ISSN:2231-0711

P.Siva et al | IJCSET(www.ijcset.net) | October 2016 | Vol 6, Issue 10, 349-355

H-MINE: Scalable Space Preserving Mining

[1] **P.Siva** , [2] **D.Geetha**

[1]*Research Scholar, Sree Saraswathi Thyagaraja College, Pollachi.*
[2]*Head & Assistant Professor, Department of Computer Application, Sree Saraswathi Thyagaraja College, Pollachi.*

Abstract – 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. H-mine has high performance, is scalable in all kinds of data, with very limited and predictable space overhead, and outperforms the previously developed algorithms with various settings. First, a major distinction of H-mine from the previously proposed methods is that H-mine re-adjusts the links when mining different "projected" databases and has very small space overhead, even counting temporary working space; whereas candidate generation and test has to generate and test a large number of candidate itemsets, and FP growth has to generate a good number of conditional (projected) databases and FP trees. The structure and space preserving philosophy of H-mine promotes the sharing of the existing structures in mining, reduces the cost of copying a large amount of data and building new data structures on such data, and reduces the cost of updating and checking such data structures as well.

Keywords: Hyper Structure mining, Frequent Pattern Mining, FP Tree, H-MINE.

1. INTRODUCTION

H-mine(Mem) (memory based hyper structure mining of frequent patterns) is the method which is extended to handle large and/or dense databases.

General idea of H-mine(Mem)

Our general idea of H-mine(Mem) is illustrated in the following example. Let the first two columns of Table-1 be our running transaction database TDB. Let the minimum support threshold be min-sup = 2.

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

Table-1: The transaction database *TDB*.

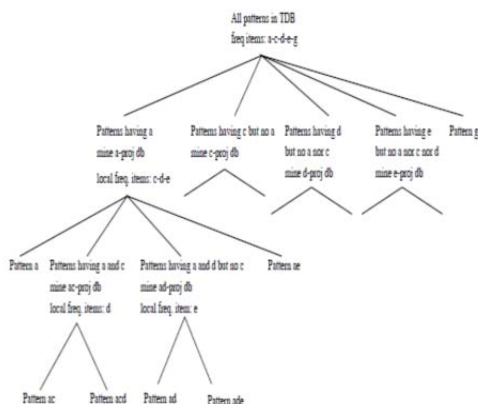

Figure-1: Divide and conquer tree for frequent patterns.

Following the Apriori property, 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 item set 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 items1 (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, as shown in Figure-1.

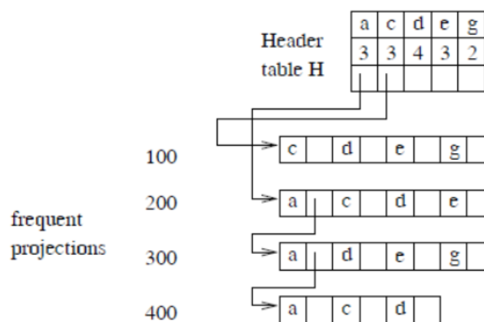

Figure-2: H-struct, the hyper structure for storing frequent item projections.

If the frequent item projections of transactions in the database can be held in main memory, they can be organized as shown in Figure-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, 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 200, 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.

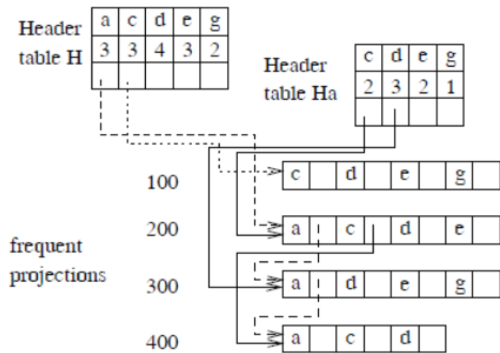

Figure-3: Header table H_a and ac-queue.

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 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, a-header table H_a is created, as shown in Figure-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 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.

By traversing the a-queue once, the set of locally frequent items, i.e., the items appearing at least twice, 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 4.3. Thus the set of frequent patterns containing item a can be further partitioned into four subsets: (1) the pattern a itself; (2) those containing a and c; (3) those containing a and d but no c; and (4) those containing a and e but no c nor d, i.e., pattern ae. The divide and conquer tree for frequent patterns is shown in Figure-1. These four subsets are mined as follows.

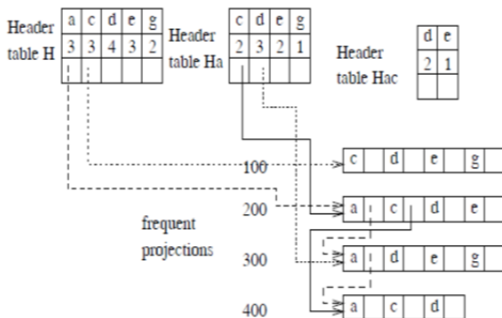

Figure-4: Header table H_{ac}.

1. The first subset contains only frequent pattern a.
2. The a-queue is traversed recursively to find frequent patterns containing a and c. First, the frequent item projections whose first local frequent item is a are linked together by the hyper-links as a queue, and the entries in the header table H_a act as the heads of the queues. Here, frequent item projections of transactions 200 and 400 are added to the ac-queue in any order. Transaction 300 is added to the ad-queue. At this moment, the ae-queue is empty. This situation is shown in Figure-3. The process continues recursively for the ac-projected database by examining the c-queue in H_a. This process creates the ac-header table Hac, as shown in Figure-4. Since only item d : 2 is a locally frequent item in the ac-projected database, only acd : 2 is output, and the search along this path is completed.

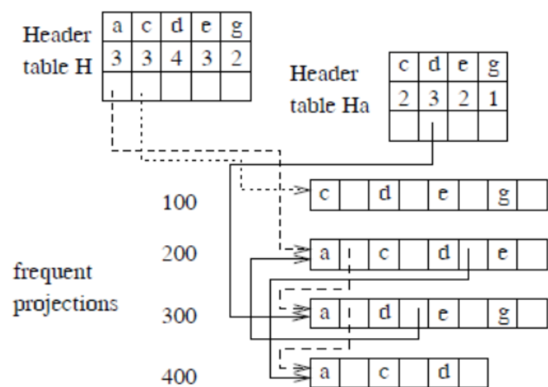

Figure-5: Header table H_a and ad-queue.

3. The recursion backtracks to find frequent 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 Figure-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 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 Had 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.

4. Since there is no transaction in the ae-projected database, the only frequent pattern in this projected database is ae itself. The search terminates.

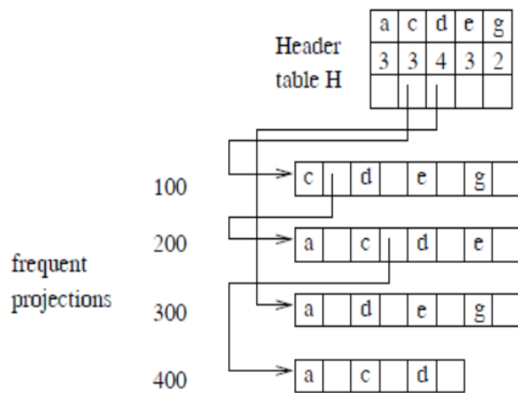

Figure-6: Adjusted hyper-links after mining a-projected database.

After the frequent patterns containing item a are found, the transactions in the a-projected database, i.e., a-queue, should be further projected to other projected databases. 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 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 Figure-6. For example, frequent item projection acde is inserted into c-queue and adeg is inserted into d-queue. 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.

2. H-MINE ALGORITHM

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 $x \prec y$ if and only if x is before y according to the F-list. For example, based on the F-list in Example, we have $a \prec c \prec d \prec e \prec g$. Frequent item projections of transactions in TDB are organized in an H-struct defined below.

1. 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.
2. An H-struct has a header table. The header table is an array of frequent items in the order of F-list. A support count and a hyper-link are attached to each item in the header table.
3. When the H-struct is created, items in the header table are the heads of queues of frequent-item projections linked by hyper-links.

Algorithm - (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: "$x_1, \ldots, x_n$" ($n = |L|$) 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 xi immediately.

Algorithm H-mine(P, H, F-list)

Step-1: traverse P-queue once, find and output its locally frequent items and derive F-list$_P$: "$x_{j1}, \ldots, x_{jn^1}$".

Step-2: construct header table HP, scan the P-projected database, and for each frequent item projection X in the projected database, use the hyper-link of x_{ji} $(1 \le i \le n^1)$ in X to link X to the Px_{ji}-queue in the header table H_P, where x_{ji} is the first locally frequent item in X according to the F-list$_P$.

Step-3: for i = 1 to n^1 do
(a) call $H - \min e(P \cup \{x_{ji}\}, H_P, \text{F-list}_P)$.
(b) traverse Px_{ji}-queue in the header table H_P, for each frequent-item projection X, link X to the x_{jk}-queue $(i < k \le n^1)$ in the header table H_P , where x_{jk} is the item in X following x_{ji} immediately according to F-list.

3. 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 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, $TDB_1, \ldots, TDB_k$, such that, for each TDB_i $(1 \le i \le k)$, 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 $min-sup_i = [min-sup \text{ x } \frac{n_i}{n}]$ (i.e., each partitioned database keeps the same relative minimum support as the global database). Let $F_i (1 \le i \le k)$ be the set of (locally) frequent patterns in TDB_i. Based on the property of partition based mining, P cannot be a (globally) frequent pattern in TDB with respect to the support threshold min sup if there exists no i $(1 \le i \le k)$ 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 as follows.

Algorithm (H-mine): Hyper-structure mining of frequent patterns in large Databases.

Input: A transaction database TDB and a support threshold min-sup.
Output: The complete set of frequent patterns.
Method:
Step-1: Scan 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 $(1 \le i \le k)$, the frequent-item projections in TDB_i can be held in main memory.
Step-3: For i = 1 to k, use H-mine(Mem) to mine frequent patterns in TDB_i with respectto the minimum support threshold

$min-sup_i = [\text{min-sup x } \frac{n_i}{n}]$, where n and n_i are the numbers 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. The only space cost of H-mine(Mem) incurred by the header tables. The maximal number of header tables as well as their space requirement is predictable (usually very small in comparison with the size of frequent-item projections). Therefore, after reserving space for header tables, the remaining main memory 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 available main memory for mining and the size of the overall frequent item projected database (in the scale of the sum of support counts of frequent items), and then partition the database relatively even to avoid the generation of skewed partitions.

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 frequent pattern	Partitions	Accumulated support count
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.

1. 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.
2. 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.
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.

1. Accumulate the global support count from local ones for the patterns frequent in every partition.
2. Only check the patterns against those partitions where they are infrequent.
3. 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.

- 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 takes two scans of TDB, whereas H-mine takes three scans. 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.

4. Handling Dense Data Sets: Dynamic Integration Of H-Struct And Fp Tree Based Mining

As indicated in several studies, 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 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 follows:

$$\text{relative support} = \frac{\text{absolute support}}{\text{\# of tran (or freq-item projections) in the (projected) database}}$$

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, for relative support values in between, it is not clear which method would be more efficient.

5. Performance Analysis

To evaluate the efficiency and scalability of H-mine, we have performed an extensive performance study.

Mining transaction databases in main memory

We report results on mining transaction databases which can be held in main memory. For FP growth, the FP trees can be held in main memory in the tests reported below. Data set Gazelle is a sparse data set. It is a web store visit (click stream) data set from Gazelle.com. It contains 59, 602 transactions, while there are up to 267 item per transaction. Figure-7 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 a dense data set. Constructing FP trees over sparse data sets recursively has its overhead.

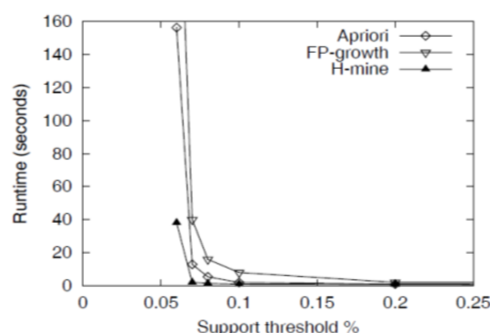

Figure-7: Runtime on data set Gazelle.

Figure-8 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 is 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. We use the synthetic data set generator to generate a data set T25I15D10k. The data set generator has been used in many studies on frequent pattern mining. Data set T25I15D10k 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.

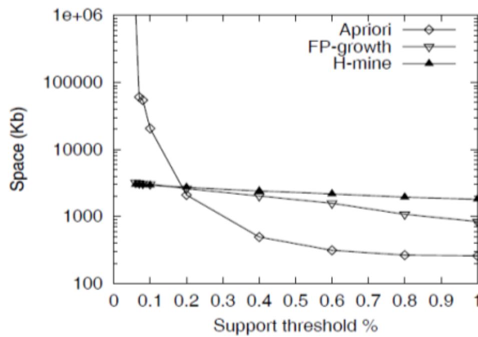

Figure-8: Space usage on data set Gazelle.

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 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-10 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 patterns 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 mean more recursions and more recursive FP trees. 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.

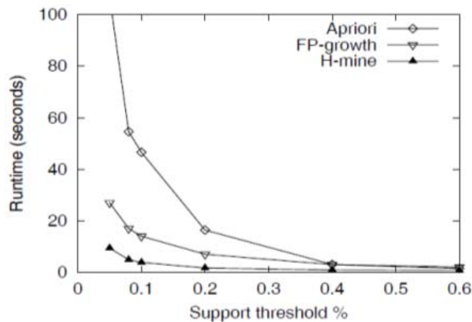

Figure-9: Runtime on data set T25I15D10k.

Figure-11 and Figure-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. That explains the scalability of the three algorithms. Among the three algorithms, H-mine has the least runtime per pattern and thus has the best performance, especially when support threshold is low. The figures also illustrate that H-mine is scalable with respect to the number of frequent patterns.

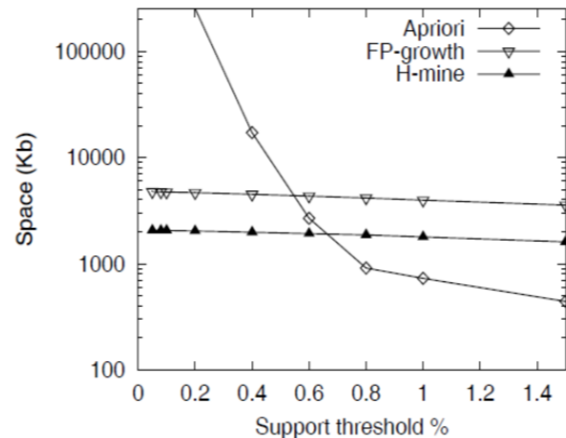

Figure-10: Space usage on data set T25I15D10k.

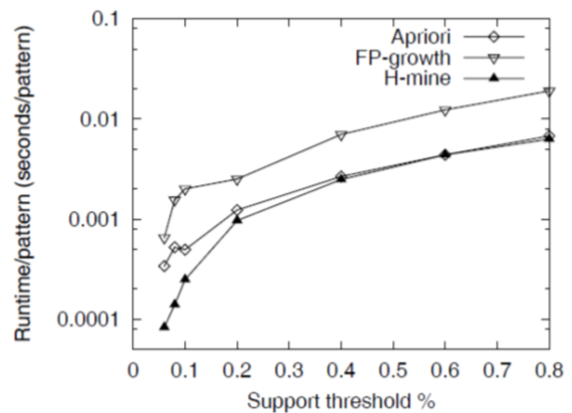

Figure-11: Runtime per pattern on data set Gazelle.

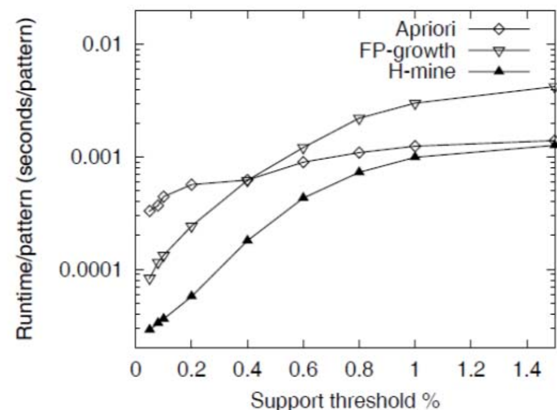

Figure-12: Runtime per pattern on data set T25I15D10k.

6. CONCLUSION

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 and data based on 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 trees. Instead, it builds and adjusts links dynamically among frequent items during mining to achieve the same effect as construction of physical projected databases. It avoids paying the cost of space and time for the (projected) database re-construction, and thus has better performance than FP growth. 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. This absorbs the benefits of FP growth which explores data compression and prefix path shared mining. Since mining on H-struct and mining on FP tree are built based on the same frequent pattern growth methodology, such a dynamic algorithm swapping can be performed naturally and easily.

REFERENCES

1. R. Agrawal, and R. Srikant, 1995. "Mining Sequential Patterns", Proc. of the Int'l Conference on Data Engineering (ICDE), Taipei, Taiwan.
2. R. Srikant , and R. Agrawal,1996. "Mining Sequential Patterns: Generalizations and performance improvements", In Proc. 5th Int'l Conference Extending Database Technology (EDBT), Avignon, France.
3. W. Li, 2001. "Classification Based on Multiple Association Rules". M.Sc. Thesis, Simon Fraser University.
4. J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu, 2001. "PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth", Proc. 2001 Int. Conf. on Data Engineering (ICDE'01), Heidelberg, Germany.
5. J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu, 2000. "FreeSpan: Frequent Pattern-Projected Sequential Pattern Mining", Proc. 2000 Int. Conf. Knowledge Discovery and Data Mining (KDD'00), Boston, MA.
6. M. J. Zaki, 2001. "SPADE: An Efficient Algorithm for Mining Frequent Sequences", in Machine Learning Journal, special issue on Unsupervised Learning (Doug Fisher, ed.), pages 31-60, Vol. 42 Nos. 1/2.
7. R. Agrawal, T. Imielinski, and A. Swami, 1993. "Mining association rules between sets of items in large databases.", Proc. of the ACM SIGMOD Conference on Management of Data, pages 207-216, Washington, D.C., May 1993.
8. M. J. Zaki, and C. Hsiao, 1999. "CHARM: An Efficient Algorithm for Closed Association Rule Mining", in Technical Report 99-10, Computer Science, Rensselaer Polytechnic Institute.
9. J. Pei, J. Han, and R. Mao, 2000. "CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets.", ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery 2000, pages 21-30, Dallas, TX, 2000.
10. Bjorn Bringmann,2009. "Mining Patterns in Structured data". PhD thesis, Katholieke University Leuven, Belgium.
11. Wei Wang and Jiong Yang, 2005. "Mining high-dimensional data". In Oded Maimon and Lior Rokach, editors, Data Mining and Knowledge Discovery Handbook, pages 793–799. Springer, 2005.
12. Marisa Thoma, Hong Cheng, Arthur Gretton, Jiawei Han, Hans Peter Kriegel, Alex Smola, Le Song, Philip S. Yu, Xifeng Yan, and Karsten Borgwardt, 2009. "Near optimal supervised feature selection among frequent sub graphs. In Proceedings of the 9th SIAM International Conference on Data Mining, pages 1075–1086.
13. Diane J. Cook and Lawrence B. Holder,1994. "Substructure discovery using minimum description length and background knowledge". Journal of Artificial Intelligence Research, 1:231–255.
14. Christian Borgelt, 2002. "Mining molecular fragments: Finding relevant substructures of molecules". In Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM), pages 51–58. IEEE Press, 2002.
15. Tias Guns, Siegfried Nijssen, and Luc De Raedt, 2011. "Itemset mining: A constraint programming perspective". Artif. Intell., 175(12-13):1951–1983, 2011.