Go Green: Recycle and Reuse Frequent Patterns

Gao Cong[1] Beng Chin Ooi[1,2] Kian-Lee Tan[1,2] Anthony K. H. Tung[1,3]
[1]Department of Computer Science
National University of Singapore
3 Science Dr 2, Singapore 117543
[2]Singapore-MIT-Alliance
National University of Singapore
Singapore 117576

[3]Contact Author: atung@comp.nus.edu.sg

Abstract

In constrained data mining, users can specify constraints to prune the search space to avoid mining uninteresting knowledge. This is typically done by specifying some initial values of the constraints that are subsequently refined iteratively until satisfactory results are obtained. Existing mining schemes treat each iteration as a distinct mining process, and fail to exploit the information generated between iterations. In this paper, we propose to salvage knowledge that is discovered from an earlier iteration of mining to enhance subsequent rounds of mining. In particular, we look at how frequent patterns can be recycled. Our proposed strategy operates in two phases. In the first phase, frequent patterns obtained from an early iteration are used to compress a database. In the second phase, subsequent mining processes operate on the compressed database. We propose two compression strategies and adapt three existing frequent pattern mining techniques to exploit the compressed database. Results from our extensive experimental study show that our proposed recycling algorithms outperform their non-recycling counterpart by an order of magnitude.

1 Introduction

Mining frequent patterns or itemsets is a fundamental and essential problem in many data mining applications [9]. Recently, many constraints (apart from the traditional support) were introduced into frequent pattern mining to give the user more freedom to express his/her preferences [18, 12, 14]. On one hand, these additional constraints can restrict the search to find only those relevant patterns. On the other hand, it often prolongs the mining process because the user may want to see the results of various combinations of constraint changes by running the mining algorithm many times. For example, consider a frequent pattern mining task with only the minimum support constraint. A user may initially set the minimum support to 5% and run a mining algorithm. After inspecting the returned results, s/he may find that 5% is too high, and rerun the algorithm with a reduced minimum support of 3%. This process is repeated several times until s/he is satisfied with the final mining results.

Existing techniques provide very little support for interactive and iterative mining process. In fact, the dataset usually has to be mined from scratch in each iteration. This is clearly inefficient because a large portion of the computation from an early iteration is repeated in the new mining process. Thus, it is critical to design mechanisms to exploit and recycle frequent patterns that have been mined in earlier iterations.

It is straightforward to obtain the new set of frequent patterns when constraints are tightened from those in a previous round of mining (the solution space is reduced), e.g., when the minimum support is increased. To obtain the new set of frequent patterns under the new constraints, we can simply check the frequent patterns from the early mining step to filter out patterns that do not satisfy the new constraints. This filtering process is sufficient because the set of new frequent patterns is only a subset of the old set. When constraints are relaxed (the solution space is expanded), recycling previous frequent patterns becomes non-trivial as re-running the mining algorithm is needed to find those additional frequent patterns. For instance, when minimum support is decreased, more patterns may be generated.

In this paper, we propose a novel technique to recycle frequent patterns to speed up subsequent frequent pattern mining. Our scheme comprises two phases. In the first phase, we use the frequent patterns from an early iteration of mining to compress the database. In the second phase, we mine the compressed database. The compression here aims to speed up subsequent

mining by utilizing the knowledge encapsulated in previous frequent patterns, rather than to save space although it does. We design two compression strategies. While the first attempts to minimize cost, the second minimizes storage space. The strategy of minimizing cost is novel in that we design a function to estimate the potential saving of using a pattern to do the compression for subsequent mining. The strategy of minimizing storage space is relatively straightforward. We also propose a naive mining algorithm that operates on the compressed database using the projected database technique. We show how the naive algorithm can be combined with algorithms that use the projected database as the underlying framework. In this paper, we adapted the H-Mine [15], FP-tree [10] and Tree Projection [4].

We conducted extensive experiments to study the performance of our recycling technique. Our experimental results show that our proposed recycling algorithms outperform their non-recycling counterparts by an order of magnitude. Our study also shows that the compression strategy that minimizes cost is more effective than the compression strategy which minimizes storage space. Another interesting finding is that the saving of recycling algorithms over non-recycling counterparts is much greater than the time that is used to mine the set of frequent patterns for recycling.

The rest of the paper is organized as follows. In Section 2, we present the problem of recycling patterns. Section 3 presents the compression techniques and how to apply the projected database techniques to mine the compressed database. In Section 4, we show how existing frequent mining algorithm can be adapted to mine a compressed database. Section 5 presents performance studies. We review some related works in Section 6, and finally, we conclude in Section 7.

2 Problem Statement

Let $I = \{i_1, i_2, ..., i_n\}$ be a set of **items** which represent attribute values in a database DB. A pattern (or itemset) X is a non-empty subset of I. Given DB, the **support** of a pattern X, denoted as $sup(X)$, is the number of tuples in DB which contains X. Given a *minimum support threshold* ξ, the problem of **frequent pattern mining** is to find the complete set of frequent patterns whose supports are greater than ξ. Given a database DB and a set of constraints C (including *minimum support threshold* ξ), the problem of **constrained frequent pattern mining** is to find the complete set of frequent patterns that satisfy C.

Minimum support threshold is an essential constraint in *frequent pattern mining*. There are many types of constraints that can be imposed on frequent pattern mining. Four categories of constraints - anti-monotone, monotone, succinct, and convertible constraints - have been effectively integrated into some mining algorithms [12, 14].

As discussed before, a typical data mining application is an iterative process. A user often runs a mining algorithm many times, each with more refined constraints. Such an iterative process provides the opportunity to recycle frequent patterns obtained in early iterations. Moreover, when there are many users in a data mining system, the frequent patterns discovered by one user also provide opportunity for the others to recycle.

Recycling frequent patterns: Given a database DB and a set of constraints C, the problem of *recycling frequent patterns* is to find the complete set of frequent patterns with the help of the set of frequent patterns, denoted as FP, discovered at a set of constraints C_{old}.

Compared with C_{old}, the set of constraints C might be tightened (e.g., the *minimum support* is increased), or relaxed (e.g., the *minimum support* is decreased). When constraints are tightened from C_{old}, the new set of frequent pattern can be filtered from the old set easily. The challenge comes when constraints are relaxed. The new set of frequent patterns cannot be obtained from the old ones. Our main approach to recycling previous patterns is to carefully select a set of frequent patterns from an early iteration and compress the data to be mined using these patterns. The selection criteria take into account the estimated saving that could occur when the database is compressed with a particular pattern. We can adapt a series of algorithms using projected database as the underlying framework to mine the compressed database.

We note that many frequent pattern discovery algorithms have been developed [5, 6, 10, 15, 11] and it is not our intention to develop yet another efficient algorithm for finding frequent patterns. *Instead, our aim here is to show that the concept of recycling patterns is useful and practical in an interactive data mining environment.* More specifically, we hope to illustrate two points: (1) Frequent patterns can be used to estimate the cost for visiting some portion of the search space that have been visited before. (2) It is possible to use such estimation to develop a mining plan such that the cost of a new round of mining is reduced.

Note that we can extend the problem statement by two cases (1) The constraints C and C_{old} are the same while a set of FP may be discovered on a database that contains more or fewer tuples than DB. This is essentially the incremental update problem. (2) Both constraints and database are changed. We should point out that our proposed technique can be applied to these two cases [20].

3 Recycling Frequent Patterns

In this section, we shall present our proposed strategy to recycle frequent patterns. We shall first look at how compression can optimize subsequent mining. We then present two compression strategies, and finally a naive algorithm to mine the compressed database.

We use the *minimum support* constraint relaxation as an example to present the proposed technique of recycling frequent patterns. Let ξ_{old} be the *minimum support* corresponding to the set of frequent patterns FP, and ξ_{new} be the current *minimum support* (relaxed from ξ_{old}). Recycling with other constraint changes can be similarly addressed as discussed in our technical report [20]. We use projected database concept (explain later) as the underlying mining framework of our proposed technique. Algorithms based on projected database concept include Tree Projection, FP-tree, H-Mine and their variations [16, 11].

3.1 Recycling frequent patterns via compression

We first illustrate how compression can be used to speed up the mining of frequent patterns with an example. The following three definitions will be used in the example.

Definition 3.1 Frequent List
Given a database DB, a *frequent list* is a list in which frequent items in the database are ordered in support ascending order. We denote frequent list as *F-list*. □

For example, with $\xi_{new} = 2$, the *F-list* of the database DB in Table 1 is $< d : 2, f : 3, g : 3, a : 3, e : 4, c : 4 >$, where the number after ":" indicates the support of the item.

Definition 3.2 Projected Database
Consider a database DB and its F-list. Let i be a frequent item in DB. The i-projected database is the subset of tuples in DB containing i, where all the occurrences of infrequent items, item i and items before i (i.e., lower support values) in the F-list are omitted. We denote the i-projected database as $PROJ_i$ □

For instance, the a-projected database in Table 1 is $< 100 : ec, 400 : ec, 500 : e >$ where ":" separates the tuple ID and tuple.

Definition 3.3 Candidate Extension
Consider a (projected) database DB and its *F-list*. Let i be an item in *F-list*. The candidate extensions of i (or the corresponding pattern of i) in DB are defined to be the items following i in the F-list. We denote candidate extensions of i as C_i □

ID	Items
100	a,c,d,e, f,g
200	b,c,d,f,g
300	c,e,f,g
400	a,c,e,i
500	a,e,h

Table 1. The example database *DB*.

Example 1 For the database in Table 1, the set of frequent patterns under $\xi_{old} = 3$ is $FP = \{f : 3, fg : 3, fgc : 3, g : 3, gc : 3, a : 3, ae : 3, e : 4, ec : 3, c; 4\}$. Table 2 is the corresponding compressed database using the set FP (we will explain how to get the compressed database shortly). The outlying items are the remaining items in each tuple after compression.

With $\xi_{new} = 2$, the fourth column in Table 2 is obtained by ranking the left items according to *F-list* after removing the infrequent items (not in *F-list*) in the third column of Table 2. We observe that compression can help to save computation in two ways.

First, computation can be saved when counting the support of a pattern. When we mine frequent patterns extended from item f (it is in group fgc), we do not need to scan the items in the group fgc (in the uncompressed database, we have to scan them tuple by tuple). Instead we can utilize the group count (here it is 3) to compute the frequent items in f-projected database. When mining frequent patterns extended from d (it does not belong to any group), we associate group fgc with a counter when scanning d-projected database, and then add the counter value (here it is 2) to the counter of each item in fgc. In this way, we require less computation to mine the compressed database than the uncompressed one. The saving is significant in practice where each group contains a large number of tuples. Similarly, it requires less computation to construct *F-list* by scanning Table 2 (the compressed database) instead of Table 1 (the uncompressed database).

Second, computation can be saved when constructing a projected database. Consider the construction of g-projected database. We can know all tuples of group fgc belonging to the g-projected database by checking fgc once. Again, if we were to operate on the uncompressed database, we have to scan every tuple in Table 1. For group ae, we need to scan ae only once and then scan the outlying items in the group. □

In summary, the new round of mining can benefit from the compression using patterns from the previous round of mining as follows. Computation can be saved when we count supports for candidate extensions of a pattern P in a (projected) database. As shown

Group	ID	Outlying items	(Ordered) Frequent Outlying Items
fgc	100	a,d,e	d,a,e
	200	b,d	d
	300	e	e
ae	400	c,i	c
	500	h	

Table 2. The compressed database CDB.

in the above example, not only items in some groups but also items not in any group can benefit when we compute the supports of candidate extensions. Computation can also be saved when we construct the projected database. Constructing projected databases and computing supports take the main computation in frequent pattern mining algorithms that employ projected database as the underlying framework. The saving from the two aspects can greatly improve mining efficiency as we shall see later in our experimental study.

3.2 Compression strategies

We now have some intuition on how compression can help in mining frequent patterns. The remaining problem is determine good strategies to compress a database given a set of frequent patterns FP. The basic framework for the compression works as follows (see Figure 1. In step 1, we determine the utility of each frequent pattern. We shall discuss the utility functions used shortly. In step 2, the patterns are ordered in descending order of their utility values. For each tuple in the database, we then select a pattern to compress it (Steps 3-5). Note that we leave a tuple as it is when it has no matching pattern. The pattern picked is the one with the highest utility.

To estimate the potential savings for subsequent mining if we use the pattern for compression, we design two functions to compute the utility of each frequent pattern X as follows:

Strategy 1: Minimize Cost Principle (MCP) The utility function is $U(X) = (2^{|X|} - 1) * X.C$, where $X.C$ is the number of tuples that contain pattern X.

MCP assumes that the potential savings of pattern X for subsequent mining can be estimated by the cost of visiting the search space to generate the pattern X at ξ_{old}. The assumption is reasonable since the larger the cost used in the old mining to discover X, the larger the potential savings can be derived from using X for the compression. The remaining problem is how to estimate the amount of processing that must be done in order to discover X at ξ_{old}. Since all subsets

Compression Algorithm
Procedure CompressDB(Database: DB, set of frequent patterns: FP)

(1) Compute the utility values of all patterns in FP;
(2) Sort patterns in FP according to the descending order of utility values;
(3) **for each** tuple t in DB **do**
(4) **for each** pattern X in FP **do**
(5) **if** tuple t contains pattern X **then**
Use X to compress t, **break**;

Figure 1. The Compression Algorithm

of X are also frequent patterns in this case and their support are at least X.C, the amount of processing to discover X can be estimated to be $(2^{|X|} - 1) * X.C$. This represents the potential savings for subsequent mining if a tuple is covered with the pattern.

Strategy 2: Maximal Length Principle (MLP) The utility function is $U(X) = |X| * |DB| + X.C$.

MLP aims to cover each tuple with the longest pattern. Among the patterns with the same maximal length, MLP will choose the pattern with the highest support to do compression. The first part of the utility function, i.e., the product of the pattern length and its frequency of occurrences, ensures that longer patterns always have larger utility values than shorter ones. The second part, i.e., the frequency of occurrences of X in database, ensures that among patterns with the same length, patterns with larger frequency have larger utility values.

The two utility functions essentially give rise to two different compression strategies that we will study later.

Example 2 *Here, we see how the compressed database in Table 2 is obtained from Table 1 using the MCP strategy. We compute the utility of patterns in FP (e.g. the utility value of $fgc:3$ is $(2^3 - 1) * 3 = 21$) and sort them in descending utility value. We get $\{fgc:21, fg:9, gc:9, ae:9, ec:9, e:4, c;4, f:3, g:3, a:3\}$ (the number after ":" is the utility value). First, we find that tuple 100 contains pattern fgc. Thus we use fgc to compress it. The same is done for tuples 200 and 300. Tuple 400 does not contain fgc, fg, and gc, but ae. We use ae to compress it. The same is done for tuple 500. Finally, the results obtained is shown in Table 2.* □

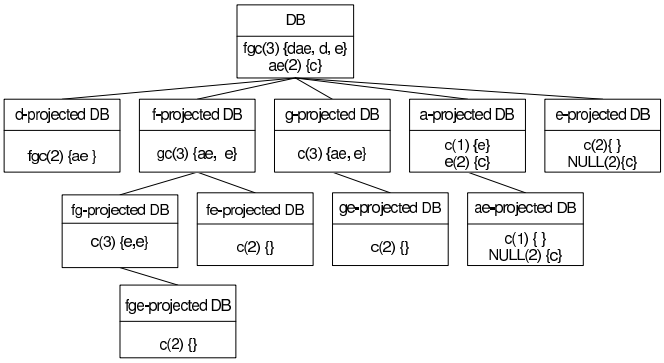

Figure 2. Mining from compressed DB

3.3 A naive algorithm for mining compressed databases

We show how to mine the compressed database using the projected databases in a naive way in this subsection. Let us illustrate our naive algorithm with an example first.

Example 3 *Figure 2 shows the mining process on the compressed database CDB in Table 2 with $\xi_{new} = 2$.*

We first find frequent items by scanning the CDB to construct the F-list. Following the order of F-list, we mine the complete search space of frequent patterns as follows (the mining process can be regarded as a depth-first traversal of all nodes of Figure 2):

(1) *Find those containing item d. The candidate extensions for d are the items after d in F-list, i.e. f, g, a, e, c. We first construct d-projected database, which is $fgc(2)\{ae\}$, where 2 registers the frequency of the group fgc in d-projected database. Each candidate extension is associated with a counter and each group is also associated with a counter. In the process of constructing projected database, we fill these counters. We then add the values of group counters to the corresponding counters for candidate extensions. We get the set of frequent items $\{f:2, g:2, c:2\}$ in d-projected database (the count of a is 1). Because all occurrences of f, g, c belong to group fgc, the frequent patterns can be generated by enumerating any combination of f, g, c, i.e. $\{dc:2, df:2, dg:2, dcf:2, dcg:2, dfg:2, dcfg:2\}$.*

(2) *Find those containing item f but not d. We first construct the f-projected database. We count the support for candidate extensions of f as in (1). The set of frequent items in f-projected database is $\{g:3, e:2, c:3\}$. Then we construct fg-projected database. The set of frequent items in fg-projected database is $\{e:2, c:3\}$. We need to construct the fge-projected database. In the step, we can get the set of frequent patterns $\{fg:3, fge:2, fgec:2, fgc:3, fe:2, fec:2, fc:3\}$.*

(3) *Find those containing g, but not f and d. The mining process is similar to (2) and is ignored here.*

(4) *Find those containing a but not g, f and d. We construct the a-projected database and get the set of frequent items $\{e:3, c:2\}$. Then we construct ae-projected database. Finally, the frequent patterns in the step are $\{ae:3, aec:2, ac:2\}$.*

(5) *Finally, the other frequent patterns are computed in a similar way and are ignored here. These include those patterns containing e but no a, g, f and d as well as those only containing c.* □

Lemma 3.1 (Single group pattern generation) Suppose that all occurrences of frequent items in a projected database is in a single group. The complete set of frequent patterns can be generated by the enumerations of all the combinations of frequent items with the count of the group as support. □

The above example assumes that the compressed database fit in memory. Although the compressed database is smaller than the original database, it is possible that it may still be too large for the available memory. In this case, the compressed database can be projected onto its set of frequent items. There are two methods for doing so. One is the partition-based projection as used in [15]. This approach projects each tuple only to its first projected database (according to item ordering). After processing the first projected database, it needs to project the first projected database to subsequent projected databases. The method is not efficient although it saves disk space. Another approach, which we adopted, is to use parallel projection to speed up the computation. This approach projects each tuple into all its projected databases. Based on the above analysis, we give the naive algorithm of recycling patterns in Figure 3. In line 1 of procedure RP-Mine(), the estimation of memory usage relies on the representation of the projected database and is discussed in [20].

4 The Mining Algorithms on Compressed database

We adapt three representative frequent pattern mining algorithms (using projected database as the underlying framework) to mine a compressed database. This section mainly introduces how to adapt H-Mine since it is the most complicated to be adapted. Due to space limitation, we only give a short introduction about adapting FP-tree and Tree Projection algorithms. Interested readers can refer to [20] for details.

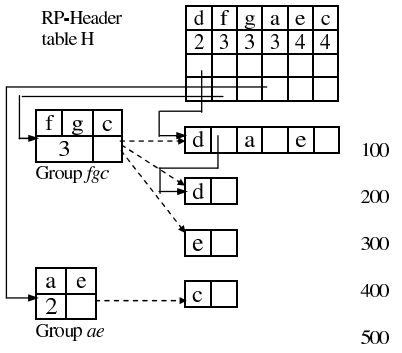

Figure 4. The Representation of Table 2 with RP-Struct

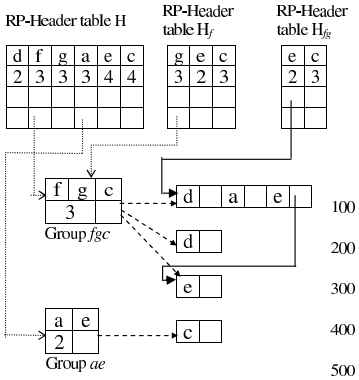

Figure 5. RP-Header tables H_f and H_{fg}

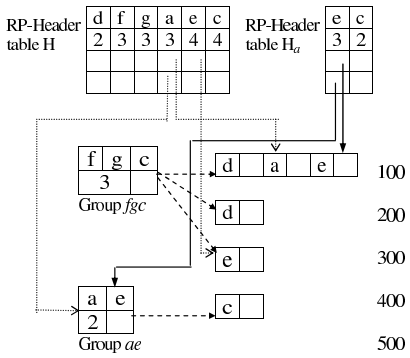

Figure 6. RP-Header table H_a

Algorithm Recycling
Input: Compressed database CDB, the support threshold ξ_{new}, and available memory M.
Output: The complete set of frequent patterns.
Method: Call Procedure RP-Mine(CDB, *null*)

Procedure RP-Mine(compressed DB: D, pattern: α)
(1) Scan D to find frequent items I_f and estimate the size of expected memory usage $EM(D)$
(2) **if** $(EM(D) > M)$ **then**
(3) Project D to items in set I_f;
(4) **for each** projected database D_i $(i \in I_f)$ **do**
(5) Generate pattern $\beta = i \cup \alpha$ with supp $= i.count$
(6) Call RP-Mine(D_i, β);
(7) **else** Count frequency of items in D & construct *F-list*
(8) Call RP-InMemory(D, *F-list*, $i \cup \alpha$);

Procedure RP-InMemory(Projected DB: $PROJ$, List :*list*, Pattern:α)
(1) **if** all occurrences of items in *list* are in a single group G in $PROJ$ **then**
(2) **for each** combination (denoted as β) of the items in the *list* **do** Generate pattern $\beta \cup \alpha$ with supp = the count of group G;
(3) **else for each** item a_i in *list* **do**
(4) Generate pattern $\beta = a_i \cup \alpha$ with supp $= a_i.count$;
(5) Construct a_i-projected database $PROJ_{a_i}$ and find the list of frequent items LF_{a_i} in $PROJ_{a_i}$;
(6) **if** $LF_{a_i} \neq null$
(7) **then** Call RP-InMemory($PROJ_{a_i}$, LF_{a_i}, β);

Figure 3. Algorithm to Recycle Patterns

4.1 Recycling using H-Mine algorithm

We use the data structure of H-Mine to represent the outlying frequent items (uncompressed part). The integration of such a data structure into recycling algorithm is non-trivial. We first use an example to illustrate how a compressed database can be mined by adapting H-Mine. Then the algorithm for frequent pattern discovery is given.

Example 4 *Consider the compressed database CDB as shown in Table 2 and $\xi_{new} = 2$. CDB can be organized as shown in Figure 4. The RP-Header table H contains the same number of items as F-list and follows the order of F-list. One compressed group fgc contains three tuples, and the other group ae contains two tuples (one tuple is null after compression).* □

The representation of the compressed database as in Figure 4 is called **RP-Struct**. It has three components:
1. Group Head: Each entry in group head consists of three fields: *group pattern*, *count*, and *tail*, where *group pattern* registers the items contained in the group, *count* registers the number of tuples in the group, and *tail* points to the tuples of the group.
2. Group Tail: It records the frequent items in the uncompressed part of each tuple. We adapt the data structure of H-Mine [15] for *group tail*s. Each frequent item is stored in an entry that contains two fields: *item-name* and *item-link*, where *item-name* registers the item the entry represents and *item-link* is used to link the same *item-name* in different group tails together.
3. RP-Header Table: Each entry in RP-Header table represents a pattern and the entries in RP-Header table follow the same order as F-list. Each entry consists of four fields: *item-name*, *count*, *item-link*, and *group-link*, where *item-name* registers the last item of the pattern represented by the entry, *count* means the number of tuples containing the pattern represented by

Algorithm FillTable

Method: Fill-RPHeader(*null*, H, *F-list*, *null*);
Procedure Fill-RPHeader(RP-Header table:H_1, PR-Header table:H_2, Item List: LI, Item:a_i)
(1) **for each** group G linked by *group-link* of entry a_i of H_1
// if $H_1 = null$, for each group used for compression
(2) **if** $G \cap LI \neq \emptyset$
(3) Let i be the first item of $G \cap LI$;
(4) Link group G to the *group-link* of entry i of H_2;
(5) **for each** group tail t of G
(6) **if** there exists an item $j, j \in LI \cap t$,
j orders before i in the order of LI **then**
// if there are several such j, we choose the first
// When i = null, i is ordered after all items in LI
(7) Link entry j in group tail t with entry j of H_2

Figure 7. Algorithm to Fill the RP-Header Table

the entry, *item-link* points to the tuple whose first item is *item-name*, and *group-link* points to the groups containing item-name. By following *item-link*, and *group-link*, we can get the projected database for the pattern represented by each entry.

One main originality of H-Mine is to construct the projected database using a set of pointers rather than physically projecting the database. The compressed database makes it non-trivial to do so since we need to consider both group heads and tails. We show how to fill the *item-link* and *group-link* of RP-Header table in Figure 4 to construct the projected database. The algorithm is described in Figure 7.

As in Example 3, d-projected database is mined first. In filling the RP-Header table, we can get d-projected database while assigning the group heads and group tails that are not in d-projected database to the other entries in RP-Header table. For group head G (lines 2-4), we assign it to the entry corresponding to the first item of $G \cap F\text{-}list$. For instance, group fgc is assigned to the entry f of RP-Header table H because f is the first item of $fgc \cap F\text{-}list$. For group tail (lines 5-7), we give two examples: (1) *Group tail* 100 in group fgc is linked by the *item-link* of entry d of H. (2) *Group tail* 300 is not linked with entry e of H. Note that *group tails* 100 and 300 are handled differently. This is because in 100 item d ranks before f (the first item of group fgc) in *F-list* and d-projected database is mined before f-projected database. However, in 300 e ranks after f and e-projected database is mined after f-projected database.

Example 5 *Let us examine the mining process for Example 3 based on the RP-Struct constructed in Example 4 as follows:*
(1) Find those containing item d. There is no group head that contains d. Therefore, by traversing the item-link of d, we can find the set of frequent items $\{f:2, g:2, c:2\}$ in d-projected database. Because all occurrences of f, g, c belong to group fgc, the frequent patterns can be generated by enumerating any combination of f, g, c.

After discovering frequent patterns in the subset, we traverse the item-link of d again to assign them to the items after d, i.e. f, g, a, e (there is no need to fill the item-link and group-link of item c because it can not be further extended). In group tail 100, the item after d is a and a ranks after item f, the first item of group fgc. Group tail 100 is not linked with the entry of item a of RP-Header table H since we mine f-projected database before a-projected database. For the similar reason, we also do not link group tail 300 with any entry of H.
(2) Find those containing item f but not d. The item-link of item f is null. The set of frequent items in f-projected database is $\{g:3, e:2, c:3\}$ by checking the group fgc. The RP-Header table H_f is constructed for f as shown in Figure 5. The group fgc contains the first item g of H_f. Therefore, the group fgc is linked with group-link of entry g of H_f. Since all items in group tails are after item g according to the order of H_f, there is no need to scan the group tails of fgc to build item-link for other items in H_f.

In order to mine the fg-projected compressed database, the RP-Header table H_{fg} is constructed as shown in Figure 5. The RP-Header table is constructed by traversing the item-link and group-link of entry g of H_f. For group fgc, its first item contained in H_{fg} is c. Since its group tails 100 and 300 contain item e and e is before c in H_{fg}, they are linked with entry e of H_{fg}. We do not link entry c with group fgc since pattern (fgc) represented by c can not be extended. The group tail 200 do not contain any items in H_{fg}. See Example 3 for the set of frequent patterns obtained. At the end of the step, the group fgc is assigned to item g of H.
(3) Find those containing g, but not f and d. The mining process is similar to (2) and is ignored here.
(4) Find those containing a but not g, f and d. Figure 6 shows the RP-Header table H_a. By traversing the item-link and group-link of entry e, we can get the set of frequent patterns (see Example 3).
(5) The step is ignored here. □

Based on the above analysis, we give the procedure Recycle-HM that recycles patterns by adapting H-Mine in Figure 8. The procedure Recycle-HM is used to replace the procedure RP-InMemory in algorithm Recycling shown in Figure 3. The a_i-projected database in line 6 of procedure Recycle-HM is obtained by following the item-link and group-link of entry a_i of H. The

Procedure Recycle-HM(PR-Struct: *Struct*,
RP-Header table :H, pattern:α)
(1) **if** RP-Header table H only contains a single group G
(2) **then for each** combination of (denoted as β)
the items in group G **do** Generate pattern $\beta \cup \alpha$
with supp = the count of group G;
(4) **else for each** item a_i in H **do**
(5) Generate pattern $\beta = a_i \cup \alpha$ with supp = $a_i.count$;
(6) Find List of frequent items LF_{a_i} in a_i-projected
database;
(7) Construct RP-Header table H_β for pattern β;
(8) Call Fill-RPHeader(H, H_β, LF_{a_i}, a_i);
(9) **if** $H_\beta \neq null$
(10) **then** Call Recycle-HM(*Struct*, H_β, β);
(11) Let A_{a_i} be the list of items ordered after a_i in H;
(12) Call Fill-RPHeader(H, H, A_{a_i}, a_i);

Figure 8. Recycling frequent patterns by adapting H-Mine

Procedure Fill-RPHeader() called in lines 8 and 12 is given in Figure 8. In line 8, the item-link and group-link of a_i are assigned to the RP-Header table in next level while in line 12 they are assigned to the entries after a_i in the same RP-Header table.

4.2 Recycling using FP-tree and Tree Projection Algorithms

We use the data structure of frequent pattern tree (or FP-tree in short), which is a prefix tree, to represent the outlying frequent items (uncompressed part). In the process of recursively constructing projected databases that are represented with FP-tree, we treat each (compressed) group head as a special item, which is in the upper of each prefix tree branch.

In Tree Projection algorithm [4], transactions are projected on each node of the tree from the root on. A matrix is maintained to count the support on the reduced set of transactions after projection. Tree Projection algorithm can mine frequent patterns in both depth-first and breath-first ways. We adopt depth-first approach in this paper.

5 Performance Studies

In this section, we will look at the performance of our approaches to recycling and reusing frequent patterns by comparing recycling algorithms with the corresponding non-recycling algorithms. As it is difficult to simulate the actual constrained mining environment, we adopt a simplified method to conduct our experiments. We perform an initial mining with a support threshold ξ_{old} to generate a set of patterns for recycling and then lower the support threshold to ξ_{new} when trying to recycle the patterns.

We have performed an extensive performance study on a wide range of data sets. We report a summary of the results here. All the experiments are performed on a 1.4GHz Pentium PC with 512M main memory, running Windows XP. All programs are developed using Microsoft VC++.

Weather [1] and *Forest* [3] are two sparse datasets used to report our results. *Connect-4* [3] and *Pumsb* [2] are dense data sets that we have used. The columns 2-4 of Table 3 list the number of tuples, the average tuple length and the total number of items in each data set. Because of the different properties of these datasets, we cannot choose the same initial support threshold ξ_{old} for all datasets. We try to choose the initial support ξ_{old} to ensure that we can have some frequent patterns to recycle.

Our argument for this is that a lack of frequent patterns for recycling will mean that little or no resources are used for the previous round of mining. It thus makes no sense to try to recycle patterns when no resources are used in the first place. This argument, based on the law of conservation, is also consistent with our observation that a lower initial support will usually give better performance of recycling. After all, we know that mining frequent patterns with low minimum support will typically require more resources in term of both CPU and I/Os. Since more resources are used, it is expected to bring more benefits when reusing the output of the mining.

5.1 Analysis of Compression Strategies

This subsection analyzes compression time and compression ratio of the two proposed compression strategies, MCP and MLP.

Columns 6 and 7 of Table 3 give some statistic on the patterns that are discovered with a minimum support ξ_{old}. We compress each database using these patterns. The last two columns of Table 3 show the compression ratio using the two strategies. The compression ratio R is computed as S_c/S_o, where S_c is the size of compressed database and S_o is the size of original database. In term of the compression ratio, $MLP \geq MCP$.

We would stress again that the compression here provides a way to speed up subsequent mining by utilizing previous frequent patterns, rather than to save space although it does. We can see that the compression ratio is not very large.

Table 3 also shows the running time for compressing the dataset in seconds. The column *"run time* (I/O)" in Table 3 includes the time used to read, write and

Dataset	#Tuples	Avg. Len.	# Items	ξ_{old}	# pattern	maximal length	Run Time(I/O) Sec.		Run Time(Pipeline) Sec.		Compression Ratio	
							MCP	MLP	MCP	MLP	MCP	MLP
Weather	1,015,367	15	7939	5%	1227	9	9.61	10.68	4.34	5.31	0.723	0.675
Forest	581,012	13	15,970	1%	523	4	2.67	4.58	0.45	2.25	0.858	0.785
Connect-4	67,557	43	130	95%	4411	10	0.32	0.32	0.06	0.06	0.773	0.773
Pumsb	49,046	74	7117	90%	2607	8	0.50	0.51	0.10	0.11	0.894	0.894

Table 3. The properties of datasets and compression statistic

compress data sets. The column *"run time (pipeline)"* deducts the I/O time from the column *run time* (I/O). We list such a column since the compression step can in fact be directly integrated into the mining algorithm when it is projecting the databases which means that the I/O time will be incurred anyway.

As shown in Table 3, the compression time is small compared with the mining time as shown in next subsection. This shows that the overhead of compression is not significant. The run time of the two strategies follows the order: $MLP \geq MCP$. The order is consistent with the compression ratio since the better ratio usually means more computation required for compression.

5.2 Mining in Main Memory

In this subsection, we assume that both the compressed databases and original databases can fit into the memory. We will evaluate the effectiveness of recycling patterns and the two compression strategies. We use HM-MCP and HM-MLP to represent the two recycling pattern algorithms adapted from H-Mine. HM-MCP and HM-MLP run on compressed database generated with the MCP and MLP strategies respectively. Similarly, FP-MCP and FP-MLP represent two algorithms adapted from FP-tree; TP-MCP and TP-MLP represent two algorithms adapted from Tree Projection.

The reported CPU time does not include the time used to output frequent patterns since it is the same for all algorithms. In any case, the mining cost dominates performance so that including them does not affect the relative performance of the various schemes.

The effectiveness of recycling patterns: Figures 9, 12, 15 and 18 compare the performance of recycling algorithm HM-MCP with H-Mine by varying the new support threshold ξ_{new} and plotting the CPU running time for each support threshold. For example, in Figure 9, HM-MCP mines the compressed dataset *weather* which is generated using the set of frequent patterns under $\xi_{old} = 5\%$. Readers can refer to Table 3 to get the related information for other figures. Note that the vertical axes of Figures 15 and 18 use logarithmic scale for clarity. These figures clearly show that HM-MCP are performing far better than H-Mine with respect to run time. In Figures 15 and 18, recycling algorithms are over two orders of magnitude faster than the non-recycling version. We also observe similar relative performance between the recycling algorithms and their non-recycling counterparts for FP-based techniques (see Figures 10, 13, 16 and 19) and Tree Projection methods (see Figures 11, 14, 17 and 20, where the vertical axes of Figures 17 and 20 use logarithmic scale). The experiment results clearly demonstrate the usefulness of recycling frequent patterns.

There are three interesting observations from our experiment results:

(1) When *minimum support* is low, the savings of HM-MCP against H-Mine are much more than the time used to generate the set of frequent patterns at ξ_{old}.[1] Considering that the compression time with pipeline in Table 3 is also small, this suggests the possibility that we could split a new mining task with low minimum support into two steps: (a) we first run it with a high minimum support; (b) we then compress the database with the strategy MCP and mine the compressed database with the actual low minimum support. We plan to explore this issue further.

(2) None of H-Mine, FP-tree and Tree Projection algorithms came out as a winner on all the datasets used. However, our recycling algorithms can always improve their performance.

(3) When the *minimum support* is low, recycling patterns using MCP performs better. This is exciting for incremental mining of frequent patterns. Existing incremental mining techniques do not work well when the data set or constraints change dramatically (e.g. sharp decrease in minimum support). HM-MCP can overcome the problem when it is applied to incremental mining.

Comparison of two compression strategies: Figures 9–20 compare the usefulness of the two compression strategies in recycling patterns. As shown, in all cases, the compression strategy MCP achieves at least the same or better performance than the other strategy MLP. As shown in subsection 5.1, MLP usually achieves better compression ratio than MCP. Therefore, we can conclude that better compression

[1] Although we do not show the time when mining with the support threshold ξ_{old}, readers can infer that it will be less than the CPU time for the lowest ξ_{new}.

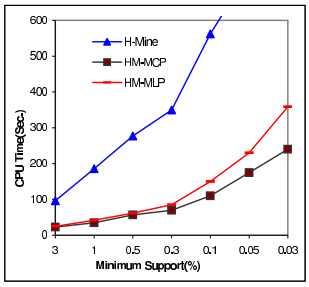

Figure 9. Adapting H-Mine on Weather

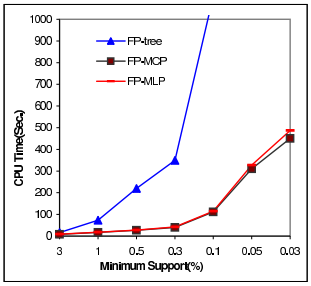

Figure 10. Adapting FP-tree on Weather

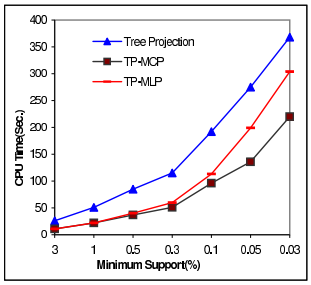

Figure 11. Adapting Tree Proj. on Weather

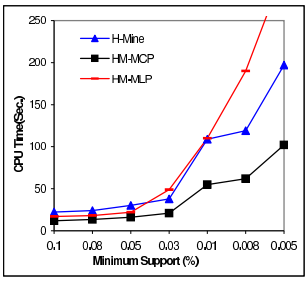

Figure 12. Adapting H-Mine on Forest

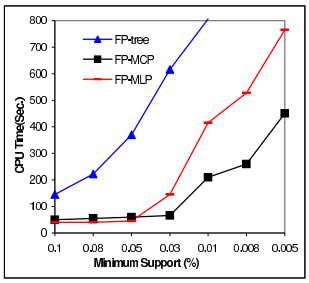

Figure 13. Adapting FP-tree on Forest

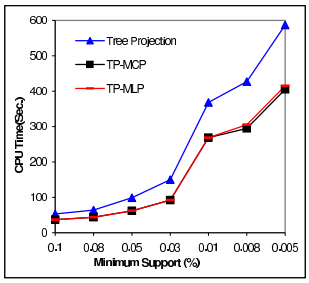

Figure 14. Adapting Tree Proj. on Forest

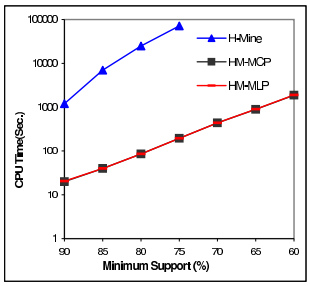

Figure 15. Adapting H-Mine on Connect-4

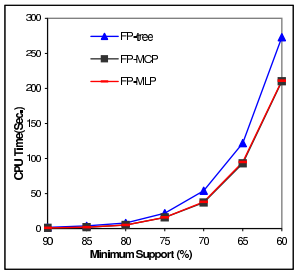

Figure 16. Adapting FP-tree on Connect-4

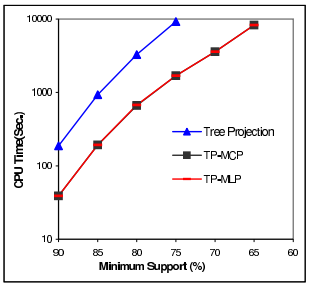

Figure 17. Adapting Tree Proj. on Connect-4

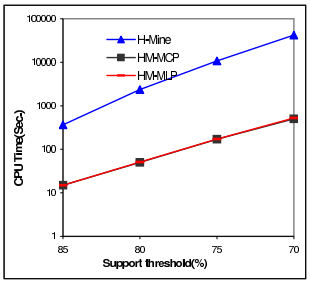

Figure 18. Adapting H-Mine on Pumsb

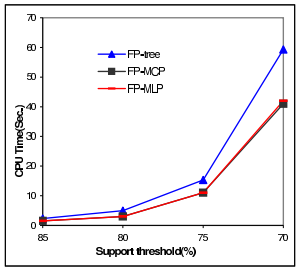

Figure 19. Adapting FP-tree on Pumsb

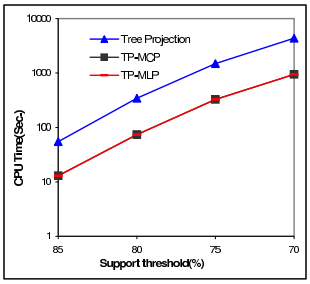

Figure 20. Adapting Tree Proj. on Pumsb

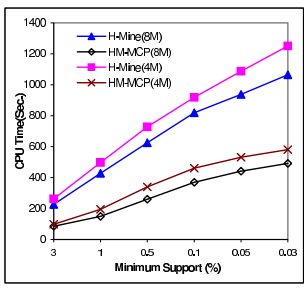

Figure 21. Weather with Memory Limitation

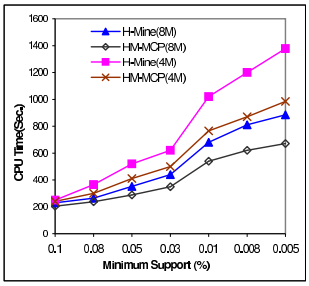

Figure 22. Forest with Memory Limitation

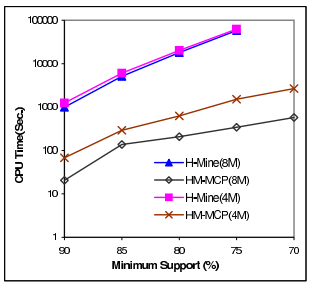

Figure 23. Connect-4 with Memory Limitation

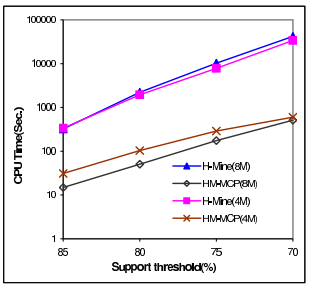

Figure 24. Pumsb with Memory Limitation

does not necessary means better performance. Our experiments also prove that minimizing mining cost (MCP) is more effective than minimizing storage space (MLP)in recycling frequent patterns. In fact, the compression ratios are not very large for both strategies as in Table 3. As given in section 3.1, the reason for the improvement of recycling algorithms against non-recycling schemes is that we can achieve savings in counting and projecting by means of compression.

In Figure 12, HM-MLP that recycles patterns based on MLP performs even worse than H-Mine. This implies that simply maximizing compression can even worsen the situation. We also observe that our mining algorithms based on the two compression strategies nearly achieve the same performance on dense data sets. This is because the two strategies nearly do the same compression as shown in Table 3.

5.3 Mining with Memory Limitation

In this subsection, we consider the case that the compressed datasets (and hence the original datasets) cannot be held in the main memory. As discussed in [15, 20], our mining algorithms HM-MLP and HM-MCP can compute the size of memory usage in the same manner as the H-Mine algorithm because they adopt similar data structure. The memory usage of FP-tree and Tree Projection algorithms can not be effectively estimated, and it is difficult to enforce memory limitation using FP-tree and Tree Projection algorithms. As a result, we do not compare FP-tree and Tree Projection with their recycling algorithms.

We enforce memory limitation to 4 and 8 megabytes. Such limitations are used because they can imitate the memory limitation situations considering the size of datasets although we realize such limitations are small compared to the available memory in current PC. The compressed databases are generated using the same set of recycled patterns as that in Section 5.1. Figures 21–24 show that HM-MCP outperforms H-Mine. Figures 23 and 24 use logarithmic scale for y-axes.

When comparing Figure 15 with Figure 23, readers may find that enforcing memory restriction on dense data set *Connect-4* even improves performance of both H-Mine and HP-MCP in some cases . Memory restriction requires that the (compressed) database be projected in the secondary storage until a level where the projected database can fit into memory. The projected (compressed) databases require less time in counting and the savings may be larger than the time used to read and write the projected (compressed) database.

Finally, all experiments show that in nearly all cases the saving of recycling algorithms over non-recycling counterparts is much greater than the time that is used to generate the set of frequent patterns for recycling.

In summary, our experimental results and performance analysis support the claim that recycling patterns is useful. Moreover, our experiments showed that the strategy of minimizing cost (MCP) is usually more effective than MLP for recycling patterns.

6 Related Work

Frequent pattern mining has been studied extensively in the past e.g. in [5, 4, 10, 17, 15, 11]. Many existing algorithms are variations of the Apriori algorithm [5]. Recently, a series of algorithms using projected database as the underlying framework were proposed [4, 10, 15] and were shown to be more efficient than the Apriori algorithm.

Some studies, such as [18, 12, 14], push constraints deep into frequent pattern mining algorithm in order to reduce computation in discovering uninteresting patterns. These constraints often prolong the frequent pattern mining process because the user may want to see the results of various combinations of constraint changes by running the mining algorithm multiple times. Studies on mining on multi-user platforms provide more opportunities for users to share their mining results. This makes recycling and reusing frequent patterns even more important.

Incremental techniques [7, 19, 13] are related to our work. They assume that the old mining process will output intermediate results for use in subsequent incremental mining. However, our approach does not make any assumption that old mining process realizes and makes preparation for subsequent mining. Our technique can be applied to incremental mining. Compared with existing incremental techniques [7, 19, 13], our techniques overcome the following disadvantages of existing incremental techniques: (1) existing incremental techniques need to store the negative border or similar information from previous computation, which can take large amount of space. (2) they are not effective when the changes of database or constraints are significant. (3) existing techniques become awkward when the size of data set reduces rather than increases. (4) they are usually not practical for recycling patterns problems with constraint changes because of huge number of candidates generated by such techniques[8].

Similarly, the techniques of mining with constraint change [8] have the same problems as incremental mining. In [8], the handling of constraint changes is dependent on the properties of constraints and is not applicable to certain constraints, for example *convertible* and *hard* constraints. The proposed technique in this paper gives a non-intrusive method of reusing patterns in previous computation no matter what type of constraints that are being used.

7 Conclusion

In this paper, we showed how frequent patterns discovered in the early round of mining (by the same user or different users) can be recycled to enhance subsequent mining. We proposed a two phase strategy that first compresses the database based on frequent patterns from an early round of mining and then mines the compressed database. We designed two compression strategies and adapted three existing mining algorithms to work on compressed databases. Our experimental results showed that the proposed strategy is effective, and the proposed recycling algorithms outperform their non-recycling counterparts significantly. Our results also showed that a cost-based compression strategy is preferred over a storage-based strategy.

References

[1] http://cdiac.esd.ornl.gov/cdiac/ndps/ndp026b.html.

[2] http://www.almaden.ibm.com/cs/quest/demos.html.

[3] http://www.ics.uci.edu/ mlearn/mlrepository.html.

[4] R. Agarwal, C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent itemsets. In *Journal of Parallel and Distributed Computing*, 2000.

[5] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB'94*, pages 487–499, Santiago, Chile, Sept. 1994.

[6] R. J. Bayardo. Efficiently mining long patterns from databases. In *SIGMOD'98*, pages 85–93, Seattle, WA, June 1998.

[7] D.W. Cheung, J. Han, V. Ng, and C.Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *ICDE'96*, pages 106–114, New Orleans, Louisiana, Feb. 1996.

[8] G. Cong and B. Liu. Speed-up iterative frequent itemset mining with constraint changes. In *ICDM'02*, 2002.

[9] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.

[10] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD'00*, pages 1–12, Dallas, TX, May 2000.

[11] J.Liu, Y. Pan, K. Wang, and J. Han. Mining frequent item sets by opportunistic projection. In *SIGKDD'02*, Alberta, Canada, July 2002.

[12] R. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *SIGMOD'98*, pages 13–24, Seattle, WA, June 1998.

[13] S. Parthasarathy, M. J. Zaki, M. Ogihara, and S. Dwarkadas. Incremental and interactive sequence mining. In *CIKM'99*, Kansas City, MO, USA,, November 1999.

[14] J. Pei, J. Han, and L. V. S. Lakshmanan. Mining frequent itemsets with convertible constraints. In *ICDE'01*, pages 433–332, Heidelberg, Germany, April 2001.

[15] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Y ang. H-mine: Hyper-structure mining of frequent patterns in large da tabases. In *ICDM'01*, San Jose, CA., Novermber 2001.

[16] J. Pei, J. Han, and R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *DMKD'00*, pages 11–20, Dallas, TX, May 2000.

[17] P. Shenoy, J. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In *SIGMOD'00*, pages 22–23, Dallas, TX, May 2000.

[18] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In *Proc. 1997 Int. Conf. Knowledge Discovery and Data Mining (KDD'97)*, pages 67–73, Newport Beach, CA, Aug. 1997.

[19] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An efficient algorithm for the incremental updation of association rules in large databases. In *KDD'97*, 1997.

[20] Anthony K. H. Tung, G. Cong, B. Ooi, and K. Tan. *Go Green: Recycle and Resuse in Human-centered, Interactive Data Mining*. Techinical Report 03-2, School of Computing, NUS, 2003.