

Summary

The paper we are presenting is “Turbo-charging Vertical Mining of Large Databases” by Pradeep Shenoy, Gaurav Bhalotia, Jayant R. Harista, Mayank Bawa, S. Sudarshan and Devevrat Shah .This introduces a new Association Rule mining algorithm “VIPER” that shows a significant improvement over the classical horizontal association rule mining algorithms that are dependent on the content and size of the underlying database. The new algorithm presented in this paper, “VIPER” does not depend on the underlying database type or size.

Previously “Association rules” mining algorithms used databases with horizontal data layout. In horizontal layout each row of data represents a customer transaction in terms of the items purchased under that transaction.

There is an alternative approach to data layout. In this approach each item is associated with a column that represents the transactions in which that item appears. This data layout is called Vertical layout. “Association rules” mining algorithms using this layout has a significant advantage over the horizontal data layout algorithms. They have smaller effective database size, compact storage of the database, better support of dynamic database. But all of these algorithms has the drawback that they perform well in some specific situation or assume some characteristics about the underlying database.

The new Vertical mining algorithm presented in this paper VIPER, (Vertical Itemset Partitioning for Efficient Rule-extraction) efficiently overcomes all these limitations.

1. Data Layout

VIPER uses vertical tid-vector(VTV) to represent items in the database. In VTV format database is organized as collection of columns where each column has an IID(item id) and a column of bits. Each position in the column represents a TID (transaction ID) and each bit represents the presence or absence of that TID in the database. An example of VTV follow here.

Suppose a database has 7 items - A, B, C , D, H, M, R and has 7 transaction and horizontal layout of the database is given as Fig 1.

TID	Item IDs				
1	1	2	3	4	5
2	3	4	6	7	
3	1	2	3	4	
4	1	2	3	6	
5	1	2	3	6	7
6	1	2	3	4	7
7	2	3	4	7	

Fig 1: Horizontal IID List

In Fig.1 item Ids of A, B, C, D, H, M and R is used which is usually done in “Market-basket” database. The correspondence between IID and item names is given in Table 1

Item Name	Item Id (IID)
A	1
B	2
C	3
D	4
H	5
M	6
R	7

Table 1: Relation Between Item Name and Item Id

The vertical tid-vector of dataset of Fig .1 is shown in Fig 2.

TID	IID						
	1	2	3	4	5	6	7
1	1	1	1	1	1	0	0
2	0	0	1	1	0	1	1
3	1	1	1	1	0	0	0
4	1	1	1	0	0	1	0
5	1	1	1	0	0	1	1
6	1	1	1	1	0	0	1
7	0	1	1	1	0	0	1

Fig 2. VTV format

Generally VTV format takes more space than VTL (vertical tid-list) . The VTL representation of the Fig 1 is given in Fig 3.

The occurrence of the items in the database are presented in VTV format. Since VTV takes more space than VTL, VIPER performs compression which takes advantage of the sparseness of the database. Same format is used to represent itemsets. Item and itemsets associated with compressed VTV is called “**snake**”.

For compression scheme used by VIPER is not Run-Length Encoding (RLE) would be assumed to be appropriate choice for compression. Because RLE works well when there are long runs of 0’s and 1’s. In the “market-basket” database there can be long runs of 1s and 0s. But in worst case all 1s can occur separately in which case RLE will double the data size. Because of this VIPER uses a new Technique named Skinning that uses Golomb Encoding Scheme.

	IID						
TID	1	2	3	4	5	6	7
1	1	1	1	1	2	2	
3	3	2	2		4	4	
4	4	3	3		5	5	
5	5	4	6			6	
6	6	5	7				
	7	6					
		7					

Fig 3. Vertical tid-List

2. Special Features of VIPER

VIPER implements a number of optimization that reduces the cost of search and intersection. For search optimization a technique called “equivalence class” clustering and for intersection optimization DAG structure is used. The description of these will be given with the algorithm.

Unlike other “Association Rules” mining algorithms VIPER is read and write algorithm. It may seem a drawback for the first time, but it significantly increases the later mining process and disk traffic is also less than the read-only algorithms.

3. VIPER Algorithm

The VIPER algorithm is stated in Fig 4.
Fig 4 is given so that the sequence of execution is clear.

3.1 First Pass:

In the first pass the database is read and HIL (horizontal item list as shown in Fig 1) is converted into snake format. During this process support of the items are determined and the items which does not meet the minimum support are removed and this way frequent 1-itemset and its accompanying snake is created.

Algorithm VIPER($D, I, minSup$)

```
1:Input :Horizontal Database (D), set of items(I)
2:output:Set of frequent itemset with Support (F)

3:F=countLevelOne(D);
4:F=F U counPairs(F1)

5:i=2; until (isEmpty(Fi )){

6:    candDAG=createDAG(level=i);

7:    Ci = Fi ;

8:    for k in i to 2i do {
9:        Ck+1 = FORC(Ck)
10:       if(isEmpty(Ck+1 )) break;
11:       candDAG=candDAG U Ck+1 ;
12:    }

13:   if (isEmpty(candDAG)) break;

14:   readList[i]=findReadList(candDAG);
15:   writeList[i]=writePrune(candDAG);

16:   FANGS(candDAG,readList[i],writeList[i]);

17:   for k in i+1 to 2i do
```

```

18:      F=F U frequentItems(candDAG,k);
19:  DeleteSnakes(writeList[i/2]);
20:  i=i*2;
    }
    }

```

Fig. 4 VIPER Algorithm

Line 3 in Fig 4, which show the algorithm, “countLevelOne” function represents this pass. If Fig 2 is considered and minimum support is 2 then after this pass the snakes that will be generated is shown in Fig 5. This figure shows snakes in a decompressed format and each column represents a snake.

1	2	3	4	6	7 <-----IID
1	1	1	1	0	0<-----VTV
0	0	1	1	1	1
1	1	1	1	0	0
1	1	1	0	1	0
1	1	1	0	1	1
1	1	1	1	0	1
0	1	1	1	0	1

Fig. 5 Frequent 1 Snakes (in uncompressed form)

For snake generation in this pass HIL database is read one row at a time. When a IID is found in a transaction it is passed to a routine that adds 0 bit all the TIDs from the last occurrence of that item in the database till the current TID. Then current TID is converted to 1.

3.2 Second Pass:

The job of second pass is to generate the frequent 2-itemset. To minimize the expense of this pass direct VTV intersection is avoided.

Disk resident frequent 1-snakes are read and decompressed. Snakes are then converted into "Horizontal tuples" in main memory. If Fig.5 is considered, the horizontal tuples for first row is {TID =1},{IID of A} , {IID of B},{IID of C}, {IID of D}. Second horizontal tuple will be as {TID=2}, {IID of C},{IID of D},{IID of M},{IID of R}.

Then for each tuple each pair of items in the tuple are enumerated and their count is updated. As example, for TID =1, the first horizontal tuple pairs , as shown below are made.

{IID of A}{IID of B},
{IID of A}{IID of C},
{IID of A}{IID of D},
{IID of B}{IID of C},
{IID of B}{IID of D},
{IID of C}{IID of D},

Then their counts are updated, and pairs the meet the support is kept. Thus frequent 2 itemset F_2 is created. In Fig 4 line 4 resembles this operation.

3.3 Subsequent Passes:

In the subsequent passes, in the first pass the frequent item set of the current level is used to generate candidate itemset of subsequent level using FORC algorithm.

$$C_{i+1} = \text{FORC}(F_i)$$

This represents the first pass of the loop at line 8.

From level $i+1, i+2 \dots 2i$ candidate itemset of current level is used to generate the candidate itemsets of subsequent pass.

The counting technique is capable of counting candidate itemsets of length $i+1$ to $2i$.

The itemsets generated are then inserted into a DAG structure. FANGS algorithm of line 16 performs the intersection and counting process. DAG structure used to hold the itemsets makes FANGS implementation easy. After some optimization the snakes to be read (ReadList) and snakes to be written (WriteList) in this pass are identified. The snakes in the ReadList are scanned in the memory and counts of

the candidate itemsets are generated by FANGS procedure and at the same time snakes in the WriteList are written in the disk.

Now various steps of this pass are described.

3.4 Snake Generation:

New snakes in vertical format can be generated just by ANDing the existing snaking. But since VTV takes larger space than VTL (vertical tid-list) after reading and decompressing the snakes into TID vectors they are on the fly converted into VTL format, ANDing is done in this format and then again VTL is converted into VTV, compression is done and snake is generated.

3.5 FORC (candidate generation Algorithms):

FORC uses “equivalent class clustering” for generation of candidate itemset. It takes frequent itemset or candidate itemset as input and generates the candidate itemset.

Suppose S_k is the input which can be candidate itemset or frequent itemset. It is supposed to generate C_{k+1} , $k+1$ length candidate itemset. Now S_k is clustered using equivalent class clustering. In equivalence class clustering itemsets in S_k with common $(k-1)$ prefix is stored in a hash table and the k -th items are stored in “extList” in ascending order.

This can be explained with an example. If we take Fig 5 into consideration we can see that at F2 will be as Fig 6. The whole F2 is not shown in the figure, only a part of it is stated.

12	13	14	16	17	23	
1	1	1	0	0	1	
0	0	0	0	0	0	
1	1	1	0	0	1	
1	1	0	1	0	1	etc
1	1	0	1	1	1	
1	1	1	0	1	1	
0	0	0	0	0	1	

Fig 7: Part of F2 (frequent 2 itemset)

If equivalence class clustering is explained on this data it will not explain the process properly. So one more step is made. F3 on the data created for F2 in Fig 6 is shown in Fig 7.

123	124	126	127	134	136	137	
1	1	0	0	1	0	0	
0	0	0	0	0	0	0	etc→
1	1	0	0	1	0	0	
1	0	1	0	0	1	0	
1	0	1	1	0	1	1	
1	1	0	1	1	0	1	
0	0	0	0	0	0	0	

Fig 8: Part of frequent 3 itemset (in decompressed form)

Suppose S_k is as given in Fig 8.

Here itemset with prefix 12 are 123,124,126,127. These items are clustered into common equivalence class g with common prefix $P_g=12$ and $extList_g=3,4,6,7$. Now candidates associated with each itemset in cluster g is to be found.

Suppose 124 is considered first. So 4's position in $extList_g$ is calculated. Its position is 2. So items with positions in $extList_g$ greater than 2 that means 3rd and 4th item is to be taken.

Now 3rd item 6 and 4th item 7 are added with 124. At this stage we get 1246 and 1247. Now it should be found whether the length 3 subsets of 1246 and 1247 are present in S_k (where $k=3$). Length 3 subsets of 1246 are 124,126,146,246 and 1247 are 124, 127, 147 and 247. 124 is excluded since 124 is the prefix of P_g . So the set of subsets 126, 146, 246, 127,147 and 247.

Suppose the presence or absence of 146 in S_3 is to be found. So cluster h with prefix P_h is accessed and its $extList_h$ is read to see whether 6 is present there or not. If 146 is present than 146 is taken otherwise it is ignored. With same access 147 can be checked too since it has the same prefix.

The advantage of using this approach over AprioriGen is that in AprioriGen the joinable itemsets are scattered all over the hash tree and searching the itemsets is expensive in AprioriGen. Also same subset of item can belong to more than one candidate set. AprioriGen will run the test for presence or absence of itemsets for duplicate itemsets.

In each pass of the database, candidate itemset are created by FORC which are presented as DAG structure. In DAG, frequent i -itemset are leaves at level i and at height r candidate itemset of length $i+r$ can be found. Fig 9 shows the DAG structure when partial request 2-itemset shown in Fig-7 is given as input.

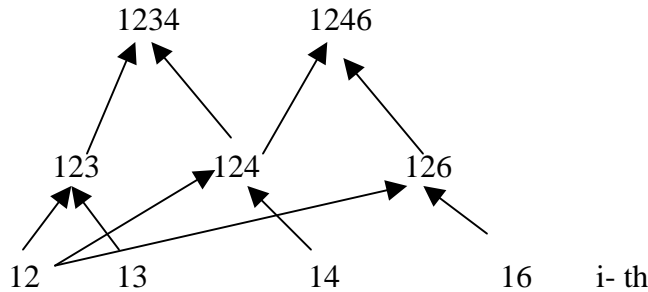


Fig 9: DAG Structure

If an itemset is a candidate all its itemset are also either candidate or frequent itemset. Each of candidate snake in DAG has an associated “latest TID”(LTID) variable and a “currentCount” (CCNT). These values are initialized to 0 and updated by FANGS procedure.

3.6 FANGS

FANGS performs the counting of the DAG structure and identifies frequent $i+1$ -itemset to frequent $2i$ -itemset. FANGS assumption is that any $i+1$ to $2i$ length candidate set can be represented by two frequent i -itemset. So support of candidate itemset of length $i+1$ to $2i$ can be calculated simultaneously in a single pass.

In FANGS, snakes corresponding to current pass are concurrently read in the main memory and dynamically converted into TID list. All the TID lists are processed one TID at a time and LTID and CCNT variables of the DAG structure are updated. First TID lists of the leaves are read and after reading each TID the LTID of the parents are updated. If parents current LTID is smaller then recently read TID then parent is marked with the new TID. If they are equal then the CCNT of the parent is increased by one and the change is applied to the parents gradually.

FANGS implement **lazy snake** write which reduces the number of snake that is to be written in the list. During the counting of candidate itemset in DAG structure it

is not known in advance that which itemset will turn out to be frequent. So not all 2^i snakes are written in the disk. Instead a pair of i -snake that can generate the 2^i -snake in the subsequent pass are written to the disk. So in the current pass (i -th pass) frequent $i/2$ -snake are read from the disk and i -th snakes are generated on-the-fly. To make it possible, with every 2^i candidate a “generator cover” is associated which includes a pair of i -snakes which are able to generate it in the next pass.

The simplest way to generate “generator cover” is write all the i -snake in the disk. At the end of the current pass all the frequent 2^i itemset are now identified and then a pair of i -snake is selected that will generate the frequent 2^i -itemset. This process can be further optimized by identifying generator cover prior to intersection. In this case only the i -snake will be further written to the disk.

3.7 Termination

When no more candidate set can be generated the algorithm terminates.

4. Prior Work

Previously a number of algorithms for vertical mining has been presented. Among them **MaxClique**, **Column Wise** and **Hierarchical BitMap** are mentionable.

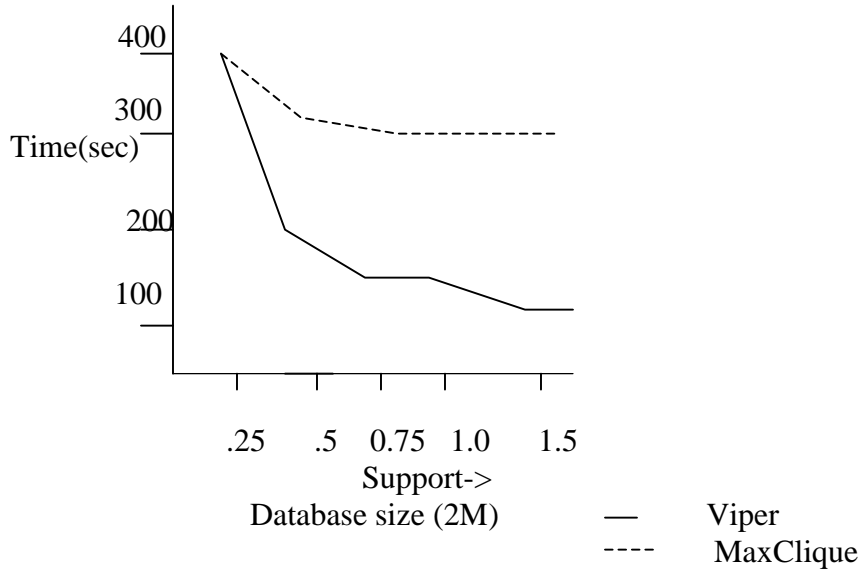
In **MaxClique** algorithm equivalence class are refined into cliques. Though cliques are more refined finding them are computationally more expensive. Also in **MaxClique** it is assumed that all the cliques can be stored in main memory which is not feasible for large database.

“**Column Wise (CW)**” works well specifically for wide and short database.

Other vertical mining algorithms also suffer from various limitations. **VIPER** overcomes most of the limitations most efficiently.

5. Results

With a synthetically generated T10I4 database the comparative performance of **MaxClique** and **VIPER** are evaluated and displayed in the accompanying figures. Comparative performance of **VIPER** and some horizontal and vertical mining is stated here. With small support (.25%) and with small database (2M) the execution time of **MaxClique** and **VIPER** are comparable.

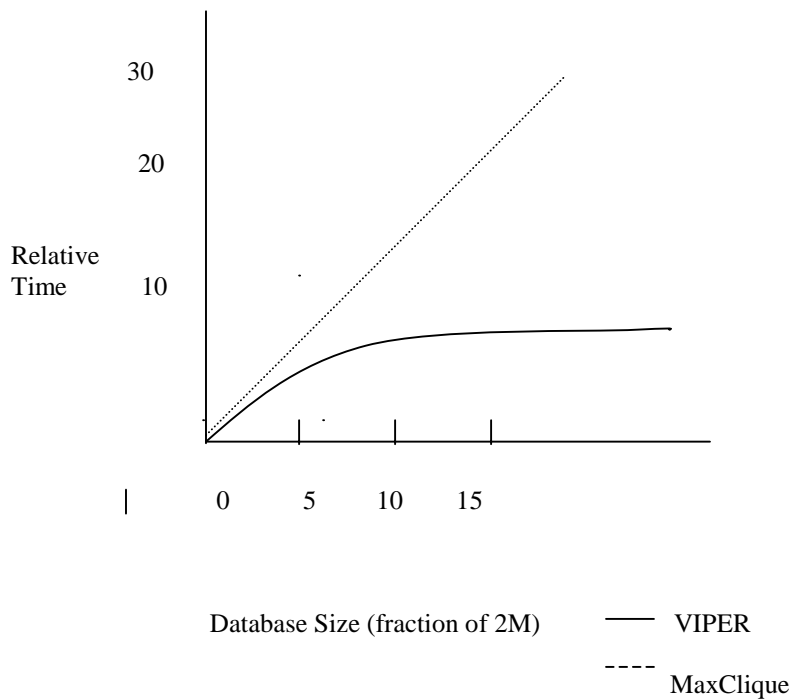


Graph 1

With the same database size but with greater support the VIPER performs significantly well over MaxClique. With increase of database size (from 2M to 5M, 10M, 15M and 25M) the VIPER significantly outperforms MaxClique. Graph 2 shows the comparative performance of VIPER and MaxClique.

So VIPER shows excellent scalability with the database size. VIPER's disk traffic is also significantly less than MaxClique.

With T10I4 database performance of AprioriGen, VIPER and ORACLE are also collected. Graph 3 shows VIPER's better performance over AprioriGen and ORACLE at lower support. At higher support VIPER's performance decreases. But key reason behind this is that the test database (T10I4) has an horizontal data layout and VIPER had to convert it to vertical data layout which is a computationally expensive process.



Graph 2

6. Conclusion

From this paper it is concluded that VIPER is the only vertical mining algorithm whose performance is not restricted by underlying database size, shape and content. It utilizes all the benefits offered by vertical data layout. VIPER outperforms MaxClique consistently and it successfully scales with the database size. It also performs better than AprioriGen and ORACLE- which represents the idealized situation for horizontal mining.

