Finding Frequent Itemsets by Transaction Mapping

Mingjun Song
Computer Science and Engineering
University of Connecticut
Storrs, CT 06269, U.S.A.
1-860-486-0849
mjsong@engr.uconn.edu

Sanguthevar Rajasekaran
Computer Science and Engineering
University of Connecticut
Storrs, CT 06269, U.S.A.
1-860-486-2428
rajasek@engr.uconn.edu

ABSTRACT

In this paper, we present a novel algorithm for mining complete frequent itemsets. This algorithm is referred to as the *TM algorithm* from hereon. In this algorithm, we employ the vertical representation of a database. Transaction ids of each itemset are mapped and compressed to continuous transaction intervals in a different space thus reducing the number of intersections. When the compression coefficient becomes smaller than the average number of comparisons for intervals intersection, the algorithm switches to transaction id intersection. We have evaluated the algorithm against two popular frequent itemset mining algorithms - FP-growth and dEclat using a variety of data sets with short and long frequent patterns. Experimental data show that the TM algorithm outperforms these two algorithms.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Data Mining

General Terms

Algorithms

Keywords

TM, Frequent Itemsets, Association Rule Mining

1. INTRODUCTION

The first algorithm for association rules mining was *AIS* [1] proposed together with the introduction of association rules mining, which was further improved later and called the *Apriori* algorithm [2]. Apriori algorithm employed a so-called downward closure property – if an itemset is not frequent, any superset containing this itemset cannot be frequent either. Based on Apriori algorithm, many others have been developed. For instance, AprioriTid algorithm [2] repeatedly replaces each transaction in the database with the set of candidate itemsets occurring in that transaction at each iteration; ArioriHybrid algorithm [2] combines both Apriori and AprioriTid; DHP (direct hashing and pruning) algorithm [3] reduced the size of candidate $k+1$-itemsets by hashing all $k+1$-subsets of each transaction to a hash table when counting the supports of candidate k-itemsets; DIC (dynamic itemset counting) algorithm [4] attempted to reduce the number of database scans by dividing the database into blocks and adding new candidate itemsets if all their subsets were already known to be frequent. Apriori algorithm needs many database scans and for each scan, frequent itemsets are searched by pattern matching, which is especially time-consuming for large frequent itemsets with long patterns. Partition algorithm [5] was another Apriori-like algorithm that used the intersection of transaction ids (*tids*) to determine the support count. Because it performed a breadth-first search, it partitioned the database into blocks in which local frequent itemsets were found to overcome the memory limitations. An extra database scan was needed to determine the global frequency of local frequent itemsets. FP-growth [6] is a well-known algorithm that uses the FP-tree data structure to achieve condensed representation of database transactions and employs a divide-and-conquer approach to decompose the mining problem into a set of smaller problems. Recursive construction of the FP-tree, however, affected the algorithm's performance. Eclat [7] was the first algorithm to find frequent patterns by a depth-first search and it has been shown to perform well. It used a vertical database representation and counted the itemset supports using the intersection of tids. However, because of the depth-first search, pruning used in Apriori algorithm was not available during the candidate itemsets generation. VIPER [8] and Mafia [9] also used the vertical database layout and the intersection to achieve a good performance. The only difference was that they used the compressed bitmaps to represent the transaction list of each itemset. However, their compression scheme has limitations especially when tids are uniformly distributed. Zaki and Gouda [10] developed a new approach called dEclat using the vertical database representation. They stored the difference of tids called *diffset* between a candidate k-itemset and its prefix $k-1$-frequent itemsets, instead of the tids intersection set, denoted here as *tidset*. This algorithm has been shown to gain significant performance improvements over Eclat. However, when the database is sparse, diffset will lose advantage over tidset.

In this paper, we present a novel approach using the vertical database layout and achieve significant improvements over existing algorithms by transaction mapping. We call the new algorithm the *TM* algorithm.

2. Lexicographic Prefix Tree

In this paper, we employ a lexicographic prefix tree data structure to efficiently generate candidate itemsets and test their frequency, which is very similar to the lexicographic tree used in the TreeProjection algorithm [11]. This tree structure was also used in many other algorithms such as Eclat [7]. An example of this tree is shown in figure 2-1. Each node in the tree stores a collection of frequent itemsets. The root contains all frequent 1-itemsets. Itemsets in level *l* (for any *l*) are frequent *l*-itemsets. Each edge in the tree is labeled with an item. Itemsets in any node are stored as singleton sets with the understanding that the actual itemset also contains all the items found on the edges from this node to the root. For example, consider the leftmost node in level 2 of the tree in Figure 2-1. There are four 2-itemsets in this node, namely, {1,2}, {1,3}, {1,4}, and {1,5}. The singleton sets in each node of the tree are stored in lexicographic order. If the root contains {1}, {2}, .., {n}, then, the nodes in level 2 will contain {2}{3}…{n}; {3}{4}…{n}; ..; {n}, and so on. For each candidate itemset, we also store a list of transaction ids (i.e., ids of transactions in which all the items of the itemset occur). This tree will not be generated in full. The tree is generated in a depth first manner. As the search progresses, if the expansion of a node cannot possibly lead to the discovery of itemsets that have minimum support, then the node will not be expanded and the search will backtrack. When a frequent itemset that meets the minimum support requirement is found, it is output. Candidate itemsets generated by depth first search are the same as those generated by the joining step (without pruning) of the Apriori algorithm.

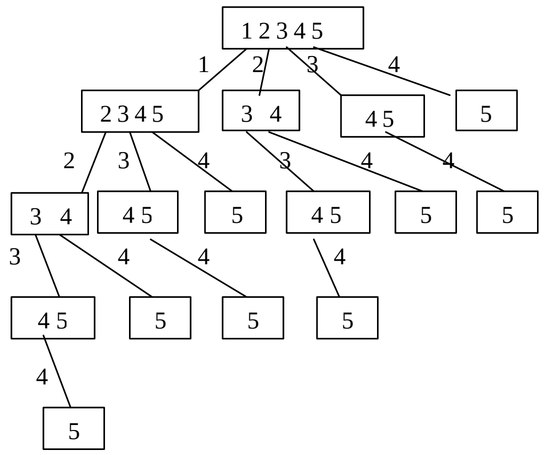

Figure 2-1. Illustration of lexicographic tree

3. TRANSACTION MAPPING AND COMPRESSION

3.1 Transaction Mapping

In our algorithm, we compress tids (transaction ids) for each itemset by mapping transactions into a different space appealing to a transaction tree. This transaction tree is similar to FP-tree [6] except that there is no header table or node link. The transaction tree can be thought of as a compact representation of all the transactions in the database. Each node in the tree has an id corresponding to an item and a counter keeping the number of transactions that contain this item in this path. Adapted from [6], the construction of the transaction tree is as follows:

1) Scan through the database once and identify all frequent 1-itemsets and sort them in descending order of frequency. At the beginning the transaction tree consists of just a single node (which is a dummy root).

2) Scan through the database for a second time. For each transaction, select items that are in frequent 1-itemsets, sort them according to the order of frequent 1-itemsets and insert them into the transaction tree. For inserting an item, start from the root. At the beginning the root is the current node. In general, if the current node has a child node whose id is equal to this item, then just increment the count of this child by 1, otherwise create a new child node and set its counter as 1.

Table 3-1 and figure 3-1 illustrate the construction of a transaction tree. Table 3-1 shows an example of a transaction database and figure 3-1 displays the constructed transaction tree assuming the minimum support count is 2.

Table 3-1. Sample transaction database

TID	Items	Select ordered items
1	2,1,5,3,19,20	1,2,3
2	2,6,3	2,3
3	1,7,8	1
4	3,1,9,10	1,3
5	2,1,11,3,17,18	1,2,3
6	2,4,12	2,4
7	1,13,14	1
8	2,15,4,16	2,4

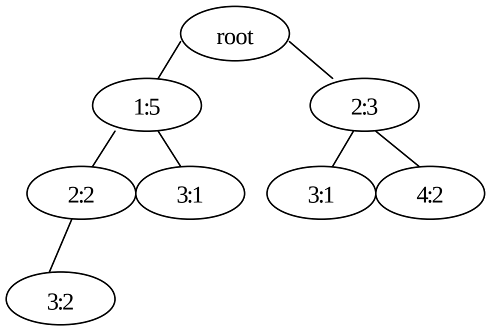

Figure 3-1. Example for transaction tree

After the transaction tree is constructed, ids of all the transactions that contain an item are represented with an interval list. Each interval corresponds to a contiguous sequence of relabeled ids. In Figure 3-1, item 1 is associated with the interval [1,5]. Item 2 is associated with the intervals [1,2] and [6,8]. A relabelling of the transaction ids is done to obtain compact intervals. For instance, in the above example, we relabel the transactions whose first select ordered item is 1 (1, 3, 4, 5, 7 in original TID) from 1 through 5. Transactions whose first two select ordered items are 1 and 2 (1 and 5 in original TID) are relabeled 1 and 2. Transactions whose first select ordered item is 2 (2, 6, 8 in original TID) are relabeled from 6 through 8, and so on. Consider a node u that has an associated interval of [*s*, *e*], where *s* is the relabeled start id, *e* is the relabeled end id. This node corresponds to $c = e - s + 1$ transactions. Assume that u has *m* children with child i having c_i transactions, (for $i = 1, 2, .., m$). It is obvious that $\sum_{i=1}^{m} c_i \le c$. The intervals associated with the children of u are: $[s_1, e_1]$, $[s_2, e_2]$, ..., $[s_m, e_m]$ with the following relationships holding:

$$s_1 = s \quad (3\text{-}1)$$

$$s_i = e_{i-1} + 1, \text{ where } i = 2,3,...,m \quad (3\text{-}2)$$

$$c_i = e_i - s_i + 1 \quad (3\text{-}3)$$

This allocation was done recursively starting from the children of the root in a depth-first order. The compressed transaction id list of each item is ordered by the start id of each associated interval. The mapped transaction interval list for each item is shown in table 3-2. In addition, if two intervals are contiguous, they will be merged and replaced with a single interval. For instance, 1-3 of item 3 results from the merging of 1-2 and 2-3.

Table 3-2. Example of transaction mapping

Item	Mapped transaction interval list
1	1-5
2	1-2, 6-8
3	1-3, 6-6
4	7-8

We now summarize a procedure that computes the interval list for each item as follows: By depth first order, transverse the transaction tree. For each node, create an interval composed of a start id and an end id. If it is the first child of its parent, then the start id of the interval is equal to the start id of the parent and the end id is computed by adding the start id and its counter minus 1, as in formula 3-1. If not, the start id is assigned to the end id of its previous child plus 1, as in formula 3-2. Insert this interval to the interval list of the corresponding item.

3.2 Interval Lists Intersection

In addition to the items described above, each element of every node in the lexicographic tree also stores a transaction interval list (corresponding to the itemset denoted by the element). By constructing the lexicographic tree in depth-first order, the support count of the candidate itemset is computed by intersecting the mapped interval lists of two elements. In contrast, Eclat used a tid set intersection. Interval lists intersection is more efficient. Note that one interval list cannot partially contain or be partially contained in another interval list. There are only three possible relationships between any two intervals A = [s_1, e_1] and B = [s_2, e_2]. 1) $A \cap B = \emptyset$. 2) $A \supseteq B$. 3) $A \subseteq B$.

3.3 Details of the TM Algorithm

Now we provide details on the steps involved in the TM algorithm. 1) Scan through the database and identify all 1-frequent itemsets. 2) Construct the transaction tree with counts for each node. 3) Construct the transaction interval lists. Merge intervals if they are mergeable (i.e., if the intervals are contiguous). 4) Construct the lexicographic tree in a depth first manner. When processing a node in the tree, for every element in the node, the corresponding interval list is computed. As the search progresses, itemsets with enough support are output.

3.4 Switching

After a certain level of the lexicographic tree, the transaction interval lists of elements in any node will be expected to become scattered. There could be many transaction intervals that contain only single tids. At this point, interval representation will lose its advantage over single tid representation, because the intersection of two segments will use three comparisons in the worst case while the intersection of two single tids only needs one comparison. Therefore, we need to switch to the single tid representation at some point. Here, we define a coefficient of compression for one node in the lexicographic tree, denoted by *coeff*, as follows: Assume that a node has m elements. Let s_i represent the support of the *i*th element, l_i represent the size of the transaction list of the *i*th element, then $coeff = \frac{1}{m}\sum_{i=1}^{m}\frac{s_i}{l_i}$. For the intersection of two interval lists, the average number of comparisons is 2, so we will switch to tid set intersection when *coeff* < 2.

4. EXPERIMENTS AND PERFORMANCE EVALUATION

We used four sets of data, where two sets are synthetic data (T10I4D100K and T25I10D10K). The synthetic data resembles market basket data with short frequent patterns. The other two datasets are real data (Mushroom and Connect-4 data) which are dense in long frequent patterns. These data sets were often used in the previous study of association rules mining and were downloaded from FIMI'03 website http://fimi.cs.helsinki.fi/testdata.html and http://miles.cnuce.cnr.it/~palmeri/datam/DCI/datasets.php. Table 4-1 shows the characteristics of data with the number of items, average transaction length, and number of transactions.

Table 4-1. Characteristics of experiment data sets

Data	#items	avg. trans. length	#transactions
T10I4D100k	1000	10	100,000
T25I10D10K	1000	25	9,219
mushroom	120	23	8,124
Connect-4	130	43	67,557

All experiments were performed on a DELL 2.4GHz Pentium PC with 1G of memory, running Windows 2000. The TM algorithm was coded in C++ using std libraries and compiled in Visual C++. dEclat and FP-growth were and downloaded from http://www.cs.helsinki.fi/u/goethals/software, implemented by Goethals, B. FP-growth code was modified somewhat to read the whole database into memory at the beginning so that the comparison of all the three algorithms is fair. We did not compare with Eclat because it was shown that dEclat outperforms Eclat [10]. Both TM and dEclat used the same optimization techniques, such as early stopping: the intersection between two tid sets can be stopped if the number of mismatches in one set is greater than the support of this set minus the minimum support threshold; dynamic ordering [12]: reordering all the items in every node at each level of the lexicographic tree in ascending order of support; combination: if the support of some itemsets is equal to the support of its subset, then any combination of these itemsets will be frequent. All times shown include time of outputting all frequent itemsets. The results are shown in tables 4-2 through 4-5 and figures 4-1 through 4-5.

Table 4-2. Run time (s) for T10I4D100k data

support(%)	FP-growth	dEclat	TM
5	0.671	0.39	0.328
2	2.375	1.734	0.984
1	5.812	5.562	2.406
0.5	7.359	9.078	4.515
0.2	7.484	11.796	7.359
0.1	8.5	12.875	8.906
0.05	11.359	15.656	10.453
0.02	20.609	33.468	14.421
0.01	33.781	73.093	21.671

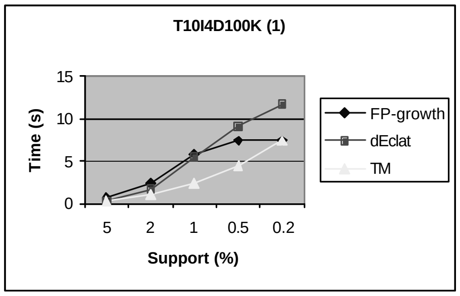

Figure 4-1. Run time for T10I4D100k data (1)

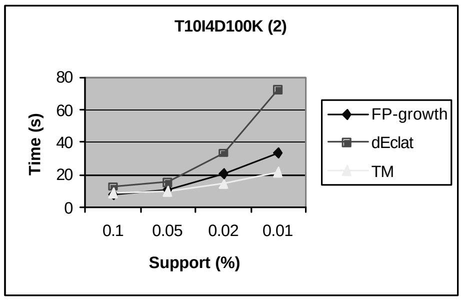

Figure 4-2. Run time for T10I4D100k data (2)

Table 4-2 shows the running time of the three algorithms on T10I4D100K data with different minimum support represented by percentage of the total transactions. Under large minimum support, dEclat runs faster than FP-Growth while slower than FP-Growth under small minimum support. TM algorithm runs faster than both algorithms under almost all minimum support values. On average, TM algorithm runs almost 2 times faster than the faster of FP-Growth and dEclat. To show the comparison more intuitively on a different scale, two graphs (Figure 4-1 and 4-2) are employed to display the performance comparison under large minimum support and small minimum support respectively.

Table 4-3. Run time (s) for T25I10D10K data

support(%)	FP-growth	dEclat	TM
5	0.25	0.14	0.093
2	3.093	3.203	1.109
1	4.406	4.921	2.718
0.5	5.187	5.296	3.953
0.2	10.328	6.937	5.656
0.1	31.219	20.953	10.906

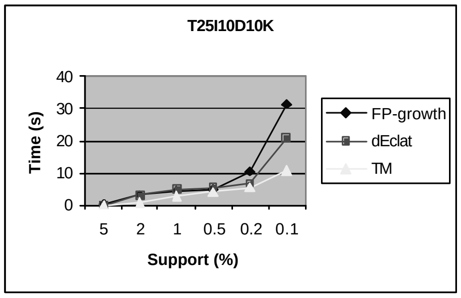

Figure 4-3. Run time for T25I10D10K data

Table 4-3 and Figure 4-3 show the performance comparison of the three algorithms on T2510D10K data in running time value and graph representation respectively. dEclat runs in general faster than FP-Growth with some exceptions at some minimum support. TM algorithm runs 2 times faster than dEclat on an average.

Table 4-4. Run time (s) for Mushroom data

support(%)	FP-growth	dEclat	TM
5	32.203	29.828	28.125
2	208.078	196.156	187.672
1	839.797	788.781	751.89
0.5	2822.11	2668.83	2640.83

Table 4-4 and figure 4-4 show the running time comparison of the three algorithms on mushroom data. dEclat is better than FP-Growth while TM is better than dEclat.

Table 4-5 and figure 4-5 show the performance comparison among the three algorithms on Connect-4 data. Connect-4 data is very dense so the smallest minimum support is 40 percent in this experiment. Similar to the result on mushroom data, dEclat is faster than FP-Growth while TM is faster than dEclat, though the difference is not significant.

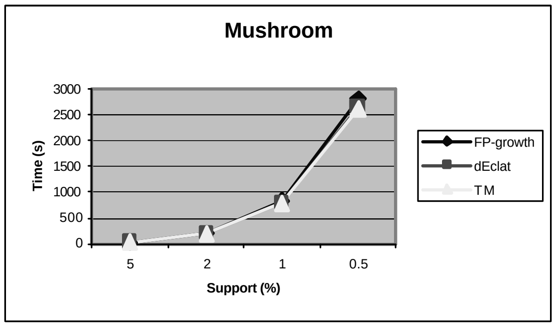

Figure 4-4. Run time for Mushroom data

Table 4-5. Run time (s) for Connect-4 data

support(%)	FP-growth	dEclat	TM
90	2.171	0.891	0.781
80	9.078	5.406	4.734
70	56.609	40.296	35.484
60	283.031	211.828	195.359
50	1204.67	935.109	871.359
40	4814.59	3870.64	3579.38

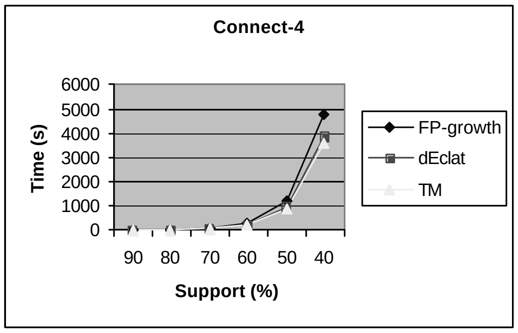

Figure 4-5. Run time for Connect-4 data

5. CONCLUSIONS

In this paper, we presented a new algorithm – TM algorithm using the vertical database representation. Transaction ids of each itemset are transformed and compressed to continuous transaction interval lists in a different space using the transaction tree, and frequent itemsets are found by transaction intervals intersection along a lexicographic tree in depth order. This compression greatly saves the intersection time, especially when the number of transactions is large with short frequent patterns. Through experiments, TM algorithm has been shown to gain significant performance improvement over FP-growth and dEclat on datasets with short frequent patterns, and some improvement on datasets with long frequent patterns.

6. REFERENCES

[1] Agrawal, R., Imielinski, T. and Swami, A.N. Mining association rules between sets of items in large databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, ACM Press, Washington DC, May 1993, 207-216.

[2] Agrawal, R.and Srikant, R. Fast algorithms for mining association rules. In *Proceedings of 20th International Conference on Very Large Data Bases*, Morgan Kaufmann, 1994, 487-499.

[3] Park, J.S., Chen, M.-S., and Yu, P.S. An effective hash based algorithm for mining association rules. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, ACM Press, San Jose, California, May 1995, 175-186.

[4] Brin, S., Motwani, R., Ullman, J.D., and Tsur, S. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, ACM Press, Tucson, Arizona, May 1997, 255-264.

[5] Savasere, A., Omiecinski, E., and Navathe, S. An efficient algorithm for mining association rules in large databases. In *Proceedings of 21th International Conference on Very Large Data Bases*. Morgan Kaufmann, 1995, 432-444.

[6] Han, J., Pei, J., and Yin, Y. Mining frequent patterns without candidate generation. In *Procedings of ACM SIGMOD Intnational Conference on Management of Data*, ACM Press, Dallas, Texas, May 2000, 1-12.

[7] Zaki, M.J., Parthasarathy, S., Ogihara, M., and Li, W. New algorithms for fast discovery of association rules. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, AAAI Press, 1997, 283-286.

[8] Shenoy, P., Haritsa, J. R., Sudarshan, S., Bhalotia, G., Bawa, M., and Shah, D. Turbo-charging vertical mining of large databases. In *Procedings of ACM SIGMOD Intnational Conference on Management of Data*, ACM Press, Dallas, Texas, May 2000, 22-23.

[9] Burdick, D., Calimlim, M., and Gehrke, J. MAFIA: a maximal frequent itemset algorithm for transactional databases. In *Proceedings of International Conference on Data Engineering*, Heidelberg, Germany, April 2001, 443-452.

[10] Zaki, M.J., and Gouda, K. Fast vertical mining using diffsets. In *Proceedings of the Nineth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Washington, D.C., ACM Press, New York, 2003, 326-335.

[11] Agrawal, R., Aggarwal, C., and Prasad, V. A Tree Projection Algorithm for Generation of Frequent Item Sets, *Parallel and Distributed Computing*, 2000, 350-371.

[12] Bayardo, R. J. Efficiently mining long patterns from databases. In *Procedings of ACM SIGMOD Intnational Conference on Management of Data*, ACM Press, Seattle, Washington, June 1998, 85-93.