Turbo-charging Vertical Mining of Large Databases

Pradeep Shenoy[‡§] Jayant R. Haritsa[†‡*] S. Sudarshan[§]
Gaurav Bhalotia[§] Mayank Bawa[§] Devavrat Shah[§]

[†]Database Systems Lab, SERC
Indian Institute of Science
Bangalore 560012, INDIA

[‡]Lucent Bell Labs
600 Mountain Avenue
Murray Hill, NJ 07974, USA

[§]Computer Science and Engg.
Indian Institute of Technology
Mumbai 400076, INDIA

Abstract

In a vertical representation of a market-basket database, each *item* is associated with a column of values representing the transactions in which it is present. The association-rule mining algorithms that have been recently proposed for this representation show performance improvements over their classical horizontal counterparts, but are either efficient only for certain database sizes, or assume particular characteristics of the database contents, or are applicable only to specific kinds of database schemas. We present here a new vertical mining algorithm called **VIPER**, which is general-purpose, making no special requirements of the underlying database. VIPER stores data in compressed bit-vectors called "snakes" and integrates a number of novel optimizations for efficient snake generation, intersection, counting and storage. We analyze the performance of VIPER for a range of synthetic database workloads. Our experimental results indicate significant performance gains, especially for large databases, over previously proposed vertical and horizontal mining algorithms. In fact, there are even workload regions where VIPER outperforms an optimal, but practically infeasible, horizontal mining algorithm.

1 Introduction

The need for efficiently mining "association rules" from large historical "market-basket" databases has been well established in the literature. Most of the algorithms developed for this purpose (e.g. [1, 2, 7]) are designed for use on databases where the data layout is *horizontal*. In a horizontal layout, the database is organized as a set of rows, with each row representing a customer transaction in terms of the items that were purchased in the transaction.

Of late, there has been considerable interest in alternative *vertical* data representations wherein each *item* is associated with a column of values representing the transactions in which it is present. Since association rule mining's objective is to discover correlated items, the vertical layout appears to be a *natural* choice for achieving this goal. Further, as explained later, vertical partitioning opens up possibilities for fast and simple support counting, for reducing the effective database size, for compact storage of the database, for better support of dynamic databases, and for asynchrony in the counting process. Based on these observations, a variety of "vertical mining" algorithms have been proposed recently [3, 4, 6, 9, 11]. Performance evaluations of these algorithms has indicated that they can provide significantly faster mining times as compared to their horizontal counterparts.

While the above-mentioned algorithms have served to highlight the utility of the vertical approach, they all suffer from a common limitation in that they are rather "specialized" – that is, they are either efficient only for certain database sizes, or assume specific characteristics of the database contents, or are applicable only to special kinds of database schemas, or place restrictions on future mining activities. For example, the ColumnWise algorithm in [3] is designed primarily for relations that are "wide" rather than "long", that is, where the number of items (i.e. columns) is significantly more than the number of transactions (i.e. rows) in the database. Similarly, the MaxEclat and MaxClique algorithms of [11] assume that users will be able to provide a lower bound on the minimum support used in all future mining activities. Finally, the performance studies have mostly been evaluated on databases that completely fit into main memory. Therefore, the ability of these algorithms to scale with database size, an important requirement for mining applications, has not been conclusively shown.

1.1 Contributions

We present here a new vertical mining algorithm called **VIPER** (Vertical Itemset Partitioning for Efficient Rule-extraction) that aims to address the above-mentioned limitations. No assumptions about the underlying database or the mining cycle are made in its design – that is, VIPER is as "general-purpose" as the classical horizontal mining algorithms. VIPER stores data in compressed bit-vectors called "snakes" and integrates a number of novel optimizations for efficient snake generation, intersection, counting and storage – these optimizations exploit the vertical data layout to a significantly greater degree as compared to the prior algorithms.

Using a synthetic database generator, we compare the response time performance of VIPER against a representative set of previously proposed vertical and horizontal mining algorithms. An important feature of our experiments

*Contact Author: haritsa@dsl.serc.iisc.ernet.in

is that they include workloads where the database is large enough that the working set of the database cannot be completely stored in memory. This situation may be expected to frequently arise in data mining applications since they are typically executed on huge historical databases.

Our experimental results indicate that VIPER provides significant performance gains, especially for large databases. Further, it shows close to linear scaleup with database size. Very interestingly, VIPER's performance improvement is to the extent that there are workload regions where it can outperform even an *idealized* horizontal mining algorithm that has *complete apriori* knowledge of the identities of all the frequent itemsets and only needs to find their counts. This is a new result that clearly establishes the power of vertical mining.

2 Background

2.1 Data Layout Alternatives

Conceptually, a market-basket database is a two-dimensional matrix where the rows represent individual customer purchase transactions and the columns represent the items on sale. This matrix can be implemented in the following four different ways, which are pictorially shown in Figure 1:

Horizontal item-vector (HIV): The database is organized as a set of rows with each row storing a transaction identifier (TID) and a bit-vector of 1's and 0's to represent for each of the items on sale, its presence or absence, respectively, in the transaction (Figure 1a).

Horizontal item-list (HIL): This is similar to HIV, except that each row stores an ordered list of item-identifiers (IID), representing only the items *actually* purchased in the transaction (Figure 1b).

Vertical tid-vector (VTV): The database is organized as a set of columns with each column storing an IID and a bit-vector of 1's and 0's to represent the presence or absence, respectively, of the item in the set of customer transactions (Figure 1c). Note that a VTV database occupies exactly the same space as an HIV representation.

Vertical tid-list (VTL): This is similar to VTV, except that each column stores an ordered list of only the TIDs of the transactions in which the item was purchased (Figure 1d). Note that a VTL database occupies exactly the same space as an HIL representation.

Virtually all the prior association rule mining algorithms, both vertical and horizontal, have opted for a *list-based* layout since this format takes much less space than the bit-vector approach (which has the overhead of explicitly representing *absence*) in sparse databases. We make the case in this paper, however, that a special form of the bit-vector-based VTV layout results in both significant performance improvements *and* reduced space requirements.

2.2 Merits of Vertical Mining

As mentioned in the Introduction, the vertical layout appears to be a natural choice for achieving association rule mining's objective of discovering correlated items. More specifically, it has the following major advantages over the horizontal layout:

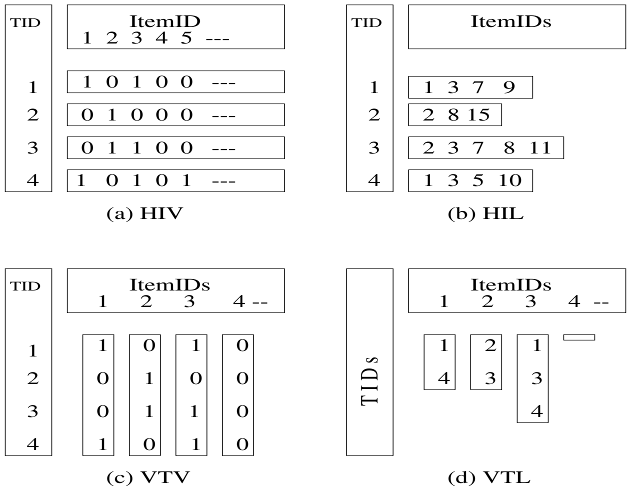

Figure 1: Comparison of Data Layouts

Firstly, computing the supports of itemsets is simpler and faster with the vertical layout since it involves only the *intersections* of tid-lists or tid-vectors, operations that are well-supported by current database systems. In contrast, complex hash-tree data structures and functions are required to perform the same function for horizontal layouts (e.g. [2]).

Secondly, with the vertical layout, there is an automatic "reduction" of the database before each scan in that only those itemsets that are *relevant* to the following scan of the mining process are accessed from disk. In the horizontal layout, however, extraneous information that happens to be part of a row in which useful information is present is *also transferred* from disk to memory. This is because database reductions are comparatively hard to implement in the horizontal layout. Further, even if reduction were possible, the extraneous information can be removed only in the scan *following the one* in which its irrelevance is discovered. Therefore, there is always a *reduction lag* of at least one scan in the horizontal layout.

Thirdly, bit-vector formats, due to their sequences of 0's and 1's, offer scope for *compression*. From this perspective also, the vertical layout is preferred since a VTV format results in higher compression ratios than the equivalent HIV format. This is because compression techniques typically perform better with larger datasets since there is greater opportunity for identifying repeating patterns – in a VTV, the length of the dataset is proportional to the number of customer transactions, whereas for HIV, it is limited to the number of items in the database, usually a fixed quantity that is small relative to the number of tuples in the database.

Finally, the vertical layout permits *asynchronous* computation of the frequent itemsets. For example, given a database with items A, B, C, once the supports of items A and B are known, counting the support of their combination AB can commence even if item C has not yet been fully counted. This is in marked contrast to the horizontal approach where the counting of all itemsets has to proceed synchronously with the scan of the database. We believe that asynchrony will prove to be an especially important advantage in *parallel* implementations of the mining process.

A careful algorithmic design is required to ensure that the above-mentioned inherent advantages of the vertical layout are translated into tangible performance benefits – we attempt this in the VIPER algorithm, which is described in the following sections.

2.3 Notation and Assumptions

For ease of exposition, we will use the following notation in the remainder of this paper:

$\mathcal{I}$	Set of items in the database
$\mathcal{D}$	Database of customer purchase transactions
$minSup$	User-specified minimum rule support
F	Set of frequent itemsets in $\mathcal{D}$
F_k	Set of frequent k-itemsets in $\mathcal{D}$
C_k	Set of candidate k-itemsets in $\mathcal{D}$

Table 1: Notation

Without loss of generality, we assume that each itemset is always represented as a lexicographically ordered sequence of items. Similarly, a set of itemsets is also always maintained in lexicographic order.

3 The VIPER Algorithm

In this section, we overview the main features and the flow of execution of the VIPER algorithm – detailed descriptions of its internal components are deferred to the following sections.

VIPER uses the vertical tid-vector (VTV) format for representing an item's occurrence in the tuples of the database. The bit-vector is stored in a *compressed* form, taking advantage of the sparseness that is typically exhibited in large databases. Exactly the same format is also used for storing the *itemsets* that are dynamically constructed and evaluated during the mining process. While this format is consistently used for disk storage, it is converted on-the-fly into alternative representations during main-memory operations, for efficiency reasons.

We will hereafter refer to an itemset and its associated compressed tid-vector as a **"Snake"**. Further, we use the term "frequent snake" to mean that the corresponding itemset is frequent, and the term "i-snake" to refer to a snake corresponding to an itemset comprised of i items.

At a macro level, VIPER is a multi-pass algorithm, wherein data (in the form of snakes) is read from and written to the disk in each pass. It proceeds in a bottom up manner and at the end of the data mining, the supports of all frequent itemsets are available. Each pass involves the simultaneous counting of *several levels* of candidates via intersections of the input snakes. A variety of techniques, described below, are implemented to improve the efficiency of this mining process.

3.1 Efficiency Features

To minimize the computational costs, VIPER implements a new candidate generation scheme called **FORC** (Fully ORganized Candidate-generation), partly based on the technique of *equivalence class* clustering [11]. The FORC scheme avoids the expensive searching associated with Apriori-Gen [2], the predominant candidate generation algorithm for both horizontal and vertical mining.

VIPER also incorporates a novel snake intersection and counting scheme called **FANGS** (Fast ANding Graph for Snakes). The FANGS scheme is based on a simple DAG structure that has a small footprint and efficiently supports *concurrent* intersection of multiple snake-pairs by using a *pipelined* strategy.

At first glance, VIPER's writing of intermediate results to disk may appear to represent an additional overhead, especially since virtually all the prior horizontal and vertical mining algorithms are "read-only". However, we claim that this is a *positive tradeoff* since the data that is written is utilized to significantly speed up the subsequent mining process. Moreover, the disk traffic is minimized in a variety of ways – with these optimizations, VIPER turns out to have, in our experiments, *less overall disk traffic* than the read-only algorithms. The optimizations include ensuring that each processed snake is read *only once*; counting the support for candidates of several levels in a single pass, resulting in a *logarithmic* reduction in the number of passes over the database; writing only a carefully chosen *subset* of processed snakes to disk; and snake *compression* through a process called **Skinning**.

Finally, the space reduction resulting from the snake compression is augmented by automatically and immediately deleting snakes that are no longer relevant to the remainder of the mining process.

3.2 The Mining Process

We now move on to discuss the flow of execution in the VIPER mining process, whose pseudocode is shown in Figure 2. Assuming the most general case where the original database is stored in the standard *horizontal item-list* (HIL) format, the following sequence of passes (over continually shrinking databases) is executed.

3.2.1 Pass 1

In the first pass over the database, the snakes for all the individual items are created and stored on the disk. That is, the database is converted from the HIL layout to the snake format. During this process, the supports of these items are counted and F_1 is determined.

3.2.2 Pass 2

In the second pass, an obvious mechanism to compute F_2 is to intersect and count the supports of all $\binom{\mid F_1 \mid}{2}$ snake pairs. However, this would be prohibitively expensive as it requires numerous snake intersections. Therefore, F_2 is determined using the following alternative approach, suggested in [11]: Temporary *horizontal* tuples (in list format) are sequentially created *in main memory* from the disk-resident collection of frequent 1-snakes. To make this clear, assume that A, B and C are the frequent 1-snakes and that the first bit (after decompression) in each of them is a 0, 1 and 1, respectively. Then, the equivalent horizontal tuple is "{TID=1},{IID of B},{IID of C}". Now, given this effectively horizontal database, for each tuple, all pairs of items in the tuple are enumerated, and their counts are simply updated in a 2-D triangular array (dimension F_1) of counters.

An important point to note in the above process is that *no snakes are constructed* during this pass. This is because writing out the snakes of all the pair-wise combinations would not only be extremely expensive, but also quite

```
Algorithm VIPER(D, I, minSup){
    Input: Horizontal Database (D), Set of items (I),
           Minimum Support (minSup)
    Output: Set of Frequent Itemsets with Supports (F)

    // Pass 1: Write all 1-snakes to disk and identify F_1
    F = countLevelOne(D);
    // Pass 2: Identify F_2
    F = F ∪ countPairs(F_1);
    // Subsequent Passes
    i = 2 ; until (isEmpty(F_i)) {
        // Create a new DAG for this level
        candDAG = createDAG(level = i);
        C_i = F_i;
        // Candidate Generation for levels k+1 to 2k
        for k in i to 2i do {
            C_{k+1} = FORC(C_k);
            if (isEmpty(C_{k+1})) break;
            candDAG = candDAG ∪ C_{k+1};
        }
        if (isEmpty(candDAG)) break;  // Terminate

        // List of snakes to be read
        readList[i] = findReadList(candDAG);
        // Trim the list of snakes to be written
        writeList[i] = writePrune(candDAG);

        // Snake intersection, counting and writing
        FANGS(candDAG, readList[i], writeList[i]);

        // Update F from levels i+1 to 2i
        for k in i + 1 to 2i do
            F = F ∪ frequentItems(candDAG,k);

        // Delete the snakes written in previous pass
        DeleteSnakes(writeList[i/2]);

        // Increment the mining level
        i = i * 2;
    }
}
```

Figure 2: The VIPER Algorithm

wasteful given that many of the combinations may eventually turn out to be infrequent. Generalizing this observation, there are two features of VIPER's snake writing (after the first pass):

- The only snakes ever written to disk are *frequent* snakes; further, only a *useful* subset of the identified frequent snakes, that is, those snakes that are potentially relevant for future passes, are written to disk.
- The writing of a frequent snake always *lags* one pass behind the pass in which the snake is identified to be frequent. Therefore, the "unwritten" snakes that are required as inputs to the current pass are *dynamically generated* using the snakes written out in the previous pass.

3.2.3 Subsequent Passes

In each subsequent pass P ($P > 2$), the first step is to generate the candidate itemsets of the current level, based on the immediately previous level's frequent itemsets, using the FORC candidate generation procedure. That is, $C_{i+1} = FORC(F_i)$. The candidate itemsets of levels $i+2, i+3, \ldots, 2i$ are then computed using the same FORC procedure, except that now the *candidate set* of each level is used to generate the candidate set of the next level. That is, $C_{k+1} = FORC(C_k)$ for $i+1 \leq k \leq 2i-1$.

We note here that our counting scheme is capable of counting candidates of length $i+1$ to $i+k$ for any k, $1 \leq k \leq i$. This is useful in the last pass when there may not remain any candidates beyond level $i+k$ ($k < i$), or in case the number of candidates turns out to be unmanageably large. In the latter case, the counting is *truncated* to consider only candidates upto length $i+k$ ($k < i$) – the appropriate value of k depends on the amount of available memory.

The generated list of candidates is inserted into the DAG structure which is the basis of the FANGS snake intersection and counting scheme. After employing a variety of pruning techniques, the set of snakes to be read ($ReadList$) and to be written ($WriteList$) in this pass are identified. The snakes in $ReadList$ are then sequentially scanned into memory and the counts of all the candidate itemsets generated in this pass ($C_{i+1}, \ldots, C_{2i}$) are concurrently computed using the FANGS procedure. Simultaneously, the snakes in $WriteList$ are written out to disk. When the database scan is over, all frequent itemsets F_{i+1} through F_{2i} will have been identified.

The last operation of the pass is to *delete* all the snakes that were written out in the *previous* pass since they are no longer required, thereby minimizing the disk space overhead.

3.2.4 Termination

The above process is repeated until there are no more candidate itemsets. Finally, the complete set of frequent itemsets, F, is returned along with the support of each of its elements. With this information, the desired association rules can be easily determined [2].

In the following sections, the details of the main components of VIPER – Skinning for snake compression, FORC for candidate generation, and FANGS for snake merging – are described.

4 Snake Generation and Compression

The snake generation process operates in the following manner: During each pass, a (page-sized) buffer is maintained in main memory for each itemset whose snake is currently being "materialized". The snake portions corresponding to these itemsets are first accumulated in these buffers – when a buffer is full, it is written to a disk-resident *common* file.[1] Within the file, the pages associated with each individual snake are chained together using a linked list of pointers. The specific set of operations in each pass is given below:

First Pass: In the first pass, the original HIL database is sequentially scanned and for each item that occurs in a transaction, the associated TID is passed to a routine which first generates 0 bits for all the tuples between the last TID in which the item occurred and

[1] The option of writing each snake into a separate file is presently not feasible since current operating systems do not permit applications to have more than a limited number of file descriptors simultaneously open. Further, there may be an actual *advantage* to writing them to a common file in that *correlated* frequent snakes may tend to have their data blocks close to each other since their buffers would fill up during similar time periods.

the current TID and then adds a 1 bit for the current TID. This bit-sequence is then compressed (using the Skinning technique described below) and added to the buffer associated with the item.

Second Pass: In the second pass, the frequent 1-snakes are decompressed to dynamically create horizontal tuples in memory, but no output snakes are constructed (as described earlier in Section 3).

Subsequent Passes: In subsequent passes, where the vertical format is exclusively used, new snakes are generated by "ANDing" of existing snakes. For example, the snake for the itemset ABC may be generated by intersecting the AB and AC snakes. This process requires decompression of the input snakes but is computationally inexpensive since it only requires simple arithmetic. For ANDing, a straightforward option is to decompress the snakes into tid-vectors and then to AND these vectors. However, as discussed later in this section, tid-vectors typically take more space than tid-lists. So, as the tid-vectors are being produced in memory, they are converted on-the-fly into tid-lists. Therefore, the ANDing reduces to "joining" tid entries, and the output is a tid-list. This tid-list is, as for the first pass described above, converted on-the-fly into a bit-vector and then a snake.

We emphasize again here that all of the above transformations between snakes, tid-vectors, and tid-lists are done only *in memory* – what is stored on disk is always a snake.

4.1 Skinning

At first glance it may seem that the classical and simple to implement *Run-Length Encoding (RLE)* would be the appropriate choice to compress the bit-vectors. However, we expect that while there may be long runs of 0's, runs of 1's which imply a *consecutive* sequence of customers purchasing the same item may be uncommon in transactional databases. In the worst-case, where all the 1's occur in an isolated manner, the RLE vector will output *two* words for each occurrence of a 1 – one word for the preceding 0 run and one for the 1 itself. This means that the resulting database will be *double* the size of the original HIL database, which would have only one word associated with each 1 (since 0's are not explicitly represented). In short, it would result in an *expansion*, rather than compression.

We have, therefore, developed an alternative snake compression technique called **Skinning**, based on the classical Golomb encoding scheme [5]. Here, runs of 0's and runs of 1's are divided into groups of size W_0 and W_1, respectively – the W's are referred to as "weights". Each such full group is represented in the encoded vector by a single "weight" bit set to 1. The last partial group (of length R mod W_i, where R is the total length of the run) is represented by a count field that stores the binary equivalent of the remainder length, expressed in $log_2 W_i$ bits. Finally, a "field separator" 0 bit is placed between the last weight-bit and the count field to indicate the transition from the former to the latter. Note that a "run separator" for distinguishing between a run of 0's and a run of 1's, is *not* required since it is implicitly known that the run symbol changes after the count field and the number of bits used for the count field ($log_2 W_i$) is fixed.[2]

[2] Without loss of generality, we assume that every bit vector starts with a run of 1's, possibly of zero length.

With appropriate choices of W_0 and W_1, Skinning results in close to an *order of magnitude* compression from the VTV format to the snake format for the databases considered in our experiments. Further, this high degree of compression is sufficient to ensure that although a VTV usually takes much more space than a VTL (or HIL) representation for sparse matrices, the snake database itself is only about *one-third* of the size obtained with these formats.

4.1.1 Frequent Snake Compression Bounds

While the above compression ratios have been empirically observed, we can go a step further in assessing the compression ratio for *frequent* snakes. Note that these are exactly the snakes of interest since, as mentioned before, VIPER is designed to only store frequent snakes.[3] Using the fact that a frequent snake, by definition, has a minimum proportion (equal to $minSup$) of 1's, we derive in [8] lower bounds on the compression ratio which show that, for realistic mining environments, a frequent snake *always* occupies less than *half* its corresponding size in a list-based format.

5 The FORC Candidate Generation Algorithm

We present here a new algorithm called **FORC** (Fully ORganized Candidate-generation) for efficiently generating candidate itemsets. FORC is based on the powerful technique of *equivalence class* clustering described in [11], but adds important new optimizations of its own.

The FORC algorithm operates as follows: Given a set S_k (which can be either a set of frequent itemsets or a set of candidate itemsets) from which to generate C_{k+1}, the itemsets in S_k are first grouped into clusters called "equivalence classes". The grouping criterion is that all itemsets in an equivalence class should share a common prefix of length $k-1$.[4] For each class, its prefix is stored in a hash table and the last element of every itemset in the class is stored in a lexicographically ordered list, called the extList.

With this framework, a straightforward mechanism of generating candidates is the following: For each prefix in the hash table, take the union of the prefix with all *ordered pairs* of items from the extList of the class (the ordering ensures that duplicates are not generated). For each of these potential candidates, check whether all its k-subsets are also present in S_k, the necessary condition for an itemset to be a candidate. This searching is simple since the $k-1$ prefix of the subset that is being searched for indicates which extList is to be searched. Finally, include those which survive the test in C_{k+1}.

5.1 Simultaneous Search Optimization

We can optimize the above-mentioned process by recognizing that since the unions are taken with ordered pairs, the prefix of the subsets of the candidates thus formed *will not* depend on the second extension item, which in turn means that all these subsets are shared and the same for each element in the extList. Hence, repeated searches for the same subsets can be avoided and they can be searched for *simultaneously*, as shown in the following example.

Example 5.1 Consider a set S_4 in which the only itemsets that begin with the prefix ABC are $ABCD$, $ABCH$,

[3] With the sole exception, of course, that all 1-snakes are stored during the first pass.

[4] As mentioned earlier, an itemset is always represented as a lexicographically ordered sequence of items.

$ABCM$ and $ABCR$. These itemsets are grouped into a common equivalence class g, with the class prefix being $P_g = ABC$ and the associated extension list being $extList_g = D, H, M, R$. We now need to find all the candidates associated with each of the itemsets in g, and we illustrate this process by showing it for $ABCH$ – the others are processed similarly.

To find the candidates associated with $ABCH$, we first identify the items that are lexicographically greater than H in $extList_g$, namely, M and R. Now, the potential candidates are $ABCHM$ and $ABCHR$, and we need to check whether all their 4-subsets are also in S_4. That means we have to search for $ABHi$, $ACHi$ and $BCHi$, where i is either M or R (we do not have to search for $ABCi$ although it is a 4-subset because its prefix is the same as P_g and therefore, by definition, will be present in S_k).

Now, to search for $ABHM$, for example, we access its group, say h, with prefix $P_h = ABH$ and then check $extList_h$ – if M exists, it means $ABHM$ exists in S_k. The important point to note now is that, having come this far, we can trivially *also determine* whether $ABHR$, which corresponds to the *other* candidate $ABCHR$, exists in S_k by verifying whether R is present in $extList_h$. □

Generalizing the above instance, we can *overlap* the subset status determination of multiple candidates by ensuring that all subsets across these candidates that belong to a common group are checked for with only one access of the associated $extList$. This is in marked contrast to the standard practice of subset status determination on a sequential (one candidate after another) basis, resulting in high computational cost.

5.2 Implementation of Simultaneous Search

```
SetOfItemsets FORC (S_k){
    Input: Set of k-itemsets (S_k)
    Output: Set of candidate k+1-itemsets (C_{k+1})

    for each itemset i in S_k do
        insert (i.prefix) into hashTable;
        insert (i.lastelement) into i.prefix →extList;

    C_{k+1} = φ ;
    for each prefix P in the hashTable do {
        E = P →extList ;
        for each element t in E do {
            newP = P ∪ t ;
            remList = {i | i ∈ E and i > t} ;
            for each (k − 1) subset subP of newP do
                remList = remList ∩ (subP →extList);
            for each element q in remList do {
                newCand = newP ∪ {q} ;
                C_{k+1} = C_{k+1} ∪ newCand ;
            }
        }
    }
    return C_{k+1} ;
}
```

Figure 3: Candidate Generation with FORC

FORC implements the simultaneous search optimization as shown in the pseudocode of Figure 3: For each (P,e) combination, where P is a prefix in the hash table and e is an element in its extList, P is extended with e to obtain the $newP$ itemset. The items in the extList that are greater than e are copied into another list called the "remnant" list, remList. The $k-1$-length subsets of $newP$ are enumerated and the associated equivalence class of each of these subsets is determined from its prefix. For each of these classes, the associated subset exists only if its last item is present in their own extList. Hence, intersecting remList with extList gives the survivors after searching in this class and the survivors are reassigned to remList. This remList updation process is executed across all the $k-1$ classes, and after completion, the $newP$ is extended with each of the elements in remList to obtain candidates of size $k+1$.

All operations in the above implementation are done in lexicographic order. It is easy to see that this feature ensures that an equivalence class, once processed, will never have to be referred to again while processing the remaining classes.

5.3 Discussion

As described above, while FORC is based on the equivalence class clustering technique proposed in [11], it adds important optimizations for efficient representation and searching. Further, although we use FORC as part of our new vertical mining algorithm, note that it can be used equally well for *horizontal mining* too since there are no format-specific features in the generation process. That is, like AprioriGen[2], it can be used for *both* vertical and horizontal mining. However, it scores over AprioriGen on the following counts:

In AprioriGen, a *hash-tree* data structure is used for storing candidate itemsets and their running counts. This results in *scattering* "joinable" itemsets (i.e. itemsets with a common prefix upto their last element) across the hashtree, making identification of such itemsets a computationally intensive task since all combinations have to be explicitly examined.

Another drawback of the AprioriGen approach is that it traverses the hashtree afresh even when multiple candidates either have common subsets or have subsets with a common prefix. So, for example, if {ABCDE}, {ABCDF} and {ABCDG} are potential candidates, then their subsets {BCDE}, {BCDF} and {BCDG} are searched for by traversing from the root in every instance although the {BCD} initial segment of the hash route is the same for all of them.

6 The FANGS Snake Processing Algorithm

The FANGS (Fast ANding Graph for Snakes) algorithm is based on the observation that any candidate of length between $i+1$ and $2i$ can be represented as the union of *some pair* of frequent i-itemsets. That is, its support can be calculated by intersecting the corresponding i-snakes. Hence, given the set of frequent i-snakes as input, the support for all candidates of length $i+1$ to $2i$ can be computed in a single pass by simultaneously intersecting all the associated pairs.

6.1 The Graph Structure

In each pass over the database, a DAG of the candidate itemsets (generated by the FORC algorithm described in the previous section) is first created. The "leaves" of the DAG are the frequent itemsets at level i. Each intermediate node at height r is a candidate of length $i+r$, and is pointed to by some pair of its subsets at height $r-1$. This is easy to arrange since if an itemset is a candidate, all its subsets are also either candidates or frequent itemsets. Finally, each of the candidate snakes in the DAG has an associated "latest TID" (LTID) variable and a "currentCount" (CCNT) variable, both of which are initialized to zero (the functions of these variables are explained later).

The intuition behind the DAG structure is as follows: We know that the union of any two $i+r-1$-subsets of a $i+r$-candidate is the candidate itself. In this sense, the pair of child nodes "covers" the candidate itemset in that these nodes can be intersected to generate (and count) the candidate. Further, an itemset has to be counted only if its immediate subsets are also present in the transaction. Hence, for an $i+r$-length candidate, we choose a pair of $i+r-1$-subsets to cover it, instead of other smaller subsets.

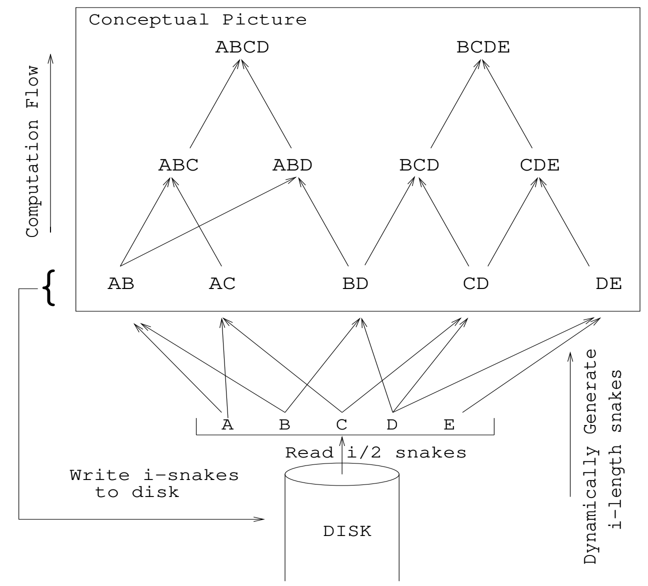

Figure 4: The DAG of Candidate Snakes

The above concepts are illustrated in the "conceptual picture" box of Figure 4, which shows a sample portion of the complete DAG structure. Here, the leaves of the DAG are the frequent 2-itemsets AB, AC, BD, CD and DE. At the next level, each of the 3-candidates is pointed to by some pair of leaves – for example, ABC and BCD are pointed to by (AB, AC) and (BD, CD), respectively. Similarly, pairs of the 3-candidates point to the candidates at level 4, namely, $ABCD$ and $BCDE$.

6.2 The Counting Process

With the above structure, the counting scheme is simple: The snakes corresponding to all the leaf itemsets are concurrently read, a page at a time, from disk into memory. During this process, as mentioned earlier in Section 4, they are dynamically converted into equivalent tid-lists. Each of these lists is processed a single TID at a time, and the processing is co-ordinated so that the TIDs are processed in *sorted* order. During the course of counting, the LTID and CCNT variables of the candidate itemsets are continuously updated.

The counting starts from the tid-lists of the leaves, and when a TID is read for a leaf, the LTID variable of its immediate parents are updated with this information. If a parent's current marking is a smaller TID, then it is simply marked with the new TID instead. However, if it is already marked with the same TID, its CCNT is incremented, and its parents are in turn updated with this TID. Intuitively, this corresponds to generating the subset-snakes of a candidate *on-the-fly*, and intersecting them at the node. The upward propagation of the updates at a node correspond to that node's participation in further intersections.

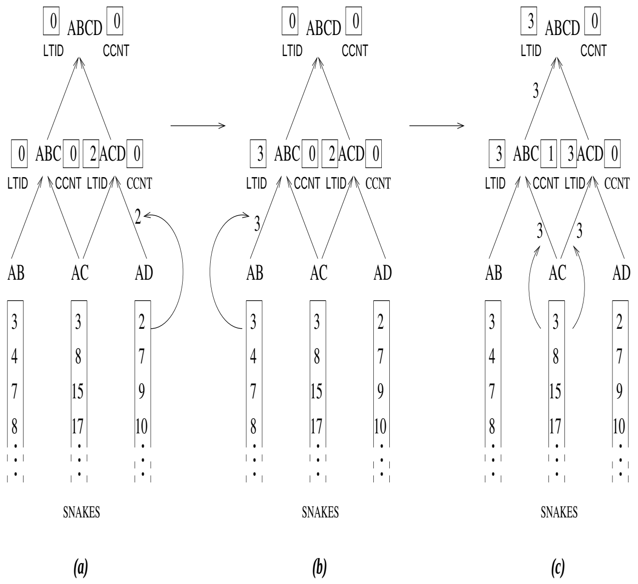

Figure 5: Snake Intersection and Counting

A pictorial example of the counting scheme is shown in Figure 5a-c.[5] Here, the snakes AB, AC and AD are being read from disk. They are read in a TID at a time, and in sorted order. The first update is from the snake AD upwards – all its parents are marked with the TID 2. In the next step, AB is read, and the candidate ABC updated with the TID 3. The third step involves reading in AC's TID, and updating the candidates ABC and ACD with this TID. At this step, since ABC has already been marked with the TID 3, its counter is incremented, and the update is propagated to the candidate $ABCD$. At the same time, since ACD is marked with a smaller TID (2), it is simply marked with 3.

The above mechanism for counting reduces the number of updates and performs them in an *on-demand* manner, thereby mitigating the expense of completely computing several intersections. This makes the overall cost of FANGS much lower than other complete-intersection-based vertical algorithms. Further, it opens up possibilities for a variety of optimizations – in the remainder of this section, we describe a few such optimizations that are currently implemented in VIPER.

6.3 Lazy Snake Writes

FANGS implements a *lazy snake write* optimization that substantially reduces the number of snakes written to disk – in fact, the only snakes that are written are those that are potentially useful in subsequent computations.

More concretely, while counting the candidates in the DAG using i-snakes, we do not know which among the top-level $2i$-candidates will turn out to be frequent, and which

[5] The DAG is, again, only partially shown.

snakes will be used to generate subsequent itemsets. Writing out all the $2i$-candidate snakes to disk can be very expensive and wasteful. Therefore, we do not write *any* $2i$-snake, but instead *dynamically regenerate* only the required snakes in the next pass. For this purpose, we associate with each $2i$-candidate a "generator cover", that is, a pair of i-snakes that can be used to dynamically generate it in the next pass. These i-snakes are written in the current pass for use in the subsequent pass. In turn, these i-snakes are regenerated in the current pass using a pair of $i/2$-snakes that had been written out in the previous pass.

Dynamic regeneration is easily incorporated into the counting process by adding an *additional* level to the DAG corresponding to the $i/2$-snakes. The modified DAG now looks like the entire picture of Figure 4, with the leaves being the $i/2$-snakes that generate the i-snakes. Specifically, though the conceptual picture shows the DAG leaves as the 2-snakes AB, AC, BD, CD and DE, in reality each of these snakes are being generated dynamically from the 1-snakes A, B, C, D and E; the 2-snakes are written to disk only during the current pass. Note that this modification does not require any changes in the counting scheme except for including an additional level of updates.

6.4 Generator Cover Selection and Writing

A simple mechanism for selecting the generator covers described above is the following: During the pass, write out *all* the i-snakes to disk. After the pass is over, which means that the frequent itemsets among the top-level $2i$-candidates have now been identified, for each of these frequent itemsets choose any pair of i-snakes whose union gives the itemset.

This simple process can be optimized, however, by observing that generator covers can be identified *prior* to performing the intersections. That is, we can associate a pair of i-snakes for each top-level candidate even before counting it. This results in a substantial benefit in that *only* those i-snakes that could *potentially* be used for re-generating a top-level candidate during the next pass need to be written to disk.

The second optimization in the generator cover identification step utilizes the fact that *several* generator cover choices may exist for a top-level candidate. For example, in Figure 4, both (AB, CD) and (AC, BD) are generator covers for $ABCD$. We can exploit this by choosing the covers in an *overlapped* fashion – that is, for each new top-level candidate, try as far as possible to use the i-snakes that have *already* been identified to cover previous itemsets. This will result in a further reduction of the number of the snakes that are written to disk.

The final optimization is related to the *order* in which the top-level candidates are processed for identifying generator covers. Note that, given the above "overlap" heuristic, the order has a bearing on the eventual assignment of generator covers. We therefore choose to process the candidates in decreasing order of their *estimated supports*.[6] Within this processing order, preference is given to generator covers comprised of leaves with higher support – the idea here is that high support leaves will be common to a larger fraction of the candidates, and therefore choosing them "early on" will eventually result in a smaller set of snakes in the global cover.

Note that a plausible alternative to the above ordering is to do exactly *the opposite* – give preference to covers comprised of leaves with *low supports*, based on the observation that such snakes will be more highly compressed, resulting in less computational effort and disk traffic. Of course, this may result in having a larger number of snakes represented in the cover.

In short, the choice is between "a small cover of high-frequency snakes" and "a bigger cover of low-frequency snakes". We evaluated both possibilities in our experiments and found that the former approach yields better results.

6.5 Snake Trimming through Top-Down Writes

We have outlined above the techniques for choosing and minimizing the number of snakes to be written to disk. We now move on to presenting an additional optimization that "trims" the chosen snakes by increasing their *sparseness*, resulting in higher compression ratios.

The key idea here is that the i-snakes that are written to disk are used *only* for regenerating the top-level candidates in the following pass. Therefore, only those TIDs which *completely* contain the top-level itemset need to be included in the leaf-snake. To make this clear, consider the following example: Suppose that snakes AB and CD are being written to disk in order to generate $ABCD$ in the next pass. Now, if a transaction has the items A, B, D, E, we would normally add this transaction to the snake AB, but not to the snake CD. However, we can exploit the information that the snake AB is being used only to generate the snake $ABCD$, and hence this transaction is useless for that purpose. Therefore, there is *no need* to add this particular transaction to the AB snake as well.

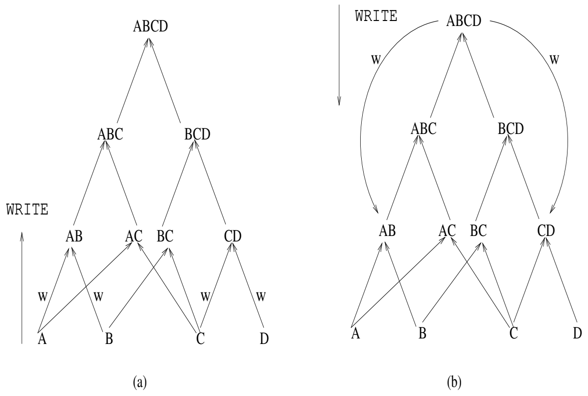

Figure 6: Bottom-up and Top-Down Approaches to Snake Writing

The above optimization is easily incorporated into the counting process described earlier in this section: Instead of the bottom-up approach of updating the i-snakes when they are detected in a transaction, adopt a *top-down* approach wherein these updates are made only when the top-level $2i$-candidate is detected in the transaction. That is, the writes are "focussed" with regard to the ultimate objective.

These contrasting approaches are shown pictorially in Figure 6. In the bottom-up approach of Figure 6a, the snakes AB and CD are appended to for all successful intersections of A, B and C, D, respectively. With the top-down approach of Figure 6b, however, these snakes are updated only when the top-level itemset $ABCD$ is detected in a transaction (through intersections of its subsets).

Note that this does *not* mean that the trimmed snake AB is identical to the snake $ABCD$ because writes to AB

[6] The estimated support is computed using the scheme presented in [1].

may be made from all of the different top-level candidates for which it forms part of the cover. It is, of course, ensured that a particular TID is added only once to a snake.

7 Related Work

Algorithms for (sequential) vertical mining have been previously presented in [6, 9, 11, 4, 3]. We restrict our attention here to the most recent among these, namely MaxClique[11], ColumnWise[3] and Hierarchical BitMap[4].

The **MaxClique** algorithm, which is a pioneering effort in the development of the vertical mining approach, is based on a vertical tid-list (VTL) format. It first generates the equivalence classes of frequent itemsets (as described for VIPER) and then refines these classes into smaller *cliques*. For each clique, the mining process operates in two phases: In the first phase, beginning with a *support-ordered* list of the itemsets of the clique, the first itemset is repeatedly intersected with each of the following itemsets in the list until an infrequent itemset is generated. In the second phase, each of the remaining itemsets in the list are combined with each of the itemsets in the first set and the supports of all these combinations are counted to identify all the additional frequent itemsets.

While cliques are more refined than equivalence classes, identifying them is computationally expensive, especially when the class graph is not sparse. Secondly, the algorithm assumes that the TID-lists of an *entire* clique can be completely stored in memory – this may not always be feasible for large databases. Thirdly, since the cliques may share individual items, the same item may have to be read in from disk *multiple* times. Finally, the problem of computing L_2 mentioned in Section 3 is circumvented by assuming an off-line "pre-processing" step that gathers the counts of *all* 2-itemsets that qualify against a user-specified *lower bound* on the minimum support. It is not clear how realistic it is to expect users to be able to choose such a bound across all future mining activities. Moreover, the pre-processing step has to be repeated every time the database is augmented.

Extensions to the basic MaxClique algorithm have been proposed in [10, 11] to address some of the above problems, but the feasibility and performance impact of these modifications have not been assessed. For example, the proposal to recursively decompose cliques until all the TID-lists in a clique fit into memory, may result in significant overlap of items across cliques, with adverse impact on the disk traffic.

The **ColumnWise (CW)** algorithm is designed for "wide and short" databases, where the number of items is significantly more than the number of transactions. For such databases it may not be possible to store the counters of all the candidates in memory and therefore using the traditional horizontal mining approach may result in significant disk traffic for paging the counters between memory and disk. To address this issue, the CW algorithm assumes a VTL format and does the counting *sequentially*, a candidate at a time, by merging the tid-lists of the individual items featured in the candidate. The rest of the algorithm is identical to Apriori. Their experimental study only considers the I/O traffic but not the total execution time of the mining process. Also, CW does not feature any special optimizations for taking advantage of the vertical format.

Finally, the **Hierarchical BitMap (HBM)** algorithm uses a VTV representation that is augmented with an *auxiliary index* indicating which "groups" (every consecutive set of 16 bits forms a group) contain only 0's. This identification helps, during the intersection process, to skip the groups for which either vector has a 0 in the auxiliary index. While this makes the intersection more efficient, it is at the cost of having to maintain auxiliary structures that are proportional to the size of the database.

From the above discussion, we conclude that the state-of-the-art in vertical mining algorithms is subject to various restrictions on the underlying database size, shape, contents or the mining process. Further, and very importantly, their ability to scale with database size has not been conclusively evaluated since their experiments have focussed on environments where the *entire database* is smaller than the main memory of their experimental platforms. A comparative summary of the algorithms, as also VIPER, is given in Table 2.

8 Performance Study

We have conducted a detailed study to assess VIPER's performance against representative vertical and horizontal mining algorithms. In particular, we compare it with MaxClique[7] and Apriori. We also include in the evaluation suite an idealized, but practically infeasible, horizontal mining algorithm, called **ORACLE**, which "magically" knows the identities of all the frequent itemsets in the database and only needs to gather the actual supports of these itemsets. Note that this algorithm represents the absolute minimal amount of processing that is necessary and therefore represents a lower bound on the execution time of horizontal mining algorithms.[8]

Our experiments cover a range of database and mining workloads, and include *all* the experiments described in [11] – the only difference is that we also consider database sizes that are *significantly larger* than the available main memory. A range of rule support threshold values between 0.25% and 2% are considered in these experiments. The primary performance metric in all the experiments is the *total execution time* taken by the mining operation. (This total execution time includes the pre-processing time in the case of the MaxClique algorithm.) Due to space limitations, we show only a few representative experiments here – the others are available in [8].

The databases used in our experiments were synthetically generated using the technique described in [2] and attempt to mimic the customer purchase behavior seen in retailing environments. The parameters used in the synthetic generator and their default values are described in Table 3.

Parameter Symbol	Parameter Meaning	Default Value
N	No. of items	1000
T	Mean transaction length	10
L	No. of frequent itemsets	2000
I	Mean frequent itemset length	4
D	No. of transactions	2M – 25M

Table 3: Parameter Table

Our experiments were conducted on a SGI Octane 225 MHz workstation running Irix 6.5, configured with a 128 MB main memory and a local 4 GB SCSI disk. For the databases with parameters $T = 10$ and $I = 4$, (see Table 3),

[7]The code for MaxClique was supplied to us by its authors.

[8]The bound applies, of course, only within the framework of the horizontal mining data and storage structures used in our study.

Mining Algorithm	Database Format	Compressed	Candidate Generation	Single/Multiple Scans of a Column	Main Restrictions
MaxClique	Tid-list	no	Clique	Multiple	Pre-processing / Small db
ColumnWise	Tid-list	no	AprioriGen	Multiple	Short-Wide Tables / Small db
HBM	Bit-vector	no	AprioriGen	Multiple	Memory-intensive / Small db
VIPER	Bit-vector	yes	FORC	Single	–

Table 2: Comparison of Vertical Mining Algorithms

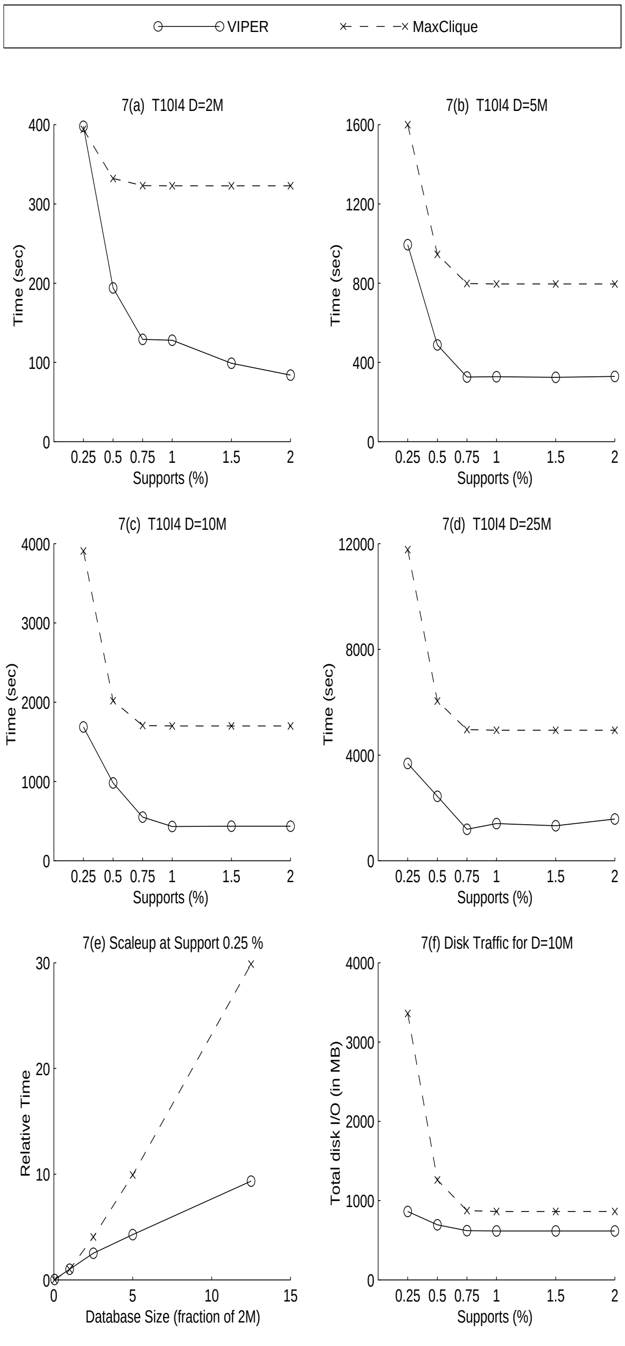

Figure 7: Comparison of VIPER and MaxClique

the associated database sizes are approximately 100MB (2M tuples), 250MB (5M tuples), 500MB (10M tuples) and 1.25 GB (25M tuples). Finally, the weights used in VIPER's skinning scheme to create compressed snakes from tid-vectors are $W_0 = 256$ and $W_1 = 1$ (the rationale for these choices is given in [8]).

8.1 Experiment 1: Comparison with MaxClique

In our first experiment, we evaluated the performance of the VIPER and MaxClique algorithms for the T10I4 database across a range of database sizes. The results of this experiment are shown in Figures 7a–d, which correspond to databases with 2M, 5M, 10M and 25M transactions, respectively. As is shown in these graphs, VIPER consistently performs better than MaxClique. Also, the difference in performance shows a marked increase with the size of the database. For example, while their performance at $minSup = 0.25\%$ is comparable for the 2M database (Figure 7a), we see for the same support a performance ratio of over 3 in favor of VIPER in the database with 25M transactions (Figure 7d).

8.1.1 Performance Scalability

In Figure 7e, the results of Figures 7a–d are combined to compare the *scalability*, with respect to database size, of VIPER and MaxClique. In this figure, which is evaluated for $minSup = 0.25\%$, the database size is shown relative to the 2M database, while the running times have been normalized with respect to the corresponding running times for the 2M database.

The results show that VIPER has excellent scalability with database size – the ratios of time taken versus database sizes are nearly equal. This conforms to our expectation – since the computation cost in VIPER is on a per-transaction basis, it should scale linearly with an increase in the number of transactions. In contrast, MaxClique shows significant degradation with increasing database size.

8.1.2 Resource Usage

The disk activity of VIPER and MaxClique for the T10I4D10M database is shown in Figure 7f over the range of support values. We see here that VIPER's disk traffic is consistently *less* than that of MaxClique, highlighting the effect of the several optimizations that VIPER incorporates to reduce disk I/O.

MaxClique, on the other hand, reads in a TID-list corresponding to a single item multiple times, depending upon the number of cliques in which it is present. As a result, the disk reads increase dramatically at low supports where there is considerable overlap between clusters. In this situation, VIPER's strategy of a single scan per snake in conjunction with lazy snake writes, appears to be the preferred choice.

Another feature of VIPER is that its main memory usage is effectively independent of the database size. This is because it only needs to store the data structures associated with the FORC and FANGS algorithms (apart from, of course, the read and write snake buffers), and the size of these data structures is dependent only on the density of patterns in the database, *not* the database size. For example, VIPER's peak memory usage across all the workloads considered in the baseline experiment is 4 MB. In contrast, MaxClique's memory usage depends on the lengths of the TID-lists and the number of TID-lists in each clique. Accordingly, MaxClique uses close to 2.5 MB for the database with 2M transactions, and as much as 23 MB for the 10M database.

8.2 Experiment 2: Comparison with Apriori and ORACLE

We now move on to compare VIPER's performance with that of the Apriori and ORACLE *horizontal* mining algorithms. The performance for all the database sizes considered in the baseline T10I4 experiment was evaluated and a representative graph for the 10M database is shown in Figure 8.1.

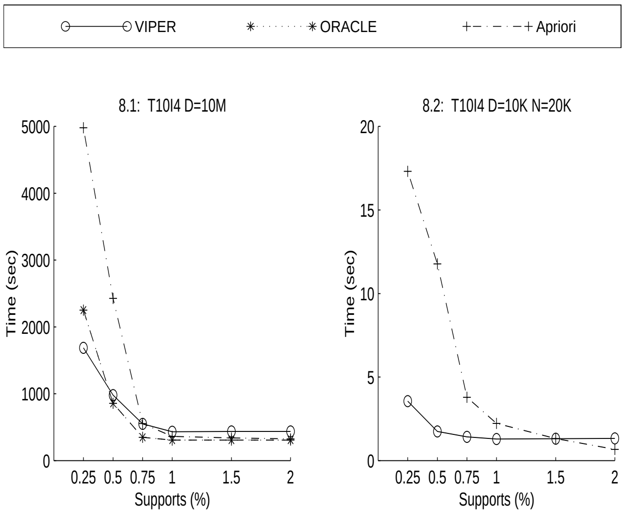

Figure 8: (.1) Comparison with Apriori and ORACLE (.2) Short and Wide Database

In Figure 8.1, we first notice that Apriori's performance, steeply degrades at lower supports and is considerably worse than that of VIPER. This is because it has to make several scans over the entire database at these lower supports. At high supports, Apriori appears to perform marginally better than VIPER. However, this is an artifact of our experimental setup wherein the original database is in horizontal format and VIPER has the overhead of converting this database to the vertical format – in particular, writing out of *all* the 1-snakes. This overhead is the predominant mining cost at high supports – when it is factored out, VIPER performs better than Apriori. In practice, we might expect that applications using vertical mining algorithms would store their databases in the vertical format itself.

Another interesting observation here is that if we compare the results with those in Figure 7c, Apriori actually outperforms MaxClique over the entire higher support region. This might seem to be at odds with the results reported in [11], wherein MaxClique always beat Apriori by substantial margins. The difference here is that the pre-processing times are included in our execution time computations – these times were ignored (for all algorithms) in [11]. However, the pre-processing step takes different amounts of time for different algorithms – in Apriori, only the "join" of F_1 is counted in the second pass, whereas in MaxClique the "join "of $\mathcal{I}$ (the set of all items in the database) is counted in the second pass, and typically $\mathcal{I} \gg F_1$ – this has a major impact at higher supports, where the preprocessing step takes up most of the overall execution time.

Finally, moving on to the performance of ORACLE, observe that VIPER's performance is close to that of ORACLE for most of the support range and, in fact, VIPER *does noticeably better* at $minSup = 0.25\%$! This behavior was also confirmed in our other experiments. Based on this, we can conclude that there are workload regions where "an on-line vertical mining algorithm can outperform even the optimal horizontal mining algorithm", clearly highlighting the power of the vertical approach.

8.3 Experiment 3: Short and Wide Database

The previous experiments were evaluated on "tall and thin" databases where the number of transactions (rows) significantly exceeded the number of items (columns). We now move on to considering a "short and wide" database[3] – in particular, a database with $N = 20,000$ items and $D = 10,000$ transactions, all the other parameters remaining the same as those of the previous experiments. This choice corresponds to a "width-ratio" (defined as N/D[3]) of 2.0, matching the maximum considered in [3]. In fact, it perhaps represents a more "stressful" environment since the number of items is an order of magnitude more than that modeled in [3] (their database had only 1000 items).

The behavior of VIPER and Apriori for the above database is shown in Figure 8.2. We see here that VIPER significantly outperforms Apriori over virtually the entire range of support values – for example, at 0.25% support, Viper completes in one-fourth the time taken by Apriori. (Only at the highest support of 2% does Apriori do marginally better, and this again is due to the artifact of our experimental setup, discussed in the previous experiment.) These results demonstrate that, unlike the CW algorithm of [3], which is specifically designed for short-and-wide databases, VIPER applies equally to both short-and-wide databases, as well as the the more traditional tall-and-thin databases.

8.4 Compression and Pruning

Finally, we present a few supporting statistics indicating the contributions of the several optimizations implemented in VIPER.

With regard to snake compression, the Skinning technique resulted in databases that were substantially smaller as compared to the original horizontal database. This is clearly brought out in the statistics of Table 4, which show the space requirements for the various alternative representations of the T10I4D10M database – here we see that VIPER is approximately *one-third* the size of the original HIL database and almost an *order of magnitude* smaller than the VTV representation.

The pruning mechanisms for reducing the number of snakes written to disk resulted in considerable savings, as demonstrated in the following extract from VIPER's output for the second pass over the T10I4D10M database:

Representation Format	Disk Space
HIL	392 MB
VTL	392 MB
VTV	1.2 GB
Snakes	135 MB

Table 4: Database Format Sizes (T10I4D10M)

```
Database:  t10i4d10m, supp:  0.25,
Starting level = 2
Candidates at level3:  3458
Candidates at level4:  2402
# 2-snakes generated:  2504
# 2-snakes written:  1474
```

What this extract means is that during this pass a total of 2504 2-snakes were dynamically regenerated while counting the supports of C_3 and C_4. Only 1474 of them were written back to disk as potential covers for the 2402 C_4 candidates, to be used in regenerating the frequent 4-snakes which are the leaves of the DAG during the following pass.

8.5 Other Experiments

We evaluated the sensitivity of the above results obtained for the T10I4 database to the choice of database parameters – these results were similar to those described here and are available in [8]. An interesting observation was that for a T20I4D10M database, wherein the transactions are longer, the 0.25% support evaluation *could not* be conducted for MaxClique since the number of TID-lists in a clique is very large in this environment, and the combined memory requirement to store the TID-lists of a clique (approximately 500 MB) heavily exceeded the available physical memory (128 MB). This result highlights the fact that MaxClique does not scale easily to databases whose active segment exceeds the available memory.

9 Conclusions

In this paper, we have addressed the problem of designing a "general-purpose" vertical mining algorithm whose applicability or efficiency, unlike previously proposed algorithms, is not subject to restrictions on the underlying database size, shape, contents, or the mining process. We presented VIPER, a new algorithm that uses a compressed bit-vector representation of itemsets, called snakes, and aggressively materializes the benefits offered by the vertical data layout. It features a novel DAG-based snake intersection scheme that permits the candidates of multiple levels to be efficiently counted in a single pass. Other optimizations include cluster-based candidate generation, single scan per snake, lazy snake writes, generator cover selection and snake trimming, all of which together result in significant savings in both computation and disk traffic.

Our experimental results demonstrate that VIPER consistently performs better than MaxClique, which represents the state-of-the-art in vertical mining – further, VIPER has the added advantage of excellent scalability, an important requirement for a viable mining algorithm. Finally, we also showed that VIPER is capable of not only outperforming Apriori, but also ORACLE, the idealized horizontal mining algorithm – this is a new result establishing the power of the vertical approach.

In our future work, we propose to explore the development of *parallel* vertical mining algorithms that can effectively exploit vertical mining's attractive feature of supporting asynchrony in the counting process.

Acknowledgments

We are very grateful to Mohammed Zaki for providing us with the MaxClique program. We thank Vikram Pudi for his insightful comments and assistance in the coding of the algorithms. The work of J. R. Haritsa was supported in part by research grants from the Dept. of Science and Technology and the Dept. of Bio-technology, Govt. of India.

References

[1] R. Agrawal, T. Imielinski, and A. Swamy. Mining association rules between sets of items in large databases. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 1993.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of 20th Intl. Conf. Very Large Databases (VLDB)*, September 1994.

[3] B. Dunkel and N. Soparkar. Data organization and access for efficient data mining. In *Proc. of 15th Intl. Conf. on Data Engineering (ICDE)*, 1999.

[4] G. Gardarin, P. Pucheral, and F. Wu. Bitmap based algorithms for mining association rules. Technical report 1998-18, University of Versailles, 1998. (http://www.prism.uvsq.fr/rapports/1998/document_1998_18.ps.gz)

[5] S.W. Golomb. Run-length encoding. *IEEE Trans. on Information Theory*, 12(3), July 1966.

[6] M. Holsheimer, M. Kersten, H. Mannila, and H. Toivonen. A perspective on databases and data mining. In *Proc. of 1st Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, August 1995.

[7] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases . In *Proc. of 21st Intl. Conf. on Very Large Databases (VLDB)*, 199 5.

[8] P. Shenoy, J. Haritsa, S. Sudarshan, M. Bawa, G. Bhalotia, and D. Shah. Turbo-charging vertical mining of large databases. Technical Report TR-2000-02, DSL, Indian Institute of Science, 2000. (http://dsl.serc.iisc.ernet.in/pub/TR/TR-2000-02.ps)

[9] S-J. Yen and A.L.P. Chen. An efficient approach to discovering knowledge from large databases. In *Proc. of 4th Intl. Conf. on Parallel and Distributed Information Systems (PDIS)*, 1996.

[10] M. J. Zaki. *Scalable Data Mining for Rules.* PhD thesis, Dept. of Computer Science, University of Rochester, July 1998.

[11] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *Proc. of 3rd Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, August 1997.