Mining Frequent Patterns without Candidate Generation *

Jiawei Han, Jian Pei, and Yiwen Yin
School of Computing Science
Simon Fraser University
{han, peijian, yiweny}@cs.sfu.ca

Abstract

Mining frequent patterns in transaction databases, time-series databases, and many other kinds of databases has been studied popularly in data mining research. Most of the previous studies adopt an Apriori-like candidate set generation-and-test approach. However, candidate set generation is still costly, especially when there exist prolific patterns and/or long patterns.

In this study, we propose a novel frequent pattern tree (FP-tree) structure, which is an extended prefix-tree structure for storing compressed, crucial information about frequent patterns, and develop an efficient FP-tree-based mining method, FP-growth, for mining *the complete set of frequent patterns* by pattern fragment growth. Efficiency of mining is achieved with three techniques: (1) a large database is compressed into a highly condensed, much smaller data structure, which avoids costly, repeated database scans, (2) our FP-tree-based mining adopts a pattern fragment growth method to avoid the costly generation of a large number of candidate sets, and (3) a partitioning-based, divide-and-conquer method is used to decompose the mining task into a set of smaller tasks for mining confined patterns in conditional databases, which dramatically reduces the search space. Our performance study shows that the FP-growth method is efficient and scalable for mining both long and short frequent patterns, and is about an order of magnitude faster than the Apriori algorithm and also faster than some recently reported new frequent pattern mining methods.

* The work was supported in part by the Natural Sciences and Engineering Research Council of Canada (grant NSERC-A3723), the Networks of Centres of Excellence of Canada (grant NCE/IRIS-3), and the Hewlett-Packard Lab, U.S.A.

1 Introduction

Frequent pattern mining plays an essential role in mining associations [3, 12], correlations [6], causality [19], sequential patterns [4], episodes [14], multi-dimensional patterns [13, 11], max-patterns [5], partial periodicity [9], emerging patterns [7], and many other important data mining tasks.

Most of the previous studies, such as [3, 12, 18, 16, 13, 17, 20, 15, 8], adopt an Apriori-like approach, which is based on an *anti-monotone* Apriori *heuristic* [3]: *if any length k pattern is not frequent in the database, its length $(k+1)$ super-pattern can never be frequent.* The essential idea is to iteratively generate the set of candidate patterns of length $(k+1)$ from the set of frequent patterns of length k (for $k \geq 1$), and check their corresponding occurrence frequencies in the database.

The Apriori heuristic achieves good performance gain by (possibly significantly) reducing the size of candidate sets. However, in situations with prolific frequent patterns, long patterns, or quite low minimum support thresholds, an Apriori-like algorithm may still suffer from the following two nontrivial costs:

- It is costly to handle a huge number of candidate sets. For example, if there are 10^4 frequent 1-itemsets, the Apriori algorithm will need to generate more than 10^7 length-2 candidates and accumulate and test their occurrence frequencies. Moreover, to discover a frequent pattern of size 100, such as $\{a_1, \ldots, a_{100}\}$, it must generate more than $2^{100} \approx 10^{30}$ candidates in total. This is the inherent cost of candidate generation, no matter what implementation technique is applied.
- It is tedious to repeatedly scan the database and check a large set of candidates by pattern matching, which is especially true for mining long patterns.

Is there any other way that one may reduce these costs in frequent pattern mining? May some novel data structure or algorithm help?

After some careful examination, we believe that the bottleneck of the Apriori-like method is at the *candidate set generation and test*. If one can avoid generating a huge set of candidates, the mining performance can be substantially improved.

This problem is attacked in the following three aspects.

First, a novel, compact data structure, called *frequent pattern tree*, or FP-tree for short, is constructed, which is an extended prefix-tree structure storing crucial, quantitative information about frequent patterns. Only frequent length-1 items will have nodes in the tree, and the tree nodes are arranged in such a way that more frequently occurring nodes will have better chances of sharing nodes than less frequently occurring ones.

Second, an FP-tree-based pattern fragment growth mining method, is developed, which starts from a frequent length-1 pattern (as an initial *suffix pattern*), examines only its *conditional pattern base* (a "sub-database" which consists of the set of frequent items co-occurring with the suffix pattern), constructs its (*conditional*) FP-tree, and performs mining recursively with such a tree. The pattern growth is achieved via concatenation of the suffix pattern with the new ones generated from a conditional FP-tree. Since the frequent itemset in any transaction is always encoded in the corresponding path of the frequent pattern trees, pattern growth ensures the completeness of the result. In this context, our method is not Apriori-like *restricted generation-and-test* but *restricted test only*. The major operations of mining are count accumulation and prefix path count adjustment, which are usually much less costly than candidate generation and pattern matching operations performed in most Apriori-like algorithms.

Third, the search technique employed in mining is a *partitioning-based, divide-and-conquer method* rather than Apriori-like *bottom-up generation of frequent itemsets combinations*. This dramatically reduces the size of *conditional pattern base* generated at the subsequent level of search as well as the size of its corresponding *conditional* FP-tree. Moreover, it transforms the problem of finding long frequent patterns to looking for shorter ones and then concatenating the suffix. It employs the least frequent items as suffix, which offers good selectivity. All these techniques contribute to substantial reduction of search costs.

A performance study has been conducted to compare the performance of FP-growth with Apriori and TreeProjection, where TreeProjection is a recently proposed efficient algorithm for frequent pattern mining [2]. Our study shows that FP-growth is at least an order of magnitude faster than Apriori, and such a margin grows even wider when the frequent patterns grow longer, and FP-growth also outperforms the TreeProjection algorithm. Our FP-tree-based mining method has also been tested in large transaction databases in industrial applications.

The remaining of the paper is organized as follows. Section 2 introduces the FP-tree structure and its construction method. Section 3 develops an FP-tree-based frequent pattern mining algorithm, FP-growth . Section 4 presents our performance study. Section 5 discusses the issues on scalability and improvements of the method. Section 6 summarizes our study and points out some future research issues.

2 Frequent Pattern Tree: Design and Construction

Let $I = \{a_1, a_2, \ldots, a_m\}$ be a **set of items**, and a **transaction database** $DB = \langle T_1, T_2, \ldots, T_n \rangle$, where T_i ($i \in [1..n]$) is a transaction which contains a set of items in I. The **support**[1] (or occurrence frequency) of a **pattern** A, which is a set of items, is the number of transactions containing A in DB. A, is a **frequent pattern** if A's support is no less than a predefined *minimum support threshold*, ξ.

Given a transaction database DB and a minimum support threshold, ξ, the problem of *finding the complete set of frequent patterns* is called the frequent pattern mining problem.

2.1 Frequent Pattern Tree

To design a compact data structure for efficient frequent pattern mining, let's first examine an example.

Example 1 Let the transaction database, DB, be (the first two columns of) Table 1 and $\xi = 3$.

A compact data structure can be designed based on the following observations.

1. Since only the frequent items will play a role in the frequent pattern mining, it is necessary to perform one scan of DB to identify the set of frequent items (with *frequency count* obtained as a by-product).

[1]Notice that *support* is defined here as *absolute* occurrence frequency, not the *relative* one as in some literature.

2. If we store the *set* of frequent items of each transaction in some compact structure, it may avoid repeatedly scanning of DB.

3. If multiple transactions share an identical frequent item set, they can be merged into one with the number of occurrences registered as *count*. It is easy to check whether two sets are identical if the frequent items in all of the transactions are sorted according to a fixed order.

4. If two transactions share a common prefix, according to some sorted order of frequent items, the shared parts can be merged using one prefix structure as long as the *count* is registered properly. If the frequent items are sorted in their *frequency descending order*, there are better chances that more prefix strings can be shared.

TID	Items Bought	(Ordered) Frequent Items
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	c, b, p
500	a, f, c, e, l, p, m, n	f, c, a, m, p

Table 1: A transaction database as running example.

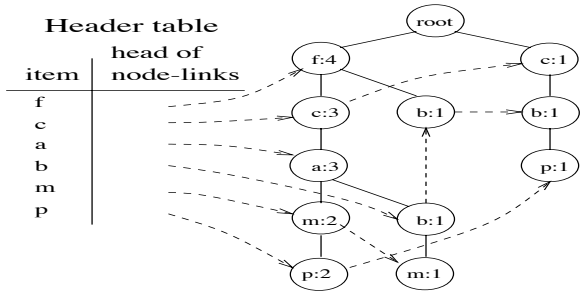

Figure 1: The FP-tree in Example 1.

With these observations, one may construct a frequent pattern tree as follows.

First, a scan of DB derives a *list* of frequent items, $\langle (f{:}4), (c{:}4), (a{:}3), (b{:}3), (m{:}3), (p{:}3) \rangle$, (the number after ":" indicates the support), in which items ordered in frequency descending order. This ordering is important since each path of a tree will follow this order. For convenience of later discussions, the frequent items in each transaction are listed in this ordering in the rightmost column of Table 1.

Second, one may create the root of a tree, labeled with "*null*". Scan the DB the second time. The scan of the first transaction leads to the construction of the first branch of the tree: $\langle (f{:}1), (c{:}1), (a{:}1), (m{:}1), (p{:}1) \rangle$. Notice that the frequent items in the transaction is ordered according to the order in the *list* of frequent items. For the second transaction, since its (ordered) frequent item list $\langle f, c, a, b, m \rangle$ shares a common prefix $\langle f, c, a \rangle$ with the existing path $\langle f, c, a, m, p \rangle$, the count of each node along the prefix is incremented by 1, and one new node $(b{:}1)$ is created and linked as a child of $(a{:}2)$ and another new node $(m{:}1)$ is created and linked as the child of $(b{:}1)$. For the third transaction, since its frequent item list $\langle f, b \rangle$ shares only the node $\langle f \rangle$ with the f-prefix subtree, f's count is incremented by 1, and a new node $(b{:}1)$ is created and linked as a child of $(f{:}3)$. The scan of the fourth transaction leads to the construction of the second branch of the tree, $\langle (c{:}1), (b{:}1), (p{:}1) \rangle$. For the last transaction, since its frequent item list $\langle f, c, a, m, p \rangle$ is identical to the first one, the path is shared with the count of each node along the path incremented by 1.

To facilitate tree traversal, an item header table is built in which each item points to its occurrence in the tree via a head of node-link. Nodes with the same item-name are linked in sequence via such node-links. After scanning all the transactions, the tree with the associated node-links is shown in Figure 1. □

This example leads to the following design and construction of a *frequent pattern tree*.

Definition 1 (FP-tree) **A frequent pattern tree** (or FP-tree in short) is a tree structure defined below.

1. It consists of one root labeled as "*null*", a set of item prefix subtrees as the children of the root, and a frequent-item header table.

2. Each node in the item prefix subtree consists of three fields: *item-name*, *count*, and *node-link*, where *item-name* registers which item this node represents, *count* registers the number of transactions represented by the portion of the path reaching this node, and *node-link* links to the next node in the FP-tree carrying the same item-name, or null if there is none.

3. Each entry in the frequent-item header table consists of two fields, (1) *item-name* and (2) *head of node-link*, which points to the first node in the FP-tree carrying the *item-name*. □

Based on this definition, we have the following FP-tree construction algorithm.

Algorithm 1 (FP-tree construction)

Input: A transaction database DB and a minimum support threshold ξ.

Output: Its frequent pattern tree, FP-tree

Method: The FP-tree is constructed in the following steps.

1. Scan the transaction database DB once. Collect the set of frequent items F and their supports. Sort F in support descending order as L, the *list* of frequent items.

2. Create the root of an FP-tree, T, and label it as "null". For each transaction $Trans$ in DB do the following.

   Select and sort the frequent items in $Trans$ according to the order of L. Let the sorted frequent item list in $Trans$ be $[p|P]$, where p is the first element and P is the remaining list. Call $insert\_tree([p|P], T)$.

   The function $insert\_tree([p|P], T)$ is performed as follows. If T has a child N such that $N.item\text{-}name = p.item\text{-}name$, then increment N's count by 1; else create a new node N, and let its count be 1, its parent link be linked to T, and its node-link be linked to the nodes with the same *item-name* via the node-link structure. If P is nonempty, call $insert\_tree(P, N)$ recursively.

Analysis. From the FP-tree construction process, we can see that one needs exactly two scans of the transaction database, DB: the first collects the set of frequent items, and the second constructs the FP-tree. The cost of inserting a transaction $Trans$ into the FP-tree is $O(|Trans|)$, where $|Trans|$ is the number of frequent items in $Trans$. We will show that the FP-tree contains the complete information for frequent pattern mining. □

2.2 Completeness and Compactness of FP-tree

Several important properties of FP-tree can be observed from the FP-tree construction process.

Lemma 2.1 *Given a transaction database DB and a support threshold ξ, its corresponding* FP-tree *contains the complete information of DB in relevance to frequent pattern mining.*

Rationale. Based on the FP-tree construction process, each transaction in the DB is mapped to one path in the FP-tree, and the frequent itemset information in each transaction is completely stored in the FP-tree. Moreover, one path in the FP-tree may represent frequent itemsets in multiple transactions without ambiguity since the path representing every transaction must start from the root of each item prefix subtree. Thus we have the lemma. □

Lemma 2.2 *Without considering the (null) root, the size of an* FP-tree *is bounded by the overall occurrences of the frequent items in the database, and the height of the tree is bounded by the maximal number of frequent items in any transaction in the database.*

Rationale. Based on the FP-tree construction process, for any transaction T in DB, there exists a path in the FP-tree starting from the corresponding item prefix subtree so that the set of nodes in the path is exactly the same set of frequent items in T. Since no frequent item in any transaction can create more than one node in the tree, the root is the only extra node created not by frequent item insertion, and each node contains one node-link and one count information, we have the bound of the size of the tree stated in the Lemma. The height of any p-prefix subtree is the maximum number of frequent items in any transaction with p appearing at the head of its frequent item list. Therefore, the height of the tree is bounded by the maximal number of frequent items in any transaction in the database, if we do not consider the additional level added by the root. □

Lemma 2.2 shows an important benefit of FP-tree: the size of an FP-tree is bounded by the size of its corresponding database because each transaction will contribute at most one path to the FP-tree, with the length equal to the number of frequent items in that transaction. Since there are often a lot of sharing of frequent items among transactions, the size of the tree is usually much smaller than its original database. Unlike the Apriori-like method which may generate an exponential number of candidates in the worst case, under no circumstances, may an FP-tree with an exponential number of nodes be generated.

FP-tree is a highly compact structure which stores the information for frequent pattern mining. Since a single path "$a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_n$" in the a_1-prefix subtree registers all the transactions whose maximal frequent set is in the form of "$a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_k$" for any $1 \leq k \leq n$, the size of the FP-tree is substantially smaller than the size of the database and that of the candidate sets generated in the association rule mining.

The items in the frequent item set are ordered in the support-descending order: More frequently occurring

items are arranged closer to the top of the FP-tree and thus are more likely to be shared. This indicates that FP-tree structure is usually highly compact. Our experiments also show that a small FP-trees is resulted by compressing some quite large database. For example, for the database *Connect-4* used in MaxMiner [5], which contains 67,557 transactions with 43 items in each transaction, when the support threshold is 50% (which is used in the MaxMiner experiments [5]), the total number of occurrences of frequent items is 2,219,609, whereas the total number of nodes in the FP-tree is 13,449 which represents a reduction ratio of 165.04, while it withholds hundreds of thousands of frequent patterns! (Notice that for databases with mostly short transactions, the reduction ratio is not that high.)

Nevertheless, one cannot assume that an FP-tree can always fit in main memory for any large databases. Methods for highly scalable FP-growth mining will be discussed in Section 5.

3 Mining Frequent Patterns using FP-tree

Construction of a compact FP-tree ensures that subsequent mining can be performed with a rather compact data structure. However, this does not automatically guarantee that it will be highly efficient since one may still encounter the combinatorial problem of candidate generation if we simply use this FP-tree to generate and check all the candidate patterns.

In this section, we will study how to explore the compact information stored in an FP-tree and develop an efficient mining method for mining *the complete set of frequent patterns* (also called *all patterns*).

We observe some interesting properties of the FP-tree structure which will facilitate frequent pattern mining.

Property 3.1 (Node-link property) *For any frequent item a_i, all the possible frequent patterns that contain a_i can be obtained by following a_i's node-links, starting from a_i's head in the* FP-tree *header.*

This property is based directly on the construction process of FP-tree. It facilitates the access of all the pattern information related to a_i by traversing the FP-tree once following a_i's node-links.

Example 2 Let us examine the mining process based on the constructed FP-tree shown in Figure 1. Based on Property 3.1, we collect all the patterns that a node a_i participates by starting from a_i's head (in the header table) and following a_i's node-links. We examine the mining process by starting from the bottom of the header table.

For node p, it derives a frequent pattern $(p{:}3)$ and two paths in the FP-tree : $\langle f{:}4, c{:}3, a{:}3, m{:}2, p{:}2\rangle$ and $\langle c{:}1, b{:}1, p{:}1\rangle$. The first path indicates that string "(f, c, a, m, p)" appears twice in the database. Notice although string $\langle f, c, a\rangle$ appears three times and $\langle f\rangle$ itself appears even four times, they only appear twice together with p. Thus to study which string appear together with p, only p's prefix path $\langle f{:}2, c{:}2, a{:}2, m{:}2\rangle$ counts. Similarly, the second path indicates string "(c, b, p)" appears once in the set of transactions in DB, or p's prefix path is $\langle c{:}1, b{:}1\rangle$. These two prefix paths of p, "$\{(f{:}2, c{:}2, a{:}2, m{:}2), (c{:}1, b{:}1)\}$", form p's sub-pattern base, which is called p's conditional pattern base (i.e., the sub-pattern base under the condition of p's existence). Construction of an FP-tree on this conditional pattern base (which is called p's conditional FP-tree) leads to only one branch $(c{:}3)$. Hence only one frequent pattern $(cp{:}3)$ is derived. (Notice that a pattern is an itemset and is denoted by a string here.) The search for frequent patterns associated with p terminates.

For node m, it derives a frequent pattern $(m{:}3)$ and two paths $\langle f{:}4, c{:}3, a{:}3, m{:}2\rangle$ and $\langle f{:}4, c{:}3, a{:}3, b{:}1, m{:}1\rangle$. Notice p appears together with m as well, however, there is no need to include p here in the analysis since any frequent patterns involving p has been analyzed in the previous examination of p. Similar to the above analysis, m's conditional pattern base is, $\{(f{:}2, c{:}2, a{:}2), (f{:}1, c{:}1, a{:}1, b{:}1)\}$. Constructing an FP-tree on it, we derive m's conditional FP-tree , $\langle f{:}3, c{:}3, a{:}3\rangle$, a single frequent pattern path. Then one can call FP-tree-based mining recursively, i.e., call $mine(\langle f{:}3, c{:}3, a{:}3\rangle|m)$.

Figure 2 shows "$mine(\langle f{:}3, c{:}3, a{:}3\rangle|m)$" involves mining three items (a), (c), (f) in sequence. The first derives a frequent pattern $(am{:}3)$, and a call "$mine(\langle f{:}3, c{:}3\rangle|am)$"; the second derives a frequent pattern $(cm{:}3)$, and a call "$mine(\langle f{:}3\rangle|cm)$"; and the third derives only a frequent pattern $(fm{:}3)$. Further recursive call of "$mine(\langle f{:}3, c{:}3\rangle|am)$" derives $(cam{:}3)$, $(fam{:}3)$, and a call "$mine(\langle f{:}3\rangle|cam)$", which derives the longest pattern $(fcam{:}3)$. Similarly, the call of "$mine(\langle f{:}3\rangle|cm)$", derives one pattern $(fcm{:}3)$. Therefore, the whole set of frequent patterns involving m is $\{(m{:}3), (am{:}3), (cm{:}3), (fm{:}3), (cam{:}3), (fam{:}3), (fcam{:}3), (fcm{:}3)\}$. This indicates *a single path* FP-tree *can be mined by outputting all the combinations of the items in the path.*

Similarly, node b derives $(b{:}3)$ and three paths: $\langle f{:}4, c{:}3, a{:}3, b{:}1\rangle$, $\langle f{:}4, b{:}1\rangle$, and $\langle c{:}1, b{:}1\rangle$. Since b's

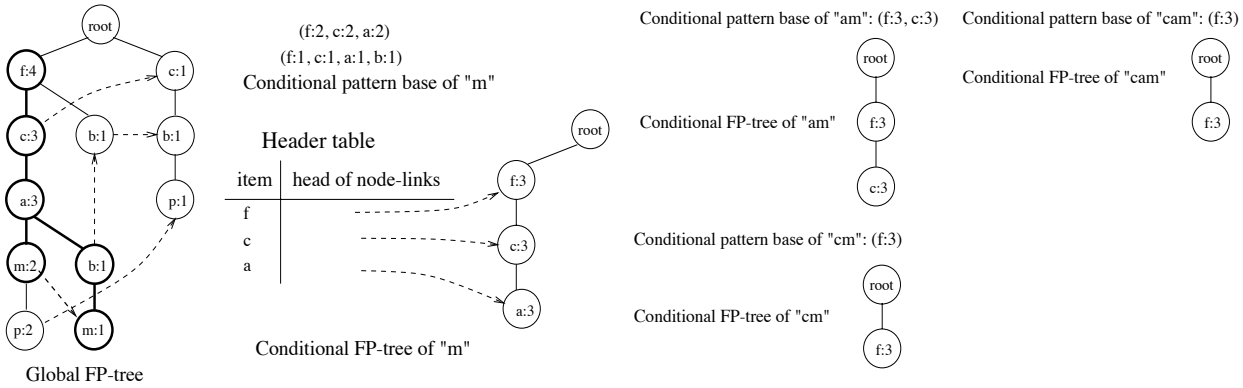

Figure 2: A conditional FP-tree built for m, i.e, "FP-tree | m"

conditional pattern base: $\{(f{:}1, c{:}1, a{:}1), (f{:}1), (c{:}1)\}$ generates no frequent item, the mining terminates. Node a derives one frequent pattern $\{(a{:}3)\}$, and one subpattern base, $\{(f{:}3, c{:}3)\}$, a single path conditional FP-tree. Thus, its set of frequent patterns can be generated by taking their combinations. Concatenating them with $(a{:}3)$, we have $\{(fa{:}3), (ca{:}3), (fca{:}3)\}$. Node c derives $(c{:}4)$ and one subpattern base, $\{(f{:}3)\}$, and the set of frequent patterns associated with $(c{:}3)$ is $\{(fc{:}3)\}$. Node f derives only $(f{:}4)$ but no conditional pattern base.

item	conditional pattern base	conditional FP-tree
p	$\{(f{:}2, c{:}2, a{:}2, m{:}2), (c{:}1, b{:}1)\}$	$\{(c{:}3)\}\|p$
m	$\{(f{:}4, c{:}3, a{:}3, m{:}2), (f{:}4, c{:}3, a{:}3, b{:}1, m{:}1)\}$	$\{(f{:}3, c{:}3, a{:}3)\}\|m$
b	$\{(f{:}4, c{:}3, a{:}3, b{:}1), (f{:}4, b{:}1), (c{:}1, b{:}1)\}$	$\emptyset$
a	$\{(f{:}3, c{:}3)\}$	$\{(f{:}3, c{:}3)\}\|a$
c	$\{(f{:}3)\}$	$\{(f{:}3)\}\|c$
f	$\emptyset$	$\emptyset$

Table 2: Mining of all-patterns by creating conditional (sub)-pattern bases

The conditional pattern bases and the conditional FP-trees generated are summarized in Table 2. □

The correctness and completeness of the process in Example 2 should be justified. We will present a few important properties related to the mining process.

Property 3.2 (Prefix path property) *To calculate the frequent patterns for a node a_i in a path P, only the* prefix subpath *of node a_i in P need to be accumulated, and the frequency count of every node in the prefix path should carry the same count as node a_i.*

Rationale. Let the nodes along the path P be labeled as $a_1, \ldots, a_n$ in such an order that a_1 is the root of the prefix subtree, a_n is the leaf of the subtree in P, and a_i $(1 \leq i \leq n)$ is the node being referenced. Based on the process of construction of FP-tree presented in Algorithm 1, for each prefix node a_k $(1 \leq k < i)$, the prefix subpath of the node a_i in P occurs together with a_k exactly $a_i.count$ times. Thus every such prefix node should carry the same count as node a_i. Notice that a postfix node a_m (for $i < m \leq n$) along the same path also co-occurs with node a_i. However, the patterns with a_m will be generated at the examination of the postfix node a_m, enclosing them here will lead to redundant generation of the patterns that would have been generated for a_m. Therefore, we only need to examine the prefix subpath of a_i in P. □

For example, in Example 2, node m is involved in a path $\langle f{:}4, c{:}3, a{:}3, m{:}2, p{:}2\rangle$, to calculate the frequent patterns for node m in this path, only the prefix subpath of node m, which is $\langle f{:}4, c{:}3, a{:}3\rangle$, need to be extracted, and the frequency count of every node in the prefix path should carry the same count as node m. That is, the node counts in the prefix path should be adjusted to $\langle f{:}2, c{:}2, a{:}2\rangle$.

Based on this property, the prefix subpath of node a_i in a path P can be copied and transformed into a count-adjusted prefix subpath by adjusting the frequency count of every node in the prefix subpath to the same as the count of node a_i. The so transformed prefix path is called the **transformed prefixed path** of a_i for path P.

Notice that the set of transformed prefix paths of a_i form a small database of patterns which co-occur with a_i. Such a database of patterns occurring with a_i is called a_i**'s conditional pattern base**, and is denoted as "*pattern_base* | a_i". Then one can compute all the frequent patterns associated with

a_i in this a_i-conditional pattern_base by creating a small FP-tree, called a_i's **conditional** FP-tree and denoted as "FP-tree | a_i". Subsequent mining can be performed on this small, conditional FP-tree. The processes of construction of conditional pattern bases and conditional FP-trees have been demonstrated in Example 2.

This process is performed recursively, and the frequent patterns can be obtained by a pattern growth method, based on the following lemmas and corollary.

Lemma 3.1 (Fragment growth) *Let α be an itemset in DB, B be α's conditional pattern base, and β be an itemset in B. Then the support of $\alpha \cup \beta$ in DB is equivalent to the support of β in B.*

Rationale. According to the definition of conditional pattern base, each (sub)transaction in B occurs under the condition of the occurrence of α in the original transaction database DB. If an itemset β appears in B ψ times, it appears with α in DB ψ times as well. Moreover, since all such items are collected in the conditional pattern base of α, $\alpha \cup \beta$ occurs exactly ψ times in DB as well. Thus we have the lemma. □

From this lemma, we can easily derive an important corollary.

Corollary 3.1 (Pattern growth) *Let α be a frequent itemset in DB, B be α's conditional pattern base, and β be an itemset in B. Then $\alpha \cup \beta$ is frequent in DB if and only if β is frequent in B.*

Rationale. This corollary is the case when α is a frequent itemset in DB, and when the support of β in α's conditional pattern base B is no less than ξ, the minimum support threshold. □

Based on Corollary 3.1, mining can be performed by first identifying the frequent 1-itemset, α, in DB, constructing their conditional pattern bases, and then mining the 1-itemset, β, in these conditional pattern bases, and so on. This indicates that the process of mining frequent patterns can be viewed as first mining frequent 1-itemset and then progressively growing each such itemset by mining its conditional pattern base, which can in turn be done similarly. Thus we successfully transform a frequent k-itemset mining problem into a sequence of k frequent 1-itemset mining problems via a set of conditional pattern bases. What we need is just pattern growth. There is no need to generate any combinations of candidate sets in the entire mining process.

Finally, we provide the property on mining all the patterns when the FP-tree contains only a single path.

Lemma 3.2 (Single FP-tree path pattern generation) *Suppose an FP-tree T has a single path P. The complete set of the frequent patterns of T can be generated by the enumeration of all the combinations of the subpaths of P with the support being the minimum support of the items contained in the subpath.*

Rationale. Let the single path P of the FP-tree be $\langle a_1{:}s_1 \rightarrow a_2{:}s_2 \rightarrow \cdots \rightarrow a_k{:}s_k \rangle$. The support frequency s_i of each item a_i (for $1 \leq i \leq k$) is the frequency of a_i co-occurring with its prefix string. Thus any combination of the items in the path, such as $\langle a_i, \cdots, a_j \rangle$ (for $1 \leq i, j \leq k$), is a frequent pattern, with their co-occurrence frequency being the minimum support among those items. Since every item in each path P is unique, there is no redundant pattern to be generated with such a combinational generation. Moreover, no frequent patterns can be generated outside the FP-tree. Therefore, we have the lemma. □

Based on the above lemmas and properties, we have the following algorithm for mining frequent patterns using FP-tree.

Algorithm 2 (FP-growth: Mining frequent patterns with FP-tree by pattern fragment growth)

Input: FP-tree constructed based on Algorithm 1, using DB and a minimum support threshold ξ.

Output: The complete set of frequent patterns.

Method: Call FP-growth (FP-tree , *null*).

Procedure FP-growth ($Tree, \alpha$)
{
(1) **if** $Tree$ contains a single path P
(2) **then for each** combination (denoted as β) of the nodes in the path P **do**
(3) generate pattern $\beta \cup \alpha$ with *support* = *minimum support of nodes in* β;
(4) **else for each** a_i in the header of $Tree$ **do** {
(5) generate pattern $\beta = a_i \cup \alpha$ with $support = a_i.support$;
(6) construct β's conditional pattern base and then β's conditional FP-tree $Tree_\beta$;
(7) **if** $Tree_\beta \neq \emptyset$
(8) **then** call FP-growth ($Tree_\beta, \beta$) }
}

Analysis. With the properties and lemmas in Sections 2 and 3, we show that the algorithm correctly finds the complete set of frequent itemsets in transaction database DB.

As shown in Lemma 2.1, FP-tree of DB contains the complete information of DB in relevance to frequent pattern mining under the support threshold ξ.

If an FP-tree contains a single path, according to Lemma 3.2, its generated patterns are the combinations of the nodes in the path, with the support being the minimum support of the nodes in the subpath. Thus we have lines (1)-(3) of the procedure. Otherwise, we construct conditional pattern base and mine its conditional FP-tree for each frequent itemset a_i. The correctness and completeness of prefix path transformation are shown in Property 3.2, and thus the conditional pattern bases store the complete information for frequent pattern mining. According to Lemmas 3.1 and its corollary, the patterns successively grown from the conditional FP-trees are the set of sound and complete frequent patterns. Especially, according to the fragment growth property, the support of the combined fragments takes the support of the frequent itemsets generated in the conditional pattern base. Therefore, we have lines (4)-(8) of the procedure. □

Let's now examine the efficiency of the algorithm. The FP-growth mining process scans the FP-tree of DB once and generates a small pattern-base B_{a_i} for each frequent item a_i, each consisting of the set of transformed prefix paths of a_i. Frequent pattern mining is then recursively performed on the small pattern-base B_{a_i} by constructing a conditional FP-tree for B_{a_i}. As reasoned in the analysis of Algorithm 1, an FP-tree is usually much smaller than the size of DB. Similarly, since the conditional FP-tree, "FP-tree $\mid a_i$", is constructed on the pattern-base B_{a_i}, it should be usually much smaller and never bigger than B_{a_i}. Moreover, a pattern-base B_{a_i} is usually much smaller than its original FP-tree, because it consists of the transformed prefix paths related to only one of the frequent items, a_i. Thus, each subsequent mining process works on a set of usually much smaller pattern bases and conditional FP-trees. Moreover, the mining operations consists of mainly prefix count adjustment, counting, and pattern fragment concatenation. This is much less costly than generation and test of a very large number of candidate patterns. Thus the algorithm is efficient.

From the algorithm and its reasoning, one can see that the FP-growth mining process is a divide-and-conquer process, and the scale of shrinking is usually quite dramatic. If the shrinking factor is around 20∼100 for constructing an FP-tree from a database, it is expected to be another hundreds of times reduction for constructing each conditional FP-tree from its already quite small conditional frequent pattern base.

Notice that even in the case that a database may generate an exponential number of frequent patterns, the size of the FP-tree is usually quite small and will never grow exponentially. For example, for a frequent pattern of length 100, "$a_1, \ldots, a_{100}$", the FP-tree construction results in only one path of length 100 for it, such as "$\langle a_1, \rightarrow \cdots \rightarrow a_{100} \rangle$". The FP-growth algorithm will still generate about 10^{30} frequent patterns (if time permits!!), such as "a_1, a_2, ..., a_1a_2, ..., $a_1a_2a_3$, ..., $a_1 \ldots a_{100}$". However, the FP-tree contains only one frequent pattern path of 100 nodes, and according to Lemma 3.2, there is even no need to construct any conditional FP-tree in order to find all the patterns.

4 Experimental Evaluation and Performance Study

In this section, we present a performance comparison of FP-growth with the classical frequent pattern mining algorithm Apriori, and a recently proposed efficient method TreeProjection.

All the experiments are performed on a 450-MHz Pentium PC machine with 128 megabytes main memory, running on Microsoft Windows/NT. All the programs are written in Microsoft/Visual C++6.0. Notice that we do not directly compare our absolute number of runtime with those in some published reports running on the RISC workstations because different machine architectures may differ greatly on the absolute runtime for the same algorithms. Instead, we implement their algorithms to the best of our knowledge based on the published reports on the same machine and compare in the same running environment. Please also note that *run time* used here means the total execution time, i.e., the period between input and output, instead of *CPU time* measured in the experiments in some literature. Also, all reports on the runtime of FP-growth include the time of constructing FP-trees from the original databases.

The synthetic data sets which we used for our experiments were generated using the procedure described in [3].

We report experimental results on two data sets. The first one is T25.I10.D10K with 1K items, which is denoted as $\mathcal{D}_1$. In this data set, the average transaction size and average maximal potentially frequent itemset size are set to 25 and 10, respectively, while the number of transactions in the dataset is set to 10K. The second data set, denoted as $\mathcal{D}_2$, is T25.I20.D100K with 10K items. There are

exponentially numerous frequent itemsets in both data sets, as the support threshold goes down. There are pretty long frequent itemsets as well as a large number of short frequent itemsets in them. They contain abundant mixtures of short and long frequent itemsets.

4.1 Comparison of FP-growth and Apriori

The scalability of FP-growth and Apriori as the support threshold decreases from 3% to 0.1% is shown in Figure 3.

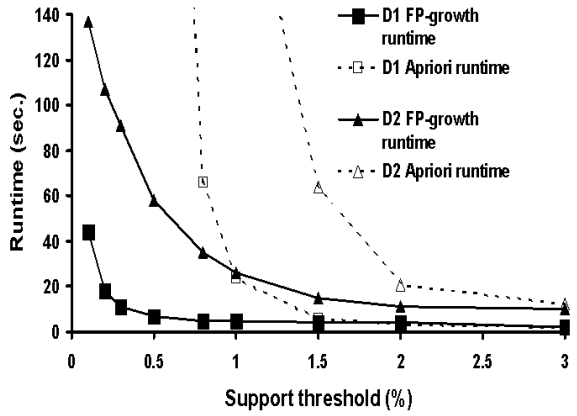

Figure 3: Scalability with threshold.

FP-growth scales much better than Apriori. This is because as the support threshold goes down, the number as well as the length of frequent itemsets increase dramatically. The candidate sets that Apriori must handle becomes extremely large, and the pattern matching with a lot of candidates by searching through the transactions becomes very expensive.

Figure 4 shows that the run time per itemset of FP-growth. It shows that FP-growth has good scalability with the reduction of minimum support threshold. Although the number of frequent itemsets grows exponentially, the run time of FP-growth increases in a much more conservative way. Figure 4 indicates as the support threshold goes down, the run time per itemset decreases dramatically (notice rule time in the figure is in exponential scale). This is why the FP-growth can achieve good scalability with the support threshold.

To test the scalability with the number of transactions, experiments on data set $\mathcal{D}_2$ are used. The support threshold is set to 1.5%. The results are presented in Figure 5.

Both FP-growth and Apriori algorithms show linear scalability with the number of transactions from

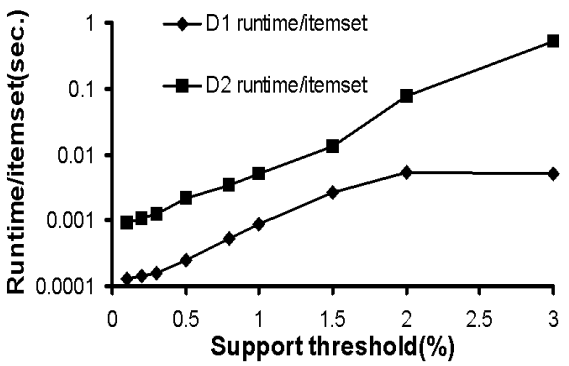

Figure 4: Run time of FP-growth per itemset versus support threshold.

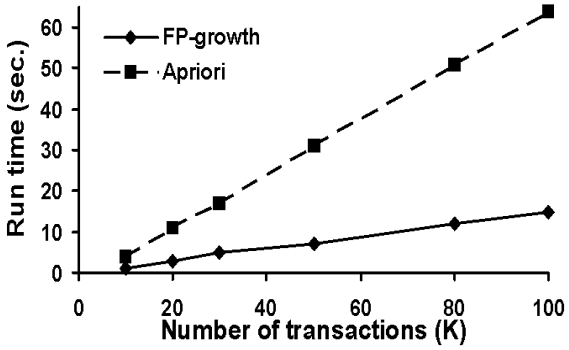

Figure 5: Scalability with number of transactions.

10K to 100K. However, FP-growth is much more scalable than Apriori. As the number of transactions grows up, the difference between the two methods becomes larger and larger. Overall, FP-growth is about an order of magnitude faster than Apriori in large databases, and this gap grows wider when the minimum support threshold reduces.

4.2 Comparison of FP-growth and TreeProjection

TreeProjection is an efficient algorithm recently proposed in [2]. The general idea of TreeProjection is that it constructs a lexicographical tree and projects a large database into a set of reduced, item-based subdatabases based on the frequent patterns mined so far. The number of nodes in its lexicographic tree is exactly that of the frequent itemsets. The efficiency of TreeProjection can be explained by two main factors: (1) the transaction projection limits the support counting in a relatively small space; and (2) the lexicographical tree facilitates the management and

counting of candidates and provides the flexibility of picking efficient strategy during the tree generation and transaction projection phrases. [2] reports that their method is up to one order of magnitude faster than other recent techniques in literature.

Based on the techniques reported in [2], we implemented a memory-based version of TreeProjection. Our implementation does not deal with *cache blocking*, which was proposed as an efficient technique when the matrix is too large to fit in main memory. However, our experiments are conducted on data sets in which all matrices as well as the lexicographic tree can be held in main memory (with our 128 megabytes main memory machine). We believe that based on such constraints, the performance data are in general comparable and fair. Please note that the experiments reported in [2] use different datasets and different machine platforms. Thus it makes little sense to directly compare the absolute numbers reported here with [2].

According to our experimental results, both methods are efficient in mining frequent patterns. Both run much faster than Apriori, especially when the support threshold is pretty low. However, a close study shows that FP-growth is better than TreeProjection when support threshold is very low and database is quite large.

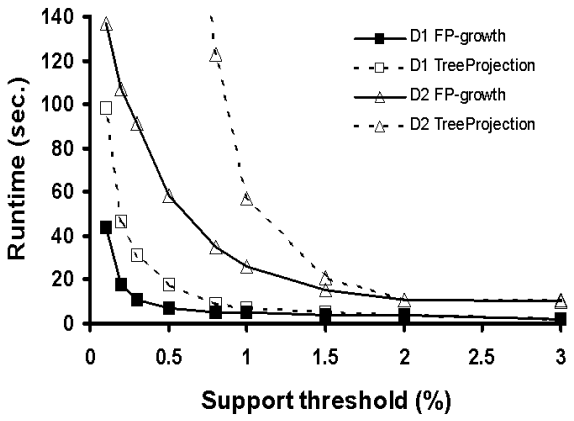

Figure 6: Scalability with support threshold.

As shown in Figure 6, both TreeProjection and FP-growth have good performance when the support threshold is pretty low, but FP-growth is better. As shown in Figure 7, in which the support threshold is set to 1%, both FP-growth and TreeProjection have linear scalability with the number of transactions, but FP-growth is more scalable.

The main costs in TreeProjection are computing of

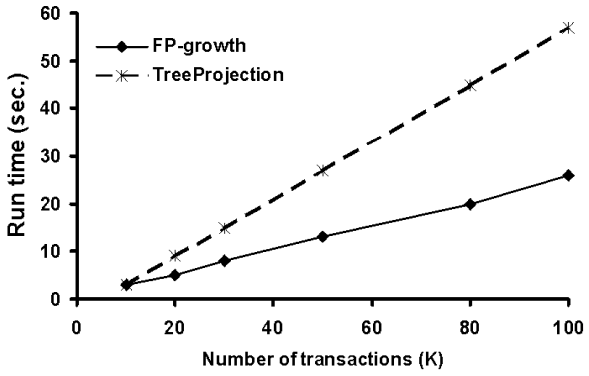

Figure 7: Scalability with number of transactions.

matrices and transaction projections. In a database with a large number of frequent items, the matrices can become quite large, and the computation cost could become high. Also, in large databases, transaction projection may become costly. The height of FP-tree is limited by the length of transactions, and each branch of an FP-tree shares many transactions with the same prefix paths in the tree, which saves nontrivial costs. This explains why FP-growth has distinct advantages when the support threshold is low and when the number of transactions is large.

5 Discussions

In this section, we discuss several issues related to further improvements of the performance and scalability of FP-growth.

1. Construction of FP-trees for projected databases.

When the database is large, and it is unrealistic to construct a main memory-based FP-tree, an interesting alternative is to first partition the database into a set of projected databases and then construct an FP-tree and mine it in each projected database.

The partition-based projection can be implemented as follows: Scan DB to find the set of frequent items and sort them into a *frequent item list* L in frequency descending order. Then scan DB again and project the set of frequent items (except i) of a transaction T into the i-projected database (as a transaction), where i is in T and there is no any other item in T ordered after i in L. This ensures each transaction is projected to at most one projected database and the total size of the project databases is smaller than the size of DB.

Then scan the set of projected databases, in the reverse order of L, and do the following. For the j-projected databases, construct its FP-tree and project

the set of items (except j and i) in each transaction T_j in the j-projected database into i-projected database as a transaction, if i is in T_j and there is no any other item in T_j ordered after i in L.

By doing so, an FP-tree is constructed and mined for each frequent item, which is much smaller than the whole database. If a projected database is still too big to have its FP-tree fit in main memory, its FP-tree construction can be postponed further.

2. Construction of a disk-resident FP-tree.

Another alternative at handling large databases is to construct a disk-resident FP-tree.

The B+-tree structure has been popularly used in relational databases, and it can be used to index FP-tree as well. The top level nodes of the B+tree can be split based on the roots of item prefix sub-trees, and the second level based on the common prefix paths, and so on. When more than one page are needed to store a prefix sub-tree, the information related to the shared prefix paths need to be registered as page header information to avoid extra page access to fetch such frequently used crucial information.

To reduce the I/O costs by following node-links, *mining should be performed in a group accessing mode*, i.e., when accessing nodes following node-links, one should exhaust the node traversal tasks in main memory before fetching the nodes on disks.

Notice that one may also construct *node-link-free* FP-trees. In this case, when traversing a tree path, one should project the prefix subpaths of *all the nodes* into the corresponding conditional pattern bases. This is feasible if both FP-tree and one page of each of its one-level conditional pattern bases can fit in memory. Otherwise, additional I/Os will be needed to swap in and out the conditional pattern bases.

3. Materialization of an FP-tree.

Although an FP-tree is rather compact, its construction needs two scans of a transaction database, which may represent a nontrivial overhead. It could be beneficial to materialize an FP-tree for regular frequent pattern mining.

One difficulty for FP-tree materialization is how to select a good minimum support threshold ξ in materialization since ξ is usually query-dependent. To overcome this difficulty, one may use a low ξ that may usually satisfy most of the mining queries in the FP-tree construction. For example, if we notice that 98% queries have $\xi \geq 20$, we may choose $\xi = 20$ as the FP-tree materialization threshold: that is, only 2% of queries may need to construct a new FP-tree. Since an FP-tree is organized in the way that less frequently occurring items are located at the deeper paths of the tree, it is easy to select only the upper portions of the FP-tree (or drop the low portions which do not satisfy the support threshold) when mining the queries with higher thresholds. Actually, one can directly work on the materialized FP-tree by starting at an appropriate header entry since one just need to get the prefix paths no matter how low support the original FP-tree is.

4. Incremental updates of an FP-tree.

Another issue related to FP-tree materialization is how to incrementally update an FP-tree, such as when adding daily new transactions into a database containing records accumulated for months.

If the materialized FP-tree takes 1 as its minimum support (i.e., it is just a compact version of the original database), the update will not cause any problem since adding new records is equivalent to scanning additional transactions in the FP-tree construction. However, a full FP-tree may be an undesirably large.

In the general case, we can register the occurrence frequency of every items in F_1 and track them in updates. This is not too costly but it benefits the incremental updates of an FP-tree as follows. Suppose an FP-tree was constructed based on a validity support threshold (called "watermark") $\psi = 0.1\%$ in a DB with 10^8 transactions. Suppose an additional 10^6 transactions are added in. The frequency of each item is updated. If the highest relative frequency among the originally infrequent items (i.e., not in the FP-tree) goes up to, say 12%, the watermark will need to go up accordingly to $\psi > 0.12\%$ to exclude such item(s). However, with more transactions added in, the watermark may even drop since an item's relative support frequency may drop with more transactions added in. Only when the FP-tree watermark is raised to some undesirable level, the reconstruction of the FP-tree for the new DB becomes necessary.

6 Conclusions

We have proposed a novel data structure, *frequent pattern tree* (FP-tree), for storing compressed, crucial information about frequent patterns, and developed a pattern growth method, FP-growth, for efficient mining of frequent patterns in large databases.

There are several advantages of FP-growth over other approaches: (1) It constructs a highly compact FP-tree, which is usually substantially smaller than the original database, and thus saves the costly database scans in the subsequent mining processes. (2) It applies a pattern growth method which avoids

costly candidate generation and test by successively concatenating frequent 1-itemset found in the (conditional) FP-trees : In this context, mining is not Apriori-like (*restricted*) *generation-and-test* but *frequent pattern* (*fragment*) *growth only*. The major operations of mining are count accumulation and prefix path count adjustment, which are usually much less costly than candidate generation and pattern matching operations performed in most Apriori-like algorithms. (3) It applies a partitioning-based divide-and-conquer method which dramatically reduces the size of the subsequent conditional pattern bases and conditional FP-trees. Several other optimization techniques, including direct pattern generation for single tree-path and employing the least frequent events as suffix, also contribute to the efficiency of the method.

We have implemented the FP-growth method, studied its performance in comparison with several influential frequent pattern mining algorithms in large databases. Our performance study shows that the method mines both short and long patterns efficiently in large databases, outperforming the current candidate pattern generation-based algorithms. The FP-growth method has also been implemented in the new version of DBMiner system and been tested in large industrial databases, such as in London Drugs databases, with satisfactory performance

There are a lot of interesting research issues related to FP-tree-based mining, including further study and implementation of SQL-based, highly scalable FP-tree structure, constraint-based mining of frequent patterns using FP-trees, and the extension of the FP-tree-based mining method for mining sequential patterns [4], max-patterns [5], partial periodicity [10], and other interesting frequent patterns.

Acknowledgements

We would like to express our thanks to Charu Aggarwal and Philip Yu for promptly sending us the IBM Technical Reports [2, 1], and to Runying Mao and Hua Zhu for their implementation of several variations of FP-growth in the DBMiner system and for their testing of the method in London Drugs databases.

References

[1] R. Agarwal, C. Aggarwal, and V. V. V. Prasad. Depth-first generation of large itemsets for association rules. *IBM Tech. Report RC21538*, July 1999.

[2] R. Agarwal, C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent itemsets. In *J. Parallel and Distributed Computing*, 2000.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB'94*, pp. 487–499.

[4] R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE'95*, pp. 3–14.

[5] R. J. Bayardo. Efficiently mining long patterns from databases. In *SIGMOD'98*, pp. 85–93.

[6] S. Brin, R. Motwani, and C. Silverstein. Beyond market basket: Generalizing association rules to correlations. In *SIGMOD'97*, pp. 265–276.

[7] G. Dong and J. Li. Efficient mining of emerging patterns: Discovering trends and differences. In *KDD'99*, pp. 43–52.

[8] G. Grahne, L. Lakshmanan, and X. Wang. Efficient mining of constrained correlated sets. In *ICDE'00*.

[9] J. Han, G. Dong, and Y. Yin. Efficient mining of partial periodic patterns in time series database. In *ICDE'99*, pp. 106–115.

[10] J. Han, J. Pei, and Y. Yin. Mining partial periodicity using frequent pattern trees. In *CS Tech. Rep. 99-10*, Simon Fraser University, July 1999.

[11] M. Kamber, J. Han, and J. Y. Chiang. Metarule-guided mining of multi-dimensional association rules using data cubes. In *KDD'97*, pp. 207–210.

[12] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A.I. Verkamo. Finding interesting rules from large sets of discovered association rules. In *CIKM'94*, pp. 401–408.

[13] B. Lent, A. Swami, and J. Widom. Clustering association rules. In *ICDE'97*, pp. 220–231.

[14] H. Mannila, H Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1:259–289, 1997.

[15] R. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *SIGMOD'98*, pp. 13–24.

[16] J.S. Park, M.S. Chen, and P.S. Yu. An effective hash-based algorithm for mining association rules. In *SIGMOD'95*, pp. 175–186.

[17] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *SIGMOD'98*, pp. 343–354.

[18] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *VLDB'95*, pp. 432–443.

[19] C. Silverstein, S. Brin, R. Motwani, and J. Ullman. Scalable techniques for mining causal structures. In *VLDB'98*, pp. 594–605.

[20] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In *KDD'97*, pp. 67–73.