Mining Frequent Itemsets Using Support Constraints

Ke Wang
National University of Singapore &
Simon Fraser University
wangk@cs.sfu.ca

Yu He
National University of Singapore
hey@comp.nus.edu.sg

Jiawei Han
Simon Fraser University
han@cs.sfu.ca

Abstract

Interesting patterns often occur at varied levels of support. The classic association mining based on a uniform minimum support, such as Apriori, either misses interesting patterns of low support or suffers from the bottleneck of itemset generation. A better solution is to exploit *support constraints*, which specify what minimum support is required for what itemsets, so that only necessary itemsets are generated. In this paper, we present a framework of frequent itemset mining in the presence of support constraints. Our approach is to "push" support constraints into the Apriori itemset generation so that the "best" minimum support is used for each itemset at run time to preserve the essence of Apriori.

1 Introduction

The *association rules mining*, first studied in [AIS93, AS94] for market-basket analysis, is to find all association rules above some user-specified minimum support and minimum confidence. The bottleneck of this problem is finding *frequent itemsets* (and support), i.e., itemsets that have a support above the minimum support. The importance of frequent itemsets goes far beyond market-basket analysis because they serve as an estimation of joint probabilities of events, thereby, useful whenever such estimation is required. For example, several recent studies have leveraged frequent itemsets to build intrusion detection models [LSM98], to construct classifiers [LHM98, MW99], to build Yahoo!-like information hierarchies [WZL99], and to discover emerging patterns [DL99]. We believe that more and more internet/web related data mining will require the ability of finding frequent itemsets.

Proceedings of the 26th VLDB Conference, Cairo, Egypt, 2000.

1.1 Apriori lives on a uniform minimum support

The best known search strategy of frequent itemests, called Apriori [AIS93, AS94], exploits the following property: if an itemset is frequent, so are all its subsets. Thus, Apriori ensures a level-wise generation of itemsets where each candidate k-itemset $\{i_1, \ldots, i_{k-2}, i_{k-1}, i_k\}$ is generated from two frequent $(k-1)$-itemsets $\{i_1, \ldots, i_{k-2}, i_{k-1}\}$ and $\{i_1, \ldots, i_{k-2}, i_k\}$. To make Apriori work, however, it is essential that all itemsets have a uniform minimum support. Consider what happens if the minimum support of $\{coffee, sugar, tea\}$ is 2% and the minimum support of $\{coffee, tea\}$, $\{sugar, tea\}$, $\{coffee, sugar\}$ is 5%: it is possible that $\{coffee, sugar, tea\}$ is frequent with respect to its minimum support, but none of $\{coffee, tea\}$, $\{sugar, tea\}$, $\{coffee, sugar\}$ is frequent with respect to their minimum support!

1.2 The reality is not uniform

In reality, the minimum support is not uniform. First, deviations and exceptions often have much lower support than general trends. For example, rules for accidents are much less supported than rules for non-accidents, but the former are often more interesting than the latter. Second, the support requirement varies with the support of items contained in an itemset. Rules containing *bread* and *milk* usually have higher support than rules containing *food processor* and *pan*. A similar scenario is that dense attributes such as *States* have less support than sparse attributes such as *Gender*. Third, item presence has less support than item absence. Fourth, the support requirement varies at different concept levels of items [HF95, SA95]. Finally, hierarchical classification like

[WZL99] requires feature terms to be discovered at different concept levels, thereby, requiring a non-uniform minimum support.

Given that existing algorithms assume a uniform minimum support, the best one can do is to apply such algorithms at the lowest minimum support ever specified and filter the result using higher minimum supports. This method will generate many candidates that are later discarded. From our experience (see Section 7), the increase of candidates often causes a non-linear increase of execution time and a drastic performance deterioration once page swapping takes place between memory and disk, during the support counting where candidates were read from disk for each transaction. In the world of non-uniform minimum support, we need a technique that finds the itemsets above their minimum supports without forcing the lowest minimum support across all itemsets.

1.3 Our approach

We propose *support constraints* as a way to specify general constraints on minimum support. Informally, a support constraint specifies what itemsets are required to satisfy what minimum support. We consider support constraints of the form $SC_i(B_1, \ldots, B_s) \geq \theta_i$, where $s \geq 0$. Each B_j, called a *bin*, is a set of items that need not be distinguished with respect to the specification of minimum support. θ_i is a minimum support in the range [0..1]. The above support constraint specifies that any itemset containing at least one item from each B_j has the minimum support θ_i. The topic of this paper is to "push" such support constraints into the itemset generation to prune candidates as early as possible. We illustrate this approach using an example.

Example 1.1 Consider four support constraints $SC_1(B_1, B_3) \geq 0.2$, $SC_2(B_3) \geq 0.4$, $SC_3(B_2) \geq 0.6$, and $SC_0() \geq 0.8$. Each bin B_i contains a disjoint set of items. We assume that if more than one constraint is applicable to an itemset, the constraint specifying the lowest minimum support is chosen. We will explain the rationale of this choice in Section 3. With this assumption, we have the following specifications. *Case (i)*: $SC_1(B_1, B_3) \geq 0.2$ specifies minimum support 0.2 for any itemset containing (probably more, same below) one item in each of B_1 and B_3. *Case (ii)*: $SC_2(B_3) \geq 0.4$ specifies minimum support 0.4 for any itemset containing one item in B_3, but no item in B_1 (otherwise, Case (i) applies). *Case (iii)*: $SC_3(B_2) \geq 0.6$ specifies minimum support 0.6 for any itemset containing one item in B_2, but no item in B_3 (otherwise, Case (ii) applies). *Case (iv)*: $SC_0() \geq 0.8$ specifies minimum support 0.8 for any other itemset (i.e., the default minimum support). There are two key issues in making use of these specifications:

Constraint pushing. On the one hand, we would like to treat these cases separately so that the highest possible minimum support is applied in each case. On the other hand, we would like to share the work done in different cases so that each itemset is generated at most once. For example, as in Apriori, we like to generate itemset $\{b_0, b_1, b_2\}$ in Case (iii) using $\{b_0, b_1\}$ generated in Case (iv) and $\{b_0, b_2\}$ generated in Case (iii), where b_i denotes an item from B_i. This requires the minimum support 0.6 of $\{b_0, b_1, b_2\}$ to be "pushed" down to $\{b_0, b_1\}$, on the ground that $\{b_0, b_1, b_2\}$ "depends on" $\{b_0, b_1\}$, and further down to $\{b_0\}$ and $\{b_1\}$. The pushed minimum support 0.6 is lower than the minimum support for $\{b_0, b_1\}$, $\{b_0\}$, $\{b_1\}$, i.e., 0.8, but is higher than the lowest minimum support 0.2. In this sense, we have pruned the minimum support 0.2 for certain itemsets and tightened up the search space.

Order sensitivity. The above has implicitly assumed that b_3 does not follow b_2 in the item ordering used by the Apriori itemset generation. If b_3 does follow b_2 in the item ordering, $\{b_0, b_1, b_2, b_3\}$ would depend on $\{b_0, b_1, b_2\}$, and the minimum support 0.2 for $\{b_0, b_1, b_2, b_3\}$ would be pushed down to $\{b_0, b_1, b_2\}$, and transitively, down to $\{b_0, b_1\}$, $\{b_0, b_2\}$, $\{b_0\}$, $\{b_1\}$, $\{b_2\}$. In this case, a lower minimum support is pushed, compared with 0.6, and more itemsets will be generated. The idea of tightening up the search space is to order items in such a way that allows the highest possible minimum support to be pushed. □

Here is the overview of our approach. We define a framework for specifying support constraints in Section 3. We then present a strategy for pushing support constraints into the Apriori itemset generation in Section 4. The constraint pushing exploits the dependency between itemsets, represented by an enumeration tree of sets of bins, and determines the highest minimum support to be pushed to each itemset. This phase makes use of the information of support constraints, but not the database. It turns out that the ordering of nodes in an enumeration tree drastically impacts the pushed minimum support. We present several ordering strategies to maximize the pushed minimum support in Section 5. At the itemset generation phase, candidates are generated as in Apriori and the pushed minimum support is used to determine whether a candidate is frequent. We call this strategy Adaptive Apriori to emphasize that the pushed minimum support is determined *individually* for each itemset and that Adaptive Apriori generalizes Apriori to non-uniform minimum support while preserving the essence of Apriori. An example in Section 6 illustrates our mining algorithm. We evaluate the effectiveness of this approach in Section 7. Finally, we conclude the paper in Section 8.

2 Related work

The support-based Apriori pruning was first studied in [AIS93, AS94], and a similar idea in [MTV94].

Nearly all later frequent itemset minings rely on Apriori as a basic pruning strategy. Another strategy is to push certain constraints into the itemset generation. However, none of these approaches considers non-uniform minimum support. The correlation approach [AY98, BMS97] considers the support requirement relative to the independence assumption, but not general support constraints or constraint pushing. Instead of abandoning the support requirement like in [C*00], our approach is to make it more realistic by allowing different support requirements for different itemsets. [HPY00] generates frequent itemsets without using the Apriori itemset generation, but still critically relies on a uniform support requirement.

To our knowledge, [LHM99] is the only work explicitly dealing with non-uniform minimum support. In [LHM99], a *minimum item support* (or MIS) is associated with each item, and the minimum support of an itemset is defined to be the lowest MIS associated with the items in the itemset. This specification is unnatural for three reasons. (i) The MIS of individual items has to reflect the minimum support of unseen itemsets at the specification time. (ii) In some applications the user may have a minimum support for an itemset, e.g., $\{white, male\}$, as a single concept, but not for individual items in the itemset (e.g., *white* or *male*). The minimum support for $\{white, male\}$ is usually lower than that for *white* or *male*. (iii) Different minimum supports cannot be specified for two itemsets, like $\{white, male\}$ and $\{white, male, grad\}$, if a common item has the lowest MIS, like *white*. We allow the support requirement to be specified directly on itemsets, thereby, overcoming these difficulties.

3 Specifying support constraints

As in [AIS93, AS94], the database is a collection of *transactions*. Each transaction is a set of *items* taken from a fixed universe. A *k-itemset* is a set of k items. The *support* of an itemset I, denoted $sup(I)$, is the fraction of the transactions containing all the items in I.

3.1 The support specification

The task of support specification is to specify the minimum support for each itemset. Clearly, it is not practical to enumerate all itemsets. Our approach is to partition the set of items into *bins*, denoted as B_j, such that items that need not be distinguished in the specification are in the same bin. Therefore, for a bag (i.e., multiset) $\beta = \{B_1, \ldots, B_k\}$ of bins, all k-itemsets $\{i_1, \ldots, i_k\}$, $i_j \in B_j$, have the same minimum support. β is called the *schema* of itemsets $\{i_1, \ldots, i_k\}$. To specify the minimum support for itemsets, we need only to specify the minimum support for schemas. This motivates the notion of support constraints.

Definition 3.1 (Support constraints) A *support constraint (SC)* has the form $SC_i(l_1, \ldots, l_s) \geq \theta_i$ (or simply $SC_i \geq \theta_i$), $s \geq 0$. Each l_j is either a bin or a variable for bins. θ_i, called a *minimum support*, is a function over $l_1, \ldots, l_s$ and returns a real in [0..1]. The order of l_j's does not matter and l_j may repeat. A SC is *ground* if it contains no variable, otherwise, *non-ground*. A non-ground SC can be *instantiated* to a ground SC by replacing each variable with a bin. A *support specification* is a non-empty set of SCs. □

There are two considerations in interpreting a SC. First, we can interpret a SC either as specifying *some* items in an itemset, called the *open interpretation*, or as specifying *all* items in an itemset, called the *closed interpretation*. Second, a choice must be made if an itemset "matches" the item specification of more than one SC. Consider itemset $I = \{b_1, b_2, b_3, b_4\}$ of support 0.15, and $SC_1(B_1, B_2) \geq 0.1$ and $SC_2(B_3, B_4) \geq 0.2$, where b_i is an item in B_i. In the open interpretation, I matches the item specification of both SCs. Therefore, whether I is frequent depends on which SC is used as the minimum support for I. Our decision is that the lower minimum support 0.1 prevails. The rationale is simple: the minimum support should not be increased by adding more items.

Definition 3.2 (Frequent itemsets) An itemset I *matches* a ground $SC_i \geq \theta_i$ in the open interpretation if I contains at least one item from each bin in SC_i and these items are distinct. An itemset I *matches* a ground $SC_i \geq \theta_i$ in the closed interpretation if I contains exactly one item from each bin in SC_i and these items are distinct. An itemset I *matches* a non-ground SC if I matches some instantiation of the SC. The *minimum support* of itemset I, denoted $minsup(I)$, is the lowest θ_i of all $SC_i \geq \theta_i$ matched by I. If I matches no SC, $minsup(I)$ is undefined. An itemset I is *frequent* if $minsup(I)$ is defined and $sup(I) \geq minsup(I)$. □

The notion of "match" and *minsup* can be extended to schemas in a natural way. A schema β *matches* a ground $SC_i \geq \theta_i$ in the open interpretation if SC_i is a sub-bag of β[1]. A schema β *matches* a ground $SC_i \geq \theta_i$ in the close interpretation if $SC_i = \beta$. A schema β *matches* a non-ground $SC_i \geq \theta_i$ if β matches some instantiation of the SC.

Let $minsup(\beta)$ denote the minimum support for (the itemsets of) schema β. In the open interpretation, for ground $SC_1(\beta_1) \geq \theta_1$ and $SC_2(\beta_2) \geq \theta_2$, if $\beta_1 \supseteq \beta_2$ and $\theta_1 > \theta_2$, $SC_1(\beta_1) \geq \theta_1$ is never used. In fact, if any itemset I matches $SC_1(\beta_1) \geq \theta_1$, I also matches $SC_2(\beta_2) \geq \theta_2$, and we always use the lower θ_2 as the minimum support for I. In this sense, $SC_1(\beta_1) \geq \theta_1$ is *redundant*. We assume that all redundant SCs are removed. With this assumption, a SC of

[1]A bag x is a sub-bag of a bag y if x is a subset of y with duplicates considered.

the form $SC_i() \geq \theta_i$, if specified, must have the highest minimum support, therefore, is used only when no other SC is matched. For this reason, $SC_i() \geq \theta_i$ is called the *default SC*.

Example 3.1 (The running example) Consider the transactions and support specification in Figure 1 in the open interpretation. Each item is represented by an integer from 0 to 8. For any itemset I containing an item from B_1 and an item from B_3, I matches both $SC_1(B_1, B_3) \geq 0.2$ and $SC_2(B_3) \geq 0.4$. But $minsup(I) = 0.2$ because the lowest minimum support of matched SCs is used. Some examples of such I are $\{0,2\}$, $\{0,2,3\}$, and $\{2,3,4\}$. $\{0,2\}$ is frequent, but $\{0,2,3\}$ and $\{2,3,4\}$ are not. The minimum support of $\{2,4,7\}$, $\{2,4,8\}$, $\{4,7,8\}$, and $\{2,4,7,8\}$ is 0.6 because these itemsets match only $SC_3(B_2) \geq 0.6$. These itemsets are frequent. $\{2,7\}$ and $\{2,8\}$ match only $SC_0() \geq 0.8$, and $\{2,7\}$ is frequent, but $\{2,8\}$ is not. □

Example 3.2 An example of non-ground SCs in the closed interpretation is $SC_i(V_1, \ldots, V_k) \geq sup(V_1) \times \ldots \times sup(V_k)$, $1 \leq k \leq 4$, where V_i are variables and each bin B_i contains the items of the same support, denoted $sup(B_i)$. Each instantiation $SC_i(B_1, \ldots, B_k) \geq sup(B_1) \times \ldots \times sup(B_k)$ specifies the minimum support relative to the independence assumption about item occurrence. Due to the closed interpretation, any itemset containing more than 4 items has an undefined minimum support. □

In $SC_i \geq \theta_i$, θ_i should be "evaluable" at the *specification time*. For example, $SC(\alpha, \beta) \geq minconf \times sup(\alpha)$ does not satisfy this requirement, where α and β are schemas and each bin contains a single item, because $sup(\alpha)$ is unknown at the specification time. Even at the itemset generation, $sup(\alpha)$ is known only for frequent itemsets α.

The notion of support constraints generalizes several existing classes of constraints. The classic uniform minimum support [AIS93, AS94] can be specified by one default $SC_i() \geq \theta_i$ with θ_i being the usual minimum support. The item constraints [SVA97] can be specified by non-default SCs in which all minimum supports are equal. To model the MIS specification in [LHM99], we can group the items of the same support into a bin and specify the non-ground $SC_i(V_1, \ldots, V_k) \geq min\{sup(V_1), \ldots, sup(V_k)\}$ in the closed interpretation, where V_j are variables for bins. However, it is not hard to see that the MIS specification cannot model the specification in Example 3.2 nor the specification: $SC_1(B_1, B_2, B_3) \geq 0.2$, $SC_2(B_1, B_3) \geq 0.3$, and $SC_2(B_2, B_3) \geq 0.4$.

3.2 Typical scenarios of specification

Until now, we have not said much about how the end user determines bins B_j and minimum support θ_i in a SC. Though this decision largely depends on applications, we consider several typical scenarios and hope that they are indicative to the end user.

Support-based specification. Example 3.2 illustrates three points. (a) A bin B_j contains similarly supported items and θ_i is a function of some representative supports of of bins (such as *max*, *min*, or *avg*). Such bins can be found by computing the support of items in one pass of the transactions and then clustering items based on their support. The number of bins can be optionally specified by the user. (b) θ_i can be either chosen from a menu of built-in functions or supplied by the user. (c) Without a particular schema in mind for specification, a generic specification given by a non-ground SC can be used.

Concept-based specification. In the presence of a item concept hierarchy, it is desirable to specify SCs based on the generality of the concept of items. For example, $SC_1(c_1, c_2) \geq 2 \times \frac{sup(c_1)}{m} \times \frac{sup(c_2)}{n}$ states that any itemset containing at least one child of c_1 and one child of c_2 has the minimum support $2 \times \frac{sup(c_1)}{m} \times \frac{sup(c_2)}{n}$, where c_1 and c_2 are variables representing concepts, and m and n are the number of child concepts of c_1 and c_2.

Attribute-based specification. For a database in the form of a relational table, it makes sense for each bin to correspond to the set of attribute/value pairs from the same attribute. For example, if *States* and *Gender* are attributes in the table, $SC_1(States, Gender) \geq \frac{N}{50} \times \frac{N}{2}$ specifies that any itemset containing a state code and a gender has the minimum support $\frac{N}{50} \times \frac{N}{2}$, where N is the number of tuples in the relational table, $\frac{N}{50}$ and $\frac{N}{2}$ are the average support of state codes and the average support of gender.

Enumeration-based specification. The most flexible specification is explicitly enumerating the items in a bin, on the basis that they are not distinguishable with respect to the specification. For example, $SC_1(B_1, B_2) \geq 0.1$, where $B_1 = \{milk, cheese\}$ and $B_2 = \{boots, sock\}$, says that any itemset containing at least one item in B_1 and one item in B_2 has minimum support 0.1. In this case, the user is interested in only *milk* and *cheese*, rather than all dairy products, and only *boots* and *sock*, rather than all footwear products.

For the rest of the paper, we assume that a support specification is given.

4 Adaptive Apriori

A key idea of our approach is to push SCs following the "dependency chain" in the Apriori itemset generation. This dependency is best described by a schema enumeration tree. In a *schema enumeration tree*, each node (except the root) is labeled by a bin B_i. A node v represents the schema given by the labels $B_1 \ldots B_k$ along the path from the root to v. If database

TID	Items
100	0,2,7
200	0,4,7,8
300	2,4,5,7,8
400	1,2,4,7,8
500	2,4,6,7,8

bins

B_0	1,7,8
B_1	2,6
B_2	4,5
B_3	0,3

a specification

$SC_0() \geq 0.8$
$SC_1(B_1, B_3) \geq 0.2$
$SC_2(B_3) \geq 0.4$
$SC_3(B_2) \geq 0.6$

Figure 1: The running example

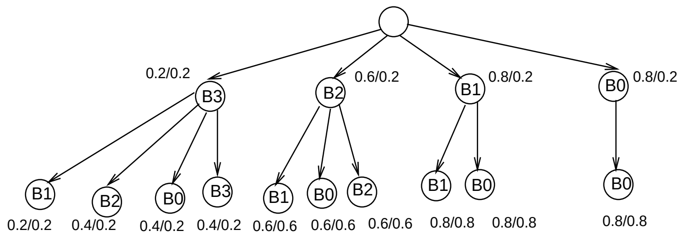


Figure 2: A schema enumeration tree, marked with *Sminsup*/*Pminsup*

a schema enumeration tree contains two sibling nodes representing schemas $s_1 = B_1 \ldots B_{k-2}B_{k-1}$ and $s_2 = B_1 \ldots B_{k-2}B_k$, where s_1 is on the left of s_2 (if $B_{k-1} \neq B_k$), the schema enumeration tree also contains the node representing schema $s = B_1 \ldots B_{k-2}B_{k-1}B_k$, as a child of the node for s_1. s_1 and s_2 are called *generating schemas* of s. Every schema *depends on* its generating schemas in that the former is constructed by the latter. For example, in Figure 2, B_2B_1 depends on B_2 and B_1, but not on B_1B_0 or B_3.

Several comments follow. (i) Unlike the static lexical ordering in a standard set enumeration tree [R92], the ordering of nodes in a schema enumeration tree is determined dynamically on a per-node basis to achieve a certain optimality of constraint pushing. We will consider the ordering issue in Section 5. (ii) There is an one-to-one correspondence between nodes and schemas represented by nodes, and the terms "schema" and "node" are interchangeable. (iii) There should be no confusion between B_i as a label and B_i as a schema of length 1. As a label, B_i can occur at several nodes (like B_2 in Figure 2), but as a schema, B_i is represented by a unique node. (iv) We can associate *minsup* with nodes, in the way of associating it with schemas. (v) A label B_i is allowed to repeat on a path to cover itemsets containing more than one item from B_i.

4.1 The pushed minimum support

Consider schema $s = B_1 \ldots B_{k-2}B_{k-1}B_k$, and its generating schemas $s_1 = B_1 \ldots B_{k-2}B_{k-1}$ and $s_2 = B_1 \ldots B_{k-2}B_k$. In the case of non-uniform minimum support, *minsup*(s), *minsup*(s_1), *minsup*(s_2) are not always the same, and the Apriori-generation of frequent itemsets of s from those of s_1 and s_2 is lost. Our approach is to replace *minsup* with a new function, denoted by *Pminsup*, such that (i) *Pminsup* defines a superset of the frequent itemsets defined by *minsup* and (ii) we can generate this superset in the manner of Apriori. Let us formalize this idea.

For any function f from schemas to [0..1], we say that an itemset I of schema s is *frequent*(f) if *sup*$(I) \geq f(s)$. Let $F(f)$ denote the set of *frequent*(f) itemsets.

Definition 4.1 (*Pminsup*) Let *Pminsup* be a function from (the schemas of) schema enumeration tree T to [0..1] satisfying:

- *Completeness*: For every schema s in T such that *minsup*(s) is defined, *Pminsup*$(s) \leq$ *minsup*(s);
- *Apriori-like*: For every schema s and its generating schemas s_1 and s_2, whenever $\{i_1, \ldots, i_{k-2}, i_{k-1}, i_k\}$ of s is *frequent*(*Pminsup*), so are $\{i_1, \ldots, i_{k-2}, i_{k-1}\}$ of s_1 and $\{i_1, \ldots, i_{k-2}, i_k\}$ of s_2;
- *Maximality*: *Pminsup* is maximal with respect to the above Completeness and Apriori-like. □

Pminsup is called the *pushed minimum support* with respect to T and *minsup*. □

Completeness ensures that F(*Pminsup*) is a superset of F(*minsup*). Apriori-like ensures the Apriori generation of F(*Pminsup*). Maximality ensures that F(*Pminsup*) is tightest to satisfy these properties. By replacing *minsup* with *Pminsup*, we are guaranteed to find a tight superset of F(*minsup*) in the manner of Apriori. This strategy is referred as to Adaptive Apriori. The novelty of Adaptive Apriori is that it breaks the barrier of uniform minimum support by defining the "best" minimum support, i.e., *Pminsup*, for each schema individually while preserving the essence of Apriori.

At this point, two questions need to be answered. First, how do we determine *Pminsup* with respect

notation	meaning
s	a node or schema
$L(s)$	the label of s
$subtree(s)$	the subtree rooted at s
$\sigma(s)$	the set of SCs in $subtree(s)$
$RS(s)$	the set of right siblings of s plus s itself
$LS(s)$	the set of left siblings of s
$minsup(s)$	the minimum support of s
$Sminsup(s)$	the lowest minimum support in $\sigma(s)$
$Pminsup(s)$	the pushed minimum support of s

Table 1: Notation for a schema enumeration tree

to a *given* schema enumeration tree T? Second, how do we generate a schema enumeration tree for which *Pminsup* is maximized? We answer the first question in the rest of this section and answer the second question in Section 5.

In the rest of the paper, we shall use the notation in Table 1. For example, for schema $s = B_3B_2$ in Figure 2, $L(s)$, the label of node s, is the B_2; $subtree(s)$ is the subtree rooted at s (not shown); $\sigma(s)$ contains all SCs except $SC_1(B_1, B_3) \geq 0.2$ because label B_1 does not occur in $subtree(s)$; $Sminsup(s)$ is the lowest minimum support in $\sigma(s)$, i.e., 0.4; $RS(s)$ contains schemas B_3B_2, B_3B_0, B_3B_3; and $LS(s)$ contains schema B_3B_1. Notice that while *minsup* only depends on the problem specification, *Pminsup* and *Sminsup* also depend on the schema enumeration tree used.

4.2 Determining *Pminsup*

Consider the running example and Figure 2. In $subtree(B_2)$, no schema will match $SC_1(B_1, B_3) \geq 0.2$ and $SC_2(B_3) \geq 0.4$ because label B_3 does not occur in the subtree. In this sense, these SCs or minimum supports are pruned from $subtree(B_2)$. The same goes for $subtree(B_1)$ and $subtree(B_0)$. In general, for generating nodes l and r (which must be siblings) with l on the left and r on the right, the node generated by l and r is a child of l and has label $L(r)$. Therefore, label $L(r)$ occurs in $subtree(l)$, but label $L(l)$ never occurs in $subtree(r)$. This has two implications stated below.

Corollary 4.1 Consider any node v in a schema enumeration tree T.

1. Only the labels of nodes in $RS(v)$ can occur in $subtree(v)$. As such, all SCs containing labels of nodes in $LS(v)$ are pruned from $subtree(v)$.

2. Only the nodes in $subtree(v)$ and in $subtree(u)$ for $u \in LS(v)$ depend on v. Such as, $Pminsup(v) = min\{Sminsup(u) \mid u \in LS(v) \cup \{v\}\}$.

Example 4.1 In Figure 2, each schema s is marked by $Sminsup(s)/Pminsup(s)$. Since label B_3 does not occur in $subtree(B_2)$, all SCs containing B_3 are pruned in $subtree(B_2)$, so $\sigma(B_2) = \{SC_0() \geq 0.8,\ SC_3(B_2) \geq 0.6\}$ and $Sminsup(B_2) = 0.6$. Similarly, $Sminsup(B_3) = 0.2$. $Pminsup(B_2) = min\{Sminsup(B_3), Sminsup(B_2)\} = 0.2$. Similarly, $Pminsup(s) = 0.6$ for $s = B_2B_1, s = B_2B_0, s = B_2B_2$ because $SC_1 \geq 0.2$ and $SC_2 \geq 0.4$ are pruned in $subtree(s)$, and $Pminsup(s) = 0.8$ for $s = B_1B_1, s = B_1B_0, s = B_0B_0$ because $SC_1 \geq 0.2$, $SC_2 \geq 0.4$, and $SC_3 \geq 0.6$ are pruned from $subtree(s)$. □

4.3 The characteristic of *Pminsup*

To get insights into the benefit of using *Pminsup*, we analyze how *Pminsup* changes in a schema enumeration tree. Refer to Table 1 for notation. As we move from a left sibling l to a right sibling r, Corollary 4.1(1) implies that $L(l)$ is pruned from $subtree(r)$, thereby, $\sigma(r) \subseteq \sigma(l)$ and $Sminsup(l) \leq Sminsup(r)$. As we move from a parent node p to a child node c, $\sigma(c)$ is the set of SCs in $\sigma(p)$ matched by at least some schema in $subtree(c)$, thereby, $\sigma(c) \subseteq \sigma(p)$ and $Sminsup(p) \leq Sminsup(c)$. The following theorems summarize these characteristics, whose proofs are given in [WHH00].

Theorem 4.1 Consider a schema enumeration tree.

1. Let $s_1, \ldots, s_k$ be the schemas at siblings from left to right. Then (a) $Sminsup(s_i) \leq Sminsup(s_{i+1})$; (b) $Pminsup(s_i) = Pminsup(s_1) = Sminsup(s_1)$.

2. Let $s_1, \ldots, s_k$ be the schemas on a path starting from the root. Then (a) $Sminsup(s_i) \leq Sminsup(s_{i+1})$; (b) $Pminsup(s_i) \leq Sminsup(s_i) \leq Pminsup(s_{i+1})$. □

Theorem 4.1(2b) tells that *Pminsup* is never decreased by moving from a parent p to a child c. The next theorem characterizes when *Pminsup* is actually increased.

Theorem 4.2 Consider a parent node p and a child node c. The following are equivalent:

1. p has a left sibling p' such that $Sminsup(p') < Sminsup(p)$;

2. p has a left sibling p' such that $Sminsup(p')$ is pruned in $subtree(p)$;

3. $Pminsup(p) < Pminsup(c)$. □

In Figure 3 (which contains only the nodes for non-empty sets of candidates), since $Sminsup(B_3) < Sminsup(B_i)$, for $i = 2, 1, 0$, every child of schema B_i has a higher *Pminsup* than B_i does. Similarly, since $Sminsup(B_3B_1) < Sminsup(B_3B_2)$, every child of B_3B_2 has a higher *Pminsup* than B_3B_2 does. The

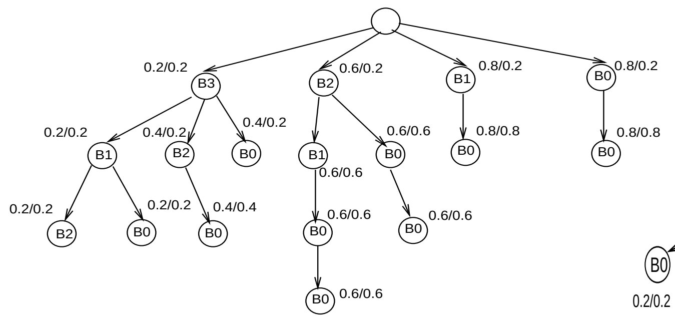


Figure 3: Nodes marked *Sminsup*/*Pminsup*

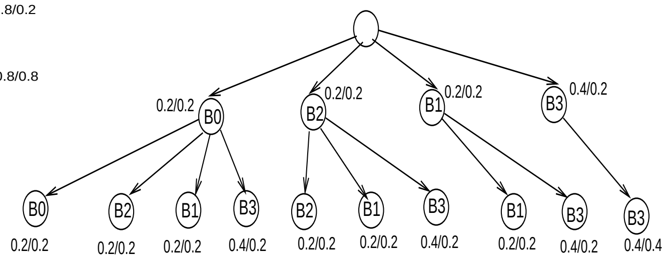


Figure 4: Nodes marked *Sminsup*/*Pminsup*

above theorems give a clear picture of how *Pminsup* changes in a schema enumeration tree: (a) All sibling nodes have the same *Pminsup*. (b) As we move down from a parent p to a child c, *Pminsup* never decreases. (c) Whether *Pminsup* is actually increased, thereby, tightening up the search space, depends on whether p has a left sibling with a lower *Sminsup*. It turns out that the ordering of siblings has a major impact on (c). We now examine how to order siblings to maximize *Pminsup*.

5 The ordering of nodes

Compare Figure 2 with Figure 3. The former is preferred because of higher *Pminsup* for most schemas. For example, $Pminsup(B_2B_1)$ and $Pminsup(B_0B_2)$ are 0.6 in Figure 2, but are 0.2 in Figure 3. This change is caused by placing labels B_1 and B_3 to the right end at level 1 in Figure 4, making $SC_1(B_1, B_3) \geq 0.2$ applicable in $subtree(B_2B_1)$ and $subtree(B_0B_2)$. Thus, the order of sibling nodes has a major impact on *Pminsup*. Unfortunately, no "optimal" order exists in general [WHH00]. Therefore, a reasonable thing to do is to order sibling nodes heuristically to maximize *Pminsup*. Let us consider such heuristic orderings.

Assume that $s_1, \ldots, s_k$ are the siblings from left to right. From Corollary 4.1(1), for $i < j$, $L(s_i)$ does not occur in $subtree(s_j)$, and all SCs containing $L(s_i)$ are pruned from $\sigma(s_j)$. Therefore, if we want to prune, as early as possible, the SCs specifying low minimum supports, label $L(s_1)$ for the first sibling s_1 should occur in such SCs. Subsequently, to determine $L(s_2)$ for the second sibling s_2, we remove the SCs containing $L(s_1)$ and repeat the same for the remaining SCs. The strategy is to greedily prune the lowest minimum support from all sibling subtrees on the right. Put another way, this strategy maximizes the chance of $Sminsup(s_i) < Sminsup(s_j)$, for all right siblings s_j of s_i, and thus, the chance of the condition in Theorem 4.2(3). This analysis leads to the first ordering strategy.

Strategy 1 Select the label specifying the lowest minimum support as the next sibling. □

Example 5.1 Consider ordering the child nodes of the root for the example in Figure 1. There is a tie between B_1 and B_3 as both specify the lowest minimum support in $SC_1(B_1, B_3) \geq 0.2$. Suppose that B_1 is selected as the first child. $SC_1(B_1, B_3) \geq 0.2$ is then pruned from $subtree(B_3)$, $subtree(B_2)$, and $subtree(B_0)$. We select B_3 as the second child because it specifies the lowest minimum support in the remaining SCs. $SC_2(B_3) \geq 0.4$ is then pruned from $subtree(B_2)$ and $subtree(B_0)$. Finally, we select B_2 and B_0 in that order. This gives the order $O_1 = B_1, B_3, B_2, B_0$ at level 1. $Sminsup(B_1) = 0.2$, $Sminsup(B_3) = 0.4$, $Sminsup(B_2) = 0.6$, and $Sminsup(B_0) = 0.8$. If we select B_3 as the first child instead, the order is $O_2 = B_3, B_2, B_1, B_0$. □

The above Strategy 1 is *dynamic* in that there is a separate round of selection for each sibling. In *static* Strategy 1 all siblings are selected in a single round, by ignoring the interaction between siblings. Our second strategy is to greedily prune as many SCs as possible. At each sibling, from left to right, we select the label that occurs in the most number of remaining SCs. In effect, this prunes all the SCs containing this label from the sibling subtrees on the right of the current sibling. By pruning as many SCs as possible, the default SC, which always specifies the highest minimum support, can be used as early as possible.

Strategy 2 Select the label specifying the most number of SCs as the next sibling. □

6 The algorithm

The algorithm expands the schema enumeration tree iteratively, one level per iteration. There are two phases in iteration k. Phase 1 generates new nodes s_i at level k and determines $Pminsup(s_i)$. This phase examines only the support specification and schemas, not the database or itemsets. Phase 2 generates $frequent(Pminsup)$ at nodes s_i, similar to Apriori. We shall focus on Phase 1. Each node p at level $k-1$ is associated with the set of SCs at p, $\sigma(p)$, and the relation T_p for $frequent(Pminsup)$ itemsets of p. (Refer to Table 1 for notation.) To expand to level k, three steps are performed in Phase 1. Step 1 creates child nodes s_i at level k and Step 2 orders these nodes according to one of the strategies proposed in Section 5.

Step 3 computes $\sigma(s_i)$ and $Pminsup(s_i)$ according to Corollary 4.1. We illustrate Step 3 by an example.

Example 6.1 As in Example 5.1, the nodes at level 1 are in the order $O_2 = B_3, B_2, B_1, B_0$. $\sigma(B_3)$ is initialized to $\sigma(root)$ because B_3 is the left-most child of the root. We delete label B_3 from the SCs in $\sigma(B_3)$ because every schema in $subtree(B_3)$ does contain B_3. Now $\sigma(B_3) = \{SC_0() \geq 0.8, SC_1(B_1) \geq 0.2, SC_2() \geq 0.4, SC_3(B_2) \geq 0.6\}$. $SC_3(B_2) \geq 0.6$ and $SC_0() \geq 0.8$ are redundant in the presence of $SC_2() \geq 0.4$, so deleted from $\sigma(B_3)$. This gives $\sigma(B_3) = \{SC_1(B_1) \geq 0.2, SC_2() \geq 0.4\}$, where $SC_2() \geq 0.4$ becomes the default SC in $subtree(B_3)$. By Corollary 4.1(2), $Pminsup(B_3) = Sminsup(B_3) = 0.2$. Similarly, for sibling B_2, $\sigma(B_2) = \{SC_3() \geq 0.6\}$, $Sminsup(B_2) = 0.6$, $Pminsup(B_2) = 0.2$; for sibling B_1, $\sigma(B_1) = \{SC_0() \geq 0.8\}$, $Sminsup(B_1) = 0.8$, $Pminsup(B_1) = 0.2$; for sibling B_0, $\sigma(B_0) = \{SC_0() \geq 0.8\}$, $Sminsup(B_0) = 0.8$, and $Pminsup(B_0) = 0.2$. □

7 Evaluation

We study the scalability with respect to the lowest minimum support specified. The scalability is measured by the *dead point*, defined as the lowest minimum support at which page swapping between memory and disk starts to takes place. We observed that whenever the available physical memory dropped to only a few Mbytes, the run did not finish within 3 hours and much longer time was needed. So, practically the dead point was taken as the lowest tested minimum support for which a run finishes within 3 hours. All experiments were performed on PII 300-MMX with 128MB memory and NT Server 4.0.

We chose Apriori and Max_Miner for comparison. Apriori provides a baseline for measuring the benefit of our approach. Max_Miner generates only maximal frequent itemsets and is a good candidate to overcome the bottleneck of itemest generation. Since neither Apriori nor Max_Miner handles general support constraints, the lowest minimum support in a support specification was used for them. There are several other high performance algorithms, by being smart in candidate generating and support counting, e.g., [BMUT97, PCY96, SON95]. Like Apriori, thoes techniques can be adopted in our itemset generation phase. So we do not compare with every such algorithm.

We borrowed the census data used in [SBMU98]. The data has 23 attributes, 77 items [2] and 126,229 transactions. Each transaction corresponds to an individual, and each item corresponds to an attribute/value pair. About an half of the items have support less than 10%, and the rest of the items have widely varied support from 10% to more than 90%.

[2] originally 63 items, but we explicitly represented the FALSE value of the 14 binary attributes as items, making 77 items in total.

To generate support specifications, we grouped the items from the same attribute into a bin, giving 23 bins $B_1, \ldots, B_{23}$. Let V_i be a bin variable and $S(V_i)$ be the smallest support of the items in the bin represented by V_i. We specified the following SCs in the closed interpretation:

$$SC_i(V_1, \ldots, V_k) \geq \theta_i(V_1, \ldots, V_k) \quad (0 < k \leq K) \quad (1)$$

where K is the maximal itemset size K specified by the user. $\theta_i(V_1, \ldots, V_k) = \gamma^{k-1} \times S(V_1) \times \ldots \times S(V_k)$ if $\gamma^{k-1} \times S(V_1) \times \ldots \times S(V_k)$ is within $[0.0000158, 1]$. If $\gamma^{k-1} \times S(V_1) \times \ldots \times S(V_k)$ is less than the lower bound or larger than the upper bound, the corresponding bound is used. The lower bound 0.0000158 corresponds to the support requirement of at least 2 transactions. Each specification is defined by a pair of γ and K. Since the occurrence of bins is symmetric, Strategy 2 does not impose a bias on the ordering of nodes and we report only "static 1" as the "dynamic 1" did not make a tangible difference. "average" refers to the average of 10 random orders for Adaptive Apriori. A more detailed study of various strategies is reported in [WHH00].

We varied γ and K to simulate different support requirements. In general, as γ decreases and K increases, the lowest minimum support in a specification decreases. The bottom of Figure 7 shows the lowest minimum support for each (γ, K) pair. In Figure 7, on the left are the measures for $\gamma = 5$, and on the right are the measures for $\gamma = 20$. In Figure 7(4a,4b), the y-value for Max_Miner is the number of maximal frequent itemsets. The dead point is represented by the right-most point on a curve. All algorithms were terminated after K iterations for a given K. In general, Apriori and Max_Miner reached the dead point earlier than "static 1" and "average". "static 1" and "average" performed better at $\gamma = 20$ than at $\gamma = 5$. This is because minimum supports are well spread at $\gamma = 20$, as shown in the table in Figure 7.

We plotted $Pminsup$ vs nodes numbered in the breath-first ordering for the dead point of "static 1" at $(\gamma = 20, K = 7)$ and $(\gamma = 5, K = 5)$, shown in Figure 7 and Figure 6, respectively. The two cases have the lowest minimum support, 0.0000158. For the case of $(\gamma = 20, K = 7)$, the minimum supports are well spread and Adaptive Apriori was able to exploit a higher $Pminsup$ for 99% of the nodes expanded! For the case of $(\gamma = 5, K = 5)$, the minimum supports tended to be crowded towards 0.0000158, and only 88% of the nodes expanded have $Pminsup$ higher than 0.0000158. This experiment strongly supports our claim that if itemsets are of varied supports, pushing support constraints is an effective strategy to deal with the bottleneck of itemset generation.

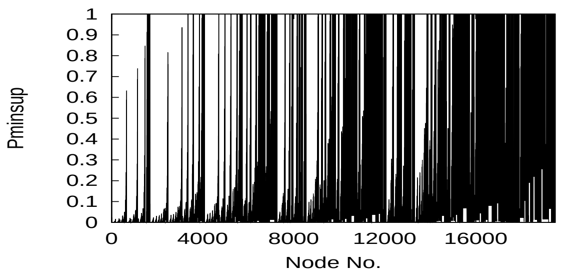


Figure 5: 99.3% nodes above 0.0000158

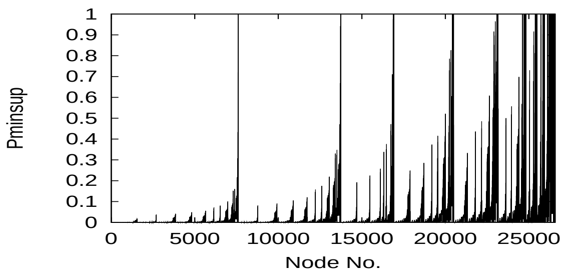


Figure 6: 88.1% nodes above 0.0000158

8 Conclusion

We motivated the need for support constraints and proposed a way of specifying support constraints. We presented a framework for pushing support constraints into the itemset generation. The challenge is that the classic Apriori is lost in the presence of non-uniform minimum support. Our approach is to use the best "run time" minimum support for each itemset so as to preserve the Apriori itemset generation. We call this strategy Adaptive Apriori. Unlike existing constraint pushing strategies, Adaptive Apriori does not rely on a uniform support requirement. A key issue for Adaptive Apriori is to order items so that the "run time" minimum support is maximized. We proposed several strategies for this. Our experiments showed that pushing support constraints is highly effective in dealing with the bottleneck of itemset generation. A meaningful future work is to study how the non-uniform support framework can be extended to frequent itemset mining without generating candidates like in [HPY00].

References

[AIS93] R. Agrawal, T. Imilienski, and A. Swami. Mining association rules between sets of items in large datasets. SIGMOD 1993, 207-216.

[AS94] R. Agrawal and R. Srikant. Fast algorithm for mining association rules. VLDB 1994, 487-499

[AY98] C. C. Aggarwal and P. S. Yu. A new framework for itemset generation. PODS 1998, 18-24

[BMS97] S. Brin, R. Motwani, and C. Silverstein. Beyond market baskets: generalizing association rules to correlations. SIGMOD 1997, 265-276

[BMUT97] S. Brin, R. Motwani, J.D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. SIGMOD 1997, 255-264

[C*00] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J.D. Ullman, C. Yang. Finding interesting associations without support pruning. ICDE 2000, 489-499

[DL99] G. Dong, J. Li. Efficient mining of emerging patterns: discovering trends and differences. SIGKDD 1999, 43-52

[HF95] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. VLDB 1995, 420-431

[HPY00] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. SIGMOD 2000, 1-12

[LHM98] B. Liu, W. Hsu, Y. Ma. Integrating classification and association rule mining. KDD 1998, 80-86

[LHM99] B. Liu, W. Hsu, Y. Ma. Mining association rules with multiple minimum supports. SIGKDD 1999, 125-134

[LSM98] W. Lee, S.J. Stolfo, K.W. Mok. Mining audit data to build intrusion detection models. KDD 1998, 66-72

[MW99] D. Meretakis, B. Wuthrich. Extending naive Bayes classifiers using long itemsets. SIGKDD 1999, 165-174

[MTV94] H. Mannila, H. Toivonen, A.I. Verkamo. Efficient algorithm for discovering association rules. KDD 1994, 181-192

[PCY96] J.S. Park, M. -S. Chen, P.S. Yu. An efficient hash based algorithm for mining association rules. SIGMOD 1995, 175-186

[R92] R. Rymon. Search through systematic set enumeration. Principles of Knowledge Representation and Reasoning, 1992, 539-550

[SA95] R. Srikant and R. Agrawal. Mining generalized association rules. VLDB 1995, 407-419

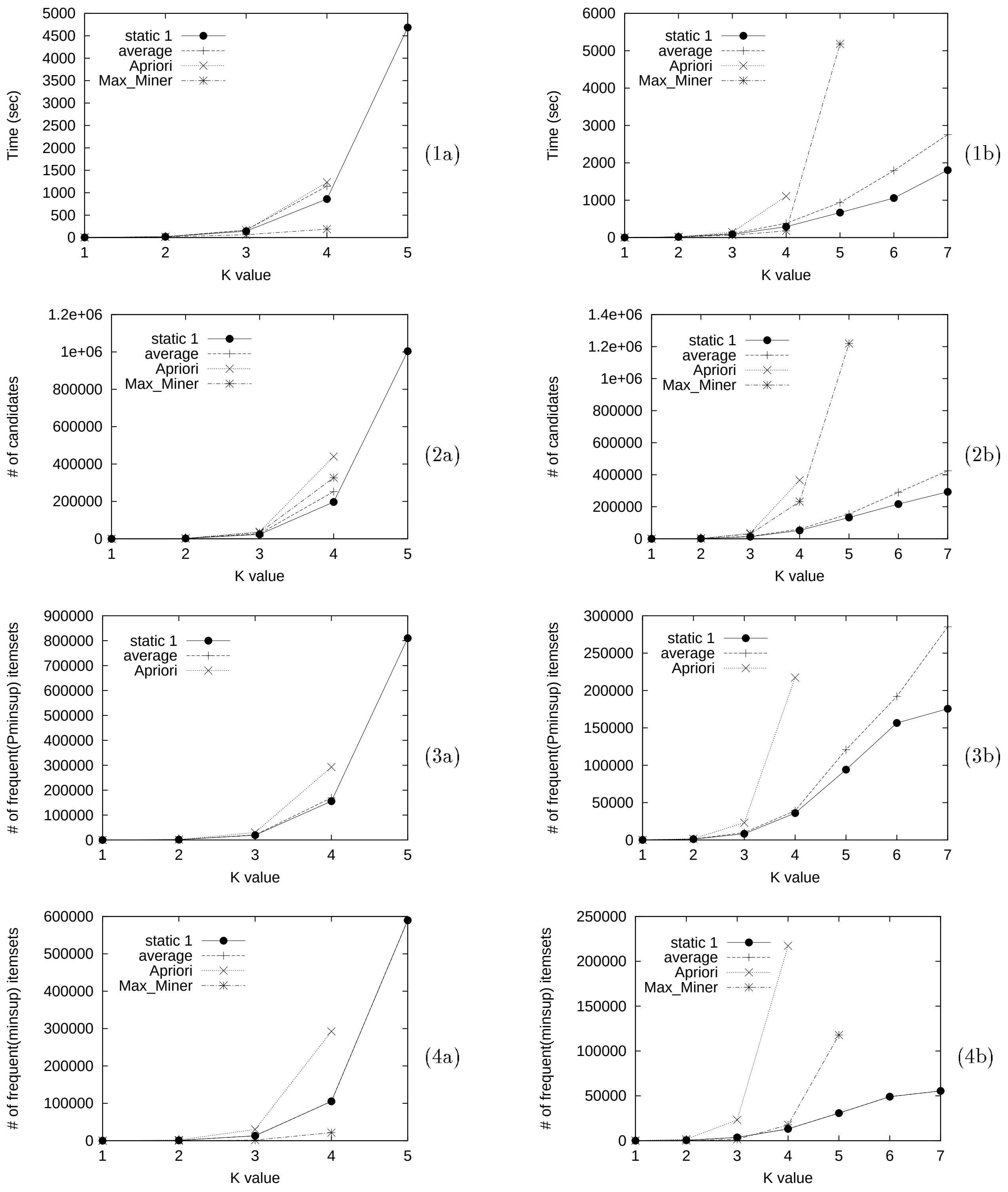
[SBMU98] C. Silverstein, S. Brin, R. Motwani, J. Ullman. Scalable techniques for mining causal structures. VLDB 1998, 594-605

[SON95] A. Savasere, E. Omiecinski, S. Navathe. An efficient algorithm for mining association rules in large databases. VLDB 1995, 432-444

[SVA97] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. KDD 1997, 67-73

[WHH00] K. Wang, Y. He, J. Han. Pushing support constraints into frequent itemset mining. School of Computing, National University of Singapore, 2000

[WZL99] K. Wang, S.Q. Zhou, S.C. Liew. Building hierarchical classifiers using class proximity. VLDB 1999, 363-374



The lowest minimum support

γ	$K=1$	$K=2$	$K=3$	$K=4$	$K=5$	$K=6$	$K=7$
5	0.0038	0.00016	0.0000158	0.0000158	0.0000158	0.0000158	0.0000158
20	0.0038	0.00064	0.00022	0.0000804	0.0000320	0.0000158	0.0000158

Figure 7: The dead points for the census dataset (the left for $\gamma = 5$ and the right for $\gamma = 20$)