

# TURBO-CHARGING VERTICAL MINING OF LARGE DATABASES

## AUTHORS

*Pradeep Shenoy*

*Jayant.R Harista*

*S.Sudarshan*

*Gaurav Bhalotia*

*Mayank Bawa*

*Devavrat Shah*

# OVERVIEW

- Introduction
- Key Contribution
- Relevant Prior Work
- Methodology
- Results
- Our Opinion

# INTRODUCTION

## ➤ Common Limitations of Vertical Mining algorithms

*Column Wise* - designed only for 'wide ' databases.

*MaxClique* - assumes that users will provide a lower bound on minimum support used in future mining activities.

## ➤ Overall drawback

- ability of the above algorithms to scale with database size, an important requirement for efficient mining has not been solved yet.

# KEY CONTRIBUTIONS

- New Vertical Mining Algorithm called *VIPER (Vertical Itemset Partitioning for Efficient Rule Extraction)* that is independent of database size and shape.

## FEATURES OF VIPER:

- **Data Layout** –Vertical-TransactionId-Vector(**VTV** format)

TID	Item ID's						
	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

## **FEATURES OF VIPER:**

- Stores data in compressed bit vectors called '**Snakes**'.
- Integrates **optimization techniques** using
  - DAG structure
  - Lazy Snake Writes
  - Generator Cover Selection and Writing
- **Significant performance** gains for large databases.

# RELATED WORK

- **The MaxClique algorithm**(A vertical mining algorithm based on vertical TID list (VTL) format)
  - It generates frequent itemsets and groups them into clusters called equivalence classes, which are further grouped into **smaller cliques** .
  - The **mining process** operates in two phases
    1. Using the sorted ordered list, the first itemsets is generated by intersecting with the remaining itemsets in the list until an infrequent set is found.
    2. Further generation of subsequent itemsets is done with the help of itemset generated in phase 1 and count is computed.
  
- **Limitations**
  - Cliques are refined but expensive to compute
  - The TID list of the entire click may not fit in the memory for large databases
  - The cliques may have common items which have to be read from the memory multiple times

# VIPER ALGORITHM

## 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;
   }
}
```

# MINING PROCESS

- **PASS 1:** Converts HIL layout into snake format, counts the items and then determines F1.
- **PASS 2:** Converts snakes to horizontal tuples and generates frequent 2 itemsets. No snakes are constructed during this pass.
- **SUBSEQUENT PASSES:**
  - Generate candidate itemsets from level  $i+1$  to  $2i$  using FORC candidate generation procedure.

## **FORC(Fully Organized Candidate Generation) Algorithm:**

- **Input :**  $S_k$  (Set of frequent item sets).
- **Description:** Given a set  $S_k$  (Set of frequent item sets), group the item sets of  $S_k$  into clusters called **equivalent classes**. Items that share a common prefix of length  $k-1$  is stored in a hash table and the last element of every item set is stored in an ordered list **extList**.

### Steps:

1. Add each item from the **extList** to the prefix in the hash table.
2. Find items greater the current '**ext List**' and store it another list by the name **remList**.
3.  $E = p \rightarrow$  ext list where P is in the hash table.
4.  $\text{newP} = P \cup t$  i.e t is from every item in E
5.  $\text{remList} = \{i, i \in t \text{ and } i > t\}$
6.  $\text{remList} = \text{remList} \cap (\text{subP} \rightarrow \text{extList})$ , for every **(k-1) subset subP of newP**.
7.  $\text{newCand} = \text{newP} \cup \{q\}$  for every element **q in remList**
8.  $C_{k+1} = C_{k+1} \cup \text{newCand}$

### Output:

$C_{k+1}$  – set of candidate k+1 itemsets

# EXAMPLE FOR FORC

❖ Part of frequent 3-itemset in decompressed form

TID	ITEM ID				
	123	124	126	127	134
1	1	1	0	0	1
2	0	0	0	0	0
3	1	1	0	0	1
4	1	0	1	0	0
5	1	0	1	1	0
6	1	1	0	1	1
7	0	0	0	0	0

->etc

## EXAMPLE FOR FORC

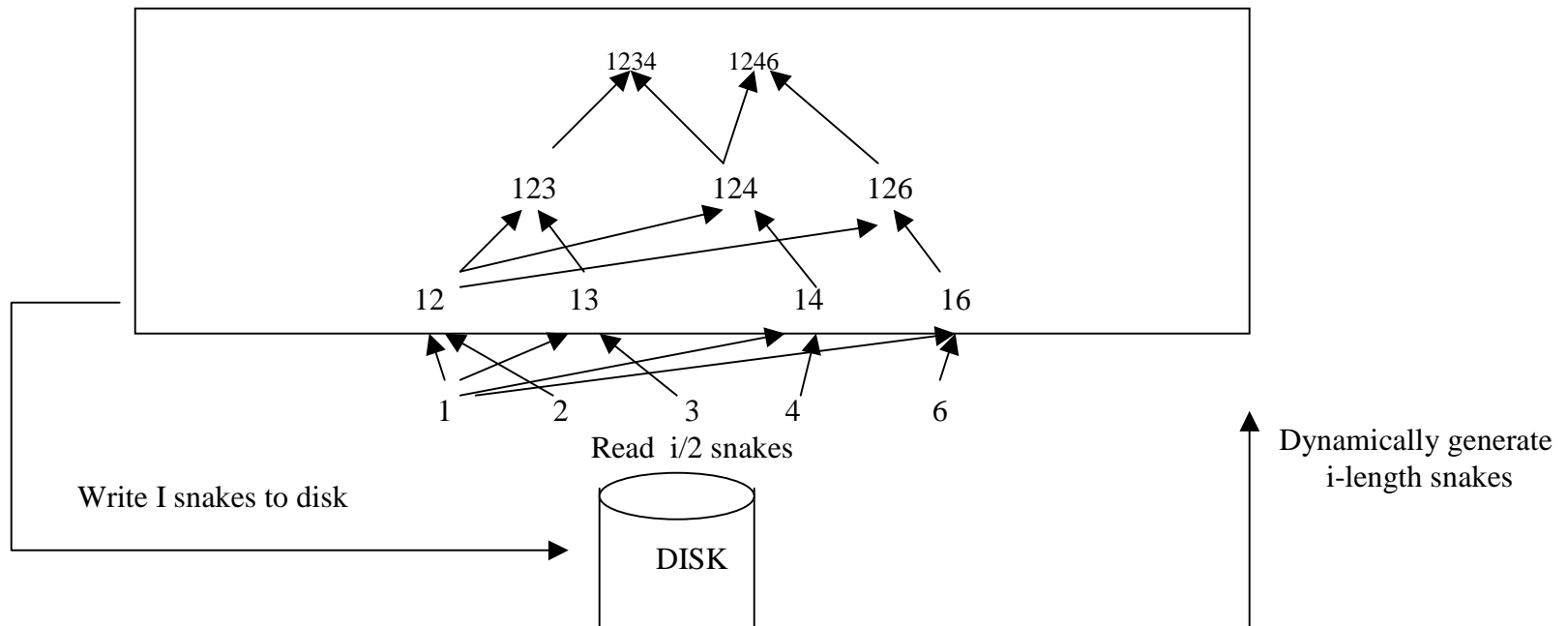
- Equivalence class  $g \rightarrow 123, 124, 126, 127$
- Common Prefix  $P_g = 12$
- $\text{extList}_g = 3, 4, 6, 7$
- For 124 itemset, Candidate 4 itemsets 1246 and 1247
- Length 3 subsets of 1246 and 1247 are 126, 146, 246, 127, 147 and 247.
- Find 146 in  $S_3$

# FANGS (FAST ANDING GRAPHS FOR SNAKES)

- **FANGS** is based on the fact that a candidate of length between  $i$  to  $2i$  can be represented as a union of some pair of frequent  $i$ -itemsets.
- It takes frequent  $i$ -snakes as input and **calculates the support** of all candidates of length  $i+1$  to  $2i$  in a **single pass** by simultaneously intersecting all the pairs.

## Steps involved in FANGS Processing Algorithm:

- **Graph Structure** – In each pass of database, a DAG(Directed Acyclic Graph) structure is created for candidate itemset generated by FORC



- The **leaves of a DAG** represent the frequent sets at level  $i$ .
- **Intermediate nodes** at height  $r$  is equivalent to a candidate of length  $i+r$  and the pair of its subsets point to nodes at height  $r-1$ .
- Each of the candidate snakes in the DAG has two variables **latest TID (LTID)** and a **current count variable (CCNT)** associated with it, which are initially initialized to zero.

➤ **The Counting process**

- The snakes associated with all the leaf itemsets are **read one page at a time** from the disk to the memory and are **dynamically** converted into equivalent TID lists.
- **LTID** and **CCNT** variables are updated.
- Reduces the number of updates, thereby **reducing the overall cost of FANGS**.

➤ **Lazy Snake Writes**

- Reduces the number of snake written to the disk by eliminating the snakes that are not required in subsequent computations.

➤ **Generator Cover Selection and Writing**

- Several generator covers exists for a top-level candidate like. In the above figure AB,CD and AC,BD are generator covers for ABCD.
- Inorder to reduce the number of snakes written to the disk, generator covers are selected in an overlapped fashion.

➤ **Snake Trimming Through Top-Down Writes**

- This optimization technique results in higher compression ratio by trimming the selected snakes, thereby increasing their sparseness

# RESULTS

## Comparison with MaxClique

- **Scalability** – The results show that VIPER excellent scalability with increase in database.
- **Resource Usage**
  - **Disk Traffic** – VIPER's disk traffic is less due to single scan per snake and lazy snake write optimization technique.
  - **Memory** –Its usage is independent of the database size, since the only data structures that need to be stored are associated with the FORC ,FANGS algorithms and read and write snake buffers and their size is dependent on only the density of patterns and not database size

## OUR OPINION

- In our opinion, VIPER clearly outperforms the prior vertical mining algorithms by overcoming *the dependency on size, shape and contents of underlying database.*
- For higher support and horizontal database, Apriori performs slightly better than VIPER
- The optimization techniques like Lazy Snake writes, etc are vaguely explained.
- We rank this paper 8 in the scale of 1-10.
- We had a chance to explore the novel concepts like Snakes, asynchronous counting, simultaneous snake intersection, etc.