IIE Transactions (2007) **39**, 593–605

ISSN: 0740-817X print / 1545-8830 online
DOI: 10.1080/07408170600897460

H-Mine: Fast and space-preserving frequent pattern mining in large databases

JIAN PEI[1,*], JIAWEI HAN[2], HONGJUN LU[3,†], SHOJIRO NISHIO[4], SHIWEI TANG[5] and DONGQING YANG[5]

[1] *School of Computing Science, Simon Fraser University, Burnaby, BC, Canada V5A 1S6*
E-mail: jpei@cs.sfu.ca
[2] *University of Illinois Urbana, IL 61801, USA*
E-mail: hanj@cs.uiuc.edu
[3] *Hong Kong University of Science and Technology, Hong Kong*
[4] *Osaka University, Osaka, Japan*
E-mail: nishio@ise.eng.osaka-u.ac.jp
[5] *Peking University, Beijing, China*
E-mail: {tsw, dqyang}@pku.edu.cn

Received February 2004 and accepted August 2005

In this study, we propose a simple and novel data structure using hyper-links, H-struct, and a new mining algorithm, H-mine, which takes advantage of this data structure and dynamically adjusts links in the mining process. A distinct feature of this method is that it has a very limited and precisely predictable main memory cost and runs very quickly in memory-based settings. Moreover, it can be scaled up to very large databases using database partitioning. When the data set becomes dense, (conditional) FP-trees can be constructed dynamically as part of the mining process. Our study shows that H-mine has an excellent performance for various kinds of data, outperforms currently available algorithms in different settings, and is highly scalable to mining large databases. This study also proposes a new data mining methodology, space-preserving mining, which may have a major impact on the future development of efficient and scalable data mining methods.

Keywords: Frequent pattern mining, FP-tree, transaction databases

1. Introduction

Frequent pattern mining plays an essential role in many data mining tasks and applications, such as mining association rules (Agrawal *et al*., 1993), correlations (Brin *et al.*, 1997; Silverstein *et al*., 1998), sequential patterns (Agrawal and Srikant, 1995), episodes (Mannila *et al*., 1997), multi-dimensional patterns (Kamber *et al*., 1997), max-patterns and frequent closed patterns (Bayardo, 1998; Pasquier *et al*., 1999), partial periodicity (Han *et al*., 1999), emerging patterns (Dong and Li, 1999), classification (Liu *et al*., 1998), and clustering (Agrawal *et al*., 1998).

The numerous studies on the fast mining of frequent patterns can be classified into two categories. The first category, *candidate-generation-and-test approaches*, such as *Apriori* (Agrawal and Srikant, 1994) and many subsequent studies, are directly based on an anti-monotone *Apriori* property (Agrawal and Srikant, 1994): *if a pattern with k items is not frequent, any of its super-patterns with (k + 1) or more items can never be frequent*. A candidate-generation-and-test approach iteratively generates a set of candidate patterns of length $(k+1)$ from a set of frequent patterns of length k ($k \geq 1$), and checks their corresponding occurrence frequencies in the database.

The *Apriori* algorithm achieves a good reduction in the size of candidate sets. However, when there exist a large number of frequent patterns and/or long patterns, candidate-generation-and-test methods may still suffer from generating a large number of candidates and taking many scans of large databases in frequency checking.

Recently, another category of methods, *pattern-growth methods*, such as *FP-growth* (Han *et al*., 2000) and *Tree Projection* (Agarwal *et al*., 2000, 2001), have been proposed. A pattern-growth method uses the *Apriori* property. However, instead of generating candidate sets, it recursively partitions the database into sub-databases according to the frequent patterns found and searches for local frequent patterns to assemble longer global ones.

*Corresponding author
†Deceased

Nevertheless, these proposed approaches may still encounter some difficulties in different cases. First, **a huge memory space is required to serve the mining, and the main memory consumption is usually hard to precisely predict**. An *Apriori*-like algorithm generates a huge number of candidates for *long or dense* patterns. To find a frequent pattern of size 100, such as $\{a_1, \ldots, a_{100}\}$, up to 5×10^{30} units of main memory space is needed to store candidates. *FP-growth* (Han *et al*., 2000) avoids candidate generation by compressing the transaction database into an *FP-tree* and pursuing partition-based mining recursively. However, if the database is *huge and sparse*, the *FP-tree* will be large and the space requirement for recursion is a challenge. Neither approach is superior in all cases.

Second, **real databases contain all the cases.** Real data sets can be sparse and/or dense in different applications. For example, for telecommunications data analysis, calling patterns for home users could be very different to that for business users: some calls are frequent and dense (e.g., to family members and close friends), but some are huge and sparse. Similar situations arise in market basket analysis, census data analysis, classification and predictive modeling, etc. It is hard to select an appropriate mining method on the fly if no algorithm fits all cases.

Last, **large applications need more scalability.** Many existing methods are efficient when the data set is not very large. Otherwise, their core data structures (such as *FP-tree*) or the intermediate results (e.g., the set of candidates in *Apriori* or the recursively generated conditional databases in *FP-growth*) may not fit into the main memory and can easily cause thrashing.

This poses a new challenge: *can we work out a better method in that: (i) the main memory requirement is precisely predictable and moderate, even for very large databases; and (ii) it is efficient in most occasions (dense vs. sparse, huge vs. memory-based data sets)?*

In this paper, we propose a new data structure, H-struct, and a new mining method, H-mine, to overcome these difficulties, with the following progress. First, a memory-based, efficient pattern-growth algorithm, H-mine(Mem), is proposed for mining frequent patterns for the data sets that can fit into the (main) memory. A simple, memory-based hyper-structure, H-struct, is designed for fast mining. Second, we show that, theoretically, H-mine(Mem) has a polynomial space complexity and is thus more space efficient than pattern-growth methods such as *FP-growth* and *TreeProjection* when mining sparse data sets, and also more efficient than *Apriori*-based methods which generate a large number of candidates. Experimental results show that, in many cases, H-mine has a very limited and exactly predictable space overhead and is faster than the memory-based *Apriori* and *FP-growth* methods. Third, based on H-mine(Mem), we propose H-mine, a scalable algorithm for mining large databases by first partitioning the database, mining each partition in the memory using H-mine(Mem), and then consolidating globally frequent patterns. Fourth, for dense data sets, H-mine is integrated with *FP-growth* dynamically by detecting the swapping condition and constructing FP-trees for efficient mining. Last, such efforts ensure that H-mine is scalable in both large and medium-sized databases and in both sparse and dense data sets. Our comprehensive performance study confirms that H-mine is highly scalable and is faster than *Apriori* and *FP-growth* on all occasions.

The rest of the paper is organized as follows. Section 2 is devoted to H-mine(Mem), an efficient algorithm for memory-based frequent pattern mining. In Section 3, H-mine(Mem) is extended to huge, disk-based databases, together with some further optimizations techniques. Our performance study is reported in Section 4. We draw conclusions in Section 5.

2. H-mine(Mem): Memory-based mining

In this section, H-mine(Mem) (memory-based hyper-structure mining of frequent patterns) is developed, and in Section 3, the method is extended to handle large and/or dense databases.

Definition 1. Let $I = \{x_1, \ldots, x_n\}$ be a set of **items**. An **itemset** X is a subset of items, i.e., $X \subseteq I$. For the sake of brevity, an itemset $X = \{x_1, x_2, \ldots, x_m\}$ is also denoted as $X = x_1 x_2 \cdots x_m$. A **transaction** $T = (tid, X)$ is a 2-tuple, where tid is a transaction-id and X an itemset. A transaction $T = (tid, X)$ is said to **contain** itemset Y if and only if $Y \subseteq X$. A **transaction database** TDB is a set of transactions. The number of transactions in TDB containing itemset X is called the **support** of X, denoted as $sup(X)$. Given a transaction database TDB and a **support threshold** min_sup, an itemset X is a **frequent pattern**, or a **pattern** in short, if and only if $sup(X) \geq min\_sup$.

The **problem of frequent pattern mining** is to *find the complete set of frequent patterns in a given transaction database with respect to a given support threshold.*

2.1. *General idea of* H-mine(Mem)

We illustrate the general idea of H-mine(Mem) using an example.

Let the first two columns of Table 1 be our running transaction database TDB. Let the minimum support threshold be $min\_sup = 2$.

Table 1. The transaction database TDB used as our running example

Transaction ID	*Items*	*Frequent-item projection*
100	*c,d,e,f,g,i*	*c,d,e,g*
200	*a,c,d,e,m*	*a,c,d,e*
300	*a,b,d,e,g,k*	*a,d,e,g*
400	*a,c,d,h*	*a,c,d*

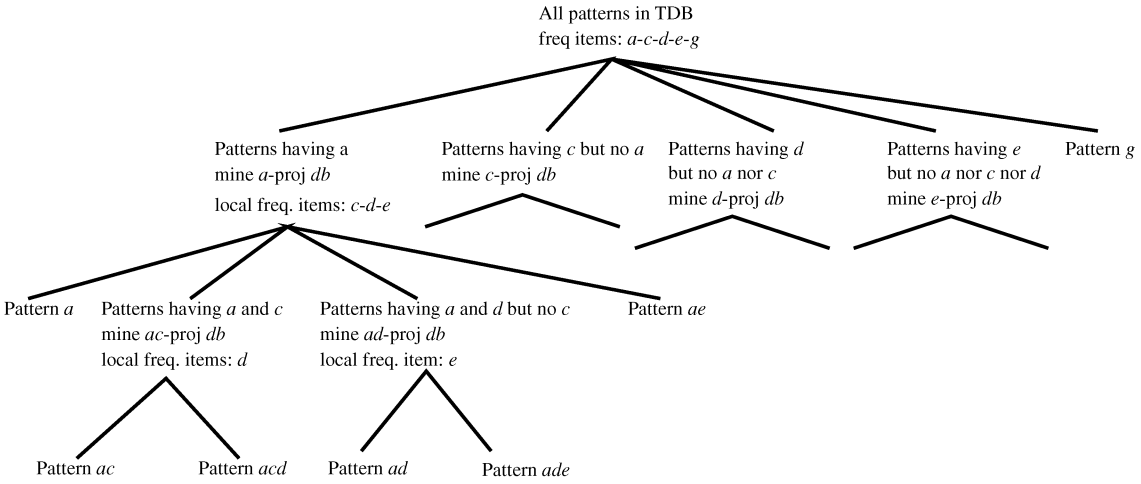

Fig. 1. Divide-and-conquer frequent patterns.

Following the *Apriori* property (Agrawal and Srikant, 1994), only frequent items play roles in frequent patterns. By scanning *TDB* once, the complete set of frequent items $\{a : 3, c : 3, d : 4, e : 3, g : 2\}$ can be found and output, where the notation $a : 3$ means item a's support (occurrence frequency) is three. Let *freq* (X) (the *frequent-item projection* of X) be the set of frequent items in itemset X. For ease of explanation, the frequent-item projections of all the transactions of Table 1 are shown in the third column of the table.

Following the alphabetical order of frequent items[1] (called an *F-list*) *a-c-d-e-g*, the complete set of frequent patterns can be partitioned into five subsets as follows: (i) those containing item a; (ii) those containing item c but not item a; (iii) those containing item d but no item a nor item c; (iv) those containing item e but no item a nor item c nor item d; and (v) those containing only item g, as shown in Fig. 1.

If the frequent-item projections of transactions in the database can be held in the main memory, then they can be organized as shown in Fig. 2. All items in frequent-item projections are sorted according to the *F-list*. For example, the frequent-item projection of transaction 100 is listed as *cdeg*. Every occurrence of a frequent item is stored in an entry with two fields: an *item-id* and a *hyper-link*.

A *header table H* is created, with each frequent item entry having three fields: an *item-id*, a *support count*, and a *hyper-link*. When the frequent-item projections are loaded into the memory, those with the same first item (in the order of the *F-list*) are linked together by the hyper-links into a queue, and the entries in header table H act as the heads of the queues. For example, the entry of item a in the header table H is the head of the a-queue, which links frequent-item projections of transactions 200, 300, and 400. These three projections all have item a as their first frequent item (in the order of the *F-list*). Similarly, the frequent-item projection of transaction 100 is linked as a *c-queue*, headed by item c in H. The d-, e- and g-queues are empty since there is no frequent-item projection that begins with any of these items.

Clearly, it takes one scan (the second scan) of the transaction database *TDB* to build such a memory structure (called the H-struct). Then the remaining mining can be performed on the H-struct only, without referencing any information in the original database. After that, the five subsets of frequent patterns can be mined one by one as follows.

First, let us consider how to find the set of frequent patterns in the first subset, i.e., all the frequent patterns containing item a. This requires us to search all the frequent-item projections containing item a, i.e., the *a-projected*

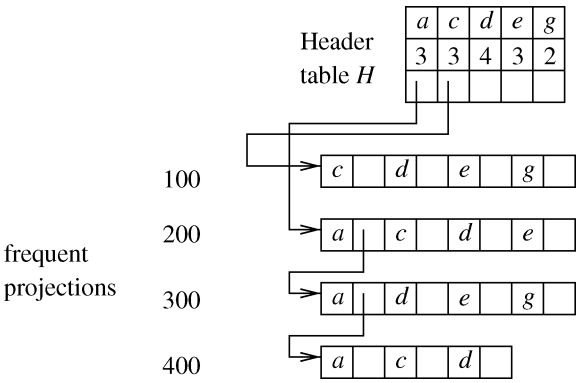

Fig. 2. H-struct, the hyper-structure to store frequent-item projections.

[1] As you may be aware, any ordering should work, and the alphabetical ordering is just for the convenience of explanation.

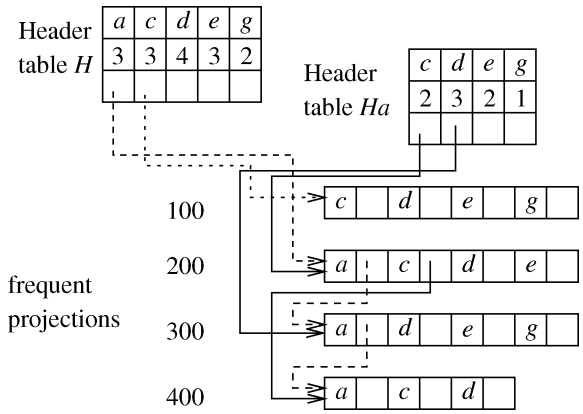

Fig. 3. Header table H_a and the ac-queue.

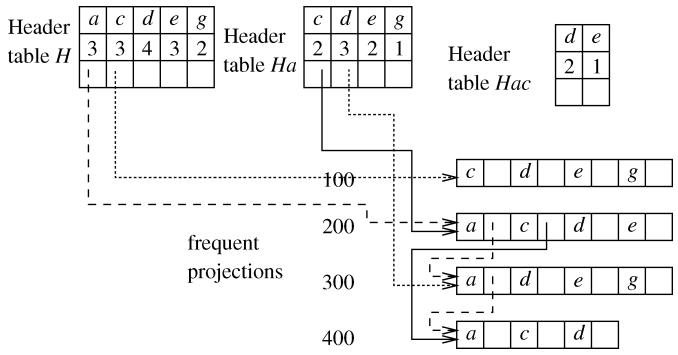

Fig. 4. Header table H_{ac}.

database[2], denoted as $TDB|_a$. Interestingly, the frequent-item projections in the a-projected database are already linked in the a-queue, which can be traversed efficiently.

To mine the a-projected database, an *a-header table* H_a is created, as shown in Fig. 3. In H_a, every frequent item, except for a itself, has an entry with the same three fields as H, i.e., *item-id, support count* and *hyper-link*. The support count in H_a records the support of the corresponding item in the a-projected database. For example, item c appears twice in the a-projected database (i.e., frequent-item projections in the a-queue), thus the support count for the entry c in H_a is two.

By traversing the a-queue once, the set of locally frequent items, i.e., the items appearing at least two times, in the a-projected database is found, which is $\{c : 2, d : 3, e : 2\}$ (note: $g : 1$ is not locally frequent and thus will not be considered further). This scan outputs frequent patterns $\{ac : 2, ad : 3, ae : 2\}$ and builds up links for the H_a header as shown in Fig. 3.

Similarly, the process continues for the ac-projected database by examining the c-queue in H_a, which creates an *ac-header table* H_{ac}, as shown in Fig. 4.

Since only item $d : 2$ is a locally frequent item in the ac-projected database, only $acd : 2$ is outputed, and the search along this path completes.

Then the recursion backtracks to find patterns containing a and d but not c. Since the queue started from d in the header table H_a, i.e., the ad-queue, links all frequent-item projections containing items a and d (but excluding item c in the projection), one can get the complete ad-projected database by inserting frequent-item projections having item d in the ac-queue into the ad-queue. This involves one more traversal of the ac-queue. Each frequent-item projection in the ac-queue is appended to the queue of the next frequent item in the projection according to *F-list*. Since all the frequent-item projections in the ac-queue have item d, they are all inserted into the ad-queue, as shown in Fig. 5.

It can be seen that, after the adjustment, the ad-queue collects the complete set of frequent-item projections containing items a and d. Thus, the set of frequent patterns containing items a and d can be mined recursively. Please note that, even though item c appears in frequent-item projections of the ad-projected database, we do not consider it as a locally frequent item in any recursive projected database since it has been considered in the mining of the ac-queue. This mining generates only one pattern $ade : 2$. Notice also that the third level header table H_{ad} can use the table H_{ac} since the search for H_{ac} was done in the previous round. Thus, we only need one header table at the third level. Later we can see that only one header table is needed for each level of the whole mining process.

For the search in the ae-projected database, since e contains no child links, the search terminates, with no patterns being generated.

After the frequent patterns containing item a are found, the a-projected database, i.e., a-queue, is no longer needed for the remaining mining processes. Since the c-queue

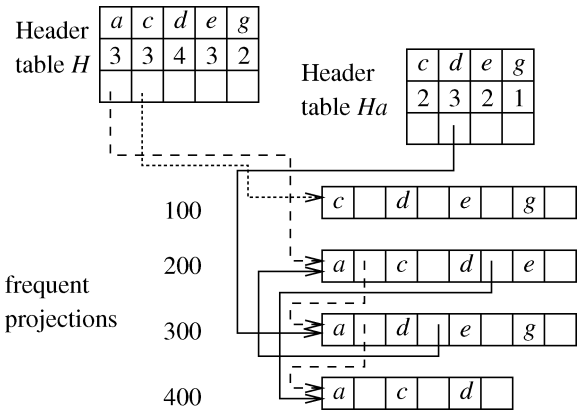

Fig. 5. Header table H_a and the ad-queue.

[2]The a-projected database consists of all the frequent-item projections containing item a, but these are all "virtual" projections since no physical projections are performed to create a new database.

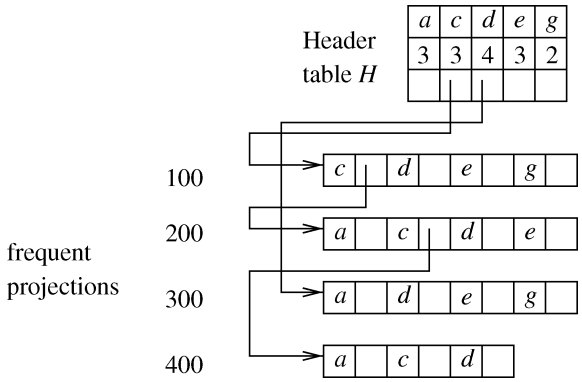

Fig. 6. Adjusted hyper-links after mining the a-projected database.

includes all frequent-item projections containing item c except for those projections containing both items a and c, which are in the a-queue. To mine all the frequent patterns containing item c but not a, and other subsets of frequent patterns, we need to insert all the projections in the a-queue into the proper queues.

We traverse the a-queue once more. Each frequent-item projection in the queue is appended to the queue of the next item in the projection following a in the *F-list*, as shown in Fig. 6. For example, frequent-item projection *acde* is inserted into the c-queue and *adeg* is inserted into the d-queue.

By mining the c-projected database recursively (with a shared header table at each level), we can find the set of frequent patterns containing item c but not a. Notice item a will not be included in the c-projected database since all the frequent patterns having item a have already been found.

Similarly, the mining goes on. In the next section, we verify that the above mining process finds the complete set of frequent patterns without duplication. The remaining mining process is left as an exercise to interested readers.

Notice also that the depth-first search for mining the first set of frequent patterns at any depth can be done in one database scan by constructing the header tables at all levels simultaneously.

2.2. H-mine(Mem)*: The algorithm*

Now, let us summarize and justify the mining process discussed in Section 2.1.

Given a transaction database *TDB* and a support threshold *min_sup*, let L be the set of frequent items. *F-list*, a list of frequent items, is a global order over L. Let x and y ($x \neq y$) be two frequent items. We denote $x \prec y$ iff x is before y according to the *F-list*. For example, based on the *F-list* in Section 2.1, we have $a \prec c \prec d \prec e \prec g$.

Frequent-item projections of transactions in *TDB* are organized in an H-struct. An H-struct contains the set of frequent-item projections of a transaction database. Each item in a frequent-item projection is represented by an entry with two fields: *item-id* and *hyper-link*.

An H-struct has a *header table*. The header table is an array of frequent items in the order of the *F-list*. A support count and a hyper-link are attached to each item in the header table. When the H-struct is created, items in the header table are the *heads* of the *queues* of frequent-item projections linked by hyper-links.

The hyper-structure shown in Fig. 2 is an example of an H-struct. For every transaction, H-struct stores its frequent-item projection. Besides frequent-item projections, H-struct also stores a header table whose size is bounded by the number of frequent items. This is all the space needed by the H-struct. Therefore, the space requirement of an H-struct is $\Theta(\sum_{t \in TDB} |freq(t)|)$, where $freq(t)$ is a frequent-item projection of a transaction t. Only two scans of a transaction database are needed to build an H-struct.

Given a transaction database *TDB* and *F-list*, the complete set of frequent patterns can be partitioned into a series of subsets without overlap, as stated in the following lemma.

Lemma 1. (*Partition of search space*.) *Given a transaction database TDB and support threshold min_sup, let F-list* "x_1-...-x_n" *be a list of frequent items. The complete set of frequent patterns can be partitioned into n subsets without overlap as follows*: *the kth subset* ($1 \leq k \leq n$) *contains patterns having item* x_k *but no item* x_i ($1 \leq i < k$).

To mine the subsets of frequent patterns, we introduce the concept of a *projected database*. Let P be a frequent pattern. The *P-projected database* is the collection of frequent-item projections containing pattern P, denoted as $TDB\}_P$.

Clearly, to mine the kth subset of frequent patterns in Lemma 1, we only need to look at the x_k-projected database $TDB|_{x_k}$ and ignore occurrences of items x_i ($1 \leq i < k$). *How can* H-struct *facilitate the construction of projected databases?* We have the following lemma.

Lemma 2. (*Projected databases*.) *Given a transaction database TDB and support threshold min_sup*, *let F-list* "x_1-$\cdots$-x_n" *be the list of frequent items. In the* H-struct: (*i*) *The* x_1*-projected database is the* x_1*-queue in the header table;* (*ii*) *the* x_2*-projected database is the* x_2*-queue in the header table and the frequent-item projections starting at item* x_2 *in the* x_1*-queue*; (*iii*) *in general, the* x_k*-projected database* ($1 < k \leq n$) *is the* x_k*-queue in the header table and the frequent-item projections starting at* x_k *in the* x_i*-queues* ($1 \leq i < k$).

Based on Lemma 2, we can first find the complete set of frequent patterns containing x_1, using the x_1-queue available in H-struct. Conceptually, we treat the queue of frequent-item projections in the x_1-projected database as a sub-H-struct and apply the techniques recursively. That is, we find the locally frequent items, further partition the subset of frequent patterns and perform recursive mining. The storage of frequent-item projections, i.e., H-struct, can be shared. We only need a new header table to be able to form queues within the x_1-projected database.

Then, we insert those frequent-item projections in the x_1-queue starting at item x_2 into the x_2-queue in the header table, and form the complete x_2-projected database. Since the projections exclude x_1 in the x_2-projected database by starting at x_2, we can find the complete set of frequent patterns containing item x_2 but not item x_1.

Similarly, we can find the complete set of frequent patterns. Based on the above reasoning, we have the following algorithm.

Algorithm 1 (H-mine(Mem)) (Main) memory-based hyper-structure mining of frequent patterns.

Input: A transaction database *TDB* and a support threshold *min_sup*.
Output: The complete set of frequent patterns.
Method:

Step 1. Scan *TDB* once, find and output L, the set of frequent items. Let *F-list*: "x_1-...-x_n" ($n = |L|$) be a list of frequent items.
Step 2. Scan *TDB* again, construct H-struct, with header table H, and with each x_i-queue linked to the corresponding entry in H.
Step 3. For $i = 1$ to n do
  (a) Call H-mine($\{x_i\}$, H, *F-list*).
  (b) traverse the x_i-queue in the header table H, for each frequent-item projection X, link X to the x_j-queue in the header table H, where x_j is the item in X following x_i immediately.

Procedure H-mine(P, H, *F-list*)//P is a frequent pattern
// Note: The frequent-item projections in the P-projected database are linked as a P-queue in the header table H.

Step 1. Traverse the P-queue once, find and output its locally frequent items and derive *F-list*$_P$: "x_{j_1}-...-$x_{j_{n'}}$".
// Note: Only the items in the *F-list* and are located to the right of P are considered. Items in the *F-list*$_P$ follow the same order as that in the *F-list*.
Step 2. Construct header table H_P, scan the P-projected database, and for each frequent-item projection X in the projected database, use the hyper-link of x_{j_i} ($1 \le i \le n'$) in X to link X to the Px_{j_i}-queue in the header table H_P, where x_{j_i} is the first locally frequent item in X according to the *F-list*$_P$.
Step 3. For $i = 1$ to n' do
  (a) Call H-mine($P \cup \{x_{j_i}\}$, H_P, *F-list*$_P$).
  (b) Traverse Px_{j_i}-queue in the header table H_P, for each frequent-item projection X, link X to the x_{j_k}-queue ($i < k \le n'$) in the header table H_P, where x_{j_k} is the item in X following x_{j_i} immediately according to *F-list*.

Now, let us analyze the space requirement of Algorithm 1. As discussed previously, the space complexity of constructing an H-struct is $\Theta(\sum_{t \in TDB} |freq(t)|)$. To mine the H-struct, the only space overhead is a set of local header tables. At first glance, the number of header tables seems to be of the order of that for the frequent patterns. However, a close look at the algorithm finds that only a very limited number of header tables exist simultaneously. For example, to find pattern $P = bcde$, only the header tables for the "prefixes" of P, i.e., H_b, H_{bc}, H_{bcd} and H_{bcde}, are needed. All the other header tables are either already used and can be freed, or have not yet been generated. The header tables for patterns with item a have already been used and can be freed since all the patterns having item a have been found before pattern $bcde$. On the other hand, all the other header tables are for patterns to be found later and thus need not be generated at this moment. Therefore, the number of header tables is bounded by the maximal length of a single frequent pattern. Thus, we have the following lemma.

Lemma 3. (*Number of header tables*.) *The maximum number of header tables needed in the hyper-structure mining of frequent patterns. i.e.*, H-mine(Mem)*, is bounded by the maximal length of a single frequent pattern that can be found.*

Since the maximal length of a single frequent pattern cannot exceed the maximal length of a transaction, and in general, the maximal length of a transaction is much smaller than the number of transactions, we have the following theorem on the space complexity of H-mine(Mem).

Theorem 1. (*Space complexity*.) *The space complexity of Algorithm 1 is* $\Theta(\sum_{t \in TDB} |freq(t)|)$, *where* $freq(t)$ *is a frequent-item projection of a transaction* t.

Comparing with other frequent pattern mining methods, the efficiency of H-mine(Mem) comes from the following aspects.

First, H-mine(Mem) avoids candidate generation and test by adopting a frequent-pattern growth methodology, a more efficient method shown in previous studies (Han *et al.*, 2000; Agrawal *et al.*, 2001). H-mine(Mem) absorbs the advantages of pattern growth.

Second, H-mine(Mem) confines its search to a dedicated space. Unlike other frequent pattern growth methods, such as *FP-growth* (Han *et al.*, 2000), it does not need to physically construct memory structures of projected databases. It fully utilizes the well organized information in the H-struct, and collects information about projected databases using the header tables, which are light-weight structures. That also saves a lot of effort on managing space.

Last, H-mine(Mem) does not need to store any frequent patterns in the memory. Once a frequent pattern is found, it is outputed on to a disk. In contrast, the candidate-generation-and-test method has to save and use the frequent patterns found in the current round to generate candidates for the next round.

3. From H-mine(Mem) to H-mine: Efficient mining on different occasions

In this section, we first extend our algorithm H-mine(Mem) to H-mine, which mines frequent patterns in large data sets that cannot fit into the main memory. Then, we explore how to integrate *FP-growth* when the data sets being mined become very dense.

3.1. H-mine: *Mining frequent patterns in large databases*

H-mine(Mem) is efficient when the frequent-item projections of a transaction database plus a set of header tables can fit into the main memory. However, we cannot expect this to always be the case. When they cannot fit into the memory, a database partitioning technique can be developed as follows.

Let TDB be the transaction database with n transactions and min_sup be the support threshold. By scanning TDB once, one can find L, the set of frequent items.

Then, TDB can be partitioned into k parts, TDB_1, $\ldots$, TDB_k, such that, for each TDB_i $(1 \leq i \leq k)$, the frequent-item projections of transactions in TDB_i can be held in main memory, where TDB_i has n_i transactions, and $\sum_{i=1}^{k} n_i = n$. We can apply H-mine(Mem) to TDB_i to find frequent patterns in TDB_i with the minimum support threshold $min\_sup_i = \lfloor min\_sup \times n_i/n \rfloor$ (i.e., each partitioned database keeps the same relative minimum support as the global database).

Let F_i $(1 \leq i \leq k)$ be the set of (locally) frequent patterns in TDB_i. Based on the property of partition-based mining (Savesere *et al.*, 1995), P cannot be a (globally) frequent pattern in TDB with respect to the support threshold min_sup if there exists no i $(1 \leq i \leq k)$ such that P is in F_i. Therefore, after mining frequent patterns in the TDB_i, we can gather the patterns in F_i and collect their (global) support in TDB by scanning the transaction database TDB one more time.

Based on the above observation, we can extend H-mine(Mem) to H-mine as follows.

Algorithm 2 (H-mine) Hyper-structure mining of frequent-patterns in large databases.

Input and output: same as Algorithm 1.
Method:

Step 1. Scan the transaction database TDB once to find L, the complete set of frequent items.
Step 2. Partition TDB into k parts, $TDB_1, \ldots, TDB_k$, such that, for each TDB_i $(1 \leq i \leq k)$, the frequent-item projections in TDB_i can be held in the main memory.
Step 3. For $i = 1$ to k, use H-mine(Mem) to mine frequent patterns in TDB_i with respect to the minimum support threshold $min\_sup_i = \lfloor min\_sup \times n_i/n \rfloor$, where n and n_i are the number of transactions in TDB and TDB_i, respectively. Let F_i be the set of frequent patterns in TDB_i.
Step 4. Let $F = \bigcup_{i=1}^{k} F_i$. Scan TDB one more time, collect support for patterns in F. Output those patterns which pass the minimum support threshold min_sup.

One important issue in Algorithm 2 is how to partition the database. As analyzed in Section 2.2, the only space cost of H-mine(Mem) is incurred by the header tables. The maximal number of header tables as well as their space requirement are predictable (usually very small in comparison with the size of the frequent-item projections). Therefore, after reserving space for header tables, the remaining main memory space can be used to build an H-struct that covers as many transactions as possible. In practice, it is good to first estimate the size of the available main memory for mining and the size of the overall frequent-item projected database (similar in size to the sum of support counts of frequent items), and then roughly evenly partition the database to avoid the generation of skewed partitions.

Note that our partition-based mining approach shares some similarities with the *partitioned Apriori* method proposed in Savasere *et al.* (1995) in which a transaction database is first partitioned, every partition is mined using *Apriori*, then all the locally frequent patterns are gathered to form globally frequent candidate patterns before counting their global support by one more scan of the transaction database. However, there are two essential differences between these two methods.

First, as also indicated in Savasere *et al.* (1995), it is not easy to obtain a good partition scheme using the *partitioned Apriori* approach since it is hard to predict the space requirement of *Apriori*. In contrast, it is straightforward for H-mine to partition the transaction database, since the space overhead is very small and predictable during mining.

Second, H-mine first finds globally frequent items. When mining partitions of a database, H-mine examines only those items which are globally frequent. In skewed partitions, many globally infrequent items can be locally frequent in some partitions, but H-mine does not spend any effort to check them unlike the *partitioned Apriori* approach which does so check.

Furthermore, we can do better in consolidating globally frequent patterns from local ones. When mining a large transaction database, if the database is partitioned relatively evenly, it is expected that many short globally frequent patterns are frequent in every partition. In this case, a pattern frequent in every partition is a globally frequent pattern, and its global support count is the sum of the counts in all the partitions. H-mine does not need to test such patterns in its third scan. Therefore, in the third scan, H-mine checks only those locally frequent patterns which are infrequent in some partitions. Furthermore, a pattern is checked against only those partitions where it is infrequent.

In general, the following factors contribute to the scalability and efficiency of H-mine.

First, as analyzed in Section 2, H-mine(Mem) has a small space overhead and is efficient in mining partitions which can be held in the main memory. With current memory technologies, it is likely that many medium-sized databases can be mined efficiently by this memory-based frequent-pattern mining mechanism.

Second, no matter how large the database, it can be mined by at most three scans of the database: the first scan finds globally frequent items; the second mines the partitioned database using H-mine(Mem); and the third verifies globally frequent patterns. Since every partition is mined efficiently using H-mine(Mem), the mining of the whole database is highly scalable.

Last, one may wonder that, since the *partitioned Apriori* algorithm of Savasere *et al*. (1995) takes two scans of *TDB*, whereas H-mine takes three scans, how can H-mine outperform it? Notice that the major cost in this process is the mining of each partitioned database. The last scan of *TDB* for collecting supports and generating globally frequent patterns is fast because the set of locally frequent patterns can be inserted into one compact structure, such as a hashing tree. Since H-mine generates fewer partitions and mines each partition very quickly, it has a better overall performance than the *Apriori*-based partition mining algorithm. This is also demonstrated in our performance study.

In summary, a major advantage of H-mine comes from the fact that it uses a more efficient data structure and algorithm than the hash-tree-based approaches discussed in the previous studies.

3.2. *Handling dense data sets: Dynamic integration of H-struct and FP-tree-based mining*

As indicated in Bayardo *et al*. (1999), Han *et al*. (2000) and Pei *et al*. (2000) finding frequent patterns in dense databases is a challenging task since it may generate dense and long patterns which may lead to the generation of a very large (and even exponential) number of candidate sets if an *Apriori*-like algorithm is used. The *FP-growth* method proposed in our recent study (Han *et al*., 2000) works well on dense databases with a large number of long patterns due to the effective compression of shared prefix paths in the mining.

In comparison with *FP-growth*, H-mine does not generate physical projected databases and conditional *FP-tree*s and thus saves space as well as time in many cases. However, *FP-tree*-based mining has its advantages over mining using H-struct since the *FP-tree* shares common prefix paths among different transactions, which may lead to space and time savings as well. As one may expect, the situation under which one method outperforms the other depends on the characteristics of the data sets: if data sharing is rare such as in sparse databases, the compression factor could be small and the *FP-tree* may not outperform mining using H-struct. On the other hand, there are many dense data sets in practice. Even though the data sets might not be dense originally, as mining progresses, the projected databases become smaller, and data often becomes denser as the relative support goes up when the number of transactions in a projected database reduces substantially. In such cases, it is beneficial to swap the data structure from H-struct to *FP-tree* since the *FP-tree*'s compression by common prefix path sharing and then mining on the compressed structures will overweigh the benefits brought by H-struct.

The questions arise of what are the situations in which one structure is more preferable than the other one and also how to determine when such a structure/algorithm swapping should happen. A dynamic pattern density analysis technique is suggested as follows.

In the context of frequent pattern mining, a (projected) database is *dense* if the frequent items in it have a *high relative support*. The *relative support* can be computed as

$$\frac{\textit{absolute support}}{\textit{number of transactions (or frequent-item projections) in the (projected) database}}.$$

When the relative support is high, such as 10% or over, i.e., the projected database is dense, and the number of (locally) frequent items is not large (so that the resulting *FP-tree* is not bushy), then an *FP-tree* should be constructed to explore the sharing of common prefix paths and database compression. On the other hand, when the relative support of frequent items is low, such as far below 1%, it is sparse, and H-struct should be constructed for an efficient H-mine. However, in the middle lies the gray area, and which structure and method should be used will depend on the size of the frequent-item projection database, the size of the main memory and other performance factors.

With this discussion, one can see that Algorithm 2 (H-mine) should be modified as follows.

In Step 3 of Algorithm 2 which mines frequent patterns in each partition TDB_i, H-mine(Mem) is called. However, instead of simply constructing H-struct and mining the H-struct iteratively till the end, H-mine(Mem) will analyze the basic characteristics of data to determine whether H-struct should be constructed or utilized in the subsequent mining or whether *FP-tree*s should be constructed for frequent-pattern growth.

4. Performance study and experimental results

To evaluate the efficiency and scalability of H-mine, we have performed an extensive performance study. In this section, we report our experimental results on the performance of H-mine in comparison with *Apriori* and *FP-growth*. It shows that H-mine outperforms *Apriori* and *FP-growth*

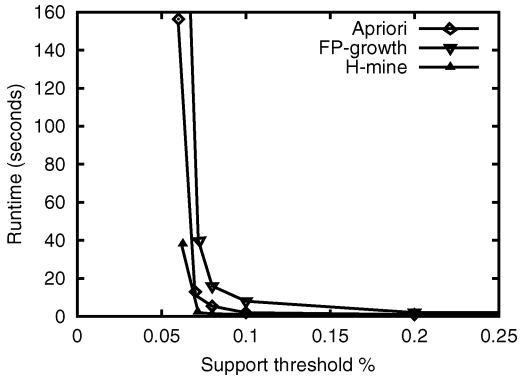

Fig. 7. Runtime on data set Gazelle.

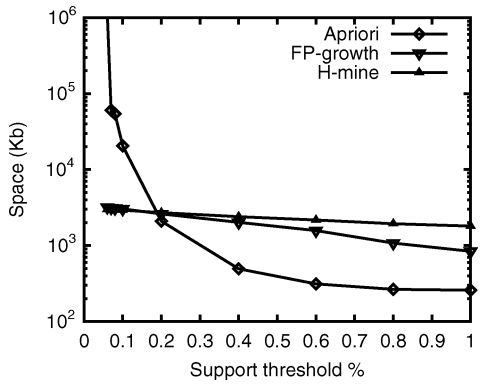

Fig. 8. Space usage on data set Gazelle.

and is efficient and highly scalable for mining very large databases.[3]

All the experiments were performed on a 466 MHz Pentium PC machine with 128 Mb main memory and 20 Gb hard disk, running Microsoft Windows/NT. H-mine and *FP-growth* were implemented by us using Visual C++6.0, while the version of *Apriori* that we used is a well-known version, *GNU Lesser General Public License*, available at http://fuzzy.cs.uni-magdeburg.de/~borgelt/. All reports of the runtime of H-mine include both the time of constructing H-struct and mining frequent-patterns. They also include both CPU time and I/O time.

We have tested various data sets, with consistent results. Limited by space, only the results on some typical data sets are reported here.

4.1. *Mining transaction databases in the main memory*

In this sub-section, we report results on mining transaction databases which can be held in the main memory. H-mine is implemented as stated in Section 2. For *FP-growth*, the *FP-tree*s can be held in the main memory in the tests reported in this sub-section. We modified the source code for *Apriori* so that the transactions are loaded into the main memory and the multiple scans of database are pursued in the main memory.

Data set Gazelle is a sparse data set. It is a web store visit (clickstream) data set from Gazelle.com. It contains 59 602 transactions, with up to 267 items per transaction.

Figure 7 shows the run-times of H-mine, *Apriori* and *FP-growth* on this data set. Clearly, H-mine is superior to the other two algorithms, and the differences (in term of seconds) become larger as the support threshold goes lower.

[3]A prototype of H-mine has also been tested by a third party in the US (a commercial company) on business data. Their results are consistent with ours. They observed that H-mine is more than ten times faster than *Apriori* and other participating methods in their test when the support threshold is low.

Apriori works well for sparse data sets since most of the candidates that *Apriori* generates turn out to be frequent patterns. However, it has to construct a hashing tree for the candidates and match them in the tree and update their counts each time when scanning a transaction that contains the candidates. That is the major cost for *Apriori*.

FP-growth has a similar performance to that of *Apriori* and sometimes it is even slightly worse. This is because when the database is sparse, the *FP-tree* cannot compress data as effectively as it can for dense data sets. Constructing *FP-tree*s over sparse data sets recursively has its overhead.

Figure 8 plots the high water mark of space usage for H-mine, *Apriori* and *FP-growth* in the mining procedure. To make the comparison clear, the space usage (Y axis) has a logarithmic scale. From the figure, we can see that H-mine and *FP-growth* have similar space requirements and are very scalable in term of space usage with respect to the support threshold. Even when the support threshold reduces to very low levels, the memory usage is still stable and moderate.

The memory usage of *Apriori* does not scale well as the support threshold goes down. *Apriori* has to store level-wise frequent patterns and generate next level candidates. When the support threshold is low, the number of frequent patterns as well as that of candidates are non-trivial. In contrast, pattern-growth methods, including H-mine and *FP-growth* , do not need to store any frequent patterns or candidates. Once a pattern is found, it is output immediately and never read back.

What are the performances of these algorithms when applied to dense data sets? We use the synthetic data set generator described in Agrawal and Srikant (1994) to generate a data set $T25I15D10k$. This data set generator has been used in many studies on frequent pattern mining. We refer readers to Agrawal and Srikant (1994) for more details on the data set generation. Data set $T25I15D10k$ contains 10 000 transactions and each transaction has up to 25 items. There are 1000 items in the data set and the average longest potentially frequent itemset has 15 items. It is a relatively dense data set.

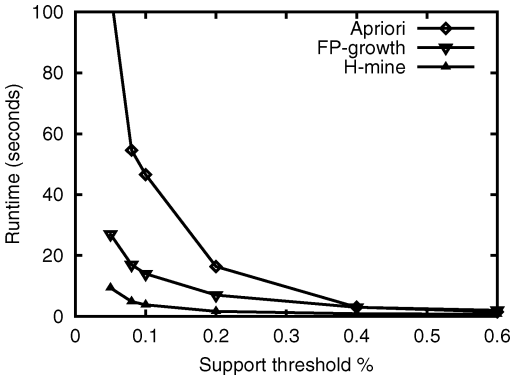

Fig. 9. Runtime on data set $T25I15D10k$.

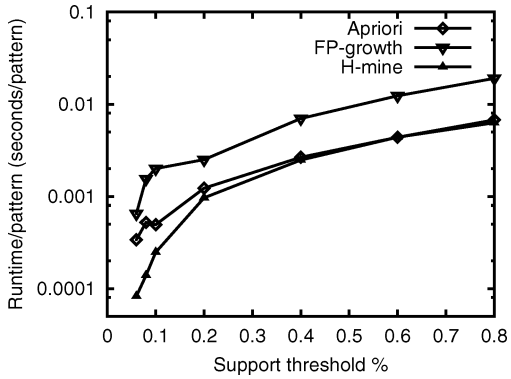

Fig. 11. Runtime per pattern on data set Gazelle.

Figure 9 shows the runtime of the three algorithms on this data set. When the support threshold is high, most patterns are of short lengths, *Apriori* and *FP-growth* have similar performances. When the support threshold becomes low, most items (more than 90%) are frequent. Then, *FP-growth* is much faster than *Apriori*. In all cases, H-mine is the fastest algorithm. It is more than ten times faster than *Apriori* and four to five times faster than *FP-growth*.

Figure 10 shows the high water mark of space usage of the three algorithms in mining this data set. Again, the space usage is plotted on a logarithmic scale. Since the number of patterns goes up dramatically as the support threshold goes down, *Apriori* requires an exponential amount of space. H-mine and *FP-growth* use a stable amount of space. For a dense data set, an *FP-tree* is smaller than the set of all frequent-item projections of the data set. However, long patterns mean more recursions and more recursive *FP-tree*s. This means that *FP-growth* will require more space than H-mine in this case. On the other hand, since the number of frequent items is large in this data set, an *FP-tree*, though compressing the database, still has many branches in various levels and becomes bushy. That also introduces non-trivial tree browsing cost.

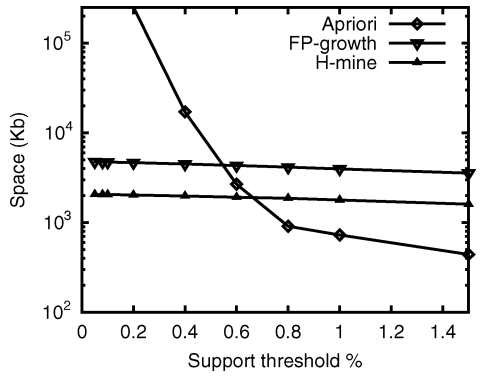

Fig. 10. Space usage on data set $T25I15D10k$.

Figures 11 and 12 explore the runtime per frequent pattern on data sets Gazelle and $T25I15D10k$, respectively. As the support threshold goes down, the number of frequent patterns goes up. As can be seen from the figures, the runtime per pattern of the three algorithms keeps going down. This observation explains the scalability of the three algorithms. Among the three algorithms, H-mine has the smallest runtime per pattern and thus has the best performance, especially when the support threshold is low.

In very dense data sets, such as *Connect-4* [4], and *pumsb*[5], H-mine builds *FP-trees* since the number of frequent items is very small. Thus, it has the same performance as *FP-growth*. Previous studies, e.g., Bayardo (1998), show that *Apriori* is incapable of mining such data sets.

We also tested the scalability of the algorithms with respect to the average number of items in transactions in the synthetic data sets. The experimental results are consistent with the results reported in Agrawal and Srikant (1995): as the average size goes up, the runtime goes up linearly. *FP-growth* and H-mine have a similar trend.

4.2. *Mining very large databases*

To test the efficiency and scalability of the algorithms to mine very large databases, we generated data set $T25I15D1280k$ using the synthetic data generator. It has 1280 000 transactions with similar statistical features to the data set $T25I15D10k$.

We enforced memory constraints on H-mine so that the total memory available is limited to 2, 4, 8 and 16 Mb, respectively. The memory covers the space for H-struct and all the header tables, as well as the related mechanisms. Since the *FP-tree* built for the data set is too big to fit into

[4]From UC-Irvine: (www.ics.uci.edu/∼mlearn/MLRepository.html)

[5]From IBM Almaden Research Center: www.almaden.ibm.com/cs/quest/demos.html

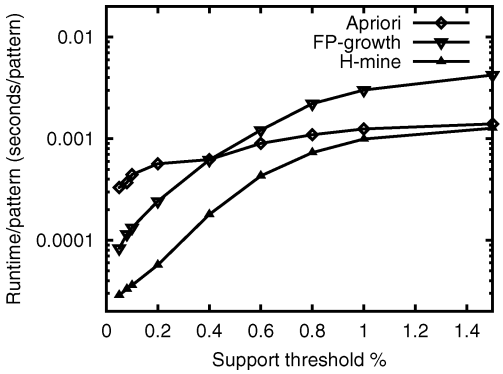

Fig. 12. Runtime per pattern on data set $T25I15D10k$.

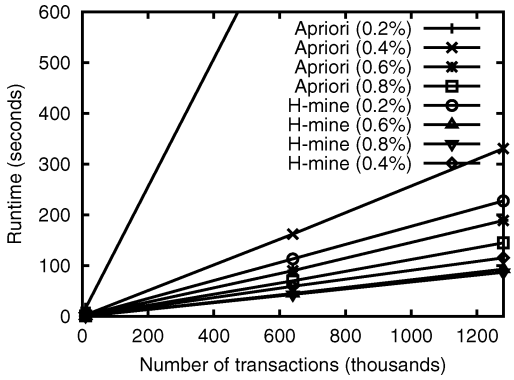

Fig. 13. Scalability with respect to the number of transactions.

the main memory, we do not report the performance of *FP-growth* on this data set. We do not explicitly impose any memory constraint on *Apriori*.

Figure 13 shows the scalability of both H-mine (with the main memory size constrained to be 2 Mb) and *Apriori* with respect to the number of transactions in the database. Various support threshold settings were tested. Both algorithms have a linear scalability and H-mine is a clear winner. From the figure, we can see that H-mine is more efficient and scalable at mining very large databases.

To study the effect of the memory size constraints on the mining efficiency and scalability of H-mine in large databases, we plot Fig. 14. The figure shows the scalability of H-mine with respect to the support threshold with various memory constraints, i.e., 2, 4, 8 and 16 Mb, respectively. As shown in the figure, the runtime is not sensitive to the memory limitation when the support threshold is high. When the support threshold goes down, as available space increases, the performance improves.

Figure 15 shows the effect of available memory size on mining large data sets. At high support levels, the performance is not sensitive to the available memory size and thus the number of partitions. When the support threshold is low, the memory size plays an important role in determining performance.

With a high support threshold, the number of frequent patterns is small and most frequent patterns are short. The dominant cost is the I/O cost and thus it is insensitive to the size of the available memory. When the support threshold is low, with a larger available memory, H-mine has less partitions and thus generates fewer locally frequent patterns, i.e., the locally frequent patterns contain more globally frequent ones and less noise. Therefore, H-mine can run faster with more memory. The results show that H-mine can fully utilize the available memory to scale up the mining process.

Does H-mine have to check all or most of the locally frequent patterns against the whole database in its third scan of the database? Fortunately, the answer is no. Our experimental results show that H-mine has a very light workload in its third scan. We consider the ratio of the number of patterns to be checked in the third scan over that of all distinct locally frequent patterns, where a locally frequent pattern is

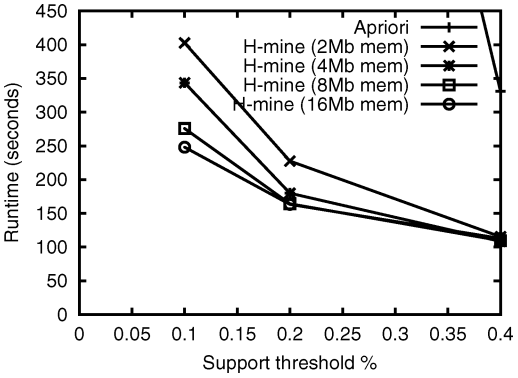

Fig. 14. Scalability of H-mine on large data set $T25I15D1280k$.

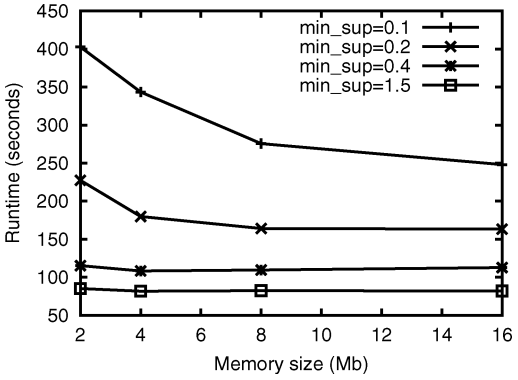

Fig. 15. Effect of memory size on mining a large data set.

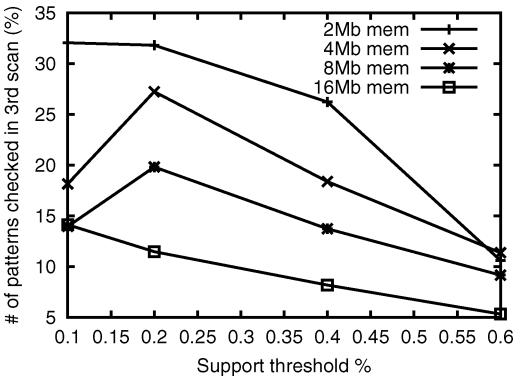

Fig. 16. The ratio of patterns to be checked by H-mine in the third scan.

to be checked in the third scan if it is not frequent in every partition. Figure 16 shows the ratio numbers. In general, as the support threshold goes down, the ratio goes up. This means that mining with a low support threshold may lead to some patterns being more frequent in certain partitions. On the other hand, less memory (small partition) leads to more partitions and also increases the ratio. As shown in the figure, only a limited portion of locally frequent patterns, e.g., less than 35% in our test case, needs to be tested in the third scan. This leads to a low cost of the third scan in our partition-based mining.

5. Conclusions

In this paper, we develop a simple and novel hyper-linked data structure, H-struct, and a new frequent pattern mining algorithm, H-mine, which takes advantage of the H-struct data structure and dynamically adjusts links in the mining process. As shown in our performance study, H-mine has a high performance and is scalable in many kinds of data, with a very limited and precisely predictable main memory overhead, and outperforms currently existing algorithms with various settings.

References

Agarwal, R.C., Agarwal, C.C. and Prasad, V.V.V. (2000) Depth first generation of long patterns, in *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ACM Press, Boston, MA, pp. 108–118.

Agarwal, R.C., Agarwal, C.C. and Prasad, V.V.V. (2001) A tree projection algorithm for generation of frequent item sets. *Journal of Parallel and Distributed Computing*, **61**(3), 350–371.

Agrawal, R., Gehrke, J., Gunopulos, D. and Raghavan, P. (1998) Automatic subspace clustering of high dimensional data for data mining applications, in *Proceedings of the 1998 ACM-SIGMOD International Conference on the Management of Data (SIGMOD'98)*, Seattle, WA, pp. 94–105.

Agrawal, R., Imielinski, T. and Swami, A. (1993) Mining association rules between sets of items in large databases, in *Proceedings of the 1993 ACM-SIGMOD International Conference on the Management of Data (SIGMOD'93)*, Washington, DC, pp. 207–216.

Agrawal, R. and Srikant, R. (1994) Fast algorithms for mining association rules, in *Proceedings of the 1994 International Conference on Very Large Data Bases (VLDB'94)*, Santiago de Chile, Chile, pp. 487–499.

Agrawal, R. and Srikant, R. (1995) Mining sequential patterns, in *Proceedings of the 1995 International Conference on the Data Engineering (ICDE'95)*, pp. 3–14.

Bayardo, R.J. (1998) Efficiently mining long patterns from databases, in *Proceedings of the 1998 ACM- SIGMOD International Conference on the Management of Data (SIGMOD'98)*, Seattle, WA, pp. 85–93.

Bayardo, R.J., Agrawal, R. and Gunopulos, D. (1999) Constraint-based rule mining on large, dense data sets, in *Proceedings of the 1999 International Conference on the Data Engineering (ICDE'99)*, Sydney, Australia, pp. 188–197.

Brin, S., Motwani, R. and Silverstein, C. (1997) Beyond market basket: Generalizing association rules to correlations, in *Proceedings of the 1997 ACM-SIGMOD International Conference on the Management of Data (SIGMOD'97)*, Tucson, AZ, pp. 265–276.

Dong, G. and Li, J. (1999) Efficient mining of emerging patterns: discovering trends and differences, in *Proceedings of the 1999 International Conference on Knowledge Discovery and Data Mining (KDD'99)*, San Diego, CA, pp. 43–52.

Han, J., Dong, G. and Yin, Y. (1999) Efficient mining of partial periodic patterns in time series database, in *Proceedings of the 1999 International Conference on Data Engineering (ICDE'99)*, Sydney, Australia, pp. 106–115.

Han, J., Pei, J. and Yin, Y. (2000) Mining frequent patterns without candidate generation, in *Proceedings of the 2000 ACM-SIGMOD International Conference on the Management of Data (SIGMOD'00)*, Dallas, TX, pp. 1–12.

Kamber, M., Han, J. and Chiang, J.Y. (1997) Metarule-guided mining of multi-dimensional association rules using data cubes, in *Proceedings of the 1997 International Conference on Knowledge Discovery and Data Mining (KDD'97)*, Newport Beach, CA, pp. 207–210.

Liu, B., Hsu, W. and Ma, Y. (1998) Integrating classication and association rule mining, in *Proceedings of the 1998 International Conference on Knowledge Discovery and Data Mining (KDD'98)*, New York, NY, pp. 80–86.

Mannila, H., Toivonen, H. and Verkamo, A.I. (1997) Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, **1**, 259–289.

Pasquier, N., Bastide, Y., Taouil, R. and Lakhal, L. (1999) Discovering frequent closed itemsets for association rules, in *Proceedings of the 7th International Conference on Database Theory (ICDT'99)*, Jerusalem, Israel, pp. 398–416.

Pei, J., Han, J. and Mao, R. (2000) CLOSET: An efficient algorithm for mining frequent closed itemsets, in *Proceedings of the 2000 ACM-SIGMOD International Workshop on Data Mining and Knowledge Discovery (DMKD'00)*, Dallas, TX, pp. 11–20.

Savasere, A., Omiecinski, E. and Navathe, S. (1995) An efficient algorithm for mining association rules in large databases, in *Proceedings of the 1995 International Conference on Very Large Data Bases (VLDB'95)*, Zurich, Switzerland, pp. 432–443.

Silverstein, C., Brin, S., Motwani, R. and Ullman, J. (1998) Scalable techniques for mining causal structures, in *Proceedings of the 1998 International Conference on Very Large Data Bases (VLDB'98)*, New York, NY, pp. 594–605.

Biographies

Jian Pei is an Assistant Professor of Computing Science at Simon Fraser University, Canada. His research interests are in various techniques of data mining, data warehousing, online analytical processing, and database systems, as well as their applications in bioinformatics.

Jiawei Han is Professor of Computer Science, University of Illinois at Urbana-Champaign. He has been working on research into data mining, data warehousing, stream data mining, spatiotemporal and multimedia data mining, biological data mining, social network analysis, text and Web mining, and software bug mining, with over 300 publications.

Hongjun Lu was Professor of Computer Science at the Hong Kong University of Science and Technology. He made numerous and notable contributions in database and knowledge base management systems, query processing and optimization, physical database design, database performance, data warehousing, and data mining, with over 200 publications.

Shojiro Nishio is a full professor at Osaka University, where he is currently serving as the Dean of the Graduate School of Information Science and Technology. He is also acting as the Program Director in the Area of Information and Networking, Ministry of Education, Culture, Sports,

Science and Technology (MEXT), Japan. Dr. Nishio has co-authored or co-edited more than 25 books, and authored or co-authored more than 200 refereed journal or conference papers, most of them in the field of databases.

Shiwei Tang, with School of Electronics Engineering and Computer Science, Peking University, is the Vice Chairman of China Computer Federation Database Society. His research interests include OLAP and data mining, Web information processing, database system implementation techniques, and database technology in specific application fields.

Dongqing Yang is the Director of Database Research Lab and the Vice Director of Institute of Networks and Information Systems in the School of Electronics Engineering and Computer Science, Peking University. Her research interests include database system implementation techniques, information integration and sharing in Web environment, data warehousing and data mining.