have $|W|/(S_q + 1)$ groups of transactions to model grouping or clustering in the retailing environment.

Each potentially large itemset in W has a weight, which is the probability that this itemset will be picked. Each weight is exponentially distributed and then normalized in such a way that the sum of all the weights is equal to one. To obtain each transaction from a series of potentially large itemsets, we used a data pool that contains between 30 and 80 potentially large itemsets. When a potentially large itemset is chosen from W and inserted into the pool, this itemset is assigned with a number which is obtained from multiplying the weight of this itemset by a factor M_f. The factor

Table 1: Meaning of various parameters

D_k	Set of transactions for large k-itemsets
$\lvert D_k \rvert$	The number of transactions in D_k
H_k	Hash table containing $\lvert H_k \rvert$ buckets for C_k
C_k	Set of candidate k-itemsets
L_k	Set of large k-itemsets
$\lvert T \rvert$	Average size of the transactions
$\lvert I \rvert$	Average size of the maximal potentially large itemsets
$\lvert L \rvert$	Number of maximal potentially large itemsets
N	Number of items

M_f is set to a value between $|D_1|/40$ and $|D_1|/80$. When a potentially large itemset is assigned to a transaction, such a number of this itemset is decreased by one. If the number of an itemset becomes 0, another potentially large itemset is randomly chosen from W and inserted into the pool. We therefore control the numbers of large k-itemsets by M_f and S_q. Same as [5], we also use a corruption level during the transaction generation to model the phenomenon that all the items in a large itemset are not always bought together. Each transaction is stored in a file system with the form of <transaction identifier, the number of items, items>.

4.2 Effect of the Size of a Hash Table

Note that the hash table size used by DHP affects the cardinality of C_2 generated. In fact, from the process of trimming the database size described in Section 3.2, it can be seen that the size of C_2 subsequently affects the determination of D_3. Table 2 shows the results from varying the hash table size, where $|D_1|$ = 100,000, $|T| = 10$, $|I| = 4$, $N = 1000$, $|L| = 2000$, and $s = 0.75\%$. We use Tx.Iy.Dz to mean that $x = |T|$, $y = |I|$, and the number of transactions in D_1 is $z \times 1000$. For notational simplicity, we use $\{H_2 \geq s\}$ in Table 2 to represent the set of buckets which are hashed into by 750 (i.e., 100*1000*0.75%) or more 2-itemsets during the execution of Part 1 in DHP, and $|\{H_2 \geq s\}|$ to denote the cardinality of $\{H_2 \geq s\}$. Also, α represents the ratio of the number of 2-itemsets hashed into $\{H_2 \geq s\}$ to the total number of 2-itemsets generated from the original database D_1. α is a factor to represent what fraction of 2-itemsets are involved in candidate 2-itemsets. The total number of 2-itemsets of the experiment at Table 2 is 5,002,249, which is slightly larger than $|D_1| \cdot \binom{|T|}{2}$.

For N = 1000, the number of distinct 2-itemsets is $\binom{N}{2}$. Note that when $n = \binom{N}{2}$ and $|H_2|$ is chosen to be the exponent of 2 which is greater than n, we have $|C_2|/|L_2|$=1.80 in Table 2. In addition, note that in the second column of Table 2, the database to be used for generating large

Table 2: Results from varying hash table sizes (T10.I4.D100)

$\lvert H_2 \rvert$	524,288	262,144	131,072	95,536	32,768	16,384
L_1	559	559	559	559	559	559
$\lvert\{H_2 \geq s\}\rvert$	58	61	75	96	182	435
C_2	81	120	199	394	1355	5602
L_2	45	45	45	45	45	45
α	0.0314	0.0320	0.0345	0.0386	0.0545	0.1027
size of D_3	498KB	500KB	507KB	539KB	603KB	888KB
$\lvert D_3 \rvert$	19,732	19,741	19,755	20,501	21,607	29,227
total time	6.44	6.43	6.24	6.77	7.23	8.35

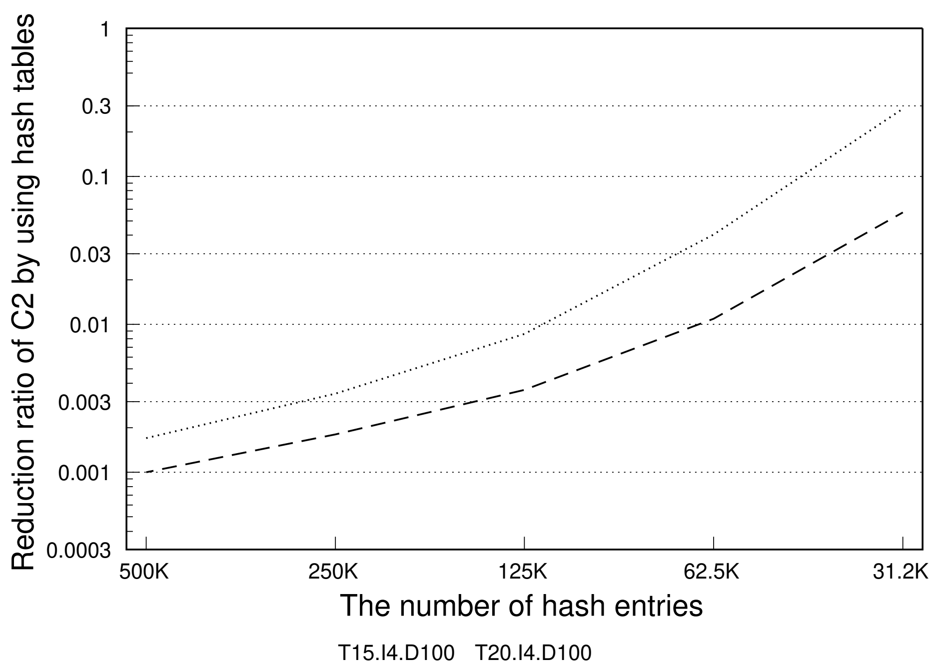

Figure 7: Reduction ratio of $|C_2|$ by DHP for different sizes of H_2

3-itemsets, i.e., D_3, is very small compared to the original database, indicating a very effective trimming in the database size by DHP. Specifically, the ratio of D_3 to D_1 is 10.55% in terms of their sizes, and 19.73% in terms of their numbers of transactions. Also, the average number of items in each transaction of D_3 is 4.33 (instead of 10 in the original database). Clearly, a small number of items in each transaction incurs fewer comparisons in a hash tree and leads to a shorter execution time.

In Table 2, the values of $|H_2|$ in DHP varies from $\binom{N}{2}$ to $\binom{N}{2}/32$. When $|H_2|$ is approximately $\binom{N}{2}$ as in the first column, $|C_2|/|L_2| = 1.80$, meaning that a larger H_2 leads to a smaller C_2 at the cost of using more memory. As $|H_2|$ decreases, $|C_2|$ and the execution time for L_2 increase. The size of D_3 then increases as well. We found that we can have fairly good overall performance till $|H_2|$ is a quarter of $\binom{N}{2}$ (i.e., till the fourth column). However, it is noted that even when the hash table has only $\binom{N}{2}/32$ buckets, the number of candidate 2-itemsets is still significantly smaller than $\binom{|L_1|}{2}$. Clearly, when either the minimum support is small or the number of total 2-itemsets is large, it is advantageous to use a large $|H_2|$ for DHP. The reduction ratios of $|C_2|$ by DHP for various sizes of H_2 are shown in Figure 7, where a logarithmic scale is used in y-axis for ease of presentation.

4.3 Comparison of DHP and Apriori

Table 3 shows the relative performance between Apriori used in [5] and DHP. Here, we use $|T| = 15$, i.e., each transaction has 15 items in average, so as to have more large itemsets in later passes for interest of presentation. The execution times of these two algorithms are shown in Figure 8. In DHP, $|H_2|$ is chosen to be the exponent of 2 which is greater than $\binom{N}{2}$ (i.e., with 524,288 buckets). In this experiment, DHP uses a hash table for the generation of C_2 (i.e., Part 2 of Figure 3). Starting from the third pass, DHP is the same as Apriori in that the same procedure for generating large itemsets (i.e., Part 3 of Figure 3) is used by both algorithms, but different from the latter in that a smaller transaction database is scanned by DHP. The last column represents the database size in the k^{th} pass (i.e., D_k) used by DHP and its cardinality (i.e., $|D_k|$). More explicitly, Apriori scans the full database D_1 for every pass, whereas DHP only scans the full database for the first 2 passes and then scans the reduced database D_k thereafter. As mentioned before, in [5] the hybrid algorithm has the option of switching from Apriori to another algorithm AprioriTID after early passes for better performance, and such an option is not adopted here. It can, nevertheless, be seen from Figure 8 that the execution time of the first two passes by Apriori is larger than the total execution time by DHP[6].

It can be seen from Table 3 that the execution time of the first pass of DHP is slightly larger

[6]The benefit of AprioriTID in later passes is complementary to the focus of DHP on initial passes. In fact, in Part 3 of DHP, AprioriTID can be used instead of Apriori if desired.

Table 3: Comparison of execution time (T15.I4.D100)

	Apriori	DHP	
	number	number	D_k, $\lvert D_k \rvert$
L_1	820	820	6,700KB, 100,000
C_2	335,790	338	6,700KB, 100,000
L_2	207	207	
C_3	618	618	659KB, 20,602
L_3	201	201	
C_4	184	184	546KB, 17,417
L_4	98	98	
C_5	30	30	332KB, 10,149
L_5	23	23	
C_6	1	1	24KB, 756
L_6	1	1	
total time	39.39	13.91	

than that of Apriori due to the extra overhead required for generating H_2. However, DHP incurs significantly smaller execution times than Apriori in later passes, not only in the second pass when a hash table is used by DHP to facilitate the generation of C_2, but also in later passes when the same procedure is used, showing the advantage of scanning smaller databases by DHP. Here, the execution time of the first two passes by Apriori is about 65% of the total execution time. This is the very motivation of employing DHP for ealry passes to achieve performance improvement. The execution times for the first three passes by the two algorithms are compared in Figure 9.

Figure 10 shows the execution time ratios of DHP to Apriori over various minimum supports, ranging from 0.75% to 1.25%. Figure 10 indicates that DHP constantly performs well for various minimum supports. Figures 11 and 12 show the effect of progressively reducing the transaction database by DHP. As pointed out earlier, this very feature of DHP is made feasible in practice due to the early reduction on the size of C_2, and turns out to be very powerful to facilitate the later itemset generations. As shown in Figure 11 where a logarithmic scale is used in y-axis, the number of transactions in the database to be scanned by DHP progressively decreases due to the elimination of transactions which are deemed useless for later large itemset generations. Note that DHP is not only reducing the number of transactions but also trimming the items in each transaction. It can be seen that the average number of items is, on one hand, reduced by the latter process (i.e., trimming each transaction size), but on the other hand, is increased by the former process (i.e., reducing the number of transactions) since transactions eliminated are usually small ones. As a result of these two conflicting factors, as shown in Case A of Figure 12 whose y-axis uses a logarithmic scale, the average number of items in each transaction in D_i remains approximately the same along the pass

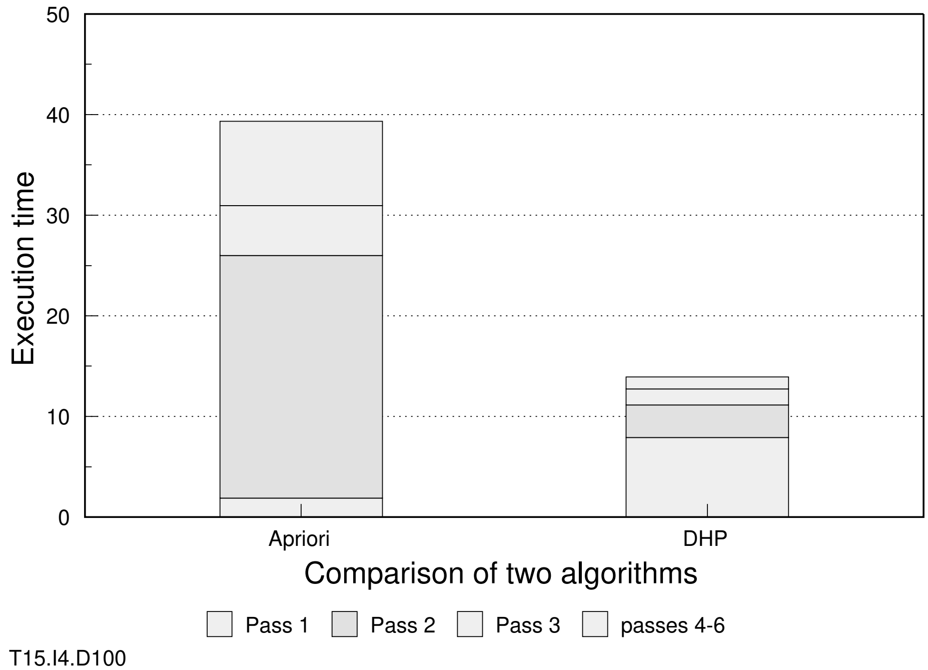

Figure 8: Execution time of Apriori and DHP

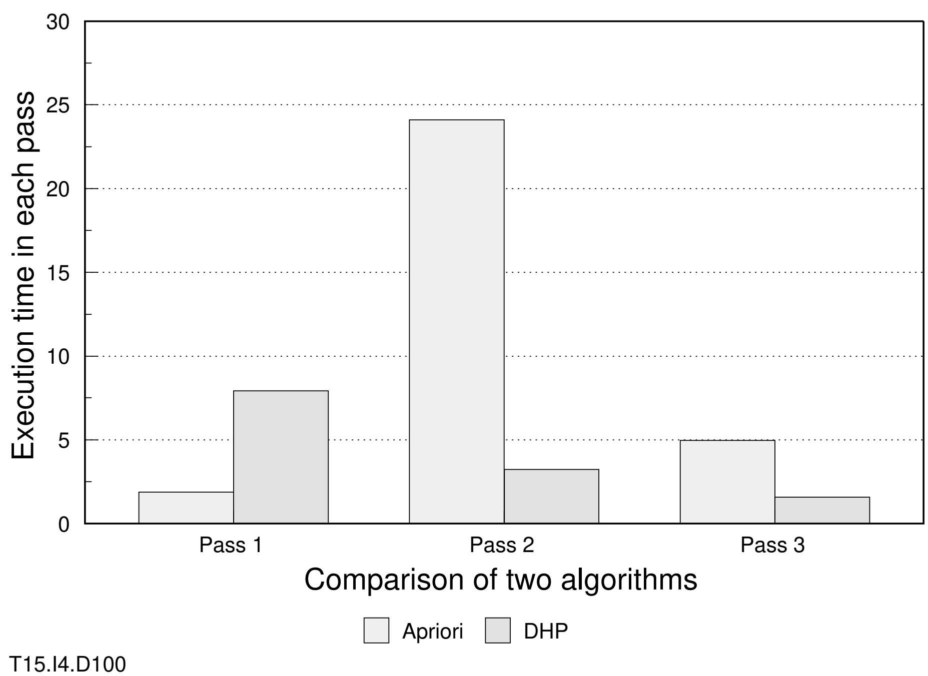

Figure 9: Execution times of the first three passes by Apriori and DHP

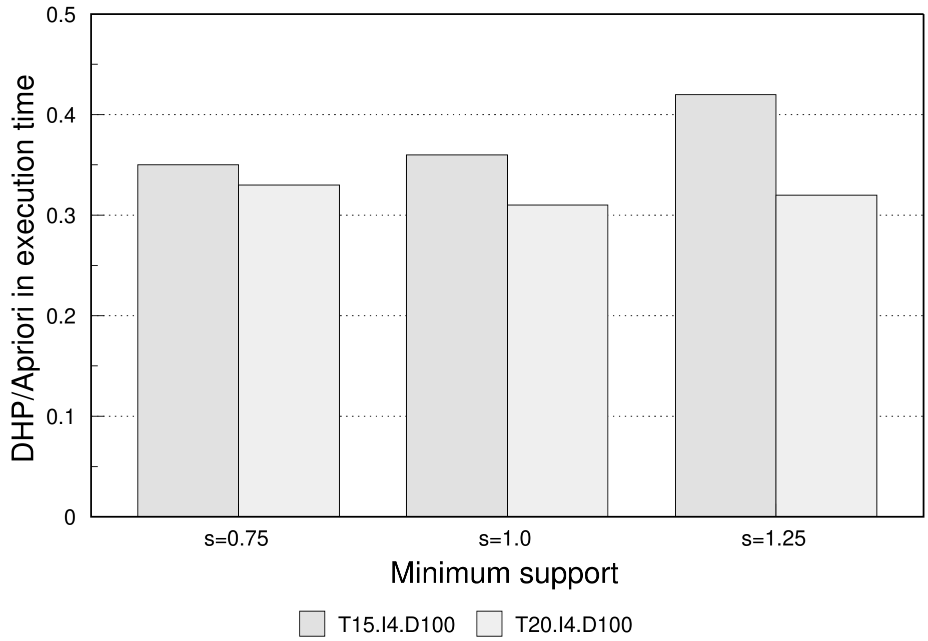

Figure 10: Execution time comparison between DHP and Apriori for some minimal supports

number i. For example, for T20.I4.D100, starting from 20 items, the average number of items in a transaction drops to 7.5, and then increases slightly since several small transactions are eliminated in later passes. To explicitly show the effect of trimming each transaction size, we conducted another experiment where transactions are only trimmed but not thrown away along the process. The average number of items in each transaction resulting from this experiment is shown by Case B of Figure 12, which indicates that the trimming method employed by DHP is very effective.

4.4 Scale-Up Experiment for DHP

Figure 13 shows that the execution time of DHP increases linearly as the database size increases, meaning that DHP possesses the same important feature as Apriori. Also, we examine the performance of DHP as the number of items, N, increases. Table 4 shows the execution times of DHP when the number of items increase from 1,000 to 10,000 for three data sets T5.I2.D100, T10.I4.D100, and T20.I6.D100. In the experiments for Table 4, the minimum support is 0.75% and the hash table size is the exponent of 2 which is greater than $\binom{1000}{2}$. Note that the portion of time on determining L_1 for the case of small transactions (e.g. T5 in Table 4) is relatively larger than that for the case of large transactions (e.g. T20 in Table 4). In other words, a large transaction has a larger likelihood of having large itemsets to process than a small transaction. Also, given a fixed minimum support, when the number of items N increases, the execution time

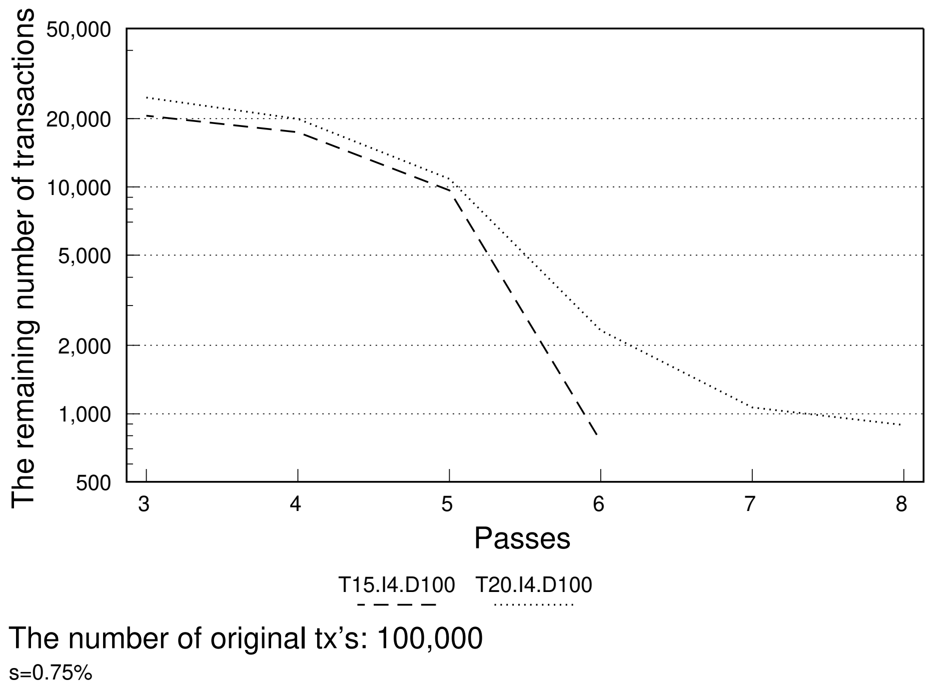

Figure 11: The remaining number of transactions in each pass

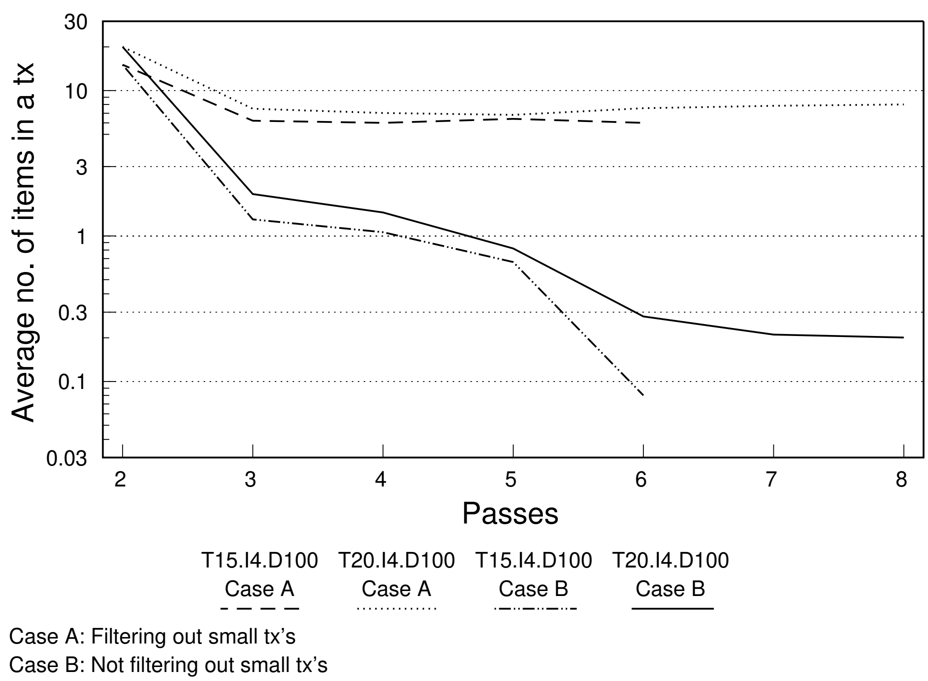

Figure 12: The average number of items in a transaction in each pass

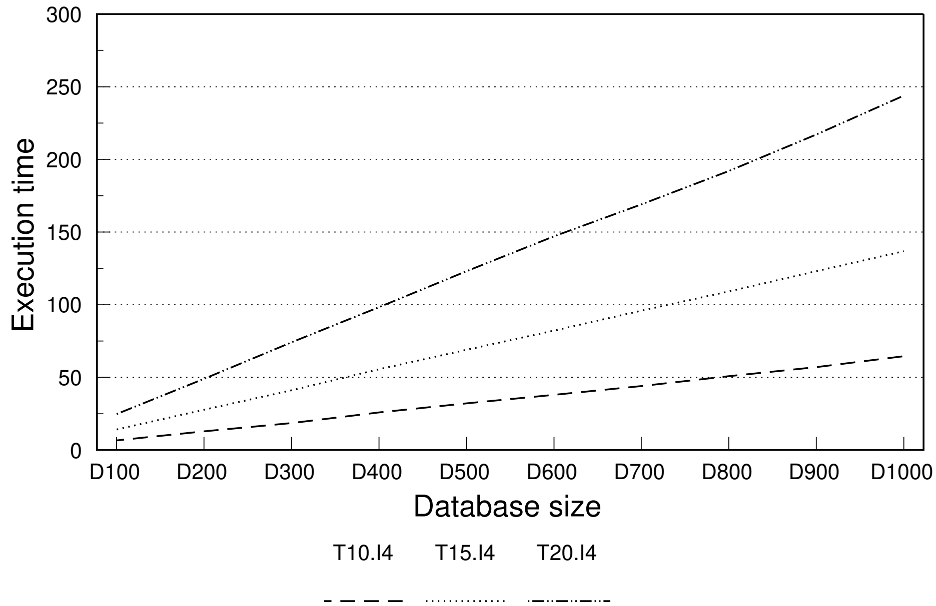

Figure 13: Performance of DHP when the database size increases

Table 4: Performance of DHP when the number of items increases

N	T5.I2	T10.I4	T20.I6
1,000	2.26	6.69	20.44
2,500	2.46	6.88	21.42
5,000	2.59	7.57	23.47
7,500	2.68	7.53	23.95
10,000	2.64	7.91	23.75

to obtain L_1 increases since the size of L_1 is usually close to N, but the execution time to obtain larger k-itemsets decreases since the support for an itemset is averaged out by more items and thus decreases. Consequently, as shown in Table 4, when N increases, execution times for small transactions increase a little more prominently than those for large transactions.

5 Conclusions

We examined in this paper the issue of mining association rules among items in a large database of sales transactions. The problem of discovering large itemsets was solved by constructing a candidate set of itemsets first and then, identifying, within this candidate set, those itemsets that meet the large itemset requirement. We proposed an effective algorithm DHP for the initial candidate set generation. DHP is a hash based algorithm and is especially effective for the generation of candidate set for large 2-itemsets, where the number of candidate 2-itemsets generated is, in orders of magnitude, smaller than that by previous methods, thus resolving the performance bottleneck. In addition, the generation of smaller candidate sets enables us to effectively trim the transaction database at a much earlier stage of the iterations, thereby reducing the computational cost for later stages significantly. Extensive simulation study has been conducted to evaluate performance of the proposed algorithm.

References

[1] R. Agrawal, C. Faloutsos, and A. Swami. Efficient Similarity Search in Sequence Databases. *Proceedings of the 4th Intl. conf. on Foundations of Data Organization and Algorithms*, October, 1993.

[2] R. Agrawal, S. Ghosh, T. Imielinski, B. Iyer, and A. Swami. An Interval Classifier for Database Mining Applications. *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 560–573, August 1992.

[3] R. Agrawal, T. Imielinski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. *Proceedings of ACM SIGMOD*, pages 207–216, May 1993.

[4] R. Agrawal and R. Srikant. Mining Sequential Patterns. *Proceedings of the 11th International Conference on Data Engineering*, March 1995.

[5] R. Agrawal and S. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. *Proceedings of the 20th International Conference on Very Large Data Bases*, September 1994.

[6] T.M. Anwar, H.W. Beck, and S.B. Navathe. Knowledge Mining by Imprecise Querying: A Classification-Based Approach. *Proceedings of the 8th International Conference on Data Engineering*, February 1992.

[7] J. Han, Y. Cai, , and N. Cercone. Knowledge Discovery in Databases: An Attribute-Oriented Approach. *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 547–559, August 1992.

[8] M. Houtsma and A. Swami. Set-Oriented Mining of Association Rules. Technical Report RJ 9567, IBM Almaden Research Laboratory, San Jose, CA, October 1993.

[9] E. G. Coffman Jr. and J. Eve. File structures using hashing functions. *Comm. of the ACM*, 13(7):427–432, 436, July 1970.

[10] R.T. Ng and J. Han. Efficient and Effective Clustering Methods for Spatial Data Mining. *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 144–155, September 1994.

[11] G. Piatetsky-Shapiro. Discovery, Analysis and Presentation of Strong Rules. *Knowledge Discovery in Databases*, 1991.

[12] J.R. Quinlan. Induction of Decision Trees. *Machine Learning*, 1:81–106, 1986.