Pushing Support Constraints Into Association Rules Mining *

Ke Wang[†]
Simon Fraser University
Yu He[‡]
National University of Singapore
Jiawei Han[§]
Simon Fraser University

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 caused by a low minimum support. A better solution lies in exploiting *support constraints*, which specify what minimum support is required for what itemsets, so that only the 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 determined for each itemset at run time to preserve the essence of Apriori. This strategy is called Adaptive Apriori. Experiments show that Adaptive Apriori is highly effective in dealing with the bottleneck of itemset generation.

Keywords. Association rules, constraints, data mining, frequent itemsets, knowledge discovery.

1 Introduction

The *association rules mining*, first studied in [2, 3] 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 their support), i.e., itemsets that have a support above the minimum support. Since frequent itemsets serve as an estimation of joint probabilities of events, the importance of mining frequent itemsets goes far beyond market-basket analysis. For

*Research was supported in part by research grants from the Natural Science and Engineering Research Council of Canada.

†Department of Computing Science, Simon Fraser University, 8888 University Drive, Burnaby, B.C., Canada, V5A 1S6. Email: wangk@cs.sfu.ca. Tel: (604)291-4667. Fax: (604)291-3045.

‡Hewlett-Packard Singapore (Private) Limited, 438-B Alexandra Road, #03-08 Alexandra Technopark, Singapore 119968. Email: yu_he@hp.com

§2123 Digital Computer Lab, Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 West Springfield Ave. Urbana, Illinois 61801. Email: hanj@cs.uiuc.edu. Tel: (217)333-6903. Fax: (217)244-6500.

example, recent studies have leveraged frequent itemsets to build intrusion detection models [11], to construct classifiers [12, 15], to build Yahoo!-like information hierarchies [25], to discover emerging patterns [8]. We believe that more and more internet/web related data mining will require the ability of finding frequent itemsets.

1.1 Apriori lives on a uniform minimum support

The key to mining frequent itemsets is to prune the number of candidate itemsets generated. The best known strategy, called Apriori [2, 3], exploits the following property: if an itemset is frequent, so are all its subsets. Thus, Apriori generates itemsets in a level-wise manner where each candidate k-itemset $\{i_1, \ldots, i_{k-2}, i_{k-1}, i_k\}$ in the kth iteration is generated from two frequent $(k-1)$-itemsets $\{i_1, \ldots, i_{k-2}, i_{k-1}\}$ and $\{i_1, \ldots, i_{k-2}, i_k\}$. A generated candidate can be further pruned if any subset of size $k-1$ is not frequent. Apriori lives on the essential assumption 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 legitimate 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! In this case, Apriori fail to find the frequent itemset $\{coffee, sugar, tea\}$!

1.2 The reality is not uniform

In reality, however, there are many good reasons that the minimum support is not uniform. First, deviation and exception 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 often 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 often varies at different concept levels of items [9, 21]. Fifth, hierarchical classification like [25] requires feature terms to be discovered at different concept levels, thereby, requiring a non-uniform minimum support. Finally, in recommender systems [23], recommendation rules are required to cater for both big and small groups of customers. In general, rules of high support are well known to the user, and it is the rules of low support that may provide interesting insights and need to be discovered.

With existing algorithms that assume a uniform minimum support, the best that one can do is to apply such algorithms at the lowest minimum support specified and filter the result using the other minimum supports. This approach will generate many candidates that are later discarded. From our experience (see Section 7), the increase in the number 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 that reads both candidates transactions from disk. In one case, as we reduced the minimum support from 0.065% to 0.060%, and to 0.047%, the execution time of Apriori increased from 940 to 9,858, and to 75,652 seconds! This clearly indicates that Apriori does not scale up well with respect to the decrease of minimum support. 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 on all itemsets.

1.3 Our approach

We propose the notion of *support constraints* as a way to specify general constraints on minimum support. Informally, a support constraint states what itemsets are required to satisfy what minimum support. We shall 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], or a function that produces such a minimum support. The above support constraint specifies that any itemset containing at least one item from each B_j must have the minimum support θ_i. The topic of this paper is to "push" such support constraints into the itemset generation to prune candidates generated. We illustrate this approach using an example.

Example 1.1 Consider four support constraints

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

Each bin B_i contains a disjoint set of items. We assume that, if more than one support constraint is applicable to an itemset, the one specifying the lowest minimum support is adopted. This is because adding more items to an itemset should not increase the minimum support of the itemset. With this in mind, we have

- Case (i): $SC_1(B_1, B_3) \geq 0.2$ specifies minimum support 0.2 for any itemset containing (at least) 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. To see this, let b_i denotes any item from B_i. 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). 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, i.e., 0.6, is lower than the specified 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. Our goal is to prune low minimum supports as much as possible while still generating all itemsets above their specified minimum supports.

Order sensitivity. The above example has implicitly assumed that b_3 does not follow b_2 in the item ordering used by the Apriori itemset generation. Suppose instead that b_3 follows b_2 in the ordering. $\{b_0, b_1, b_2, b_3\}$ would then 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 key idea of tightening up the search space is to order items so that the highest possible minimum support is pushed in each case. □

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 bin sets, and determines the highest minimum support to be pushed to each itemset. This phase makes use of the information of given 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 but 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 the case of non-uniform minimum support while preserving the Apriori itemset generation. The mining algorithm is presented in Section 6. 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 [2, 3], and a similar idea in [14]. Nearly all later frequent itemset minings rely on Apriori as a basic pruning strategy. Constraints other than the minimum support are considered in [16, 22]. However, none of these approaches considers pushing support constraints like ours. The correlation approach [1, 5] 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 [7], our approach is to make the requirement more realistic by allowing it different for different itemsets. [10] abandons the Apriori itemset generation, but still critically relies on a uniform support requirement.

[13] deals with a non-uniform minimum support. In [13], 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 as a single concept, e.g., $\{white, male\}$, but not for individual items in the itemset (e.g., *white* or *male*). This "minimum itemset support" is usually lower than the minimum item support. (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 overcome these difficulties by specifying the minimum support directly for itemsets. We will show that our specification can model the MIS specification, but the converse is not true.

Our conference paper [24] reports the preliminary work of the approach considered here. In this paper, we extend that report by presenting the mining algorithm and detailed experimental studies.

3 Specifying support constraints

As in [2, 3], 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, given a bag or multiset $\beta = \{B_1, \ldots, B_k\}$ of bins, all k-itemsets $\{i_1, \ldots, i_k\}$, where $i_j \in B_j$, 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 will 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 of an itemset 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 one item from each bin in SC_i and these items are distinct, and I contains no other items. 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*. From now on, we assume that all redundant SCs are removed. With this assumption, a SC of the form $SC_i() \geq \theta_i$, if specified, must have the highest minimum support and 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 $SC_2(B_3) \geq 0.4$. $minsup(I) = 0.2$ because the lowest minimum support of matched SCs is used. Some examples of such I are $\{0, 2\}$, $\{0, 2, 3\}$, and $\{2, 3, 4\}$. $\{2, 4, 7\}$, $\{2, 4, 8\}$, $\{4, 7, 8\}$, and $\{2, 4, 7, 8\}$ all have minimum support 0.6, because they match only $SC_3(B_2) \geq 0.6$, and 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 As an example of non-ground SCs and the closed interpretation, consider $SC_i(V_1, \ldots, V_k) \geq sup(V_1) \times \ldots \times sup(V_k)$, $1 \leq k \leq 4$, where V_i are variables. Each B_i contains the items of the same support, denoted $sup(B_i)$. This SC specifies the minimum support relative to the independence assumption on item occurrence. With the closed interpretation, any itemset containing more than 4 items has an undefined minimum support. □

We like to comment that, in $SC_i \geq \theta_i$, θ_i is required to be "evaluable" at the *specification time*. A constant θ_i satisfies this requirement, so does any θ_i defined by values associated with bins B_i

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

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

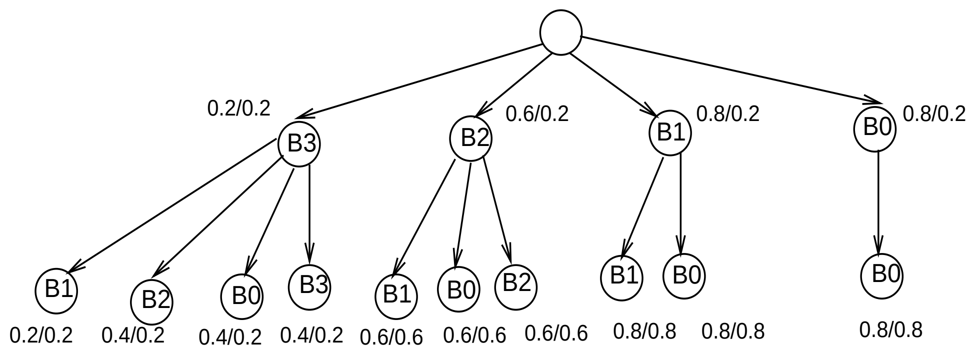


Figure 2: A schema enumeration tree, marked with $Sminsup/Pminsup$

that are known at the specification time, such as the maximum, minimum, or average support of the items in B_i. However, this requirement is not satisfied if we specify the minimum confidence of rules $\alpha \rightarrow \beta$ by $SC(\alpha, \beta) \geq minconf \times sup(\alpha)$, where α and β are schemas and each bin contains a single item (thus, schemas and itemsets coincide). This is 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 in the context of association rules mining. The classic uniform minimum support [2, 3] can be specified by one default $SC_i() \geq \theta_i$ with θ_i being the usual minimum support. The item constraints [22] can be specified by non-default SCs in which all minimum supports are equal. To model the MIS specification in [12], 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 a specification such as $SC_1(B_1, B_2, B_3) \geq 0.2$, $SC_2(B_1, B_3) \geq 0.3$, and $SC_2(B_2, B_3) \geq 0.4$.

We can construct association rules from frequent itemsets. There are three approaches to the construction, depending on which parts of rules the SCs are specified. Let $minconf$ denote the user-specified minimum confidence for association rules.

Definition 3.3 (Association rules) *For each pair of frequent itemsets I and I' such that $I \subset I'$,*

- *if $sup(I')/sup(I) \geq minconf$,* Type I association rule *$I \rightarrow I' - I$ is constructed.*
- *if $sup(I')/sup(I' - I) \geq minconf$,* Type II association rule *$I' - I \rightarrow I$ is constructed.*

- *if $sup(I')/sup(I'-I) \geq minconf$ and $I'-I$ is frequent,* Type III association rule $I'-I \rightarrow I$ *is constructed.* □

For all types of rules $X \rightarrow Y$, SCs are enforced over XY because XY (i.e., I') is always frequent [2]. In addition, for Type I, II, III, respectively, SCs are further enforced over the antecedent X, the consequent Y, and both the antecedent and the consequent. For Type I and Type III rules $X \rightarrow Y$, the confidence $sup(XY)/sup(X)$ can be computed directly using frequent itemsets because both XY and X are frequent. For Type II rules $X \rightarrow Y$, the antecedent X (i.e., $I'-I$) is not necessarily frequent and an additional database scan is needed to find $sup(X)$. If only the default SC is specified, all types degenerate to the classic association rules.

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. Typically, the minimum support for an itemset is a function of the support of some or all items contained in the itemset. Example 3.2 and the MIS specification are based on this idea. These examples illustrate three useful points. First, a bin B_j usually contains similarly supported items. Such bins can be found by computing the support of items in one pass of the transactions and then clustering the items based on their supports. Second, θ_i is usually a function of some representative supports of bins (such as the maximum, minimum, or average support in the bin), and the function of θ_i can be either chosen from a menu of built-in functions or supplied by the user. Third, if the user does not have particular schemas in mind for specification, a generic specification in the form of a non-ground SC can be used.

Concept-based specification. In the presence of an item concept hierarchy, it is desirable to specify SCs based on the generality of the item concepts. 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

[2] XY is the shorthand of the union of X and Y.

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 chosen.

4 Adaptive Apriori

A key idea of our approach is to push SCs following the "dependency chain" of itemsets in the itemset generation in Apriori. 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 $B_1 \dots B_k$ along the path from the root to v. If a schema enumeration tree contains two sibling nodes representing schemas $s_1 = B_1 \dots B_{k-2}B_{k-1}$ and $s_2 = B_1 \dots B_{k-2}B_k$, 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 $s = B_1 \dots B_{k-2}B_{k-1}B_k$, 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. 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 [18], 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 the schemas represented by them. Thus, 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 those itemsets containing more than one item from B_i.

4.1 The pushed minimum support

Consider schema $s = B_1 \dots B_{k-2}B_{k-1}B_k$, and its generating schemas $s_1 = B_1 \dots B_{k-2}B_{k-1}$ and $s_2 = B_1 \dots B_{k-2}B_k$. In the case of a uniform minimum support, if an itemset $I = \{i_1, \dots, i_{k-2}, i_{k-1}, i_k\}$ of s is frequent, so are $I_1 = \{i_1, \dots, i_{k-2}, i_{k-1}\}$ of s_1 and $I_2 = \{i_1, \dots, i_{k-2}, i_k\}$ of s_2. This property enables Apriori to generate candidate k-itemsets I using *frequent* $(k-1)$-itemsets I_1 and I_2. However, this generation is not available for non-uniform minimum support because $minsup(s)$, $minsup(s_1)$, $minsup(s_2)$ are not always the same. Our approach is to replace $minsup$ with a new function, $Pminsup$, called the "pushed minimum support", such that $Pminsup$ defines a superset of the frequent itemsets and this superset can be computed in the manner of Apriori. Let us formalize this idea.

Consider any function f from schemas s 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 an itemset $\{i_1, \ldots, i_{k-2}, i_{k-1}, i_k\}$ 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 Completeness and Apriori-like. □

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

Intuitively, Completeness ensures that $F(Pminsup)$ is a superset of $F(minsup)$, Apriori-like preserves the Apriori itemset generation of candidates, and Maximality ensures that $F(Pminsup)$ is tightest to satisfy Completeness and Maximality. Therefore, by replacing *minsup* with *Pminsup*, we are able to generate a tight superset of $F(minsup)$ in the same manner as Apriori. This strategy is referred as to Adaptive Apriori. A benefit of preserving the Apriori itemset generation is that the improvements of Apriori studied over the last several years (e.g., [6, 17, 19]) are immediately applicable to Adaptive Apriori. The novelty of Adaptive Apriori, however, is that it breaks the barrier of uniform minimum support by defining the *best* minimum support, i.e., *Pminsup*, for each schema *individually* with respect to the preservation of Apriori. In fact, Apriori is the special case of Adaptive Apriori where *Pminsup* is equal to the given uniform minimum support and the schema enumeration tree has a single path of the form $root, B, \ldots, B$, where the only bin B contains all the items.

Unlike Apriori, Adaptive Apriori does not assure that every subset of a $frequent(Pminsup)$ itemset be $frequent(Pminsup)$. This is both good news and bad news. The good news is that the number of $frequent(Pminsup)$ itemsets may not be necessarily exponential. Indeed, a $frequent(f)$ itemset $\{i_1, \ldots, i_{k-2}, i_{k-1}, i_k\}$ only assures that the subsets of the form $\{i_1, \ldots, i_{j-1}, i_j, i_p\}$ be $frequent(f)$, where $j < p \leq k$. There are only $k(k-1)/2$ such subsets [3]. Note that this does not mean that the other subsets are not $frequent(f)$. Characterizing those f's that do not define exponentially many $frequent(f)$ itemsets is an interesting problem in itself. The bad news is that pruning a candidate I of size k by checking a subset I' of size $k-1$, as in Apriori, is now possible only if $Pminsup(s) \geq Pminsup(s')$, where s and s' are the schemas of I and I'. This is a natural generalization of the subset based pruning in the case of non-uniform minimum support.

At this point, two questions need to be answered. First, how do we determine *Pminsup* with respect to a given schema enumeration tree T? Second, how do we generate a schema enumeration tree for which *Pminsup* is maximized? We will 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, $subtree(s)$ is the subtree rooted at s

[3] For $p = 1$, there is 0 subset; for $p = 2$, there is 1 subset; for $p = 3$, there are 2 subsets; ....; for $p = k$, there are $k-1$ subsets.

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 matched by some schema 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

(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 matches $SC_1(B_1, B_3) \geq 0.2$ 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 two 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 $L(r)$, and $L(r)$ occurs in $subtree(l)$, but not 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 the labels of nodes in $LS(v)$ are pruned from $subtree(v)$.

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

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 $Sminsup(B_2) = 0.6$. $Sminsup(B_3) = 0.2$. $Pminsup(B_2) = min\{Sminsup(B_3),\ Sminsup(B_2)\} = 0.2$. $Pminsup(s) = 0.6$ 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 $subtree(s)$. By using $Pminsup$ as the run time minimum support, we are able to tie up the support requirement and, at the same time, still enjoy the Apriori generation of frequent itemsets. □

4.3 The characteristic of *Pminsup*

We now analyze how *Pminsup* changes in a schema enumeration tree. This information can help us to find a schema enumeration tree that maximizes *Pminsup*. 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 label $L(l)$ 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.

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) $Pminsup(s_i) \leq Sminsup(s_i) \leq Pminsup(s_{i+1})$.

Proof: 1a and 2a follow immediately from the discussion preceding the theorem. 1b follows from 1a and the definition of *Pminsup*. Now we show 2b. Let s'_{i+1} be the left-most sibling of s_{i+1}. We have $Sminsup(s_i) \leq Sminsup(s'_{i+1})$ from 2a, and $Sminsup(s'_{i+1}) = Pminsup(s_{i+1})$ from 1b. From Corollary 4.1, $Pminsup(s_i) \leq Sminsup(s_i)$. The transitivity of these equalities and inequalities imply $Pminsup(s_i) \leq Sminsup(s_i) \leq Pminsup(s_{i+1})$, i.e., 2b. □

From Theorem 4.1(2b), *Pminsup* is never decreased by moving from a parent p to a child c. The next theorem tells 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. $Pminsup(p) < Pminsup(c)$.

An intuitive proof of Theorem 4.2 is: the schemas in $subtree(p')$ depend on p, but not on c, therefore, $Pminsup(p)$ is constrained by the lowest minimum support in $subtree(p')$, but $Pminsup(c)$ is not. Then 1 and 2 are two equivalent conditions for this difference to have effect on $Pminsup(p)$ and $Pminsup(c)$. For example, in Figure 3 (which contains only the nodes for non-empty sets of candidates), $Sminsup(B_3) < Sminsup(B_i)$, for $i = 2, 1, 0$, and every child of schema B_i has a higher *Pminsup* than B_i does. This is because $Sminsup(B_3)$, i.e., 0.2, is pruned in $subtree(B_i)$, for $i = 2, 1, 0$.

Proof of Theorem 4.2: The equivalence of 1 and 2 follows from Theorem 4.1(1a) and the definition of *Sminsup*. We show that 1 implies 3. Assume that 1 holds, that is, that p has a left sibling p' such that $Sminsup(p') < Sminsup(p)$. By definition, $Pminsup(p) \leq Sminsup(p')$. From Theorem 4.1(2b), $Sminsup(p) \leq Pminsup(c)$. Then 3 follows from the assumption $Sminsup(p') <$

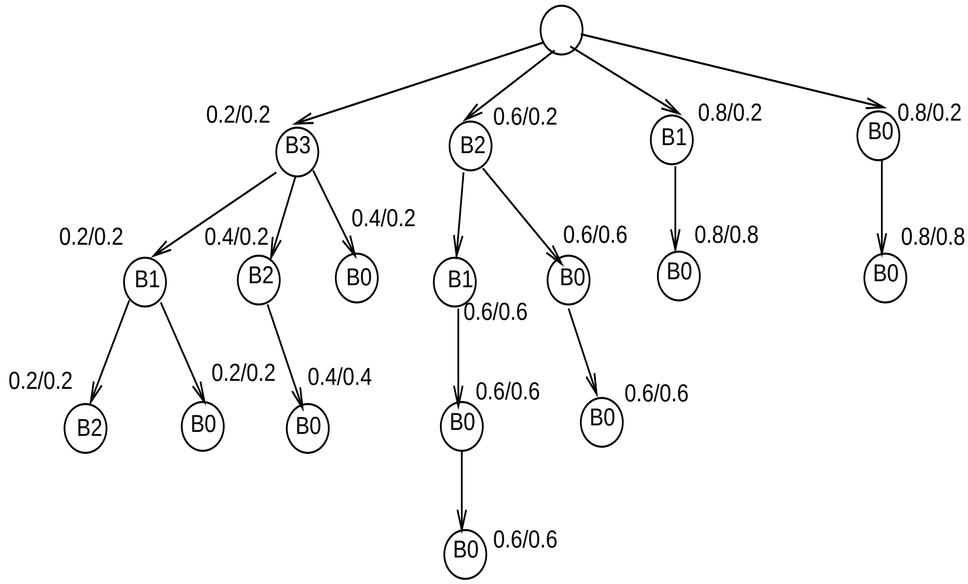


Figure 3: A schema enumeration tree for non-empty nodes, marked with $Sminsup/Pminsup$

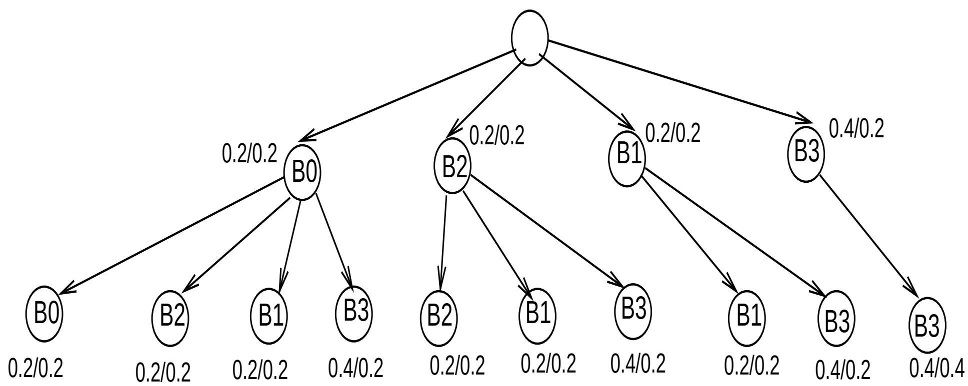


Figure 4: A different schema enumeration tree, marked with $Sminsup/Pminsup$

$Sminsup(p)$. We now show that 3 implies 1. Let c' and p' be the left-most siblings of c and p. Suppose that 1 fails. Then $Sminsup(p') = Sminsup(p)$. From Theorem 4.1(1b), $Pminsup(p) = Sminsup(p')$ and $Pminsup(c) = Sminsup(c')$, by definition, $Sminsup(c') = Sminsup(p)$. These equalities together imply $Pminsup(c) = Pminsup(p)$, i.e., the failure of 3. This proves the equivalence of 1 and 3. □

The 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). Section 5 will study this issue.

5 The ordering of nodes

Now we answer the second question: how to construct a schema enumeration tree to maximize $Pminsup$. Compare Figure 2 with Figure 4. The schema enumeration tree in Figure 2 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 4. This change is caused by placing labels B_1 and B_3 at the right end at level 1 in Figure 4, which makes $SC_1(B_1, B_3) \geq 0.2$ applicable in $subtree(B_2B_1)$ and $subtree(B_0B_2)$. Clearly, this example shows that the order of sibling nodes has an impact on $Pminsup$. In general, however, no "optimal" order exists, as the next theorem shows. Therefore, a reasonable thing to do is to order sibling nodes heuristically to maximize $Pminsup$. In the rest of this section, we consider several such heuristics.

Theorem 5.1 (No optimal ordering) There exists a support specification such that for any schema enumeration tree T_1, there exists another schema enumeration tree T_2 and two schemas s_1 and s_2 such that $Pminsup_1(s_1) > Pminsup_2(s_1)$ and $Pminsup_2(s_2) > Pminsup_1(s_2)$, where $Pminsup_i$ denotes the pushed minimum support with respect to T_i. □

Proof: Consider the support specification: $SC_1() \geq \theta_1$ and $SC_2(B_1, B_2, B_3) \geq \theta_2$, where $\theta_1 > \theta_2$. For any schema enumeration tree T_1 with nodes B_1, B_2, B_3 from left to right at level 1, there is schema enumeration tree T_2 with nodes B_2, B_1, B_3 from left to right at level 1. We can show that $Pminsup_1(B_2B_3) = \theta_1$ and $Pminsup_2(B_2B_3) = \theta_2$. An intuitive proof is that, in T_2 schema $B_2B_1B_3$ depends on (thus, θ_2 is pushed down to) B_2B_3, but in T_1 schema $B_1B_2B_3$ does not depend on B_2B_3. Similarly, we can show that $Pminsup_2(B_1B_3) = \theta_1$ and $Pminsup_1(B_1B_3) = \theta_2$. □

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 consideration 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). The following ordering strategy is based on this idea.

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 2 by Strategy 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 then 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 $Sminsup(B_0) = 0.8$. From Theorem 4.2, $Pminsup$ is increased by moving from nodes B_3, B_2, B_0

to their child nodes. 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, ignoring the interaction between siblings. Our second strategy is to greedily prune as many SCs as possible, in the hope that the default SC, which always specifies the highest minimum support, can be used as early as possible. Thus, 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. Like Strategy 1, this strategy can be either dynamic or static. Unlike Strategy 1, the information about the minimum support in SCs is not used here.

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

We can go one step further to maximize the number of s_i such that the default SC is used in $subtree(s_i)$. A necessary and sufficient condition for such s_i is that a "cover" of $\sigma(p)$ is on the left of s_i, where p is the parent node of s_i. A *cover* C of a set of SCs is a set of labels such that each non-default SC contains at least one label in C. A *minimum cover* is a cover of the minimum size. If a cover of $\sigma(p)$ is on the left of s_i, all non-default SCs in $\sigma(p)$ are pruned in $subtree(s_i)$ because at least one label is missing for each SC. In this case, $Sminsup(s_i)$ is either the default SC or undefined. On the other hand, if no cover of $\sigma(p)$ is on the left of s_i, some non-default SC remains applicable in $subtree(s_i)$. To determine the relative order within a selected minimum cover, either Strategy 1 or Strategy 2 can be applied. This strategy is computationally feasible only for a specification of a small size because finding a minimum cover is NP-complete. For a specification of a large size, we can use a "small" cover to substitute for a minimum cover. Strategy 2 can be considered as such a substitution, as it greedily selects the label that covers the most number of SCs.

Strategy 3 Select a minimum cover of $\sigma(p)$ as the first few siblings, for the parent p. □

Example 5.2 In Example 5.1, if Strategy 3 is applied, the minimum cover $C = \{B_2, B_3\}$ of $\sigma(root)$ is selected as the first two siblings at level 1. To determine the relative order of B_3 and B_2, we apply Strategy 1 or Strategy 2, both selecting B_3 first. Thus, the order is the same as O_2 in Example 5.1. □

We conclude this section by making a few remarks. First, it is possible to have a hybrid strategy that combines more than one of the above rationales. For example, Strategy 1 can be used to break the tie arising from Strategy 2, or vice versa. More generally, one can define some scoring function to take such combinations into account. Such a scoring function can be easily incorporated without affecting the rest of our algorithm. Second, the rationale of our node ordering is different from that of the item ordering in [6, 4]. The purpose of the item ordering is to reduce the cost of traversing the enumeration tree of itemsets during the support counting [6] or to maximize the chance of hitting a long pattern [4]. Our purpose is to maximize $Pminsup$ for each node or schema.

```
for each node p at level k − 1 do

/* Step 1: Generate child nodes */
    if p is the root then
        for each bin B_i do create one child s_i of p with L(s_i) = B_i;
    else
        for each p' in RS(p) do create one child s_i of p with L(s_i) = L(p');

/* Step 2: Order child nodes */
    order the child nodes s_i by one of the strategies in Section 5;

/* Step 3: Compute σ(s_i) and Pminsup(s_i) for child nodes */
    for each child s_i from left to right do
        σ(s_i) := {SC_i ≥ θ_i ∈ σ(p) | SC_i contains only the labels for nodes in RS(s_i)};
        delete one occurrence of L(s_i) from the SCs in σ(s_i);
        delete all redundant SCs from σ(s_i);
        if σ(s_i) is empty then
            delete node s_i from the schema enumeration tree
        else
            Pminsup(s_i) := min{Sminsup(s_j) | s_j ∈ LS(s_i) ∪ {s_i}};
```

Figure 5: Phase 1

6 The algorithm

The algorithm expands the schema enumeration tree iteratively, one level per iteration. Each iteration k has two phases. 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. In the following discussion, we assume that each node p at level $k-1$ is associated with the set of SCs at p, $\sigma(p)$, and relation T_p for $frequent(Pminsup)$ itemsets of p. Please refer to Table 1 for notation. We explain the expansion from level $k-1$ to level k.

6.1 Phase 1

Figure 5 gives the code for generating nodes s_i and determining $Pminsup(s_i)$ and $\sigma(s_i)$. To expand to level k, three steps are performed. 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 $Pminsup(s_i)$. We explain Step 3 using 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 $subtree(B_3)$ contains 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 are 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, $Pminsup(B_3) =$

$Sminsup(B_3) = 0.2$. 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 $Pminsup(B_0) = 0.2$. □

In Step 3, deleting one occurrence of $L(s_i)$ from the SCs in $\sigma(s_i)$ is necessary for the correctness at the next level. To see this, suppose that we have not deleted label B_2 from $\sigma(B_2)$ in the above example. $\sigma(B_2)$ would contain $SC_3(B_2) \geq 0.6$, rather than $SC_3() \geq 0.6$. At node B_2B_0, since B_2 is not a label of any node in $RS(B_2B_0)$ (see Figure 2), $SC_3(B_2) \geq 0.6$ would not be included in $\sigma(B_2B_0)$ in Step 3. As a result, the default minimum support 0.8 would be used for $Pminsup(B_2B_0)$. This is wrong because B_2B_0 matches $SC_3(B_2) \geq 0.6$.

6.2 Phase 2

In this phase, we compute $frequent(Pminsup)$ itemsets for all schemas s_i at level k. The detail is given in Figure 6. This part is similar to the Apriori itemset generation. If $k = 1$, we find all $frequent(Pminsup)$ 1-itemsets in one scan of the transactions. Assume $k > 1$. Consider any schema $s_i = B_1 \ldots B_{k-2}B_{k-1}B_k$ at level k. Let $p_1 = B_1 \ldots B_{k-2}B_{k-1}$ and $p_2 = B_1 \ldots B_{k-2}B_k$ be the generating schemas of s_i. To generate the candidates of s_i, denoted by T_{s_i}, we "join" T_{p_1} and T_{p_2} as in Apriori [3] and scan the database for computing the support of candidates. To compute the support of candidates, the hash-tree implementation for subset function [3] can be used. For Adaptive Apriori, however, two new pruning strategies, not shown in Figure 6, are available. First, before joining T_{p_1} and T_{p_2}, if $Pminsup(s_i) > Pminsup(p_1)$ or $Pminsup(s_i) > Pminsup(p_2)$, we can skip over those tuples having support less than $Pminsup(s_i)$. This pruning is not available in Apriori where the minimum support of all itemsets is the same. The second new pruning strategy is that if, for some i, $Pminsup(B_1 \ldots B_{i-1}B_{i+1} \ldots B_k) \leq Pminsup(B_1 \ldots B_k)$, we can prune all candidates of schema $B_1 \ldots B_k$ whose projection on $B_1 \ldots B_{i-1}B_{i+1} \ldots B_k$ was not generated. This is a generalized form of Apriori's subset pruning in the case of non-uniform minimum support.

Example 6.2 Continue with Example 6.1 and the schema enumeration tree in Figure 3, which is produced by Strategy 3. Table 2 shows the work in Phase 1 and Phase 2. In the last column, $frequent(Pminsup)$ itemsets are marked by $\surd$, and $frequent(minsup)$ itemsets are marked by $\triangle$. The column $Pminsup$ shows that, out of the 17 nodes expanded in the enumeration tree, 8 nodes have used $Pminsup$ higher than the lowest minimum support 0.2. The number of candidates generated is 29, the number of $frequent(Pminsup)$ itemsets is 22, and the number of $frequent(minsup)$ itemsets is 17. In comparison, if we apply Apriori at $minsup = 0.2$, the number of candidates generated and the number of $frequent(minsup)$ itemsets are 85 and 73. If we adopt the "adversary" strategy, that is, first apply Strategy 3 to determine the order of siblings and then use the reversed order, the number of candidates generated, the number of $frequent(Pminsup)$ itemsets, and the number of $frequent(minsup)$ itemsets are 89, 65, and 17, respectively. □

```
/* Step 1: Generate candidates */
for each node s_i at level k do
        generate the candidate set T_{s_i} by joining T_{p_1} and T_{p_2} for generating schemas p_1 and p_2;

/* Step 2: Find frequent(Pminsup) itemsets */
compute sup(I) of all candidates I generated in Step 1 in one pass of transactions;
prune all candidates I with sup(I) < Pminsup(s_i);

/* Step 3: Delete empty nodes */
for each node s_i at level k do
        if T_{s_i} is empty then
            delete node s_i from the schema enumeration tree;
            for each s_j ∈ LS(s_i) do delete the SCs containing L(s_i) from σ(s_j);
```

Figure 6: Phase 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. In our experiments, we observed that whenever the available physical memory dropped to 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.

A major advantage of preserving the Apriori itemset generation is that nearly all improvements of Apriori over the last several years, by being smart in candidate generating and support counting, e.g., [6, 17, 19], are immediately applicable to Adaptive Apriori. Therefore, it is not necessary to compare Adaptive Apriori with every such improvement. We chose only two algorithms for comparison: Apriori [3] and Max_Miner [4]. Apriori provides a baseline for measuring the benefit of our approach. Max_Miner generates only maximal frequent itemsets, so a good candidate to overcome the bottleneck of itemest generation. Also, the ability of Max_Miner to mine long itemsets makes Max_Miner attractive in dealing with low minimum support. Since neither Apriori nor Max_Miner handles general support constraints, the lowest minimum support in a support specification was used for these algorithms.

7.1 The synthetic dataset

Our first experiment is to study the effectiveness of Adaptive Apriori over a range of support specifications. We used the synthetic dataset from [3] with the following settings: 100K transactions of

Phase 1					Phase 2
k	Node s	$\sigma(s)$	$Pminsup(s)$	$minsup(s)$	Candidates I at s ($sup(I)$)
	root	$SC_1(B_1B_3) \geq 0.2$ $SC_2(B_3) \geq 0.4$ $SC_3(B_2) \geq 0.6$ $SC_0() \geq 0.8$			
1	B_3	$SC_1(B_1) \geq 0.2$ $SC_2() \geq 0.4$	0.2	0.4	$\{0\}$ (0.4) $\surd\triangle$ $\{3\}$ (0.0)
	B_2	$SC_3() \geq 0.6$	0.2	0.6	$\{4\}$ (0.8) $\surd\triangle$ $\{5\}$ (0.2) $\surd$
	B_1	$SC_0() \geq 0.8$	0.2	0.8	$\{2\}$ (0.8) $\surd\triangle$ $\{6\}$ (0.2) $\surd$
	B_0	$SC_0() \geq 0.8$	0.2	0.8	$\{1\}$ (0.2) $\surd$ $\{7\}$ (1.0) $\surd\triangle$ $\{8\}$ (0.8) $\surd\triangle$
2	B_3B_1	$SC_1() \geq 0.2$	0.2	0.2	$\{0,2\}$ (0.2) $\surd\triangle$ $\{0,6\}$ (0.0)
	B_3B_2	$SC_2() \geq 0.4$	0.2	0.4	$\{0,4\}$ (0.2) $\surd$ $\{0,5\}$ (0.0)
	B_3B_0	$SC_2() \geq 0.4$	0.2	0.4	$\{0,1\}$ (0.0) $\{0,7\}$ (0.4) $\surd\triangle$ $\{0,8\}$ (0.2) $\surd$
	B_2B_1	$SC_3() \geq 0.6$	0.6	0.6	$\{4,2\}$ (0.6) $\surd\triangle$
	B_2B_0	$SC_3() \geq 0.6$	0.6	0.6	$\{4,7\}$ (0.8) $\surd\triangle$ $\{4,8\}$ (0.8) $\surd\triangle$
	B_1B_0	$SC_0() \geq 0.8$	0.8	0.8	$\{2,7\}$ (0.8) $\surd\triangle$ $\{2,8\}$ (0.6)
	B_0B_0	$SC_0() \geq 0.8$	0.8	0.8	$\{7,8\}$ (0.8) $\surd\triangle$
3	$B_3B_1B_2$	$SC_1() \geq 0.2$	0.2	0.2	$\{0,2,4\}$ (0.0)
	$B_3B_1B_0$	$SC_1() \geq 0.2$	0.2	0.2	$\{0,2,7\}$ (0.2) $\surd\triangle$ $\{0,2,8\}$ (0.0)
	$B_3B_2B_0$	$SC_2() \geq 0.4$	0.4	0.4	
	$B_2B_1B_0$	$SC_3() \geq 0.6$	0.6	0.6	$\{4,2,7\}$ (0.6) $\surd\triangle$ $\{4,2,8\}$ (0.6) $\surd\triangle$
	$B_2B_0B_0$	$SC_3() \geq 0.6$	0.6	0.6	$\{4,7,8\}$ (0.8) $\surd\triangle$
4	$B_2B_1B_0B_0$	$SC_3() \geq 0.6$	0.6	0.6	$\{4,2,7,8\}$ (0.6) $\surd\triangle$

Table 2: Computation of $frequent(Pminsup)$ itemsets

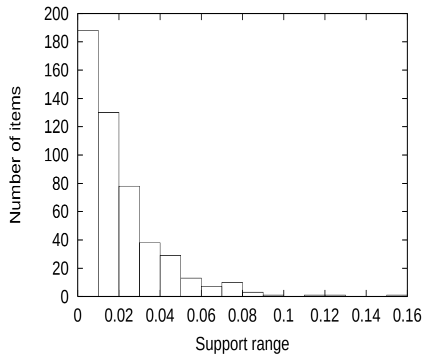
average length 10, 500 items, and the default settings for all other parameters. As shown in Figure 7(a), a characteristic of this dataset is that most items have low support, less than 0.04 (or 4%). The same characteristic remains even if other settings are used. This presents an adversary case to our approach that relies on exploiting a large variance of support in the data.

To generate a range of support specifications, we partitioned the support range into 4 intervals such that B_i contains the items with support in the ith interval and the number of items in B_i is approximately equal. We defined the minimum support of $SC_i(B_{i_1}, \ldots, B_{i_k}) \geq \theta_i$, $k > 0$, as follows:

$$\theta_i = min\{\gamma^{k-1} \times S(B_{i_1}) \times \ldots \times S(B_{i_k}), 1\} \quad (1)$$

where $S(B_j)$ denotes the lowest item support for B_j (see Figure 7(b)), and γ is an integer larger than 1. The term γ^{k-1} was used to slow down the decrease of $S(B_{i_1}) \times \ldots \times S(B_{i_k})$ for large k, and to simulate different support requirements. Figure 7(c) shows the 7 SCs corresponding to the

(a)



(b)

Bins	Support range
B_1	[0.00001, 0.00535]
B_2	[0.00536, 0.01397]
B_3	[0.01398, 0.02759]
B_4	[0.02765, 0.15113]

(c)

SCs
$SC_1(B_2, B_3, B_4) \geq \theta_1$
$SC_2(B_2, B_3) \geq \theta_2$
$SC_3(B_2, B_4) \geq \theta_3$
$SC_4(B_3, B_4) \geq \theta_4$
$SC_5(B_2) \geq \theta_5$
$SC_6(B_3) \geq \theta_6$
$SC_7(B_4) \geq \theta_7$

Figure 7: SCs for the synthetic dataset

γ	θ_1	θ_2	θ_3	θ_4	θ_5	θ_6	θ_7
1	0.0000021	0.000075	0.00015	0.00039	0.0054	0.014	0.028
2	0.0000083	0.00015	0.00030	0.00077	0.0054	0.014	0.028
3	0.000019	0.00023	0.00044	0.0012	0.0054	0.014	0.028
4	0.000033	0.00030	0.00059	0.0016	0.0054	0.014	0.028
5	0.000052	0.00038	0.00074	0.0019	0.0054	0.014	0.028
8	0.00013	0.00060	0.0012	0.0031	0.0054	0.014	0.028
9	0.00016	0.00067	0.0013	0.0035	0.0054	0.014	0.028
10	0.00020	0.00075	0.0015	0.0039	0.0054	0.014	0.028
13	0.00035	0.00097	0.0019	0.0050	0.0054	0.014	0.028
15	0.00047	0.0011	0.0022	0.0058	0.0054	0.014	0.028
17	0.00060	0.0013	0.0025	0.0066	0.0054	0.014	0.028
18	0.00065	0.0014	0.0027	0.0070	0.0054	0.014	0.028
20	0.00083	0.0015	0.0030	0.0077	0.0054	0.014	0.028

Figure 8: The minimum support θ_i for the synthetic dataset

non-empty subsets of $\{B_2, B_3, B_4\}$ and Figure 8 shows the minimum support in these SCs. B_1 was excluded because $S(B_1)$ is too low. For each non-empty subset of the 7 SCs, we created one support specification by adding $SC_0() \geq 0.03$ as the default SC. In this way, we generated all the 127 support specifications not involving B_1.

7.1.1 Benchmarking against Apriori

The benefit of Adaptive Apriori is measured by benchmarking it against the classic Apriori. We considered four measures: the execution time, the number of candidates generated, the number of $frequent(Pminsup)$ itemsets, and the number of $frequent(minsup)$ itemsets. A *relative measure* is the ratio of the measure for Adaptive Apriori to the measure for Apriori. Figure 9 plotted the four relative measures for the 127 support specifications, where $\gamma = 15$ and the static Strategy 1

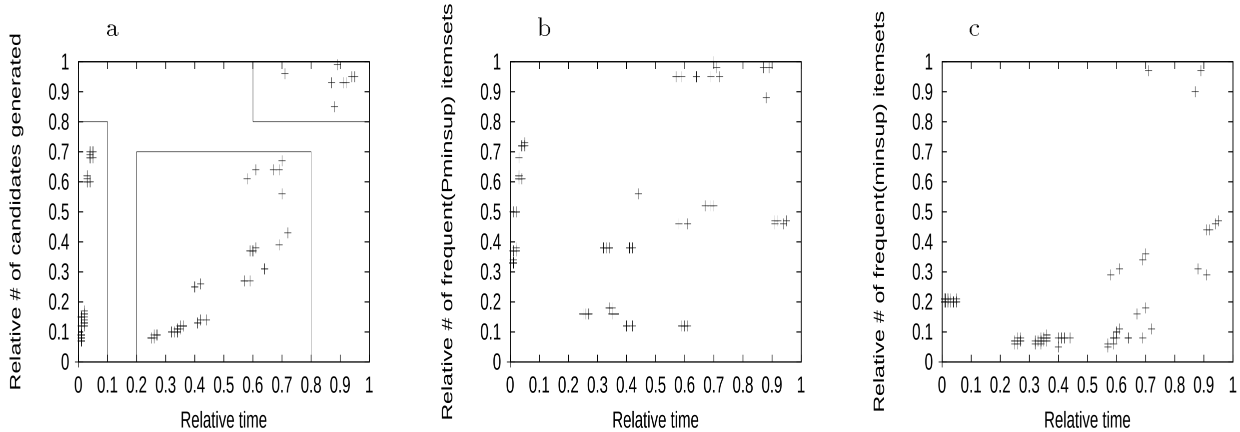


Figure 9: The measures relative to Apriori

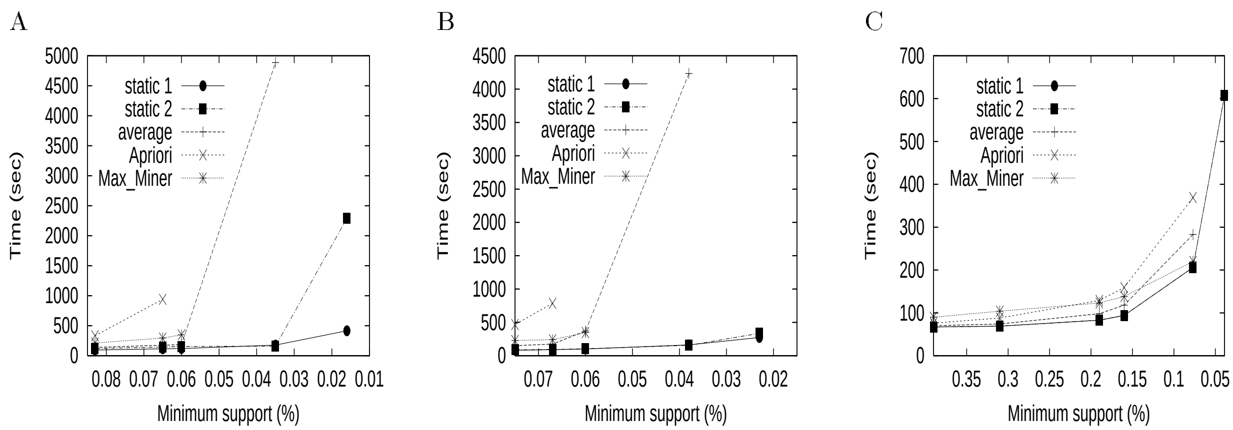


Figure 10: The dead points for the synthetic dataset

was used for Adaptive Apriori. In the figures, each support specification was represented by a point (x, y). The x-value represents the relative execution time, and the y-value represents the other three relative measures. Here are the main findings.

- All points lie southwest of the corner point (1,1). This shows that Adaptive Apriori is more efficient than Apriori in both time and space for all support specifications considered.
- There are three clusters of points in Figure 9(a), indicated by the three boxes. Cluster 1 contains the 64 points with $0 \leq x \leq 0.1$, corresponding to the 64 specifications containing the SC of length 3, i.e., $SC_1 \geq \theta_1$. Cluster 2 contains the 54 points with $0.2 \leq x \leq 0.8$ and $0 \leq y \leq 0.7$, mostly representing the specifications that contain more SCs of length 2 than SCs of length 1. Cluster 3 contains the 9 points with $x \geq 0.7$ and $y \geq 0.8$, mostly representing the specifications that contain more SCs of length 1 than SCs of length 2. Intuitively, Clusters 1, 2, and 3 correspond to large, medium, and small variances of minimum supports in support specification, thereby, good, average, and bad cases for Adaptive Apriori.

- Cluster 1 has a small relative time. At $\gamma = 15$, the minimum support for Apriori is $\theta_1 = 0.00047$. At such a low minimum support, page swapping between memory and disk took place when the hash-tree was traversed for counting the support of candidates. This drastically increased the execution time of Apriori. In fact, we had to stop Apriori after 3 hours of running and used the measures obtained for the higher minimum support 0.0006, which finished in 9,858 seconds, as the replacement in computing the above relative measures. On the other hand, all runs of Adaptive Apriori finished in less than 538 seconds without page swapping, by benefiting from using higher minimum supports in a specification.

- Cluster 2 represents the normal case where no page swapping took place in Apriori. In this case, the execution time was proportional to the number of candidate generated, and both Apriori and Adaptive Apriori were reasonably fast. As a result, the relative time is not very small.

- Adaptive Apriori did not benefit much for Cluster 3. To see this, consider the representative specification at point (0.95,0.95): $SC_4 \geq 0.0058$, $SC_5 \geq 0.0054$, $SC_6 \geq 0.014$, $SC_7 \geq 0.028$, $SC_0 \geq 0.03$. For this specification, the minimum support used for Apriori is $\theta_5 = 0.0054$ (recall that $\gamma = 15$). Only 815 itemsets satisfied this minimum support. As a result, the other minimum supports, i.e., 0.0058, 0.014, 0.028, and 0.03, are too high for most itemsets, and Adaptive Apriori could not benefit from using them.

7.1.2 The scalability with respect to minimum support

The scalability is measured by the dead point as defined at the beginning of this section. We consider the three representative specifications (all refer to Figure 9(a)):

- **Specification A** at point (0.01,0.09) from Cluster 1: contains all 7 SCs. We set γ at 20, 18, 17, 13, 9, and 8, corresponding to the lowest minimum supports 0.00083, 0.00065, 0.00060, 0.00035, 0.00016, and 0.00013.

- **Specification B** at point (0.36,0.12) from Cluster 2: contains $SC_2 \geq \theta_2$, $SC_3 \geq \theta_3$, $SC_5 \geq \theta_5$. We set γ at 10, 9, 8, 5, 3, and 2, corresponding to lowest minimum supports being 0.00075, 0.00067, 0.00060, 0.00038, 0.00023, and 0.00015.

- **Specification C** at point (0.95,0.95) from Cluster 3: contains $SC_4 \geq \theta_4$, $SC_5 \geq \theta_5$, $SC_6 \geq \theta_6$, $SC_7 \geq \theta_7$. We set γ at 10, 8, 5, 4, 2, and 1, corresponding to lowest minimum supports being 0.0039, 0.0031, 0.0019, 0.0016, 0.00077, and 0.00039.

Shown in Figure 10(A), (B), (C) are the execution time for specifications A, B, C, respectively. The x-value represents the lowest minimum support in a specification. "static 1" refers to static Strategy 1 in Adaptive Apriori, etc., and "average" refers to the average of all nodes orderings in Adaptive Apriori. The dynamic strategies have a behavior similar to their static counterparts and were omitted. The right-most point on each curve represents the dead point, with the understanding that for the *next* lowest minimum support tested, the run did not finish within 3 hours.

For specification A, Apriori first reached the dead point (0.00065), followed by Max_Miner (0.00060), "average" (0.00035), and "static 2" and "static 1"(0.00016). In fact, at the dead point of "static 1", for 22% of the nodes expanded, *Pminsup* is higher than the lowest minimum support 0.00016. This explains why "static 1" has a much smaller dead point. The experiment also shows that even the random ordering of nodes can do better than not pushing support constraints at all. For Max_Miner, as the minimum support became very low, the number of candidates grew fast because most lookahead tests failed. For Max_Miner, the execution time does not include the post-processing time for computing the support of all (not necessarily maximal) frequent itemsets.

The result for specification B in Figure 10(B) is similar to specification A, except that the difference between "static 1" and "static 2" diminished. The dead points are: 0.00067 for Apriori, 0.00060 for Max_Miner, 0.00038 for "average", 0.00023 for "static 1" and "static 2". For specification C, the dead points are: 0.00077 for Apriori, Max_Miner, and "average", and 0.00039 for "static 1" and "static 2". As mentioned in Section 7.1.2, the problem with specification C is that Adaptive Apriori could not exploit the higher minimum supports due to low support in the data. For example, at the dead point of "static 1", only 5% of the nodes expanded used a *Pminsup* larger than the lowest minimum support 0.00039. As γ was reduced, $\theta_5, \theta_6, \theta_7$ remained unchanged, and so did this problem.

7.2 The census dataset

We also experimented on the census data used in [20], which is a 5% random sample of the data collected in Washington state in the 1990 census. The data has 23 attributes, 77 items [4] and 126,229 transactions. Each transaction corresponds to an individual, and each item corresponds to an attribute/value pair. Figure 11(a) shows the distribution of item support. Unlike the synthetic dataset in Section 7.1, many items have a high support, say above 0.1, and the support varies over a wide range. We like to verify that Adaptive Apriori will benefit from this favorable case.

To generate the support specification, we grouped the items from the same attribute into a bin, yielding 23 bins $B_1, \ldots, B_{23}$ for the 23 attributes. Figure 11(b) shows the lowest support, denoted $S(B_i)$, and the size for each bin B_i. We specified the following SCs in the closed interpretation:

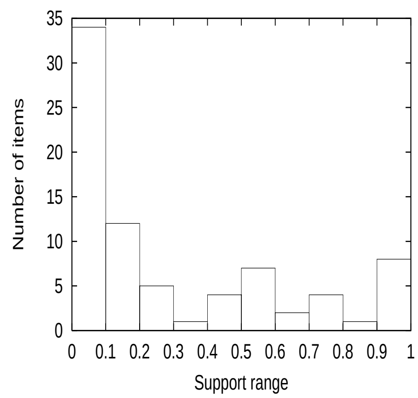
$$SC_i(V_1, \ldots, V_k) \geq \theta_i(V_1, \ldots, V_k) \quad (k > 0) \tag{2}$$

where

$$\theta_i(V_1, \ldots, V_k) = \begin{cases} 0.0000158 & \text{if } \gamma^{k-1} \times S(V_1) \times \ldots \times S(V_k) < 0.0000158 \\ 1 & \text{if } \gamma^{k-1} \times S(V_1) \times \ldots \times S(V_k) > 1 \\ \gamma^{k-1} \times S(V_1) \times \ldots \times S(V_k) & \text{otherwise} \end{cases}$$

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

(a)



(b)

Bins	$S(B_i)$	Size		Bins	$S(B_i)$	Size
B_{19}	0.0038	15		B_{14}	0.0772	4
B_{21}	0.0084	6		B_{11}	0.0846	2
B_{12}	0.0175	2		B_7	0.0848	2
B_{18}	0.0180	2		B_{20}	0.1046	3
B_5	0.0199	5		B_{13}	0.2263	2
B_3	0.0232	4		B_{17}	0.2360	2
B_{16}	0.0266	4		B_{22}	0.2717	2
B_6	0.0349	2		B_4	0.4107	2
B_{23}	0.0388	2		B_{10}	0.4503	2
B_8	0.0403	2		B_1	0.4589	2
B_{15}	0.0487	3		B_2	0.4949	2
B_9	0.0733	4				

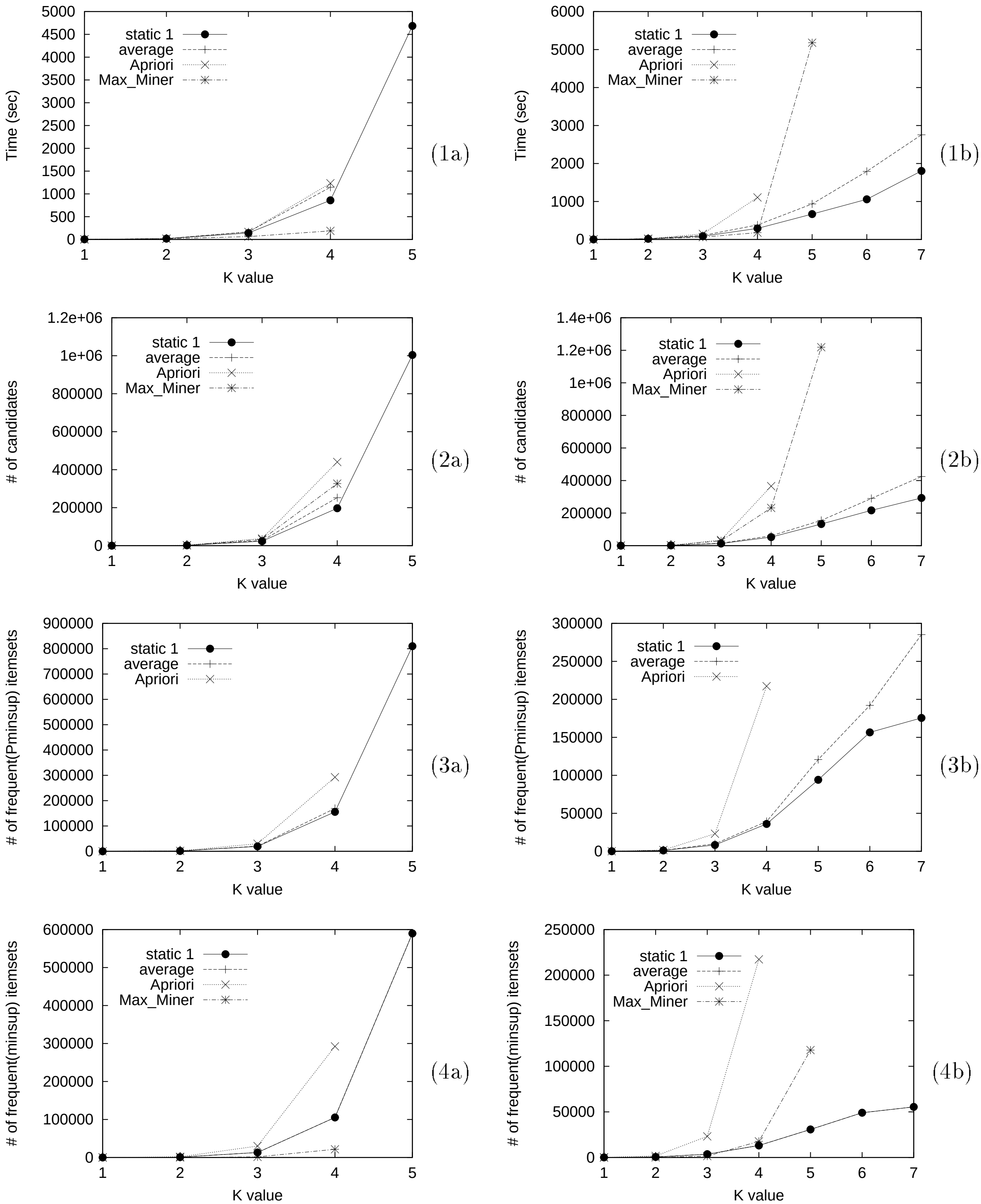
Figure 11: The bins for the census dataset

where V_i is a bin variable and $k \leq K$ for the maximal itemset size K specified by the user. Each specification is defined by a pair of γ and K values. The lower bound of minimum support is 0.0000158, corresponding to the support requirement of at least 2 transactions. Since the occurrence of bins is symmetric, Strategy 2 and Strategy 3 do not impose a bias on the ordering of nodes, so are not considered here. 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.

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 12 shows the lowest minimum support for each (γ, K) pair. In Figure 12, on the left are the measures for $\gamma = 5$, and on the right are the measures for $\gamma = 20$. In Figure 12(4a,4b), the y-value for Max_Miner is the number of maximal frequent itemsets. As before, the dead point is represented by the right-most point on a curve. All algorithms were terminated after K iterations for the given K. For a small K, Max_Miner worked very well. But as K increased, it lost to Adaptive Apriori because most lookahead tests failed. 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 12.

To get an insight into how $Pminsup$ is actually distributed in the schema enumeration tree, we plotted $Pminsup$ vs nodes numbered in the breath-first ordering for the dead point of "static 1" at the settings $(\gamma = 20, K = 7)$ and $(\gamma = 5, K = 5)$. See Figure 13 and Figure 14. Though the two cases have the same 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% (still a lot) of the nodes expanded have $Pminsup$ higher than 0.0000158.

In summary, these experiments strongly supported our claim that if itemsets are of varied



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

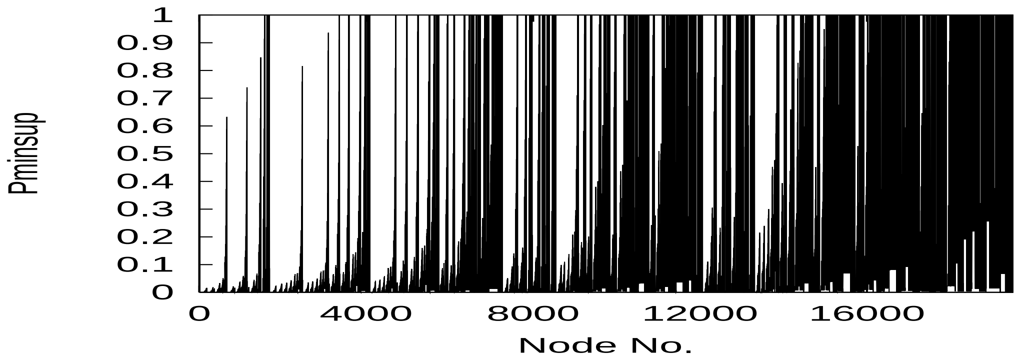


Figure 13: $\gamma = 20$ and $K = 7$: $Pminsup > 0.0000158$ for 99.3% of nodes

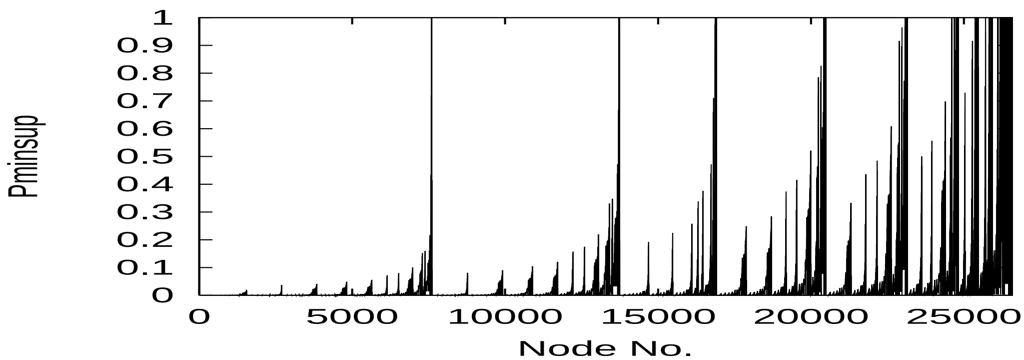


Figure 14: $\gamma = 5$ and $K = 5$: $Pminsup > 0.0000158$ for 88.1% of nodes

supports, pushing support constraints is an effective strategy to deal with the bottleneck of itemset generation. Often, the difference is not an order of magnitude, but the feasibility of solving a problem using given resources.

8 Conclusion

One contribution of this work is introducing the notion of support constraints into frequent itemset mining. We motivated the need for support constraints and discussed the representation and specification of support constraints. Another contribution is the framework for pushing support constraints into the Apriori itemset generation. The challenge is that the classic Apriori is lost in the presence of a non-uniform minimum support. Instead of using the lowest minimum support specified, our approach is to use the best "run time" minimum support pushed for each itemset that preserves the Apriori itemset generation. We call this strategy Adaptive Apriori. A major advantage of preserving the Apriori itemset generation is that nearly all improvements of Apriori over the last several years are immediately applicable to Adaptive Apriori. Unlike earlier constraint pushings, 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" pushed minimum support is maximized. We proposed several strategies for this and studied their effectiveness. Experiments showed that pushing support constraints is highly effective in dealing with the bottleneck of itemset generation. The effectiveness is not in an order of magnitude, but the feasibility of problem solving using given resources. As a future work, we like to study how the mining framework for non-uniform minimum support can be extended beyond the Apriori itemset generation. For example, [10] finds frequent itemsets without generating candidates like in Apriori. It is interesting to see how our approach can be extended in this direction.

References

[1] C. C. Aggarwal and P. S. Yu. A new framework for itemset generation. In *PODS*, pages 18–24, Sept 1998.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large datasets. In *SIGMOD*, pages 207–216, May 1993.

[3] R. Agrawal and R. Srikant. Fast algorithm for mining association rules. In *VLDB*, pages 487–499, Sept 1994.

[4] R. J. Bayardo. Efficiently mining long patterns from databases. In *SIGMOD*, pages 85–93, May 1998.

[5] S. Brin, R. Motwani, and C. Silverstein. Beyond market baskets: generalizing association rules to correlations. In *SIGMOD*, pages 265–276, May 1997.

[6] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *SIGMOD*, pages 255–264, May 1997.

[7] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. Ullman, and C. Yang. Finding interesting associations without support pruning. In *ICDE*, pages 489–499, Feb 2000.

[8] G. Dong and J. Li. Efficient mining of emerging patterns: discovering trends and differences. In *SIGKDD*, pages 43–52, August 1999.

[9] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *VLDB*, pages 420–431, Sept 1995.

[10] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, pages 1–12, May 2000.

[11] W. Lee, S. Stolfo, and K. Mok. Mining audit data to build intrusion detection models. In *KDD*, pages 66–72, Sept 1998.

[12] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *KDD*, pages 80–86, Sept 1998.

[13] B. Liu, W. Hsu, and Y. Ma. Mining association rules with multiple minimum supports. In *SIGKDD*, pages 125–134, Sept 1999.

[14] H. Mannila, H. Toivonen, and A. Verkamo. Efficient algorithm for discovering association rules. In *KDD*, pages 181–192, Sept 1994.

[15] D. Meretakis and B. Wuthrich. Extending naive bayes classifiers using long itemsets. In *SIGKDD*, pages 165–174, August 1999.

[16] R. Ng, L. V. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *SIGMOD*, pages 13–24, May 1998.

[17] J. Park, M. S. Chen, and P. Yu. An efficient hash based algorithm for mining association rules. In *SIGMOD*, pages 175–186, May 1995.

[18] R. Rymon. Search through systematic set enumeration. In *Principles of Knowledge Representation and Reasoning*, pages 539–550, 1992.

[19] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *VLDB*, pages 432–444, Sept 1995.

[20] C. Silverstein and J. U. S. Brin, R. Motwani. Scalable techniques for mining causal structures. In *VLDB*, pages 594–605, Sept 1998.

[21] R. Srikant and R. Agrawal. Mining generalized association rules. In *VLDB*, pages 407–419, Sept 1995.

[22] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In *KDD*, pages 67–73, August 1997.

[23] H. Varian and P. Resnick. Recommender systems – introduction to the special section. *Communications of the ACM*, 40(3):56–58, January 1997.

[24] K. Wang, Y. He, and J. Han. Pushing support constraints into frequent itemset mining. In *VLDB*, pages 43–52, Sept 2000.

[25] K. Wang, S. Zhou, and S. Liew. Building hierarchical classifiers using class proximity. In *VLDB*, pages 363–374, Sept 1999.