H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases

Jian Pei[†‡], Jiawei Han[‡], Hongjun Lu[*], Shojiro Nishio[§], Shiwei Tang[†], Dongqing Yang[†]
[†] Peking University, Beijing, China (pei@db.pku.edu.cn, {tsw, dqyang}@pku.edu.cn)
[‡] Simon Fraser University, B.C., Canada ({peijian, han}@cs.sfu.ca)
[*] Hong Kong University of Science and Technology, Hong Kong (luhj@cs.ust.hk)
[§] Osaka University, Osaka, Japan (nishio@ise.eng.osaka-u.ac.jp)

Abstract

Methods for efficient mining of frequent patterns have been studied extensively by many researchers. However, the previously proposed methods still encounter some performance bottlenecks when mining databases with different data characteristics, such as dense vs. sparse, long vs. short patterns, memory-based vs. disk-based, etc.

In this study, we propose a simple and novel hyper-linked data structure, 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 very limited and precisely predictable space overhead and runs really fast in memory-based setting. Moreover, it can be scaled up to very large databases by database partitioning, and 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 high performance in various kinds of data, outperforms the previously developed algorithms in different settings, and is highly scalable in mining large databases. This study also proposes a new data mining methodology,* space-preserving mining*, which may have strong impact in the future development of efficient and scalable data mining methods.*

1 Introduction

As an important data mining problem, frequent pattern mining plays an essential role in many data mining tasks, such as mining associations [3, 8], sequential patterns [15, 12], max-patterns and frequent closed patterns [4, 11, 17], classification [7, 16], and clustering [2].

There have been many algorithms developed for fast mining of frequent patterns, which can be classified into two categories. The first category, *candidate generation-and-test approach*, such as Apriori [3] as well as many subsequent studies, are directly based on an anti-monotone *Apriori* property [3]: *if a pattern with k items is not frequent, any of its super-pattern with $(k+1)$ or more items can never be frequent*. A candidate-generation-and-test approach iteratively generates the set of candidate patterns of length $(k+1)$ from the set of frequent patterns of length k $(k \geq 1)$, and check their corresponding occurrence frequencies in the database.

The *Apriori* algorithm achieves good reduction on 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 huge numbers of candidates and taking many scans of large databases for frequency checking.

Recently, another category of methods, *pattern-growth methods*, such as *FP-growth* [6] and *TreeProjection* [1], 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.

Nevertheless, these proposed approaches may still encounter some difficulties in different cases.

First, *huge space is required to serve the mining*. 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 $50 \times \binom{100}{50}$ units of space is needed to store candidates. *FP-growth* [6] 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. None is superior in all the cases.

Second, *real databases contain all the cases.* Real data sets can be sparse and/or dense in different applications. For example, for telecommunication data analysis, calling patterns for home users vs. business users could be very different: some are frequent and dense (e.g., to family members and close friends), but some are huge and sparse. Similar situations arise for 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.

Third, *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 in main memory and easily cause thrashing.

This poses a new challenge: "*Can we work out a better method which is (1) efficient in all occasions (dense vs. sparse, huge vs. memory-based data sets), and (2) space requirement is small, even for very large databases?*"

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 in (main) memory. A simple, memory-based hyper-structure, H-struct, is designed for fast mining. H-mine(Mem) has polynomial space complexity and is thus more space efficient than pattern-growth methods like *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 very limited and exactly predictable space overhead and is faster than memory-based *Apriori* and *FP-growth*.

Then, 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 memory using H-mine(Mem), and then consolidating global frequent patterns.

Third, for dense data sets, H-mine is integrated with *FP-growth* dynamically by detecting the swapping condition and constructing FP-trees for efficient mining.

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* in all the occasions.

The remaining 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 discuss related issues and conclude our study in Section 5.

2 H-mine(Mem): Memory-Based Hyper-Structure 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.

We first define the problem of frequent pattern mining.

Definition 2.1 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.*

Our general idea of H-mine(Mem) is illustrated in the following example.

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

Trans 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

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

Following the *Apriori* property [3], 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 3. Let $freq(X)$ (the *frequent-item projection* of X) be the set of frequent items in itemset X. For the 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 *F-list*): a-c-d-e-g, the complete set of frequent patterns can be partitioned into 5 subsets as follows: (1) those containing item a; (2) those containing item c but no item a; (3) those containing item d but no item a nor c; (4) those containing item e but no item a nor c nor d; and (5) those containing only item g.

If the frequent-item projections of transactions in the database can be held in main memory, they can be organized as shown in Figure 1. 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*.

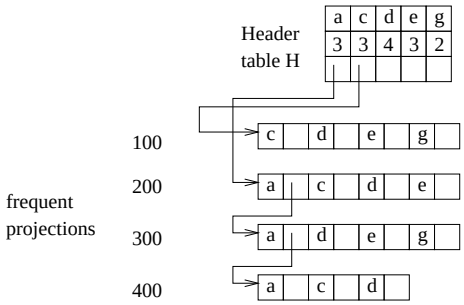

Figure 1. H-struct.

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

A *header table* H is created, where each frequent item entry has three fields: an *item-id*, a *support count*, and a *hyper-link*. When the frequent-item projections are loaded into memory, those with the same first item (in the order of *F-list*) are linked together by the hyper-links as 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 a-queue, which links frequent-item projections of transactions 100, 300, and 400. These three projections all have item a as their first frequent item (in the order of *F-list*). Similarly, frequent-item projection of transaction 100 is linked as *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 H-struct). Then the remaining of the 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 to search all the frequent-item projections containing item a, i.e., the *a-projected 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 Figure 2. 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 a-projected database (i.e., frequent-item projections in the a-queue), thus the support count in the entry c of H_a is 2.

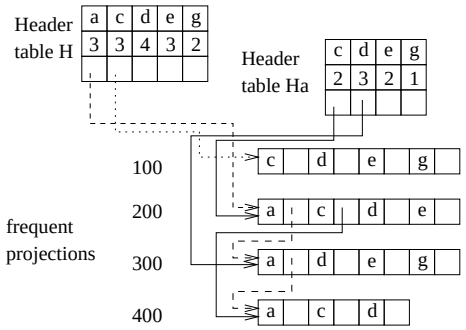

Figure 2. Header table H_a and ac-queue.

By traversing the a-queue once, the set of locally frequent items, i.e., the items appearing at least 2 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 H_a header as shown in Figure 2.

[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.

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 Figure 3.

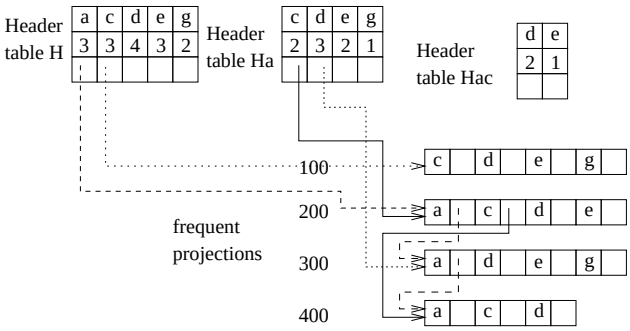

Figure 3. Header table H_{ac}.

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

Then the recursion backtracks to find patterns containing a and d but no 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 Figure 4.

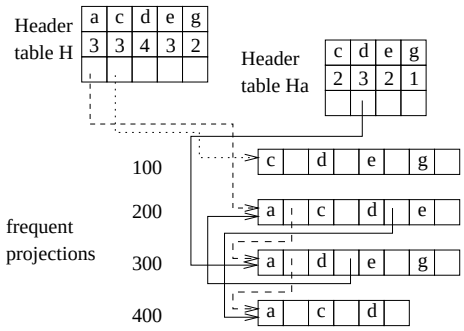

Figure 4. Header table H_a and ad-queue.

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 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 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 in the whole mining process.

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

After the frequent patterns containing item a are found, the a-projected database, i.e., a-queue, is no longer needed in the remaining of mining. Since the c-queue 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 no a, and other subsets of frequent patterns, we need to insert all the projections in the a-queue to 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 Figure 5. For example, frequent-item projection $acde$ is inserted into c-queue and $adeg$ is inserted into d-queue.

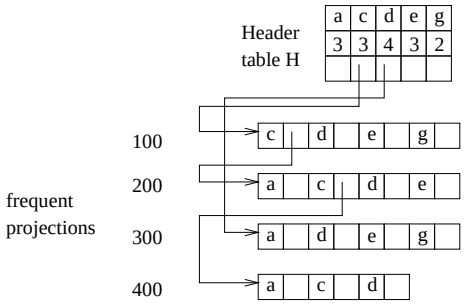

Figure 5. Adjusted hyper-links after mining a-projected database.

By mining the c-projected database recursively (with shared header table at each level), we can find the set of frequent patterns containing item c but no a. Notice item a will not be included in the c-projected database since all the frequent patterns having a have already been found.

Similarly, the mining goes on. It is easy to see 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.

The general idea of H-mine(Mem) is shown in the above example. Limited by space, we omit a formal presentation of H-mine(Mem) as well as a complexity analysis of the algorithm. 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 [6, 1]. H-mine(Mem) absorbs the advantages of pattern growth.

Second, H-mine(Mem) confines its search in a dedicated space. Unlike other frequent pattern growth methods, such as *FP-growth* [6], it does not need to physically construct memory structures of projected databases. It fully utilizes the information well organized in the H-struct, and collects information about projected databases using header tables, which are light-weight structures. That also saves a lot of efforts on managing space.

Third, H-mine(Mem) does not need to store any frequent patterns in memory. Once a frequent pattern is found, it is output to 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.

The above analysis is verified by our extensive performance study, as presented in Section 4.

3 From H-mine(Mem) to H-mine: Efficient Mining in 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 in main memory. Then, we explore how to integrate *FP-growth* when the data sets being mined become very dense.

3.1 H-mine: Mining 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 in main memory. However, we cannot expect this is always the case. When they cannot fit in 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, ..., 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 $min\_sup_i = \lfloor min\_sup \times \frac{n_i}{n} \rfloor$ (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 [13], 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 TDB_i's, we can gather the patterns in F_i's 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.

Note that our partition-based mining share some similarities with a *partitioned Apriori* method proposed in [13] 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.

As also indicated in [13], it is not easy to get a good partition scheme using the *partitioned Apriori* [13] 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.

On the other hand, 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, H-mine does not spend any effort to check them but the *partitioned Apriori* [13] does.

Furthermore, we can do better in consolidating globally frequent patterns from local ones, as illustrated in the following example.

Example 2 A large transaction database TDB is partitioned into four parts, P_1, P_2, P_3 and P_4. Let the support threshold be 100. The four parts are mined respectively using H-mine(Mem). The locally frequent patterns as well as the partition-ids where they are frequent are shown in Table 2. The accumulated support count for a pattern is the sum of support counts from partitions where the pattern is locally frequent.

Local freq. pat.	Partitions	Accumulated sup. cnt
ab	P_1, P_2, P_3, P_4	280
ac	P_1, P_2, P_3, P_4	320
ad	P_1, P_2, P_3, P_4	260
abc	P_1, P_3, P_4	120
$abcd$	P_1, P_4	40
...	...	...

Table 2. Local frequent patterns in partitions.

Pattern ab is frequent in all the partitions. Therefore, it is globally frequent. Its global support count is its accumulated support count, i.e., 280. So do patterns ac and ad.

Pattern abc is frequent in all partitions except in P_2. The accumulated support count of abc covers the occurrences of the pattern in partitions P_1, P_3 and P_4. Thus, the pattern should be checked only in P_2. The global support count of abc is its accumulated count plus its support count in P_2. Similarly, pattern $abcd$ need to be checked in only partitions P_2 and P_3.

In the third scan of H-mine, after scanning partition P_2, suppose the support count of pattern $abcd$ in partition P_2 is 20. Since $abcd$ is not frequent in partition P_3, its support count in P_3 must be less than the local support threshold. If the local support threshold is 30, we do not need to check pattern $abcd$ in partition P_3, since $abcd$ has no hope to be globally frequent.

As can be seen from the example, we have the following optimization methods on consolidating globally frequent patterns.

First, accumulate the global support count from local ones for the patterns frequent in every partition.

Second, only check the patterns against those partitions where they are infrequent.

Third, use local support thresholds to derive the upper bounds for the global support counts of locally frequent patterns. Only check those patterns whose upper bound pass the global support threshold.

With the above optimization, the number of patterns to be consolidated can be reduced dramatically. As shown in our experiments, when the data set is relatively evenly distributed, only up to 20% of locally frequent patterns have to be checked in the third scan of H-mine.

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

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

No matter how large the database is, it can be mined by at most three scans of the database: the first scan finds globally frequent items; the second mines 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.

One may wonder that, since the *partitioned Apriori* [13] takes two scans of TDB, whereas H-mine takes three scans, how can H-mine outperform the one proposed in [13]? 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 less partitions and mines each partition very fast, it has better overall performance than the Apriori-based partition mining algorithm. This is also demonstrated in our performance study.

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

As indicated in several studies [5, 6, 11], 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 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 [6] works well in dense databases with a large number of long patterns due to the effective compression of shared prefix paths in mining.

In comparison with *FP-growth*, H-mine does not generate physical projected databases and conditional *FP-trees* and thus saves space as well as time in many cases. However, *FP-tree*-based mining has its advantages over mining on H-struct since *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 *FP-tree* may not outperform mining on 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 *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 question becomes what should be the appropriate situations that one structure is more preferable over the other and 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 *high relative support*. The *relative support* can be computed as the ratio of absolute support against 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 *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 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.

4 Performance Study

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

All the experiments are performed on a 466MHz Pentium PC machine with 128 megabytes main memory and 20G hard disk, running Microsoft Windows/NT. H-mine and *FP-growth* are 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.

[3] A prototype of H-mine is also tested by a third party in US (a commercial company) on business data. Their results are consistent with ours. They observed that H-mine is more than 10 times faster than *Apriori* and other participating methods in their test when the support threshold is low.

4.1 Mining in main memory

In this sub-section, we report results on mining transaction databases which can be held in main memory. H-mine is implemented as stated in Section 2. For *FP-growth*, the *FP-tree*s can be held in main memory in the tests reported in this sub-section. We modified the source code for *Apriori* so that the transactions are loaded into main memory and the multiple scans of database are pursued in 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, while there are up to 267 item per transaction.

Figure 6 shows the run time of H-mine, *Apriori* and *FP-growth* on this data set. Clearly, H-mine wins the other two algorithms, and the gaps (in term of seconds) become larger as the support threshold goes lower.

Apriori works well in such 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 as *Apriori* and sometime is even slightly worse. This is because when the database is sparse, *FP-tree* cannot compress data as effectively as what it does on dense data sets. Constructing *FP-tree*s over sparse data sets recursively has its overhead.

Figure 7 plots the high water mark of space usage of H-mine, *Apriori* and *FP-growth* in the mining procedure. To make the comparison clear, the space usage (axis Y) is in logarithmic scale. From the figure, we can see that H-mine and *FP-growth* use similar space and are very scalable in term of space usage with respect to support threshold. Even when the support threshold reduces to very low, 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 performance of these algorithms over dense data sets? We use the synthetic data set generator described in [3] to generate a data set $T25I15D10k$, which contains $10,000$ transactions and each transaction has up to 25 items. There are $1,000$ items in the data set and the average longest potentially frequent itemset is with 15 items. It is a relatively dense data set.

Figure 8 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 performance. 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 one. It is more than 10 times faster than *Apriori* and 4-5 times faster than *FP-growth*.

Figure 9 shows the high water mark of space usage of the three algorithms in mining this data set. Again, the space usage is drawn in logarithmic scale. As the number of pat-

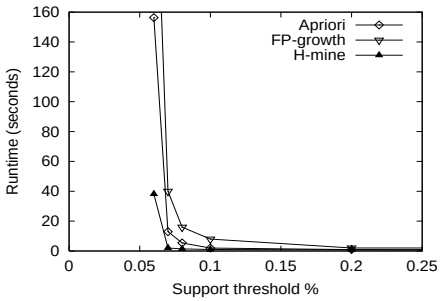

Figure 6. Runtime on data set Gazelle.

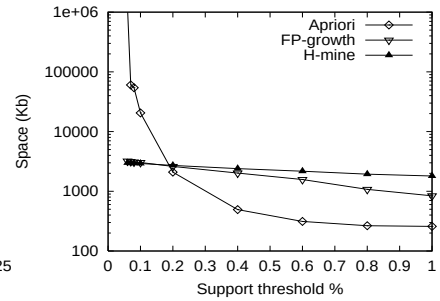

Figure 7. Space usage on data set Gazelle.

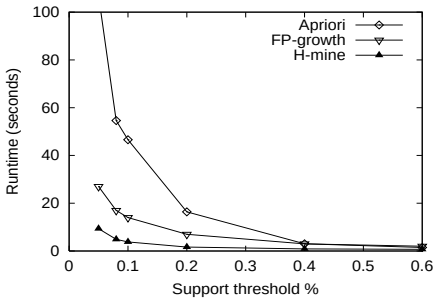

Figure 8. Runtime on data set $T25I15D10k$.

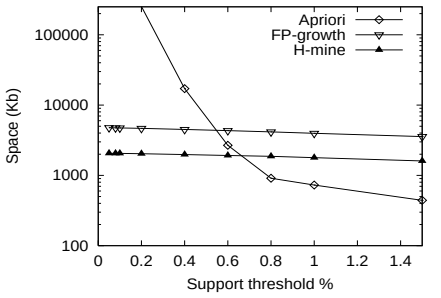

Figure 9. Space usage on data set $T25I15D10k$.

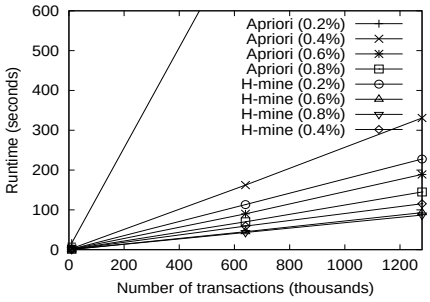

Figure 10. Scalability with respect to number of transactions.

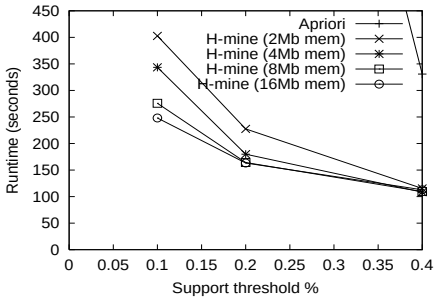

Figure 11. Scalability of H-mine on large data set $T25I15D1280k$.

terns goes up dramatically as support threshold goes down, *Apriori* requires an exponential amount of space. H-mine and *FP-growth* use stable amount of space. In dense data set, an *FP-tree* is smaller than the set of all frequent-item projections of the data set. However, long patterns means more recursions and more recursive *FP-tree*s. That makes *FP-growth* 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.

In very dense data set, such as *Connect-4* (from UC-Irvine: (www.ics.uci.edu/∼mlearn/MLRepository.html), and *pumsb* (from IBM Almaden Research Center: www.almaden.ibm.com/cs/quest/demos.html), H-mine builds *FP-tree*s since the numbers of frequent items are very small. Thus it has the same performance as *FP-growth.* Previous studies, e.g., [4], show that *Apriori* is incapable of mining such data sets.

4.2 Mining very large databases

To test the efficiency and scalability of the algorithms on mining very large databases, we generate data set $T25I15D1280k$ using the synthetic data generator. It has $1,280,000$ transactions with similar statistic features as the data set $T25I15D10k$.

We enforce memory constraints on H-mine so that the total memory available is limited to 2, 4, 8 and 16 megabytes, 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 in main memory, we do not report the performance of *FP-growth* on this data set. We do not explicitly compose any memory constraint on *Apriori*.

Figure 10 shows the scalability of both H-mine (with main memory size constraint 2 megabytes) and *Apriori* with respect to number of transactions in the database. Various support threshold settings are tested. Both algorithms have 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 memory size constraint on the mining efficiency and scalability of H-mine in large databases, we plot Figure 11. The figure shows the scalability of H-mine w.r.t. support threshold with various memory constraints, i.e., 2, 4, 8 and 16 megabytes, respectively. As shown in the figure, the runtime is not sensitive to the memory limitation when support threshold is high. When the support threshold goes down, as available space increases, performance gets better.

Our experimental results also show that H-mine has a very light workload in its third scan to consolidate global

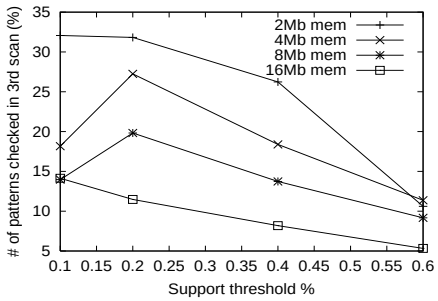

Figure 12. The ratio of patterns to be checked by H-mine **in the** 3**-rd scan.**

frequent patterns. 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 to be checked in the third scan if it is not frequent in every partition. Figure 12 shows the ratio numbers. In general, as the support threshold goes down, the ratio goes up. That means mining with low support threshold may lead to more patterns frequent in some partitions. On the other hand, less memory (small partition) leads to more partitions and also increase 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.

In summary, our experimental results and performance study verify our analysis and support our claim that H-mine is an efficient algorithm for mining frequent patterns. It is highly scalable in mining very large databases.

5 Discussion and Conclusions

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

A major distinction of H-mine from the previously proposed methods is that H-mine re-adjusts the links at mining different "projected" databases and has very small space overhead, even counting temporary working space. H-mine absorbs the nice features of *FP-growth*. It is essentially a frequent-pattern growth approach since it partitions its search space according to both patterns to be searched for and the data set to be searched on, by a divide-and-conquer methodology, without generating and testing candidate patterns. However, unlike *FP-growth*, H-mine does not create any physical projected databases nor constructing conditional (local) *FP-tree*s. H-mine is not confined itself to H-struct only. Instead, it watches carefully the changes of data characteristics during mining and dynamically switches its data structure from H-struct to *FP-tree* and its mining algorithm from mining on H-struct to *FP-growth* when the data set becomes dense and the number of frequent items becomes small.

H-mine can be scaled-up to very large databases due to its small and precisely predictable run-time memory overhead and its database partitioned mining technique.

Based on the above analysis, one can see that H-mine represents a new, highly efficient and scalable mining method. Its *structure- and space-preserving mining* methodology may have strong impact on the development of new, efficient and scalable data mining methods for mining other kinds of patterns, such as closed-itemsets [11], max-patterns [4], sequential patterns [14, 12], constraint-based mining [9, 10], etc. This should be an interesting direction for further study.

References

[1] R. Agarwal, C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent itemsets. In *J. of Parallel and Distributed Computing (Special Issue on High Performance Data Mining)*, 2000.

[2] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *SIGMOD'98*, pages 94–105.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB'94*, pages 487–499.

[4] R. J. Bayardo. Efficiently mining long patterns from databases. In *SIGMOD'98*, pages 85–93.

[5] R. J. Bayardo, R. Agrawal, and D. Gunopulos. Constraint-based rule mining on large, dense data sets. In *ICDE'99*.

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

[7] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *KDD'98*, pages 80–86.

[8] H. Mannila, H. Toivonen, and A. I. Verkamo. Efficient algorithms for discovering association rules. In *KDD'94*, pages 181–192.

[9] 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.

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

[11] J. Pei, J. Han, and R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *Proc. 2000 ACM-SIGMOD Int. Workshop Data Mining and Knowledge Discovery (DMKD'00)*, pages 11–20.

[12] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *ICDE'01*, pages 215–224.

[13] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *VLDB'95*, pages 432–443.

[14] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In *SIGMOD'96*, pages 1–12.

[15] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *EDBT'96*, pages 3–17.

[16] K. Wang, S. Zhou, and S. C. Liew. Building hierarchical classifiers using class proximity. In *VLDB'99*, pages 363–374.

[17] M. Zaki. Generating non-redundant association rules. In *KDD'00*, pages 34–43.