Q-VIPER: Quantitative Vertical Bitwise Algorithm to Mine Frequent Patterns

Patrick Hamzaj

Università degli Studi di Verona

January 4, 2024

1 Introduction

Frequent pattern mining aims to discover frequently occurring sets of items from big data. Given a series of transactions containing a set of items, frequent pattern mining seeks to determine the sets of items, which occur in many transactions. Frequent patterns can be discovered horizontally by transaction-centric mining algorithms or vertically by item-centric mining algorithms. Regardless of their mining direction, traditional frequent pattern mining algorithms aim to discover boolean patterns in the sense that patterns capture the presence (or absence) of items within the discovered patterns (e.g. A-Priori algorithm). However, there are many real-life situations in which quantities of items within the patterns are important. For example, the quantity of items may affect profits of selling the items, Hence, Q-VIPER is a quantitative vertical bitwise algorithm to mine frequent patterns representing the big data as a collection of bitmaps. Each item-centric bitmap captures the presence (or absence) of a transaction containing the item, as well as the quantity in each transaction. With this representation, this algorithm then vertically mines quantitative frequent patterns.

2 Background

The A-Priori algorithm is an example of horizontal transaction-centric frequent pattern mining algorithm, where data is represented as a collection of transactions. Each transaction captures the presence or absence of items. The core concept relies on the "monotonicity" property, which states that if an itemset is frequent, then all its subsets must also be frequent. The algorithm operates in two phases: first, it identifies frequent individual items by scanning the dataset and counting their occurrences. Subsequently, in the second phase, candidate item sets are generated and tested to determine their frequency. This involves iteratively combining frequent item sets from the previous phase to form larger sets, which are then verified against the dataset, pruning the search space by eliminating candidate item sets that do not meet the minimum support threshold. While traditional frequent pattern mining is useful in many contexts, it has a major limitation, in that we assume that every transaction either contains an item or does not contain the item: an item is contained in a transaction 0 or 1 times. For this reason, we can also refer to traditional frequent pattern mining as Boolean frequent pattern mining.

3 Vertical Representation of Data

Let D be a horizontal transaction-centric database defined as $D = (T_1, T_2, \ldots, T_i)$, where each T is a record containing multiple items $K = \{k_1, k_2, \ldots, k_m\}$ for some integer m. For instance, $T_1 = \{a, b\}$ and $T_2 = \{b, c\}$ represent two records within a transaction-centric database, amenable to algorithms like A-Priori.

Additionally, we denote a vertical item-centric database as a collection $D = (bitmap(a), bitmap(b), \ldots)$, where, for any $K = \{k_1, k_2, \ldots, k_m\}$, each record is stored as the bitmap of element k_m, i.e. bitmap(k_m), indicating the presence or absence of item k_m in transaction T_i. A bit of 1 n the i-th position indicates the presence of the item in the i-th transaction, whereas a bit of 0 in the j-th position indicates the absence of the item from the j-th transaction. An advantage of such a bitmap representation is that the size of the bitmap collection is independent of the density of the data. Dense data contain more 1s than 0s, and vice versa for sparse data.

For example, for transactions $T_1 = \{a, b\}, T_2 = b$ and $T_3 = a, c$, the corresponding vertical representation of the transaction database is *bitmap(a)* = [101], *bitmap(b)* = [110] and *bitmap(c)* = [001]. Each bitmap is of the same length, and its length equals to the number of transactions.

4 Working with Quantitative Data

As mentioned in the previous paragraph, A-Priori algorithm takes in input a transaction database. This means records are composed of sets of items, which are identified by their transaction number (or transaction id): an example can be represented by $T_1 = \{a, b\}$ and $T_2 = \{b\}$.

On the other hand, quantitative transaction database extend the concept of transaction database by embedding a quantity within each transaction item. Suppose that $E = \{e_1, e_2, \ldots, e_m\}$ is the set of all items that can be found in a transaction database for some positive integer m. Then, a transaction can be represented as $T = \{(e_1, f_1), (e_2, f_2), \ldots, (e_m, f_m)\}$ where each $e_i \in E$, such that $e_i = e_j$ whenever $i = j$, and each f_i is a positive integer. The quantitative transaction database is $D = (T_1, T_2, \ldots, T_n)$, which is the set of all transactions.

To represent quantitative transaction databases in a vertical format, for each item that occurs in the transaction database, we store it as a set of pairs. Each pair contains a transaction id associated with that item and the number of occurrences of the item in the transaction. For example, if we have two transactions $T_1 = \{(a, 1)\}$ and $T_2 = \{(a, 3)\}$, then the transaction database can be represented vertically using $pairset(a) = \{(T_1, 1), (T_2, 3)\}$.

Thereafter, expressions must be constructed. An itemexp (short for item-expression) is an ordered triplet of the form $(p \otimes q)$, where $p \in E$, $\otimes \in \{=, \geq, \leq\}$, and q is a positive integer. Then, an itemexpset (item-expression-set) can be defined as a set $X = \{x_1, x_2, \ldots, x_k\}$ for some positive integer k, where each $x_i = (p_i, \otimes_i, q_i)$ is an itemexp such that $p_i = p_j$ whenever $i = j$. For a transaction T and an itemexpset X, T satisfies X for every $i \in \{1, 2, \ldots, k\}$, there exists some $j \in \{1, 2, \ldots, t\}$ such that $p_i = e_j$ and the expression $(f_i \otimes_i q_i)$ is true.

That said, for any itemexpset X, *bitmap(X)* is defined as the set of transaction ids corresponding to transactions which satisfy X. When X is an itemexpset containing at least two itemexps, we can break down X as $X = W \cup \{y\} \cup \{z\}$, where W is an itemexpset with two fewer elements than X and y and z are itemexps. The support of an itemexpset X can be computed by simply counting or summing the number of 1-bits in its bitmap, which is defined as the number of transactions in D that satisfy X.

Finally, let *minsup* be some non-negative real number: then X is a frequent itemexpset if $sup(X) \geq minsup$.

5 Pruning Rules

Once computed the set containing the frequent itemexpsets, some redundant elements can be removed. Assume that X is an itemexpset in set L_k, the two pruning rules are described:

1. Suppose that X contains an itemexp of the form $(z \leq r)$, where z is an item and r is a positive integer. The first pruning rule states that if there is another itemexpset Y in L_k with the same support as X which is the same as X except that $(z \leq r)$ is replaced by $(z \leq r + s)$ for some positive integer s, then Y can be pruned from L_k.
2. Suppose that X contains an itemexp of the form $(z \geq s)$. The second pruning rule states that if there is another itemexpset Y in L_k with the same support as X which is the same as X except that $(z \geq r)$ is replaced by $(z \geq r - s)$ for some positive integer s, then Y can be pruned from L_k.

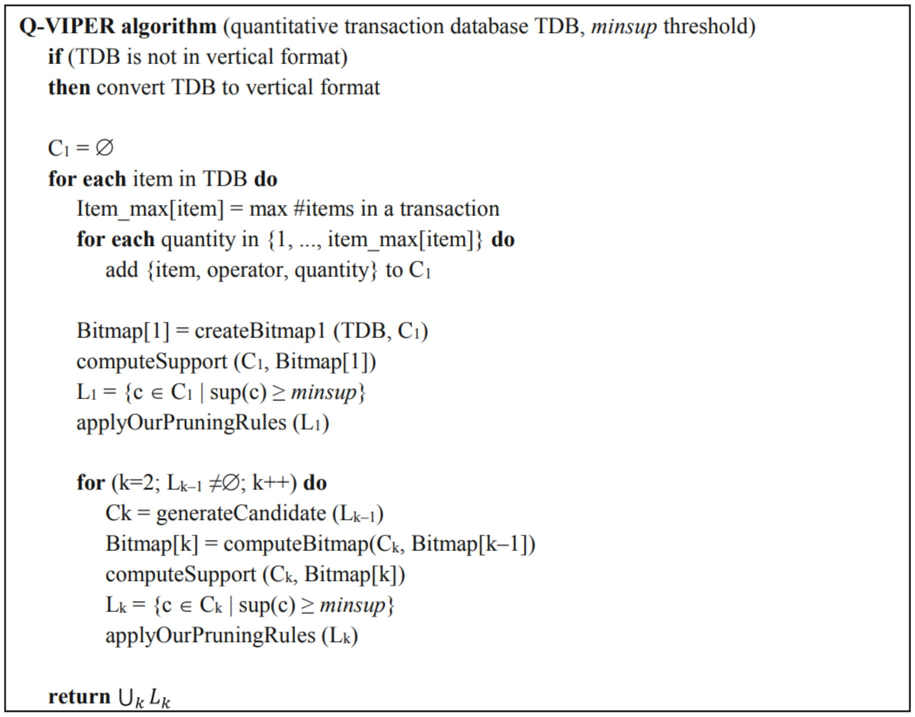
As an example, suppose that set L_2 contains itemexpsets $X = \{(a = 1), (b \geq 6)\}$ and $Y = \{(a = 1), (b \geq 3)\}$ before pruning and that those itemexpsets have the same support. Using the pruning rules, we would prune X from L_2.

6 Q-VIPER Algorithm

Now let's describe how Q-VIPER algorithm discovers quantitative frequent patterns vertically, which can be followed step by step with the pseudo-code provided in Figure 1. For any integer $k \geq 1$, define C_k to be the set of candidate itemexpsets with size k and L_k to be the set of frequent itemexpsets with size k.

First, we convert the quantitative transaction database into a vertical format if it is not already. The vertical format is useful for computing the bitmaps corresponding to the itemexpsets in C_1. The next step is to compute all itemexpsets in C_1. Each of those itemexpsets consist of a single itemexp of the form (item, operation, quantity), where *item* is an item in the transaction database, *operation* $\in \{=, \geq, \leq\}$, and *quantity* $\in \{1, \ldots, item\_max[item]\}$. We compute *item_max[item]* as the maximum number of times item appears in a transaction, over all transactions in the transaction database.

After creating C_1, we compute the bitmap associated with each itemexpset. The bitmaps can be easily computed from the vertical representation. We then



```
Q-VIPER algorithm (quantitative transaction database TDB, minsup threshold)
    if (TDB is not in vertical format)
    then convert TDB to vertical format

    $C_1 = \varnothing$
    for each item in TDB do
        Item_max[item] = max #items in a transaction
        for each quantity in {1, ..., item_max[item]} do
            add {item, operator, quantity} to $C_1$

        Bitmap[1] = createBitmap1 (TDB, $C_1$)
        computeSupport ($C_1$, Bitmap[1])
        $L_1 = \{c \in C_1 \mid sup(c) \geq minsup\}$
        applyOurPruningRules ($L_1$)

        for (k=2; $L_{k-1} \neq \varnothing$; k++) do
            Ck = generateCandidate ($L_{k-1}$)
            Bitmap[k] = computeBitmap($C_k$, Bitmap[k–1])
            computeSupport ($C_k$, Bitmap[k])
            $L_k = \{c \in C_k \mid sup(c) \geq minsup\}$
            applyOurPruningRules ($L_k$)

    return $\bigcup_k L_k$
```

Figure 1: Pseudo-code of Q-VIPER algorithm.

calculate the support of each itemexpset by counting (or summing) the number of 1-bits in its corresponding bitmaps. The itemexpsets in L_1 are the ones in C_1 with a support $\geq$ *minsup*, which is defined as the minimum support. Finally, we remove some itemexpsets in L_1 based on the two pruning rules.

Next, we set $k = 2$ and begin executing the main loop. The first step in the main loop body is to generate C_k using L_{k-1}. We initially create C_k by performing a self-join on L_{k-1}: if there are two itemexpsets in L_{k-1} where the first *(k - 1)* itemexps are the same and the last itemexp in both refer to different items, then we add an itemexpset in C_k consisting of those first *k - 1* itemexps as well as the last itemexp of both itemexpsets. Afterwards, we prune any itemexpset in C_k that contains a sub-itemexpset with *(k - 1)* that is not in L_{k-1}. Next, we create bitmaps: this can be done using the recursive definition for bitmaps, for which when X is an itemexpset containing at least two itemexps, we can break down X as $X = W \cup \{y\} \cup \{z\}$, where W is an itemexpset with two fewer elements than X and y and z are itemexps. We can then compute the bitmap of the candidate by performing a dot product between the bitmaps of $W \cup \{y\}$ and $W \cup \{z\}$.

We can then compute the support of each itemexpset in C_k. Any itemexpset in C_k with a support $\geq$ *minsup* is added to L_k.

Using the two pruning rules, we remove some uninteresting itemexpsets from

L_k, if necessary. After the pruning process, we have reached the end of the loop body, so we increment k and repeat the main steps until L_k is empty. Q-VIPER algorithm returns $\bigcup_k L_k$, which contains all interesting frequent itemexpsets.

7 Evaluation

For testing purposes, two datasets from FIMI Repository have been taken: *chess* and *mushroom* datasets. Whenever an item occurs in a transaction, instead of it only occurring once, its number of occurrences follows a $Poisson(\lambda = 1)$ distribution plus 1.

In the experiment conducted in the paper, a comparison between runtime of both Q-VIPER and MQM-A algorithms have been made for benchmarking. Figure 2 shows the runtime of each of the two algorithms for a variety of values of *minsup* for both quantitative transaction databases. The runtime (in seconds) is shown on the y-axis, while the value of *minsup* is given on the x-axis. The performance of the implementation for this project has shown to be similar for the *chess* dataset, while it has shown to be unexpectedly faster for the *mushroom* dataset, as demonstrated in Figure 3. A seed for reproducibility was not provided.

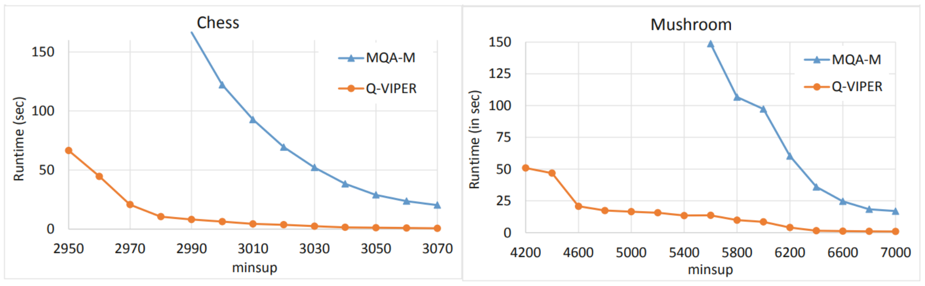


Figure 2: Performance evaluation of the experiment.

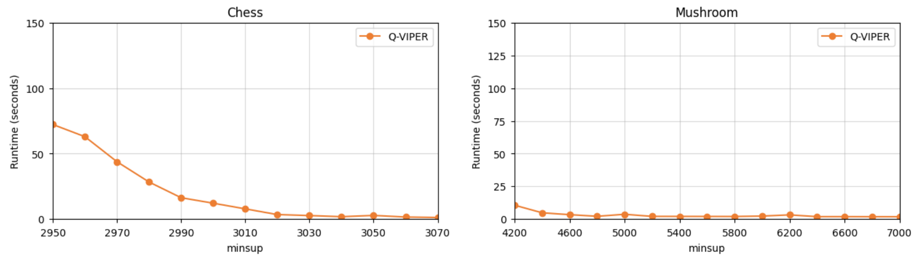


Figure 3: Performance of the project.