Appears in Proc. of the 1998 ACM-SIGMOD Int'l Conf. on Management of Data, 85-93.

Efficiently Mining Long Patterns from Databases

Roberto J. Bayardo Jr.
IBM Almaden Research Center
http://www.almaden.ibm.com/cs/people/bayardo/
bayardo@alum.mit.edu

Abstract

We present a pattern-mining algorithm that scales roughly linearly in the number of maximal patterns embedded in a database irrespective of the length of the longest pattern. In comparison, previous algorithms based on Apriori scale exponentially with longest pattern length. Experiments on real data show that when the patterns are long, our algorithm is more efficient by an order of magnitude or more.

1. Introduction

Finding patterns in databases is the fundamental operation behind several common data-mining tasks including association rule [1] and sequential pattern mining [4]. For the most part, pattern mining algorithms have been developed to operate on databases where the longest patterns are relatively short. This leaves data outside this mold unexplorable using current techniques. Interesting data-sets with long patterns include those composed of questionnaire results (people tend to answer similarly to many questions), sales transactions detailing the purchases made by regular customers over a large time window, and biological data from the fields of DNA and protein analysis. Most categorically-valued data-sets used for classification problems (e.g. targeted marketing campaigns) also tend to have long patterns because they contain many frequently occurring items and have a wide average record length.

Almost every recently-proposed pattern-mining algorithm is a variant of Apriori [2]. Two recent papers have demonstrated that Apriori-like algorithms are inadequate on data-sets with long patterns. Brin et al. [6] applied their association-rule miner DIC to a data-set composed of PUMS census records. To reduce the difficulty of this data-set, they removed all items appearing in over 80% of the transactions yet still could only mine efficiently at high support. We [5] previously applied an Apriori-inspired algorithm to several data-sets from the Irvine Machine Learning Database Repository. In order to mine efficiently, this algorithm had to sometimes apply pruning strategies that rendered the search incomplete.

Apriori involves a phase for finding patterns called *frequent itemsets*. A frequent itemset is a set of items appearing together in a number of database records meeting a user-specified threshold. Apriori employs a bottom-up search that enumerates every single frequent itemset. This implies in order to produce a frequent itemset of length l, it must produce all 2^l of its subsets since they too must be frequent. This exponential complexity fundamentally restricts Apriori-like algorithms to discovering only short patterns.

To address this problem, this paper proposes the Max-Miner algorithm for efficiently extracting only the maximal frequent itemsets, where an itemset is maximal frequent if it has no superset that is frequent. Because any frequent itemset is a subset of a maximal frequent itemset, Max-Miner's output implicitly and concisely represents all frequent itemsets. Max-Miner is shown to result in two or more orders of magnitude in performance improvements over Apriori on some data-sets. On other data-sets where the patterns are not so long, the gains are more modest. In practice, Max-Miner is demonstrated to run in time that is roughly linear in the number of maximal frequent itemsets and the size of the database, irrespective of the size of the longest frequent itemset.

Max-Miner is successful because it abandons a strict bottom-up traversal of the search space, and instead always attempts to “look ahead” in order to quickly identify long frequent itemsets. By identifying a long frequent itemset early on, Max-Miner can prune all its subsets from consideration. Max-Miner uses a heuristic to tune its search in an effort to identify long frequent itemsets as early as possible. It also uses a technique that can often determine when a new candidate itemset is frequent before accessing the database. The idea is to use information gathered during previous database passes to compute a good lower-bound on the number of transactions that contain the itemset.

The techniques we introduce in this paper are flexible and can be extended in various ways and applied to other algorithms. To demonstrate this point, we optimize Apriori with the lower-bounding technique mentioned above. While the fundamental limitations of Apriori with respect to pattern length remain, performance is improved by an order of magnitude on several data-sets. We also show how Max-Miner can be extended to exploit additional pattern constraints during its search by creating a variant that identifies only the longest of the maximal frequent itemsets in a data-set. This algorithm efficiently identifies all of the longest maximal frequent itemsets even when the space of all maximal frequent itemsets is itself intractably large.

1.1 Related Work

There are many variants of Apriori that differ in how they check “candidate” itemsets against the database. Apriori in its purest form checks itemsets of length l for frequency during database pass l. DIC [6] is more eager and begins checking an itemset shortly after all its subsets have been determined frequent, rather than waiting until the database pass completes. Partition [11] identifies all frequent-itemsets in memory-sized partitions of the database, and then checks these against the entire database during a final pass. DIC considers the same number of candidate itemsets as Apriori, and Partition can consider more but never fewer candidate itemsets than Apriori, potentially exacerbating problems associated with long patterns.

Park et al. [9] enhance Apriori with a hashing scheme that can identify (and thereby eliminate from consideration) some candidates that will turn up infrequent if checked against the database. It also uses the hashing scheme to re-write a smaller database after each pass in order to reduce the overhead of subsequent passes. Still, like Apriori, it considers every frequent itemset.

Gunopulos et al. [7] present a randomized algorithm for identifying maximal frequent itemsets in memory-resident databases. Their algorithm works by iteratively attempting to extend a working pattern until failure. A randomized version of the algorithm that does not guarantee every maximal frequent itemset will be returned is evaluated and found to efficiently extract long frequent itemsets. Unfortunately, it is not clear how this algorithm would be scaled to disk resident data-sets since each attempt at extending an itemset requires a scan over the data. It also remains to be seen how the proposed complete version of the algorithm would perform in practice.

Zaki et al. [16] present the algorithms MaxEclat and MaxClique for identifying maximal frequent itemsets. These algorithms are similar to Max-Miner in that they also attempt to look ahead and identify long frequent itemsets early on to help prune the space of candidate itemsets considered. The important difference is that Max-Miner attempts to look ahead throughout the search, whereas MaxEclat and MaxClique look ahead only during an initialization phase prior to a purely bottom-up Apriori-like search with exponential scaling. The initialization phase of MaxClique is also prone to problems with long frequent itemsets since it uses a dynamic programming algorithm for finding maximal cliques in a graph whose largest clique is at least as large as the length of the longest frequent itemset.

Concurrent to our work, Lin and Kedem [8] have proposed an algorithm called Pincer-Search for mining long maximal frequent itemsets. Like Max-Miner, Pincer-Search attempts to identify long patterns throughout the search. The difference between these algorithms is primarily in the long candidate itemsets considered by each. Max-Miner uses a simple, polynomial time candidate generation procedure directed by heuristics, while Pincer-Search uses an NP-hard reduction phase to ensure no long candidate itemset contains any known infrequent itemset. Our comparison with Pincer-Search is thus far only preliminary, and more work is needed to fully understand the advantages offered by each technique.

1.2 Overview

Section 2 begins with an introduction into the basic search strategy used by Max-Miner and the techniques used to restrict the search space. Section 3 formalizes these techniques through pseudo-code, provides implementation details, and establishes Max-Miner's correctness and efficiency characteristics. Section 4 discusses and exploits a technique for lower-bounding the support of candidate itemsets. Section 5 discusses the integration of additional pattern constraints into the search, culminating in a description of the Max-Miner-LO algorithm for finding only the longest maximal frequent itemsets. The algorithms are evaluated experimentally in Section 6 followed by a summary of contributions and avenues for future work in Section 7.

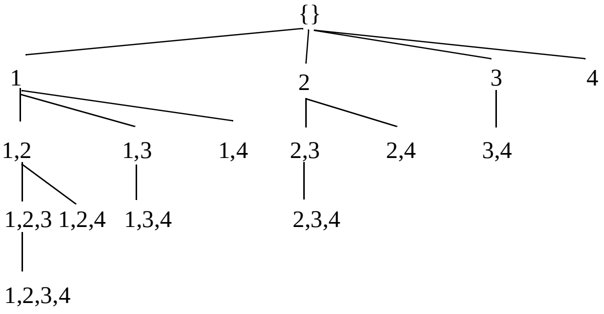
2. Introducing Max-Miner

We begin with defining the necessary terminology for describing the Max-Miner algorithm. For simplicity of presentation, we will be dealing only with the problem of identifying frequent itemsets. The application of our techniques to finding other patterns (e.g. sequential patterns) is similar.

A *data-set* is a set of *transactions* that are sets over a finite item domain. Transactions can represent things such as the supermarket items purchased by a customer during a shopping visit, or the characteristics of a person as described by his or her replies in a census questionnaire. A set of items is more succinctly called an *itemset*, and a *frequent itemset* is one that is contained in a number of transactions above or equal to the minimum support (*minsup*) specified by the user. An itemset with k items will be more succinctly referred to as a k-itemset. The *support* of an itemset I, denoted $\sup(I)$, is the number of transactions that contain it. The minsup parameter will sometimes be specified as a percentage of the transactions in the data-set instead of as an absolute number of transactions.

Max-Miner can be described using Rymon's generic set-enumeration tree search framework [10]. The idea is to expand sets over an ordered and finite item domain as illustrated in Figure 1 where four items are denoted by their position in the ordering. The particular ordering imposed on the item domain affects the parent/child relationships in the set-enumeration tree but not its completeness. The figure assumes a static lexical ordering of the items, but later we describe an optimization that dramatically improves performance by heuristically ordering the items and dynamically reordering them on a per-node basis. Set-enumeration trees are not data-structures like the hash tree or trie, but instead are used to illustrate how sets of items are to be completely enumerated in a search problem. Note that the tree could be traversed depth-first, breadth first, or even best-first as directed by some heuristic. Max-Miner employs a purely breadth-first search of the set-enumeration tree in order to limit the number of passes made over the data.

Figure 1. Complete set-enumeration tree over four items.



The key to an efficient set-enumeration search is the pruning strategies that are applied to remove entire branches from consideration. Without pruning, a set-enumeration tree search for frequent itemsets will consider every itemset over the set of all items. Max-Miner uses pruning based on subset infrequency, as does Apriori, but it also uses pruning based on superset frequency.

To aid in our pruning efforts, we will represent each node in the set enumeration tree by what we call a *candidate group*. A candidate group g consists of two itemsets. The first, called the head and denoted $h(g)$, represents the itemset enumerated by the node. The second itemset, called the tail and denoted $t(g)$, is an ordered set and contains all items not in $h(g)$ that can potentially appear in any sub-node. For example, the node enumerating itemset $\{1\}$ in the figure has $h(g) = \{1\}$ and $t(g) = \{2, 3, 4\}$. The ordering of tail items reflect how the sub-nodes are to be expanded. In the case of a static lexical ordering without pruning, the tail of any candidate group is trivially the set of all items following the greatest item in the head according to the item ordering. When we are applying pruning and dynamic item reordering, it becomes necessary to make the tail items explicit.

When we say we are *counting the support* of a candidate group g, we are computing the support of itemsets $h(g)$, $h(g) \cup t(g)$, and $h(g) \cup \{i\}$ for all $i \in t(g)$. The supports of itemsets other than $h(g)$ are used for pruning. For example, consider first the itemset $h(g) \cup t(g)$. Since $h(g) \cup t(g)$ contains every item that appears in any viable sub-node of g, if it is frequent, then any itemset enumerated by a sub-node will also be frequent but not maximal. Superset-frequency pruning can therefore be implemented by halting sub-node expansion at any candidate group g for which $h(g) \cup t(g)$ is frequent. Consider next the itemset $h(g) \cup \{i\}$ for some $i \in t(g)$. If $h(g) \cup \{i\}$ is infrequent, then any head of a sub-node that contains item i will also be infrequent. Subset-infrequency pruning can therefore be implemented by simply removing any such tail item from a candidate group before expanding its sub-nodes.

3. Formalizing Max-Miner

We now provide a pseudo-code description of Max-Miner followed by the motivation behind and description of the item ordering policy. Implementation details describing how Max-Miner can use the same data-structures as Apriori are provided in the following subsection, and the last subsection provides correctness and efficiency arguments.

3.1 Pseudo-Code for Max-Miner

The pseudo-code description of Max-Miner appears in figures 2 through 4. The body (Figure 2) accepts a data-set and (implicitly) the minimum support specified by the user. The while loop implements a breadth-first search of the set-enumeration tree that maintains every frequent itemset encountered so long as it is potentially maximal. The function Gen-Initial-Groups (Figure 3) performs the initial scan over the data-set to identify the item domain and seed the search at the second level of the tree. Superset-frequency based pruning is performed by only expanding the sub-nodes of a candidate g if $h(g) \cup t(g)$ is infrequent. Another instance of superset-frequency pruning is any candidate group g is pruned if $h(g) \cup t(g)$ is a subset of some already-know-to-be frequent itemset I.

Sub-nodes are generated by Gen-Sub-Nodes in Figure 4. Subset-infrequency pruning is performed here through the removal of any tail item i from a candidate group g if $h(g) \cup \{i\}$ is infrequent. Gen-Sub-Nodes and Gen-Initial-Groups return the sub-node with an empty tail as a frequent itemset instead of a candidate group since its frequency is already known and it has no children in the search tree.

Figure 2. Max-Miner at its top level.

```
MAX-MINER(Data-set $T$)
    ;; Returns the set of maximal frequent itemsets present in $T$
    Set of Candidate Groups $C \leftarrow \{\ \}$
    Set of Itemsets $F \leftarrow \{$GEN-INITIAL-GROUPS($T, C$)$\}$
    while $C$ is non-empty do
        scan $T$ to count the support of all candidate groups in $C$
        for each $g \in C$ such that $h(g) \cup t(g)$ is frequent do
            $F \leftarrow F \cup \{h(g) \cup t(g)\}$
        Set of Candidate Groups $C_{new} \leftarrow \{\ \}$
        for each $g \in C$ such that $h(g) \cup t(g)$ is infrequent do
            $F \leftarrow F \cup \{$GEN-SUB-NODES($g, C_{new}$)$\}$
        $C \leftarrow C_{new}$
        remove from $F$ any itemset with a proper superset in $F$
        remove from $C$ any group $g$ such that $h(g) \cup t(g)$
                has a superset in $F$
    return $F$
```

Figure 3. Generating the initial candidate groups.

```
GEN-INITIAL-GROUPS(Data-Set $T$, Set of Candidate Groups $C$)
    ;; $C$ is passed by reference and returns the candidate groups
    ;; The return value of the function is a frequent 1-itemset
    scan $T$ to obtain $F_1$, the set of frequent 1-itemsets
    impose an ordering on the items in $F_1$ ;; see section 3.2
    for each item $i$ in $F_1$ other than the greatest item do
        let $g$ be a new candidate with $h(g) = \{i\}$
            and $t(g) = \{j | j$ follows $i$ in the ordering$\}$
        $C \leftarrow C \cup \{g\}$
    return the itemset in $F_1$ containing the greatest item
```

Figure 4. Generating sub-nodes.

```
GEN-SUB-NODES(Candidate Group $g$, Set of Cand. Groups $C$)
    ;; $C$ is passed by reference and returns the sub-nodes of $g$
    ;; The return value of the function is a frequent itemset
    remove any item $i$ from $t(g)$ if $h(g) \cup \{i\}$ is infrequent
    reorder the items in $t(g)$ ;; see section 3.2
    for each $i \in t(g)$ other than the greatest do
        let $g'$ be a new candidate with $h(g') = h(g) \cup \{i\}$
            and $t(g') = \{j | j \in t(g)$ and $j$ follows $i$ in $t(g)\}$
        $C \leftarrow C \cup \{g'\}$
    return $h(g) \cup \{m\}$ where $m$ is the greatest item in $t(g)$,
            or $h(g)$ if $t(g)$ is empty.
```

3.2 Item Ordering Policies

The motivation behind item (re)ordering is to increase the effectiveness of superset-frequency pruning. Recall that superset-frequency pruning can be applied when a candidate group g is found such that $h(g) \cup t(g)$ is frequent. We therefore want to make it likely that many candidate groups will have this property. A good heuristic for accomplishing this is to force the most frequent items to appear in the most candidate groups. This is simply because items with high frequency are more likely to be part of long frequent itemsets. Items that appear last in the ordering will appear in the most candidate groups. For instance, item 4 from Figure 1 appears either in the head or tail of every single node. Item ordering is therefore used to position the most frequent items last.

Gen-Initial-Groups orders the items in increasing order of $\sup(\{i\})$. Items in the tails of candidate groups are re-ordered prior to sub-node generation in Gen-Sub-Nodes. This function orders the tail items of a group g in increasing order of $\sup(h(g) \cup \{i\})$. This strategy tunes the frequency heuristic by having it consider only the subset of transactions relevant to the given node.

Interestingly, the same item reordering heuristic is used by Slagel et al. [12] in a set-enumeration algorithm for identifying prime implicants in CNF propositional logic expressions. The fact that the same policy works well for both problems is likely due to their close relationship. Finding prime implicants in CNF expressions is similar to the problem of generating hypergraph transversals, and Gunopulos et al. [7] have previously shown that hypergraph transversal algorithms can be used as a component of an algorithm for mining maximal frequent itemsets.

3.3 Implementation Details

Max-Miner can use the same data-structures as Apriori (as detailed in [3]) for efficiently computing itemset supports. The primary data-structure used by Apriori is the hash tree to index candidate itemsets. Max-Miner uses the hash tree to index only the head of each candidate group. For each transaction in the data-set, Max-Miner uses the hash tree to quickly look up all candidate groups whose head appears in the transaction. Then, for each candidate group g identified, it traverses down its tail items one by one, incrementing the support of $h(g) \cup \{i\}$ if tail item i is present in the transaction. If every tail item appears in the transaction, then the support of $h(g) \cup t(g)$ is also incremented. We found this implementation to be substantially faster than individually storing each itemset within the hash tree. Hash trees are also used by our implementation of Max-Miner for efficiently identifying the subsets of frequent itemsets in F and C.

Our implementation of Max-Miner diverges from the pseudo-code description in only one way. During the second pass over the data-set, we use a two-dimensional array for quickly computing the support of all 2-itemsets as suggested in [3], and do not compute the support of the long itemsets $h(g) \cup t(g)$. We have found that the long itemsets $h(g) \cup t(g)$ almost always turn up infrequent at this stage, so computing their support offers no benefit.

3.4 Correctness and Efficiency Claims

THEOREM (CORRECTNESS): Max-Miner returns all and only the maximal frequent itemsets in the given data-set.

Proof: The fact that Max-Miner identifies and stores every maximal frequent itemset follows from the completeness of a set-enumeration tree search and the fact that branches of the set-enumeration tree are pruned if and only if they lead to only infrequent itemsets or non-maximal frequent itemsets. The fact that Max-Miner returns only those itemsets that are maximal frequent follows from the operation within the body of Max-Miner that continuously removes any itemset I if a frequent superset of I is known. □

Max-Miner, like Apriori, easily handles disk-resident data because it requires only one transaction at a time in memory. Also like Apriori, the number of passes over the data made by Max-Miner is bounded by the length of the longest frequent itemset. For Apriori this bound is nearly tight. Max-Miner, on the other hand, often performs substantially fewer database passes, as will be demonstrated in our evaluation.

THEOREM (EFFICIENCY): Max-Miner makes at most $l + 1$ passes over the data-set, where l denotes the length of the longest frequent itemset.

Proof: We establish the claim by demonstrating why Max-Miner terminates after pass $l + 1$ if it happens to get that far. By the manner in which Max-Miner generates candidate groups, it is easy to see that during pass k, the head of any candidate group has exactly $k - 1$ items. After pass $l + 1$, for any candidate group g and any of its tail items i, $h(g) \cup \{i\}$ will be infrequent otherwise we would have a frequent itemset longer than l. This implies that the next set of candidate groups will be empty, in which case the algorithm terminates. □

4. Support Lower-Bounding

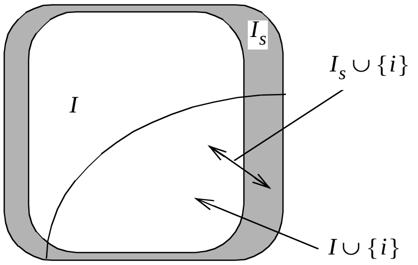
This section first describes a general technique that can be used to obtain a lower-bound on the support of an itemset by exploiting the support information provided by its subsets. The subsequent subsections describe how this technique is used to improve the performance of Max-Miner and Apriori.

4.1 Computing Support Lower-Bounds

The idea behind our method of computing a lower-bound on the support of an itemset is to exploit, as much as possible, the available support information provided by its subsets. The following function is useful in this endeavor, as established by the subsequent theorem. The function computes the number of transactions that are "dropped" from the supporting set of an itemset when it is extended with a given item.

DEFINITION: $\mathrm{drop}(I_s, i) = \mathrm{sup}(I_s) - \mathrm{sup}(I_s \cup \{i\})$, where i is an item not in itemset I_s.

Figure 5. Illustration of support drop resulting from extending itemsets I and I_s with i.



Because $\mathrm{sup}(I \cup \{i\}) = \mathrm{sup}(I) - \mathrm{drop}(I, i)$, we can get a lower-bound on the value of $\mathrm{sup}(I \cup \{i\})$ given the value of $\mathrm{sup}(I)$ and an upper-bound on $\mathrm{drop}(I, i)$. This fact is exploited by our next theorem.

THEOREM (SUPPORT LOWER-BOUNDING): $\mathrm{sup}(I) - \mathrm{drop}(I_s, i)$ is a lower-bound on the support of itemset $I \cup \{i\}$ when $I_s \subset I$.

Proof: We show that $\mathrm{drop}(I_s, i)$ is an upper-bound on $\mathrm{drop}(I, i)$ from which the claim follows. The argument is geometric and is illustrated in Figure 5. The outer square represents the space of all transactions supporting itemset I_s, and the inner square the space of all transactions supporting itemset I. Because the space occupied by I is entirely contained within the space occupied by I_s, the set of transactions dropped by extending itemset I with i must be a subset of the transactions dropped by extending itemset I_s with i. The fact that $\mathrm{drop}(I_s, i)$ is an upper-bound on $\mathrm{drop}(I, i)$ is immediate. □

The support lower-bounding theorem, as stated, can only be applied to lower-bound the support of a k-itemset if we know the support of two $k - 1$ item subsets. Below we generalize the theorem to apply when only the supports of smaller subsets are available.

THEOREM (GENERALIZED SUPPORT LOWER-BOUNDING): The following equation computes a lower-bound on the support of itemset $I \cup T$ where T is an itemset disjoint from I and $I_s \subset I$.

$$\mathrm{sup}(I) - \sum_{i \in T} \mathrm{drop}(I_s, i)$$

Proof: The equation is simply the result of applying the previous theorem repeatedly for each item i in T. □

4.2 Support Lower-Bounding in Max-Miner

Support lower-bounding can be used by Max-Miner within the Gen-Sub-Nodes function for additional superset-frequency pruning (Figure 6). If sub-nodes of a candidate group are generated while traversing the tail items in item order, a sub-node g_2 generated after g_1 will have the property that $h(g_2) \cup t(g_2) \subset h(g_1) \cup t(g_1)$. This means if $h(g_1) \cup t(g_1)$ is a frequent itemset, then any sub-node generated after g_1 can lead only to non-maximal itemsets. Max-Miner thus avoids generating sub-nodes following any sub-node g for which $h(g) \cup t(g)$ can be lower-bounded above minsup.

Figure 6. Generating sub-nodes with support lower-bounding.

```
GEN-SUB-NODES(Candidate Group $g$, Set of Cand. Groups $C$)
    ;; $C$ is passed by reference and returns the sub-nodes of $g$
    ;; The return value of the function is a frequent itemset
    remove any item $i$ from $t(g)$ if $h(g) \cup \{i\}$ is infrequent
    reorder the items in $t(g)$
    for each $i \in t(g)$ in increasing item order do
        let $g'$ be a new candidate with $h(g') = h(g) \cup \{i\}$
            and $t(g') = \{j | (j \in t(g))$ and $j$follows $i$ in $t(g)\}$
        if COMPUTE-LB($g'$, $h(g)$) $\geq$ minsup
            then return $h(g') \cup t(g')$ ;; this itemset is frequent
        else $C \leftarrow C \cup \{g'\}$
    return $h(g)$ ;; This case arises only if $t(g)$ is empty
```

Figure 7. Computing the support lower-bound.

```
COMPUTE-LB(Candidate Group $g$, Itemset $I_s$)
    ;; Returns a lower-bound on the support of $h(g) \cup t(g)$
    ;; Itemset $I_s$ is a proper subset of $h(g)$
    Integer $d \leftarrow 0$
    for each $i \in t(g)$ do
        $d \leftarrow d + \mathrm{drop}(I_s, i)$
    return $\mathrm{sup}(h(g)) - d$
```

Figure 7 shows how Max-Miner computes a lower-bound on $h(g) \cup t(g)$ using only support information provided by its subsets. The function directly applies the generalized support lower-bounding theorem. Note that the support values required for this computation are available through the itemsets whose supports were computed while counting the support of the parent node.

4.3 Support Lower-Bounding in Apriori

Apriori generates candidate itemsets of length k whose supports are computed during database pass k. An itemset is made a candidate if every $k - 1$ item subset was found frequent during the previous database pass. Suppose we are using Apriori to identify the maximal frequent itemsets. If Apriori can be made to lower-bound the support of a candidate itemset above or equal to minsup, then its support does not have to be explicitly counted. However, the candidate itemset cannot be entirely pruned because it must remain present for the subsequent candidate generation phase. Nevertheless, the savings that result from counting the support of fewer candidate itemsets can be substantial. Also, for a particular database pass, if ever Apriori can lower-bound the support of all candidate itemsets above minsup, then it can skip the database pass altogether.

A support lower-bound can be obtained on a candidate itemset I_c by plugging into the support lower-bounding theorem any proper subset I of I_c, and any proper subset I_s of I. The tightest bounds will be obtained by using only subsets with one less item. If the candidate itemset I_c has k items, there are then $(k-1)(k-2)$ subset combinations for lower-bounding the support of I_c. Our lower-bounding enhanced Apriori algorithm, Apriori-LB, considers every one of these possibilities and compares the best (largest) bound obtained to minsup.

Apriori-LB does not always have access to the exact support of every proper subset of a candidate because one or more subsets of a candidate itemset may have been left uncounted due to previous lower-bounding efforts. When this is the case, Apriori-LB uses a new function to bound support drop. The new function, drop-b, uses support bounds instead of exact support values. Because drop-b clearly computes an upper-bound on the value of drop given the same arguments, it can be used in place of drop in the lower-bounding theorems.

DEFINITION: $\text{drop-b}(I_s, i) = \text{ub-sup}(I_s) - \text{lb-sup}(I_s \cup \{i\})$, where $\text{ub-sup}(I)$ is the best-known upper-bound on $\text{sup}(I)$, and $\text{lb-sup}(I)$ is the best-known lower-bound on $\text{sup}(I)$.

The support of an itemset can be upper-bounded by the support (or upper-bound on support) of any of its subsets. Apriori-LB sets $\text{ub-sup}(I)$ of an uncounted itemset I with k items to the smallest of the support upper-bounds or (if computed) support values of each of its $k-1$ item subsets.

5. Exploiting Additional Constraints

Though the frequent patterns themselves are often of interest to the end-user, often they need to be digested further before being presented. *Association rules* are digested frequent itemsets that can be useful for prediction. The *confidence* of an association rule $i_1, i_2, \ldots, i_k \rightarrow i_c$ is equal to the support of the itemset $\{i_1, i_2, \ldots, i_k\}$ divided by the support of the itemset $\{i_1, i_2, \ldots, i_k, i_c\}$. Typically the user is interested in finding only those association rules with high confidence and support, and these are produced by searching the entire space of frequent itemsets. Another database pass is required after finding all maximal frequent itemsets in order to obtain the supports of all frequent itemsets for producing association rules. If the frequent itemsets are long, even if implemented efficiently using specialized data structures, this step is hopelessly intractable.

Max-Miner can be used to identify many, though not all, high-confidence association rules during its execution. After counting the support of a candidate group g, Max-Miner has all the supports necessary to compute the confidence of any association rule of the form $h(g) \rightarrow i_c$ where $i_c \in t(g)$. Another approach at incomplete rule-mining could be to use the set of maximal frequent itemsets to define the space of rules searched by a randomized algorithm.

Though incomplete techniques at association rule mining may sometimes be sufficient, completeness is more desirable. We believe that incorporating additional constraints into the search for frequent patterns is the only way to achieve completeness on complex data. Association rule confidence is a constraint that Bayardo [5] uses to prune some itemsets from consideration. Other constraints that have been used during the search for patterns include item constraints [15] and information-theoretic constraints [13]. Interestingness constraints thus far applied only during post-processing (e.g. [6]) might also be exploitable during search to improve efficiency.

Max-Miner provides a framework in which additional constraints can often be easily integrated into the search. Consider as an example the problem of finding only the longest frequent itemsets. This constraint is quite powerful because data sets with long frequent itemsets usually have very many maximal frequent itemsets, of which only a small fraction are longest. We have implemented a version of Max-Miner that exploits this constraint called Max-Miner-LO ("Longest Only"). It determines the cardinality l of the longest itemset in the set of frequent itemsets F after each database pass. Any frequent itemsets in F that are shorter than l are then pruned, as are any candidate groups g in C such that $|h(g) \cup t(g)| < l$.

6. Evaluation

We selected data-sets from several domains for evaluating the performance of Max-Miner, all but one of them being publicly accessible through the web. The data-set that is not publicly available was provided by a retailer. This data-set contains records listing all of the items purchased by a customer over a period of time. We produced another of our data-sets from PUMS census data available at http://augustus.csscr.washington.edu/census/comp_013.html. Following Brin et al. [6], we discretized continuous attributes and removed uninformative attributes. We created another version of the data-set where all items with 80% or greater support were discarded. The raw data from the web consists of both housing and person records, but we extracted only the person records. The remaining data-sets used in our evaluation were taken from the Irvine Machine Learning Database Repository (http://www.ics.uci.edu/~mlearn/MLRepository.html). We favored those with categorically-valued attributes, relatively wide record length, and a substantial number of records. These data-sets include connect-4, chess, splice, and mushroom. The splice data set contains DNA data, the mushroom database contains records describing the characteristics of various mushroom species, and Connect-4 and chess are compiled from game state information. Table 1 lists the width and height of each of these data-sets. Pumsb* is the same data set as pumsb minus all items with 80% or more support.

Table 1. Width and height of the evaluation data-sets.

Data-set	Records	Avg. Record Width
chess	3,196	37
connect-4	67,557	43
mushroom	8,124	23
pumsb	49,046	74
pumsb*	49,046	50
retail	213,972	31
splice	3,174	61

The data-set with the longest patterns was pumsb. This data-set also had the most maximal frequent patterns -- intractably many even at relatively high levels of support. We thus focus on pumsb* for evaluating Max-Miner, and use pumsb only to evaluate Max-Miner-LO. Even though items with 80% or more support are removed, pumsb* still a challenging data set with long frequent itemsets. A similar but smaller data-set was used by Brin et al. to evaluate DIC, though only at supports of 36% and higher. Their data-set was not available to us at the time of this writing, so we created pumsb* to use in its place.

All experiments were performed on a lightly loaded 200MHz Power-PC with 256 megabytes of RAM. The algorithms were implemented in C++ atop the same optimized hash tree implementation. Apriori and Apriori-LB were optimized for finding only maximal frequent itemsets by having them discard from memory any frequent itemset that was found to be non-maximal. The Max-Miner algorithm evaluated here uses support lower-bounding. In subsection 6.3 we describe the effects of disabling this and other optimizations.

Figure 8. CPU time on pumsb*.

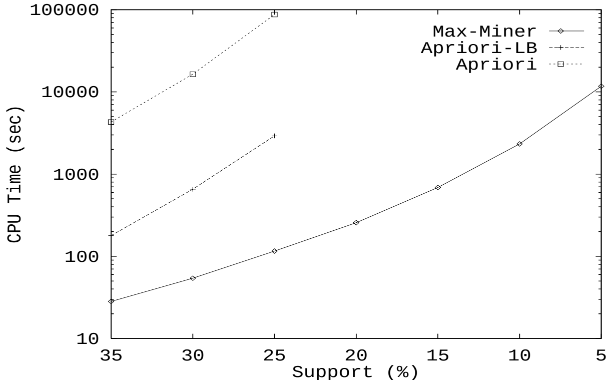


Figure 9. CPU time on mushroom.

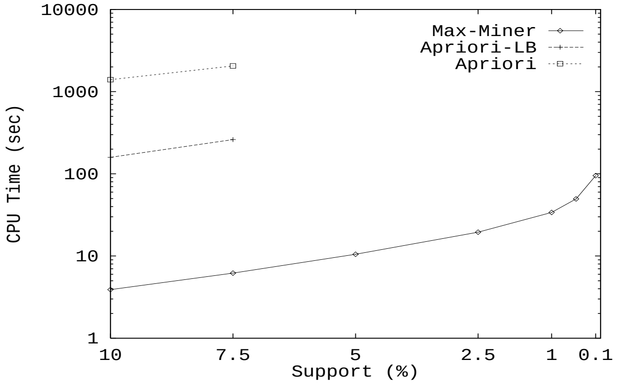


Figure 10. CPU time on chess.

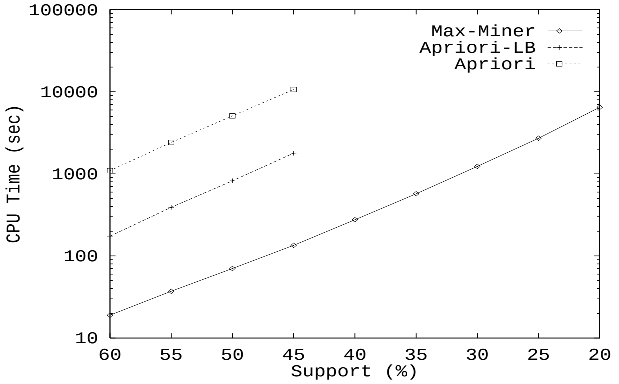
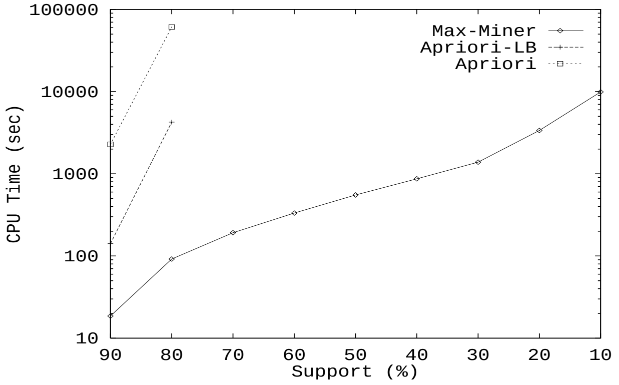


Figure 11. CPU time on connect-4.



6.1 Max-Miner versus Apriori and Apriori-LB

Figures 8 through 11 compare the performance of Max-Miner, Apriori-LB, and Apriori on the most difficult of the evaluation data-sets. While Apriori-LB is performing far better than Apriori with respect to run-time (note the logarithmically scaled y axes), because of its space complexity, we were unable to run it at lower support than Apriori without exceeding the memory of our machine. At several data points, Max-Miner is over two orders of magnitude faster than Apriori at identifying maximal frequent patterns, and an order of magnitude faster than Apriori-LB. Had we allowed Apriori to perform candidate paging, the speedups at lower supports would be even more substantial. Even though these data sets are from distinctly different domains, the performance trends are all identical. What these data-sets all have in common are long patterns at relatively high values of support.

Figure 12. CPU time on retail.

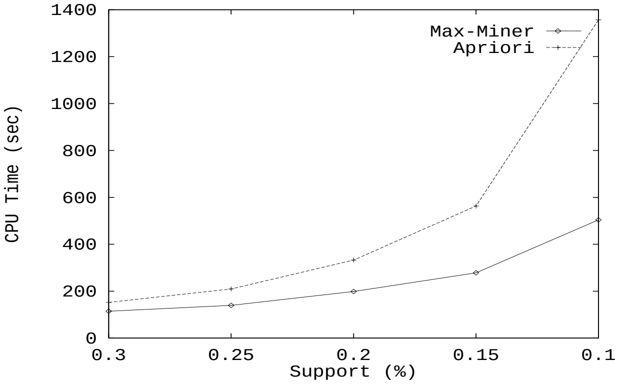
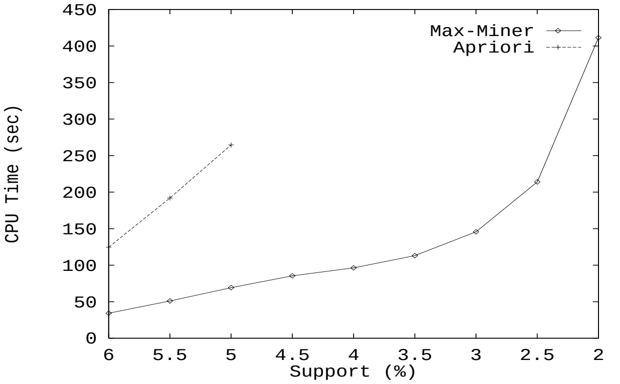
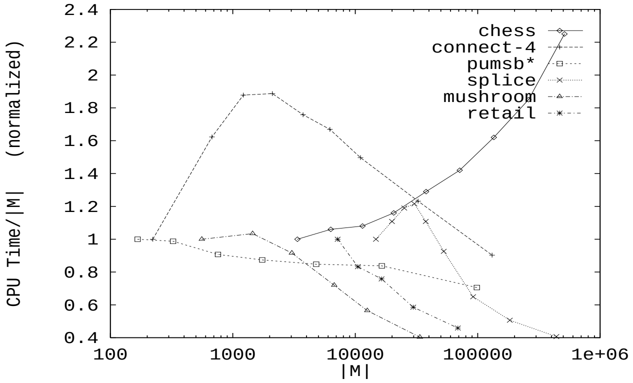


Figure 13. CPU time on splice.



The two remaining evaluation data-sets, retail and splice, have comparatively small patterns at low support levels. Even so, Max-Miner still outperforms both Apriori and Apriori-LB. We plot only the performance of Apriori on these data sets, since Apriori-LB offered no advantages. Apriori-LB appears to be more effective at tightly lower-bounding support when the patterns are long. Most of the maximal frequent itemsets in these data sets are of length 2 to 4, and the longest are of length 8 in splice and 6 in retail at the lowest support values. The superior performance of Max-Miner on these data-sets arises from considering fewer candidate itemsets and reduced index overhead resulting from the indexing of candidate groups using only head items. This suggests that Apriori should also index candidate itemsets that share a common prefix as a group rather than individually. We are currently investigating whether this optimization pays off more generally.

Figure 14. CPU time per maximal frequent itemset.

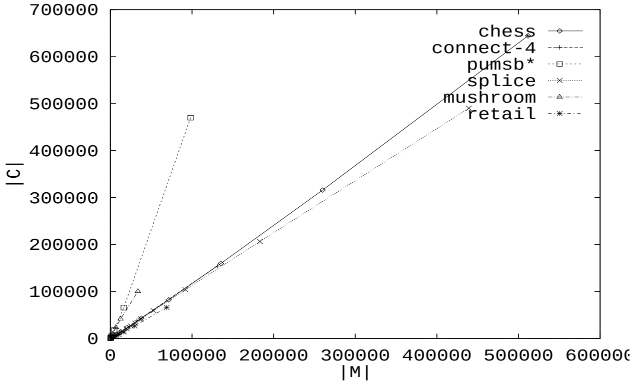


6.2 Max-Miner Scaling

For every data set at the support levels we looked at, the lower the support, the more maximal-frequent patterns were found. Figure 14 plots the amount of time spent by Max-Miner per maximal frequent itemset against the number of maximal frequent itemsets found during a given run of the algorithm. The support values used to generate this data are the same ones that appear in the previous graphs. Max-Miner's performance per maximal frequent itemset remains relatively constant as the number of maximal frequent itemsets increases, even though the size of the longest itemsets varies significantly. For instance, for the pumsb* data set, the left-most point arises from a run at 35% support where the longest frequent itemsets contain 15 items. The right-most point is for a run at 5% support where the longest frequent itemsets contain 40 items. Even so, the performance per maximal frequent itemset varies no more than 25%, indicating Max-Miner is scaling roughly linearly with the number of maximal frequent itemsets.

The chess data-set exhibits the most non-linear increase in difficulty with the number of maximal frequent itemsets, though this increase is still relatively subdued (within a factor of 2.5) considering the large increase in the number of maximal frequent itemsets and their length. It is possible that the curve for the chess data-set would begin to decrease had we mined at even lower levels of support. We were unable to determine this because of the explosion in the number of maximal frequent itemsets at low supports -- already there are over half a million maximal frequent itemsets at a minsup of 20%. The other data-sets all begin to get easier as the number of maximal frequent patterns increases beyond some point. This is certainly in part due to the fact that itemsets with lower support incur less overhead since they are contained by fewer database transactions. However, as support decreases, maximal frequent itemsets become longer, which leads to an increase in storage and indexing overhead.

Figure 15. Number of candidate groups considered per maximal frequent itemset.



To remove the effect of indexing and support-counting overhead on runtime scaling, and to demonstrate how Max-Miner scales in its space consumption, we also compared the number of candidate groups whose support was counted to the number of maximal frequent sets identified (Figure 15)[1]. For every data-set there is a strong linear relationship. In fact, the number of candidates considered is always within a small (at most 3.7 on pumsb*) constant of the number of maximal frequent itemsets.

Figure 16. Database passes performed by Max-Miner compared to the longest frequent itemset.

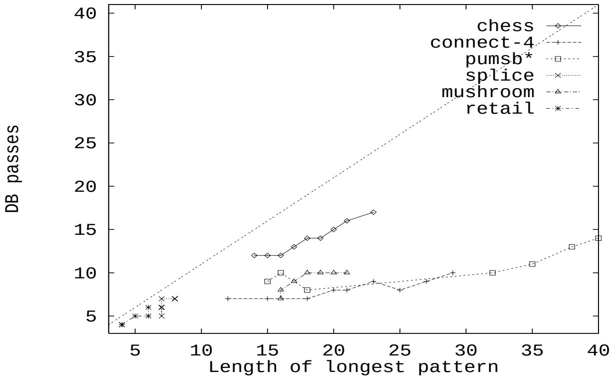


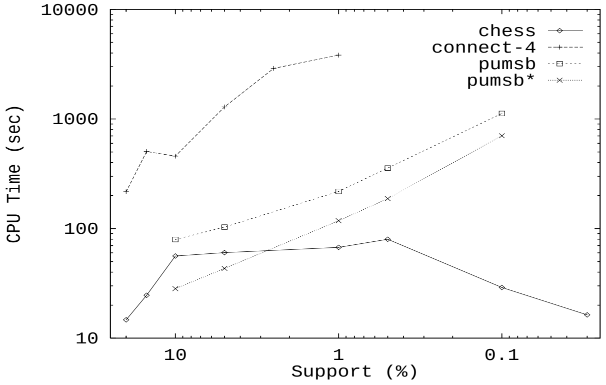
Figure 16 plots the number of database passes performed by Max-Miner against the length of the longest patterns identified during each run. The worst-case number of passes is represented by the diagonal. While the number of database passes increases with pattern length, it is usually far from the worst-case. Though not illustrated, Apriori-LB was often able to skip one or two database passes because all candidates were lower-bounded above minsup. This had a relatively small effect on performance compared to the amount of reduction in overhead from computing the support of fewer candidates. The reductions in database passes tended not to have as dramatic an effect on performance as might be expected from experiments on small-pattern data because the overhead of candidate itemset lookup largely exceeded that of data access.

6.3 Effects of Max-Miner Optimizations

Support lower-bounding is a beneficial optimization on the data-sets with long patterns for Max-Miner as well as Apriori. For example, after turning off support lower-bounding, Max-Miner's performance dropped by approximately four-fold at all levels of support on the chess data-set due to a four-fold increase in the number of candidate groups to be counted. On the retail data-set, the optimization's effects were negligible. The item-(re)ordering heuristic was beneficial, usually dramatically so, on every data-set we looked at. For example, at 80% support on the chess data set, turning off item ordering (which caused the algorithm to default to a static lexical item ordering) resulted in an order of magnitude decrease in performance, with the performance gap widening even further as support was decreased. The decrease in performance was due to an increase in candidate groups resulting primarily from less opportunities for superset-frequency pruning.

[1] The plot for the connect-4 data-set is difficult to make out because it lies directly along the lower-left portion of the plot for chess.

Figure 17. Performance of Max-Miner-LO



6.4 Max-Miner-LO

Figure 17 shows the runtime of Max-Miner-LO on the data sets with the longest frequent itemsets. We chose very low support values for these runs to illustrate how additional constraints can make mining feasible even when the number of maximal frequent itemsets is too large. Note that Max-Miner-LO can mine on the pumsb data-set without removing items with 80% or more support. The longest frequent itemsets in pumsb at .1% support contain 72 items. Frequent itemsets were very long even at 10% support where they contained 61 items. For most of these data points, the number of longest frequent itemsets was under 300.

6.5 Comparison with Pincer-Search

Because our work was done concurrently, we have thus far only had the opportunity to perform a preliminary comparison with Lin and Kedem's Pincer-Search algorithm [8]. We have run Max-Miner on the data-set from their evaluation with the longest frequent itemsets (T20.I15.D100K). Unfortunately, it was not challenging enough for either algorithm to draw any solid conclusions. The only clear case of one algorithm appearing superior over the other is at 8% minsup where Pincer-Search requires over 20,000 seconds and considers over 150,000 candidate itemsets. At this same support value Max-Miner requires 125 seconds and considers less than 22,000 candidate itemsets (within 4,894 candidate groups)[1].

Pincer-Search uses a candidate itemset generation procedure that couples Apriori candidate generation with another technique for generating long candidate itemsets used for superset-frequency pruning. The long candidate generation procedure is seeded with one itemset that contains every frequent item. For each infrequent itemset encountered after a database pass, any long itemset containing the infrequent itemset is replaced by itemsets that do not contain the infrequent itemset. The new itemsets are formed by removing a single item from the original, and keeping only those itemsets that are not subsets of any other long candidate itemset. The process is iterated until no long itemset contains any known infrequent itemset.

After the second database pass, the Pincer-Search long candidate itemset generation procedure amounts to identifying all the maximal cliques in a graph where the nodes are the frequent 1-itemsets and the edges are the frequent 2-itemsets, much as Zaki's MaxClique algorithm [16]. Unlike MaxClique, Pincer-Search uses a top-down instead of bottom-up approach for finding the cliques. But due to the fact that the problem is NP-hard, any approach at generating candidates in this manner may be prone to performance problems when the frequent itemsets (and hence the maximal cliques) are long.

Though similar in principle, Max-Miner and Pincer-Search are quite different in their details. We look forward to performing a more exhaustive comparison and feel it is likely that the algorithms will prove complementary.

7. Conclusions

We have presented and evaluated the Max-Miner algorithm for mining maximal frequent itemsets from large databases. Max-Miner applies several new techniques for reducing the space of itemsets considered through superset-frequency based pruning. The result is orders of magnitude in performance improvements over Apriori-like algorithms when frequent itemsets are long, and more modest though still substantial improvements when frequent itemsets are short. Max-Miner is also easily made to incorporate additional constraints on the set of frequent itemsets identified. Incorporating these constraints into the search is the only way to achieve tractable completeness at low supports on complex data-sets. It is therefore these extensions which we feel warrant the most future work. In particular, we hope there exists a clean way to exploit many of the wide variety of interestingness constraints during the search rather than applying them only in a post-processing filtering step.

Acknowledgments

I thank Ramakrishnan Srikant, Rakesh Agrawal, Sunita Sarawagi, and Dimitrios Gunopulos for many helpful suggestions and comments, Robert Schrag for directing me to the work of Ron Rymon, and Dao-I Lin for sharing details of the experiments used to evaluate Pincer-Search.

References

[1] Agrawal, R.; Imielinski, T.; and Swami, A. 1993. Mining Association Rules between Sets of Items in Large Databases. In *Proc. of the 1993 ACM-SIGMOD Conf. on Management of Data*, 207-216.

[2] Agrawal, R.; Mannila, H.; Srikant, R.; Toivonen, H.; and Verkamo, A. I. 1996. Fast Discovery of Association Rules. In *Advances in Knowledge Discovery and Data Mining*, AAAI Press, 307-328.

[3] Agrawal, R., and Srikant, R. 1994. *Fast Algorithms for Mining Association Rules*. IBM Research Report RJ9839, June 1994, IBM Almaden Research Center, San Jose, CA.

[4] Agrawal, R. and Srikant, R. 1995. Mining Sequential Patterns. In *Proc. of the 11th Int'l Conf. on Data Engineering*, 3-14.

[5] Bayardo, R. J. 1997. Brute-Force Mining of High-Confidence Classification Rules. In *Proc. of the Third Int'l Conf. on Knowledge Discovery and Data Mining*, 123-126.

[6] Brin, S.; Motwani, R.; Ullman, J.; and Tsur, S. 1997. Dynamic Itemset Counting and Implication Rules for Market Basket Data. In *Proc. of the 1997 ACM-SIGMOD Conf. on Management of Data*, 255-264.

[7] Gunopulos, G.; Mannila, H.; and Saluja, S. 1997. Discovering All Most Specific Sentences by Randomized Algorithms. In *Proc. of the 6th Int'l Conf. on Database Theory*, 215-229.

[8] Lin, D.-I and Kedem, Z. M. 1998. Pincer-Search: A New Algorithm for Discovering the Maximum Frequent Set. In *Proc. of the Sixth European Conf. on Extending Database Technology*, to appear.

[9] Park, J. S.; Chen, M.-S.; and Yu, P. S. 1996. An Effective Hash Based Algorithm for Mining Association Rules. In *Proc. of the 1995 ACM-SIGMOD Conf. on Management of Data*, 175-186.

[10] Rymon, R. 1992. Search through Systematic Set Enumeration. In *Proc. of Third Int'l Conf. on Principles of Knowledge Representation and Reasoning*, 539-550.

[11] Savasere, A.; Omiecinski, E.; and Navathe, S. 1995. An Efficient Algorithm for Mining Association Rules in Large Data-bases. In *Proc. of the 21st Conf. on Very Large Data-Bases*, 432-444.

[1] Candidate itemset counts for both Pincer-Search and Max-Miner do not include the 1 and 2-itemsets. Runtimes are not directly comparable due to differences in hardware and implementation details.

[12] Slagel, J. R.; Chang, C.-L.; and Lee, R. C. T. 1970. A New Algorithm for Generating Prime Implicants. *IEEE Trans. on Computers*, C-19(4):304-310.

[13] Smythe, P. and Goodman, R. M. 1992. An Information Theoretic Approach to Rule Induction from Databases. *IEEE Transactions on Knowledge and Data Engineering*, 4(4):301-316.

[14] Srikant, R. and Agrawal, R. 1996. Mining Sequential Patterns: Generalizations and Performance Improvements. In *Proc. of the Fifth Int'l Conf. on Extending Database Technology*, 3-17.

[15] Srikant, R.; Vu, Q.; and Agrawal, R. 1997. Mining Association Rules with Item Constraints. In *Proc. of the Third Int'l Conf. on Knowledge Discovery in Databases and Data Mining*, 67-73.

[16] Zaki, M. J.; Parthasarathy, S.; Ogihara, M.; and Li, W. 1997. New Algorithms for Fast Discovery of Association Rules. In *Proc. of the Third Int'l Conf. on Knowledge Discovery in Databases and Data Mining*, 283-286.