# Chapter 2
# Frequent Pattern Mining Algorithms: A Survey

**Charu C. Aggarwal, Mansurul A. Bhuiyan and Mohammad Al Hasan**

**Abstract** This chapter will provide a detailed survey of frequent pattern mining algorithms. A wide variety of algorithms will be covered starting from *Apriori*. Many algorithms such as *Eclat*, *TreeProjection*, and *FP-growth* will be discussed. In addition a discussion of several maximal and closed frequent pattern mining algorithms will be provided. Thus, this chapter will provide one of most detailed surveys of frequent pattern mining algorithms available in the literature.

**Keywords** Frequent pattern mining algorithms · *Apriori* · *TreeProjection* · *FP-growth*

## 1 Introduction

In data mining, frequent pattern mining (FPM) is one of the most intensively investigated problems in terms of computational and algorithmic development. Over the last two decades, numerous algorithms have been proposed to solve frequent pattern mining or some of its variants, and the interest in this problem still persists [45, 75]. Different frameworks have been defined for frequent pattern mining. The most common one is the support-based framework, in which itemsets with frequency above a given threshold are found. However, such itemsets may sometimes not represent interesting positive *correlations* between items because they do not normalize for the absolute frequencies of the items. Consequently, alternative measures for interestingness have been defined in the literature [7, 11, 16, 63]. This chapter will focus on the support-based framework because the algorithms based on the interestingness

C. C. Aggarwal (✉)
IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, USA
e-mail: charu@us.ibm.com

M. A. Bhuiyan · M. A. Hasan
Indiana University–Purdue University, Indianapolis, IN, USA
e-mail: mbhuiyan@cs.iupui.edu

M. A. Hasan
e-mail: alhasan@cs.iupui.edu

```
Algorithm Baseline Mining(Database: $\mathcal{T}$, Minimum Support: $s$)
  begin
    $\mathcal{FP} = \{\}$;
    Insert length-one frequent pattern in $\mathcal{FP}$
    until all frequent patterns in $\mathcal{FP}$ are explored do
    begin
      Generate a candidate pattern $P$ from one (or more) frequent
              pattern(s) in $\mathcal{FP}$
      if support($P, \mathcal{T}$) $\geq s$
        Add $P$ to frequent pattern set $\mathcal{FP}$;
    end
end
```

**Fig. 2.1** A generic frequent pattern mining algorithm

framework are provided in a different chapter. Surveys on frequent pattern mining may be found in [26, 33].

One of the main reasons for the high level of interest in frequent pattern mining algorithms is due to the computational challenge of the task. Even for a moderate sized dataset, the search space of FPM is enormous, which is exponential to the length of the transactions in the dataset. This naturally creates challenges for itemset generation, when the support levels are low. In fact, in most practical scenarios, the support levels at which one can mine the corresponding itemsets are limited (bounded below) by the memory and computational constraints. Therefore, it is critical to be able to perform the analysis in a space- and time-efficient way. During the first few years of research in this area, the primary focus of work was to find FPM algorithms with better computational efficiency.

Several classes of algorithms have been developed for frequent pattern mining, many of which are closely related to one another. In fact, the execution tree of all the algorithms is mostly different in terms of the order in which the patterns are explored, and whether the counting work done for different candidates is independent of one another. To explain this point, we introduce a primitive "baseline" algorithm that forms the heart of most frequent pattern mining algorithms.

Figure 2.1 presents the pseudocode for a very simple "baseline" frequent pattern mining algorithm. The algorithm takes the transaction database $\mathcal{T}$ and a user-defined support value $s$ as input. It first populates all length-one frequent patterns in a frequent pattern data-store, $\mathcal{FP}$. Then it generates a candidate pattern and computes its support in the database. If the support of the candidate pattern is equal or higher than the minimum support threshold the pattern is stored in $\mathcal{FP}$. The process continues until all the frequent patterns from the database are found.

In the aforementioned algorithm, candidate patterns are generated from the previously generated frequent patterns. Then, the transaction database is used to determine which of the candidates are truly frequent patterns. The key issues of computational efficiency arise in terms of generating the candidate patterns in an orderly and carefully designed fashion, pruning irrelevant and duplicate candidates, and using well chosen tricks to minimize the work in counting the candidates. Clearly, the effectiveness of these different strategies depend on each other. For example, the effectiveness of a pruning strategy may be dependent on the order of exploration of the candidates (level-wise vs. depth first), and the effectiveness of counting is also dependent on the order of exploration because the work done for counting at the higher levels (shorter itemsets) can be reused at the lower levels (longer itemsets) with certain strategies, such as those explored in *TreeProjection* and *FP-growth*. Surprising as it might seem, virtually all frequent pattern mining algorithms can be considered complex variations of this simple baseline pseudocode. The major challenge of all of these methods is that the number of frequent patterns and candidate patterns can sometimes be large. This is a fundamental problem of frequent pattern mining although it is possible to speed up the counting of the different candidate patterns with the use of various tricks such as database projections. An analysis on the number of candidate patterns may be found in [25].

The candidate generation process of the earliest algorithms used joins. The original *Apriori* algorithm belongs to this category [1]. Although *Apriori* is presented as a join-based algorithm, it can be shown that the algorithm is a breadth first exploration of a structured arrangement of the itemsets, known as a *lexicographic tree* or *enumeration tree*. Therefore, later classes of algorithms explicitly discuss tree-based enumeration [4, 5]. The algorithms assume a lexicographic tree (or enumeration tree) of candidate patterns and explore the tree using breadth-first or depth-first strategies. The use of the enumeration tree forms the basis for understanding search space decomposition, as in the case of the *TreeProjection* algorithm [5]. The enumeration tree concept is very useful because it provides an understanding of how the search space of candidate patterns may be explored in a systematic and non-redundant way. Frequent pattern mining algorithms typically need to evaluate the support of frequent portions of the enumeration tree, and also rule out an additional layer of infrequent extensions of the frequent nodes in the enumeration tree. This makes the candidate space of all frequent pattern mining algorithms virtually invariant unless one is interested in particular types of patterns such as maximal patterns.

The enumeration tree is defined on the prefixes of frequent itemsets, and will be introduced later in this chapter. Later algorithms such as *FP-growth* perform suffix-based recursive exploration of the search space. In other words, the frequent patterns with a particular pattern as a suffix are explored at one time. This is because *FP-growth* uses the opposite item ordering convention as most enumeration tree algorithms though the recursive exploration order of *FP-growth* is similar to an enumeration tree.

Note that all classes of algorithms, implicitly or explicitly, explore the search space of patterns defined by an enumeration tree of frequent patterns with different strategies such as joins, prefix-based depth-first exploration, or suffix-based depth-first exploration. However, there are significant differences in terms of the order in which the search space is explored, the pruning methods used, and how the counting is performed. In particular, certain projection-based methods help in reusing the counting work for $k$-itemsets for $(k + 1)$-itemsets with the use of the notion of projected databases. Many algorithms such as *TreeProjection* and *FP-growth* are able to achieve this goal.

**Table 2.1** Toy transaction database and frequent items of each transaction for a minimum support of 3

| tid | Items | Sorted frequent items |
|---|---|---|
| 2 | a,b,c,d,f,h | a,b,c,d,f |
| 3 | a,f,g | a,f |
| 4 | b,e,f,g | b,f,e |
| 5 | a,b,c,d,e,h | a,b,c,d,e |

This chapter is organized as follows. The remainder of this chapter discusses notations and definitions relevant to frequent pattern mining. Section 2 discusses join-based algorithms. Section 3 discusses tree-based algorithms. All the algorithms discussed in Sects. 2 and 3 extend prefixes of itemsets to generated frequent patterns. A number of methods that extend suffixes of frequent patterns are discussed in Sect. 4. Variants of frequent pattern mining, such as closed and maximal frequent pattern mining, are discussed in Sect. 5. Other optimized variations of frequent pattern mining algorithms are discussed in Sect. 6. Methods for reducing the number of passes, with the use of sampling and aggregation are proposed in Sect. 7. Finally, Sect. 8 concludes chapter with an overall summary.

## *1.1 Definitions*

In this section, we define several key concepts of frequent pattern mining (FPM) that we will use in the remaining part of the chapter.

Let, $\mathcal{T} = \{T_1, T_2, \dots, T_n\}$ be a transaction database, where each $T_i \in \mathcal{T}, \forall i = \{1 \dots n\}$ consists of a set of items, say $T_i = \{x_1, x_2, x_3, \dots x_l\}$. A set $P \subseteq T_i$ is called an itemset. The size of an itemset is defined by the number of items it contains. We will refer an itemset as $l\text{-}itemset$ (or $l\text{-}pattern$), if its size is $l$. The number of transactions containing $P$ is referred to as the *support* of $P$. A pattern $P$ is defined to be frequent if its support is at least equal to the the minimum threshold.

Table 2.1 depicts a toy database with 5 transactions ($T_1$, $T_2$ $T_3$, $T_4$ and $T_5$). The second column shows the items in each transaction. In the third column, we show the set of items that are frequent in the corresponding transaction for a minimum support value of 3. For example, the item $h$ in transaction with *tid* value of 2 is an infrequent item with a support value of 2. Therefore, it is not listed in the third column of the corresponding row. Similarly, the pattern $\{a, b\}$ (or, $ab$ in abbreviated form) is frequent because it has a support value of 3.

The frequent patterns are often used to generate *association rules*. Consider the rule $X \Rightarrow Y$, where $X$ and $Y$ are sets of items. The confidence of the rule $X \Rightarrow Y$ is the equal to the ratio of the support of $X \cup Y$ to that of the support of $X$. In other words, it can be viewed as the conditional probability that $Y$ occurs, given that $X$ has occurred. The support of the rule is equal to the support of $X \cup Y$. Association rule-generation is a two-phase process. The first phase determines all the frequent patterns at a given minimum support level. The second phase extracts all the rules from these patterns. The second phase is fairly trivial and with limited sophistication. Therefore, most of the algorithmic work in frequent pattern mining focusses on the

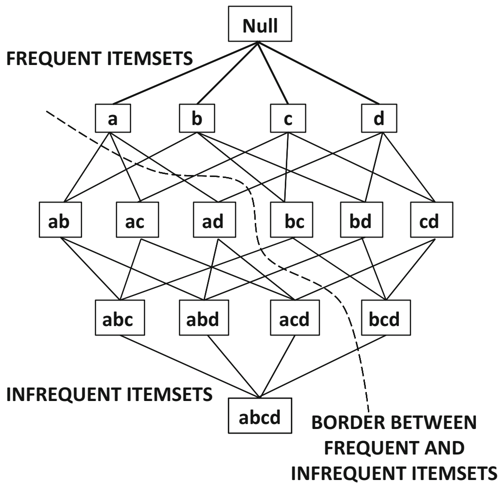

**Fig. 2.2** The lattice of itemsets

first phase. This chapter will also focus on the first phase of frequent pattern mining, which is generally considered more important and non-trivial.

Frequent patterns satisfy a *downward closure property*, according to which every subset of a frequent pattern is also frequent. This is because if a pattern $P$ is a subset of a transaction, then every pattern $P' \subseteq P$ will also be a subset of $T$. Therefore, the support of $P'$ can be no less than that of $P$. The space of exploration of frequent patterns can be arranged as a lattice, in which every node is one of the $2^d$ possible itemsets, and an edge represents an immediate subset relationship between these itemsets. An example of a lattice of possible itemsets for a universe of items corresponding to $\{a, b, c, d\}$ is illustrated in Fig. 2.2. The lattice represents the search of frequent patterns, and all frequent pattern mining algorithms must, in one way or another, traverse this lattice to identify the frequent nodes of this lattice. The lattice is separated into a frequent and an infrequent part with the use of a *border*. An example of a border is illustrated in Fig. 2.2. This border must satisfy the downward closure property.

The lattice can be traversed with a variety of strategies such as breadth-first or depth-first methods. Furthermore, *candidate nodes* of the lattice may be generated in many ways, such as using joins, or using lexicographic tree-based extensions. Many of these methods are conceptually equivalent to one another. The following discussion will provide an overview of the different strategies that are commonly used.

## 2 Join-Based Algorithms

Join-based algorithms generate $(k+1)$-candidates from frequent $k$-patterns with the use of joins. These candidates are then validated against the transaction database. The *Apriori* method uses joins to create candidates from frequent patterns, and is one of the earliest algorithms for frequent pattern mining.

## 2.1 Apriori Method

The most basic join-based algorithm is the *Apriori* method [1]. The *Apriori* approach uses a *level-wise* approach in which all frequent itemsets of length $k$ are generated before those of length $(k + 1)$. The main observation which is used for the *Apriori* algorithm is that every subset of a frequent pattern is also frequent. Therefore, *candidates* for frequent patterns of length $(k + 1)$ can be generated from *known* frequent patterns of length $k$ with the use of joins. A join is defined by pairs of frequent $k$-patterns that have at least $(k - 1)$ items in common. Specifically, consider a frequent pattern $\{i_1, i_2, i_3, i_4\}$ that is frequent, but has not yet been discovered because only itemsets of length 3 have been discovered so far. In this case, because the patterns $\{i_1, i_2, i_3\}$ and $\{i_1, i_2, i_4\}$ are frequent, they will be present in the set $\mathcal{F}_3$ of all frequent patterns with length $k = 3$. Note that this particular pair also has $k - 1 = 2$ items in common. By performing a join on this pair, it is possible to create the *candidate* pattern $\{i_1, i_2, i_3, i_4\}$. This pattern is referred to as a *candidate* because it might *possibly* be frequent, and one most either rule it in or rule it out by support counting. Therefore, this candidate is then *validated* against the transaction database by counting its support. Clearly, the design of an efficient support counting method plays a critical role in the overall efficiency of the process. Furthermore, it is important to note that the same candidate can be produced by joining multiple frequent patterns. For example, one might join $\{i_1, i_2, i_3\}$ and $\{i_2, i_3, i_4\}$ to achieve the same result. Therefore, in order to avoid duplication in candidate generation, two itemsets are joined only whether first $(k - 1)$ items are the same, based on a lexicographic ordering imposed on the items. This provides all the $(k + 1)$-candidates in a non-redundant way.

It should be pointed out that some candidates can be pruned out in an efficient way, without validating them against the transaction database. For any $(k+1)$-candidates, it is checked whether *all* its $k$ subsets are frequent. Although it is already known that two of its subsets contributing to the join are frequent, it is not known whether its remaining subsets are frequent. If all its subsets are not frequent, then the candidate can be pruned from consideration because of the downward closure property. This is known as the *Apriori* pruning trick. For example, in the previous case, if the itemset $\{i_1, i_3, i_4\}$ does not exist in the set of frequent 3-itemsets which have already been found, then the candidate itemset $\{i_1, i_2, i_3, i_4\}$ can be pruned from consideration with no further computational effort. This greatly speeds up the overall algorithm. The generation of 1-itemsets and 2-itemsets is usually performed in a specialized way with more efficient techniques.

Therefore, the basic *Apriori* algorithm can be described recursively in level-wise fashion. the overall algorithm comprises of three steps that are repeated over and over again, for different values of $k$, where $k$ is the length of the pattern generated in the current iteration. The four steps are those of (i) generation of candidate patterns $\mathcal{C}_{k+1}$ by using joins on the patterns in $\mathcal{F}_k$, (ii) the pruning of candidates from $\mathcal{C}_{k+1}$, for which all subsets to not lie in $\mathcal{F}_k$, and (iii) the validation of the patterns in $\mathcal{C}_{k+1}$ against the transaction database $\mathcal{T}$, to determine the subset of $\mathcal{C}_{k+1}$ which is truly frequent. The algorithm is terminated, when the set of frequent $k$-patterns $\mathcal{F}_k$ in a given iteration is empty. The pseudo-code of the overall procedure is presented in Fig. 2.3.

**Fig. 2.3** The *Apriori* algorithm

```
Algorithm Apriori(Database: $\mathcal{T}$, Support: $s$)
begin
  Generate frequent 1-patterns and 2-patterns
    using specialized counting methods and
    denote by $\mathcal{F}_1$ and $\mathcal{F}_2$;
  $k := 2$;
  while $\mathcal{F}_k$ is not empty do
  begin
    Generate $\mathcal{C}_{k+1}$ by using joins on $\mathcal{F}_k$;
    Prune $\mathcal{C}_{k+1}$ with Apriori subset pruning trick;
    Generate $\mathcal{F}_{k+1}$ by counting candidates in
      $\mathcal{C}_{k+1}$ with respect to $\mathcal{T}$ at support $s$;
    $k := k + 1$;
  end
  return $\cup_{i=1}^{k} \mathcal{F}_i$;
end
```

The computationally intensive procedure in this case is the counting of the candidates in $\mathcal{C}_{k+1}$ with respect to the transaction database $\mathcal{T}$. Therefore, a number of optimizations and data structures have been proposed in [1] (and also the subsequent literature) to speed up the counting process. The data structure proposed in [1] is that of constructing a *hash-tree* to maintain the candidate patterns. A leaf node of the hash-tree contains a list of itemsets, whereas an interior node contains a hash-table. An itemset is mapped to a leaf node of the tree by defining a path from the root to the leaf node with the use of the hash function. At a node of level $i$, a hash function is applied to the $i$th item to decide which branch to follow. The itemsets in the leaf node are stored in sorted order. The tree is constructed recursively in top–down fashion, and a minimum threshold is imposed on the number of candidates in the leaf node.

To perform the counting, all possible $k$-itemsets which are subsets of a transaction are discovered in a *single* exploration of the hash-tree. To achieve this goal *all possible* paths in the hash tree that could correspond to subsets of the transaction, are followed in recursive fashion, to determine which leaf nodes are relevant to that transaction. After the leaf nodes have been discovered, the itemsets at these leaf nodes that are subsets of that transaction are isolated and their count is incremented. The actual selection of the relevant leaf nodes is performed by recursive traversal as follows. At the root node, all branches are followed such that *any* of the items in the transaction hash to one of branches.At a given interior node, if the $i$th item of the transaction was last hashed, then all items *following it* in the transaction are hashed to determine the possible children to follow. Thus, by following all these paths, the relevant leaf nodes in the tree are determined. The candidates in the leaf node are stored in sorted order, and can be compared efficiently to the hashed sequence of items in the transaction to determine whether they are relevant. This provides a count of the itemsets relevant to the transaction. This process is repeated for each transaction to determine the final support count for each itemset. It should be pointed out that the reason for using a hash function at the intermediate nodes is to reduce the branching factor of the hash tree. However, if desired, a trie can be used explicitly, in which the degree of a

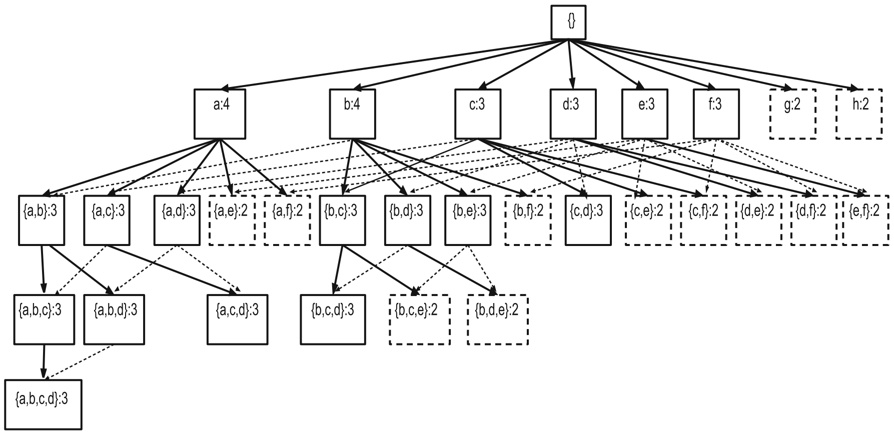

**Fig. 2.4** Execution tree of *Apriori* algorithm

node is potentially of the order of the total number of items. An example of such an implementation is provided in [12], and it seems to work quite well. An algorithm that shares some similarities to the *Apriori* method, was independently proposed in [44], and subsequently a combined work was published in [3].

Figure 2.4 illustrates the execution tree of the join-based *Apriori* algorithm over the toy transaction database mentioned in Table 2.1 for minimum support value 3. As mentioned in the pseudocode of *Apriori*, a candidate $k$-patterns are generated by joining two frequent itemset of size $(k - 1)$. For example, at level 3, the pattern $\{a, b, c\}$ is generated by joining $\{a, b\}$ and $\{a, c\}$. After generating the candidate patterns, the support of the patterns is computed by scanning every transaction in the database and determining the frequent ones. In Fig. 2.4, a candidate patterns is shown in a box along with its support value. A frequent candidate is shown in a solid box, and an infrequent candidate is shown in a dotted box. An edge represents the join relationship between a candidate pattern of size $k$ and a frequent pattern of size $(k-1)$ such that the latter is used to generate the earlier. The figure also illustrates the fact that a pair of frequent patterns are used to generate a candidate pattern, whereas no candidates are generated from an infrequent pattern.

### 2.1.1 Apriori Optimizations

Numerous optimizations were proposed for the *Apriori* algorithm [1] that are referred to as *AprioriTid* and *AprioriHybrid* respectively. In the *AprioriTid* algorithm, each transaction is replaced by a shorter transaction or null transaction) during the $k$th phase. Let the set of $k + 1$-candidates in $\mathcal{C}_{k+1}$ that are contained in transaction $T$ be denoted by $\mathcal{R}(T, \mathcal{C}_{k+1})$. This set $\mathcal{R}(T, \mathcal{C}_{k+1})$ is added to a newly created transaction database $\mathcal{T}'_k$. If the set $\mathcal{R}(T, \mathcal{C}_{k+1})$ is null, then clearly, a number of different tradeoffs exist with the use of such an approach.

- Because each newly created transaction in $\mathcal{T}_k'$ is much shorter, this makes subsequent support counting more efficient.
- In some cases, no candidate may be a subset of the transaction. Such a transaction can be dropped from the database because it does not contribute to the counting of support values.
- In other cases, more than one candidate may be a subset of the transaction, which will actually increase the overhead of the algorithm. Clearly, this is not a desirable scenario.

Thus, the first two factors improve the efficiency of the new representation, whereas the last factor worsens it. Typically, the impact of the last factor is greater in the early iterations, whereas the impact of the first two factors is greater in the later iterations. Therefore, to maximize the overall efficiency, a natural approach would be to *not* use this optimization in the early iterations, and apply it only in the later iterations. This variation is referred to as the *AprioriHybrid* algorithm [1]. Another optimization proposed in [9] is that the support of many patterns can be inferred from those of key patterns in the data. This is used to significantly enhance the efficiency of the approach.

Numerous other techniques have been proposed that use different techniques to optimize the original implementation of the *Apriori* algorithm. As an example, the method in [1] and [44] share a number of similarities but are somewhat different at the implementation level. A work that combines the ideas from these different pieces of work is presented in [3].

## 2.2 DHP Algorithm

The DHP algorithm, also known as the *Direct Hashing and Pruning* method [50], was proposed soon after the *Apriori* method. It proposes two main optimizations to speed up the algorithm. The first optimization is to prune the candidate itemsets in each iteration, and the second optimization is to trim the transactions to make the support-counting process more efficient.

To prune the itemsets, the algorithm tracks partial information about candidate $(k+1)$-itemsets, while explicitly counting the support of candidate $k$-itemsets. During the counting of candidate $k$-itemsets, all $(k+1)$ subsets of the transaction are found and hashed into a table that maintains the counts of the number of subsets hashed into each entry. During the phase of counting $(k+1)$-itemsets, the counts in the hash table are retrieved for each itemset. Clearly, these counts are overestimates because of possible collisions in the hash table. Those itemsets for which the counts are below the user-specified support level are then pruned from consideration.

A second optimization proposed in DHP is that of transaction trimming. A key observation here is that if an item does not appear in at least $k$ frequent itemsets in $\mathcal{F}_k$, then no frequent itemset in $\mathcal{F}_{k+1}$ will contain that item. This follows from the fact that there should be at least $k$ (immediate) subsets of each frequent pattern in $\mathcal{F}_{k+1}$

containing a particular item that also occur in $\mathcal{F}_k$ and also contain that item. This implies that if an item does not appear in at least $k$ frequent itemsets in $\mathcal{F}_k$, then that item is no longer relevant to further support counting for finding frequent patterns. Therefore, that item can be trimmed from the transaction. This reduces the width of the transaction, and increases the efficiency of processing. The overhead from the data structures is significant, and most of the advantages are obtained for patterns of smaller length such as 2-itemsets. It was pointed out in later work [46, 47, 60] that the use of triangular arrays for support counting of 2-itemsets in the context of the *Apriori* method is even more efficient than such an approach.

## 2.3 Special Tricks for 2-Itemset Counting

A number of special tricks can be used to improve the effectiveness of 2-itemset counting. The case of 2-itemset counting is special and is often similar for the case of join-based and tree-based algorithms. As mentioned above, one approach is to use a triangular array that maintains the counts of the $k$-patterns explicitly. For each transaction, a nested loop can be used to explore all pairs of items in the transaction and increment the corresponding counts in the triangular array. A number of caching tricks can be used [5] to improve data locality access during the counting process. However, if the number of possible items are very large, this will still be a very significant overhead because it is needed to maintain an entry for each pair of items. This is also very wasteful, if many of the 1-items are not frequent, or some of the 2-item counts are zero. Therefore, a possible approach would be to first prune out all the 1-items which are not frequent. It is simply not necessary to count the support of a 2-itemset unless both of its constituent items are frequent. A hash table can then be used to maintain the frequency counts of the corresponding 2-itemsets. As before, the transactions are explored in a double nested loops, and all pairs of items are hashed into the table, with the caveat, that each of the individual items must be frequent. The set of itemsets which satisfy the support requirements are reported.

## 2.4 Pruning by Support Lower Bounding

Most of the pruning tricks discussed earlier prune itemsets when they are guaranteed *not* meet the required support threshold. It is also possible to skip the counting process for an itemset if the itemset is guaranteed to meet the support threshold. Of course, the caveat here is that the exact support of that itemset will not be available, beyond the knowledge that it meets the minimum threshold. This is sufficient in the case of many applications.

Consider two $k$-itemsets $A$ and $B$ that have $k-1$ items $A \cap B$ in common. Then, the union of the items in $A$ and $B$, denoted by $A \cup B$ will have exactly $k+1$ items. Then, if $sup(\cdot)$ represent the support of an itemset, then the support of $A \cup B$ can