# From Path Tree To Frequent Patterns: A Framework for Mining Frequent Patterns

Yabo Xu, Jeffrey Xu Yu
Chinese University of Hong Kong
Hong Kong, China
{ybxu,yu}@se.cuhk.edu.hk

Guimei Liu, Hongjun Lu
The Hong Kong University of Science and Technology
Hong Kong, China
{cslgm,luhj}@cs.ust.hk

## Abstract

*In this paper, we propose a new framework for mining frequent patterns from large transactional databases. The core of the framework is of a novel coded prefix-path tree with two representations, namely, a memory-based prefix-path tree and a disk-based prefix-path tree. The disk-based prefix-path tree is simple in its data structure yet rich in information contained, and is small in size. The memory-based prefix-path tree is simple and compact. Upon the memory-based prefix-path tree, a new depth-first frequent pattern discovery algorithm, called PP-Mine, is proposed in this paper that outperforms FP-growth significantly. The memory-based prefix-path tree can be stored on disk using a disk-based prefix-path tree with assistance of the new coding scheme. We present efficient loading algorithms to load the minimal required disk-based prefix-path tree into main memory. Our technique is to push constraints into the loading process, which has not been well studied yet.*

## 1. Introduction

Recent studies show pattern-growth method is one of the most effective methods for frequent pattern mining [1, 2, 4, 5, 8, 7, 9]. As a divide-and-conquer method, this method partitions (projects) the database into partitions recursively, but does not generate candidate sets. This method also makes use of Apriori property [3]: if any length $k$ pattern is not frequent in the database, its length $(k+1)$ super-patterns can never be frequent. It *counts* frequent patterns in order to decide whether it can assemble longer patterns. Most of the algorithms use a tree as the basic data structure to mine frequent patterns, such as the lexicographic tree [1, 2, 4, 5] and the FP-tree [8]. Different strategies were extensively studied such as depth-first [2, 1], breath-first [2, 4], top-down [11] and bottom-up [8]. Coding techniques are also used. In [1], bit-patterns are used for efficient counting. In [5], a vertical tid-vector is used, in which a bit of 1 and 0 represent the presence and absence, respectively, of the items in the set of transactions. Other data layout such as vertical tid-list, horizontal item-vector, horizontal item-list were also studied [10, 6, 12].

In this paper, we study a general framework for a multi-user environment where a large number of users might issue different mining queries from time to time. In brief, the main tasks in our general framework are listed below.

§1. Constructing an initial tree in memory for a transactional database.
§2. Mining using the tree constructed in main memory.
§3. Converting the in-memory tree to a disk-based tree.
§4. Loading a portion of the tree on disk into main memory for mining. (Note the mining is the same as §2.)

We observe that the existing algorithms become deficient in such an environment, due to the fact that all of the algorithms aim at mining a single task in a one-by-one manner. In other words, the existing algorithms repeat the first two tasks, §1 and §2, for every mining query, even though the mining queries are the same. In order to efficiently process mining queries in a multi-user environment, it is highly desirable to i) have an even *faster* algorithm when mining in main memory (task §1 and §2), and ii) reduce the cost of reconstructing a tree (task §3 and §4). Both motivate us to study new mining algorithms and new data structures which differentiate from the existing FP-growth algorithm and its data structure, FP-tree, because the complex node-links cross the FP-tree in a unpredictable manner, and the bottom-up FP-growth algorithm makes FP-tree difficult to be efficiently implemented on disk.

The main contribution of our work is given below. We propose a novel *coded* prefix-path tree, *PP*-tree, as the core of our framework. This prefix-path tree has two representations, a disk-based representation and a memory-based representation. Both are node-link-free. It is worth noting that the memory-based representation and the disk-based representation are designed for different purposes. The former

is for fast mining and the latter is for efficiently loading a portion of the tree into main memory. The novel coding scheme assists conversion between memory-representation and disk-representation of the prefix-path tree, and assists loading the minimum subtree from disk into memory. For task §2, we propose a novel mining algorithm, called $PP$-Mine, which does not generate any conditional FP-tree, and outperforms FP-growth significantly. A collection of novel loading algorithms are also proposed by which constraints can be further pushed into the loading process (task §4). We will address task §1 and §3, which are straightforward, and report our finding in our experimental studies later in this paper.

## 2. Frequent Pattern Mining

Let $I = \{x_1, x_2, \cdots, x_n\}$ be a set of items. An itemset $X$ is a subset of items $I$, $X \subseteq I$. A transaction $T_X = (tid, X)$ is a pair, where $X$ is an itemset and $tid$ is its unique identifier. A transaction $T_X = (tid, X)$ is said to contain $T_Y = (tid, Y)$ if and only if $Y \subseteq X$. A transaction database $TDB$ is a set of transactions. The number of transactions in $TDB$ that contains $X$ is called the support of $X$, denoted as $sup(X)$. An itemset $X$ is a frequent pattern, if and only if $sup(X) \geq \tau$, where $\tau$ is a threshold called a minimum support. The frequent pattern mining problem is to find the complete set of frequent patterns in a given transaction database with respect to a given support threshold, $\tau$.

**Example 1** *Let the first two columns of Table 1 be our running transaction database $TDB$. Let the minimum support threshold be $\tau$ = 2. The frequent items are shown in the third column of Table 1.*

| Trans ID | Items | Frequent items |
|---|---|---|
| 100 | c,d,e,f,g,i | c,d,e,g |
| 200 | a,c,d,e,m | a,c,d,e |
| 300 | a,b,d,g,k | a,d,e,g |
| 400 | a,c,h | a,c |

**Table 1. The transaction database $TDB$**

Given a threshold $\tau$ and a non-empty itemset $V$. In this paper, we consider three primary types of mining queries.

- **Frequent Itemsets Mining**: mining frequent patterns whose support is greater than or equal to $\tau$.
- **Frequent Superitemsets Mining**: mining frequent patterns that include all items in $V$, and have a support that is greater than or equal to $\tau$. Examples include how to find causes of a certain rule, for example, $* \rightarrow X$, where $*$ indicates any sets.
- **Frequent Subitemsets Mining**: mining frequent patterns that are included in $V$, and have a support that is greater than or equal to $\tau$. Examples include mining rules for a limited set of products, for example, daily products.

For conducting the three frequent itemsets mining, we propose a new novel coded prefix-path tree, $PP$-tree. which has two representations: a memory-based representation ($PP_M$-tree) and a disk-based representation ($PP_D$-tree). In our framework, a $PP_D$-tree, with a threshold $\tau_m$, called a materialization threshold, is possibly maintained on disk for the database $TDB$. The $PP_D$-tree is built on disk by i) constructing a $PP_M$-tree with $\tau_m$ in memory (task §1), and ii) converting $PP_M$-tree to $PP_D$-tree (task §3). The materialization threshold, $\tau_m$, is selected as the minimum threshold to support most mining tasks. With $\tau_m = 1$, the whole database can be materialized.

There are two main cases when processing one of the three types of mining queries with a threshold $\tau$ and a possible itemset $V$.

- When $PP_D$ is not available or $PP_D$ is available but $\tau < \tau_m$, the mining is conducted as constructing an initial $PP_M$-tree from the raw $TDB$ (task §1) and mining the $PP_M$-tree in memory (task §2). We propose a novel mining algorithm, $PP$-Mine, that mines $PP_M$-tree efficiently in memory. $PP$-Mine outperforms both FP-growth [8] and H-Mine [9], as shown in our experimental studies later in this paper.
- When $PP_D$ is available and $\tau \geq \tau_m$, the mining is conducted in two steps: loading (task §4) and mining (task §2).
  - In the loading phase, a minimum subtree of $PP_D$-tree is loaded from disk, and a $PP_M$-tree is constructed in memory. The given $\tau$ and $V$ are pushed into the loading phase. We propose three primary loading algorithms: $PP_\tau$-load, $PP_\supseteq$-load and $PP_\subseteq$-load. The $PP_\tau$-load algorithm supports loading for frequent itemsets mining. The integration of $PP_\supseteq$-load with $PP_\tau$-load supports loading for frequent superitemsets mining. The integration of $PP_\subseteq$-load with $PP_\tau$-load supports loading for frequent subitemsets mining.
  - In the mining phase, as above, $PP$-Mine mines the $PP_M$-tree efficiently in memory. It is important to know that, because $\subseteq$ ($\supseteq$) is pushed into the loading phase, here, $PP$-Mine does not need to check $\subseteq$ ($\supseteq$) in the mining phase.

In the following, we concentrate on the coded prefix-path tree, the mining algorithm, $PP$-Mine, and the three loading algorithms.

## 3. A Coded Prefix-Path Tree

**Definition 1** *A Prefix-Path tree (or PP-tree in short) is an order tree. Let $F$ be a set of frequent items (1-itemsets) in a total order ($\prec$).*[1] *A node in the tree is labelled for a frequent item in $F$. The root of the tree represents "null" item. The children of a node are listed following the order. A path of length $l$ from the root to a node in the tree represents a $l$-itemset. The rank of a PP-tree is the number of frequent 1-itemsets.*

**Definition 2** *A complete prefix-path tree of rank $N$ is a prefix-path tree with $2^N$ nodes, denoted as $\overline{PP}$-tree. Each node is encoded with a number (of the* `pre-order` *of traversal of the tree). The number associated with a node is called the* code *of that node. The code for the root is 0.*

**Definition 3** *A PP-tree is coded using the code of the corresponding node in the complete $\overline{PP}$-tree with the same rank.*

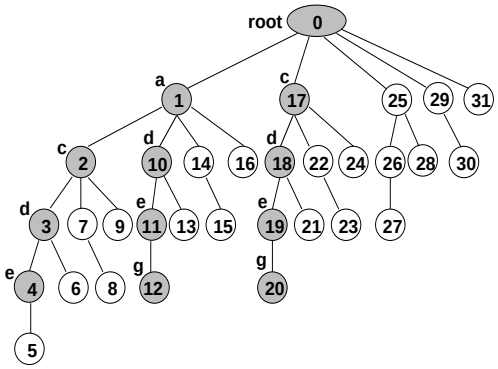



**Figure 1. The $\overline{PP}$-tree for Example 1**

In the following, a $PP$-tree is a coded prefix-path tree, unless otherwise specified. The $PP$-tree for the frequent items in the third column of Table 1 is shown as the shaded subtree in Figure 1. The rank of this $PP$-tree is 5, because five frequent items, `a`, `c`, `d`, `e` and `g`, are represented in frequency order – their support is greater than or equal to the minimum support ($\tau = 2$). Its complete prefix-path tree, $\overline{PP}$-tree has 32 ($= 2^5$) nodes in total. The root is numbered 0 and its five children, `a`, `c`, `d`, `e` and `g`, are numbered 1, 17, 25, 29 and 31, respectively. The first subtree of the root, `a`, has four children, `c`, `d`, `e` and `g`, and are numbered 2, 10, 14 and 16. A code in a $PP$-tree uniquely represents a path from the root and therefore an itemset. The code 3 represents a path (a frequent itemset) `acd`, and 19 represents `cde`.

Given a $PP$-tree of rank $N$ where $F$ is a set of frequent 1-itemsets kept in the $P$-tree. Some observations can be made below.

[1]The order can be any order like frequency order, lexicographic order.

- A $PP$-tree of rank $N$ is built for a database with a given minimum support, $\tau_m$, called materialization threshold, where $N$ is the number of frequent 1-itemsets. When $\tau_m = 1$, the whole database is maintained as the $PP$-tree.
- The $PP$-tree can be used to mine the database with a minimum support $\tau \geq \tau_m$.
- It has $N$ subtrees and the size of the $k$-th subtree is $2^{N-k}$ ($1 \leq k < N$).
- A function $kth(N, n_i)$ is defined, which indicates that code $n_i$, $1 \leq n_i < 2^N$, is in the $k$-th subtree. $kth(N, n_i) = N - k_r$ where $k_r$ is the maximum number satisfying $2^N - n_i - (2^{k_r} - 1) \geq 0$. Recall 0 is the code of the root.
- The code of its $k$-th child, $1 \leq k < N$, can be calculated with a function $code(N, k) = 1 + \sum_{i=1}^{k-1} 2^{N-i} = 1 + 2^N - 2^{(N-k+1)}$. The function $code$ can be easily calculated using bit shift operator.
- The item that the $k$-th child represents, $1 \leq k < N$, is the $k$-th item in $F$.
- All codes in the $k$-th subtree are ranged between $code(N, k)$ and $code(N, k+1)$ for $k < N - 1$. The last subtree has no children.

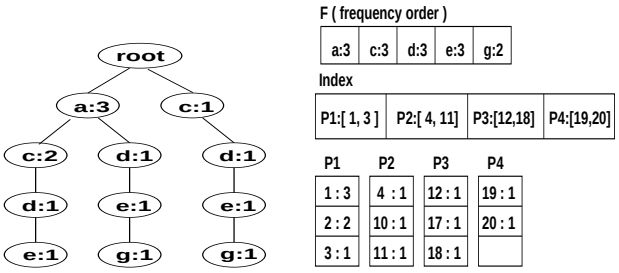
It is important to know that, given a $PP$-tree of rank $N$, the codes/itemsets along the path from the root to a node, $n$, can be computed from the code of the node, $\overline{n}$. For example, as shown in Figure 1, code 19 represents an itemset, {`c, d, e`}.

In our framework, we use the notion of complete prefix-path tree to code nodes. In practice, a $PP$-tree of rank $N$ is much smaller than the corresponding complete prefix-path tree. We only deal with prefix-path trees.

### 3.1 $PP$-tree Representations and Its Construction

A prefix-path tree has its memory-based and disk-based representations. The in-memory representation of $PP$-tree, denoted $PP_M$-tree, is of a tree. Despite the pointers to the children nodes, a node in $PP_M$-tree consists of item-name, count, and a node-link. The count registers the number of transactions represented by the portion of the path reaching from the root to this node. The disk representation of $PP$-tree of rank $N$, denoted $PP_D$-tree, is represented as $(T, F, I, \tau_m)$. Here, $T$ is a heap for the tree structure in which an element consists of a code and its count. $F$ stores $N$ frequent 1-itemsets with their counts in order. $I$ is an index indicating the ranges of codes in disk-pages. $\tau_m$ is the minimum support used to build $PP_D$-tree on disk. This $PP_D$-tree can be used for mining frequent itemsets with a minimum $\tau \geq \tau_m$.

The $PP_M$-tree and $PP_D$-tree for Example 1 ($\tau = 2$) are shown in Figure 2 (a) and (b), respectively. Recall, when




(a) The memory representation ($PP_M$-tree)

(b) The disk representation ($PP_D$-tree)

**Figure 2. The $PP$-tree representations for Example 1**

$\tau = 2$, the frequent items are shown in the third column of Table 1, and are represented as shaded nodes in Figure 1. In the $PP_M$-tree, `i:s` represents item:count. All node-links in the $PP_M$-tree are initialized as null. Those node-links are used when mining. In the $PP_D$-tree, $T$ is stored in four pages, where `c:s` represents code:count. In $F$, `i:s` represents item:count. As mentioned above, we can simply compute the item(s) a code represents. Therefore, we do not necessarily store items in $T$. The index $I$ indicates that code 1-3 are stored in page $P_1$, and so on so forth. The minimum support $\tau_m$ to build this tree is 2.

Given a transactional database $TDB$ and a minimum support ($\tau_m$), an initial $PP_M$-tree can be constructed as follows. First, we scan the database to find all the frequent items, then, we scan the database again to construct $PP_M$-tree in memory. For each transaction, the infrequent items are removed. The remaining frequent items are sorted in a total order, and are inserted into $PP_M$-tree. The constructing time for $PP_M$-tree is slightly less than FP-Tree, because it does not need to build node-links in the tree initially. $PP_M$-tree can be converted to $PP_D$-tree and maintained on disk continuously using our coding scheme. We omit the details here.

## 4. $PP$-Mine: Mining In-Memory

In this section, we propose a novel mining algorithm, called $PP$-Mine, using a $PP_M$-tree. For simplicity, we use a prefix-path to identify a subtree. Here, the prefix-path is expressed as a dot-notation to concatenate items. For example, in Figure 3, $a$-prefix identifies the leftmost subtree containing $a$, and $a.c$-prefix identifies the second subtree rooted at $a$-prefix. In the following, we use $i_j$ and $i_k$ for a single item prefix-path, and use $\alpha$, $\beta$ and $\gamma$ for a prefix-path in general which are possible empty.

The $PP$-Mine algorithm is based on two properties. The first property states the Apriori property as below.

**Property 1** *Given a $PP_M$-tree of rank $N$ for a set of frequent itemsets $I = (i_1, i_2, \cdots, i_N)$, where a total order ($\preceq$) is defined on $I$. A pattern represented by $\alpha.i_j.i_k$-prefix can be frequent if the pattern represented by $\alpha.i_j$-prefix is frequent, where $i_j \preceq i_k$.*

The second property specifies subtrees that need to be mined for a pattern. The second property is given on top of two concepts: containment and coverage. We describe them below. Given a $PP_M$-tree of rank $N$ for a set of frequent itemsets $I = (i_1, i_2, \cdots, i_N)$, where a total order ($\preceq$) is defined on $I$. We say a prefix-path (representing a subtree), $i_k.\alpha$-prefix, is contained in $i_j.\alpha$-prefix, denoted $i_k.\alpha$-prefix $\subseteq i_j.\alpha$-prefix, if $i_j \preceq i_k$. In addition, $\alpha$-prefix $\subseteq \gamma$-prefix, if $\alpha$-prefix $\subseteq \beta$-prefix and $\beta$-prefix $\subseteq \gamma$-prefix. A *coverage* of a prefix-prefix $\alpha$-prefix is defined as all the $\beta$-prefixes that contain $\alpha$-prefix (including $\alpha$-prefix itself).

**Property 2** *Given a $PP_M$-tree of rank $N$ for a set of frequent itemsets $I = (i_1, i_2, \cdots, i_N)$, where a total order ($\preceq$) is defined on $I$. Mining a pattern represented by a path-prefix $\alpha$-prefix is to mine the coverage of $\alpha$-prefix.*

For example, Figure 3 shows a $PP$-tree with four items {a, b, c, d}. Assume they are in lexicographic order. The coverage of $b.c.d$-prefix includes $b.c.d$-prefix and $a.b.c.d$-prefix. It implies that we only need to check these two subtrees, in order to determine whether the pattern, {b, c, d}, is frequent. Also, the coverage of $c.d$-prefix includes $c.d$-prefix, $b.c.d$-prefix, $a.c.d$-prefix and $a.b.c.d$-prefix. It implies that we only need to check these four subtrees, in order to determine whether the pattern, {c, d}, is frequent.

Based on the above two properties, we derive three main features including two pushing operations and a no-counting strategy below.

- **Push-down**: Processing at a node in a $PP_M$-tree is to check an itemset represented by the path-prefix from the root to the node in question. Pushing-down to one of its children is to check the itemset with one more item. Property 1 states the Apriori heuristic. We implement it as a depth-first traversal with building a sub header-table.
- **Push-right**: Mining an itemset requires to identify a minimal coverage in $PP_M$-tree to mine. Property 2 specifies such a minimal coverage for any path-prefix. Pushing-right is a technique that helps to identify the coverage transitively, based on Property 2. In other words, the push-right strategy is to push the child to its corresponding sibling. We implement it as a dynamic link-justification. It is the best to illustrate it using an example. In Figure 3, after we have mined all the patterns in the leftmost subtree ($a$-prefix), we push-right $a.b$-prefix to the subtree $b$-prefix, push-right $a.c$-prefix to the subtree $c$-prefix, and push-right $a.d$-prefix to the subtree $d$-prefix. After mining the subtree

($b$-prefix), $b.c$-prefix is pushed to $c$, as well as $a.b.c$-prefix transitively. It is worth noting that the subtree $a.c$-prefix does not need to be pushed into the subtree $b.c$-prefix, because the former is to check the itemset $\{a, c, d\}$ excluding $\{b\}$, whereas the latter is to check the item $\{b, c, d\}$ excluding $\{a\}$.

- **No-counting**: Counting is done as a side-effort of pushing-right (dynamic link-justification) in an accumulated manner. For example, after we push-right $a.b$-prefix to the subtree $b$-prefix, all the prefix-paths and their support counts for $b$-prefix are collected by dynamic link-justification automatically. Therefore, all the counting cost is minimized. No extra counting is needed.

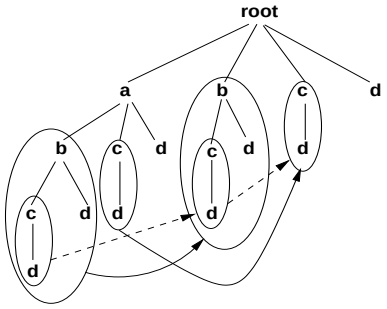



**Figure 3. A $PP_M$-tree with four items**

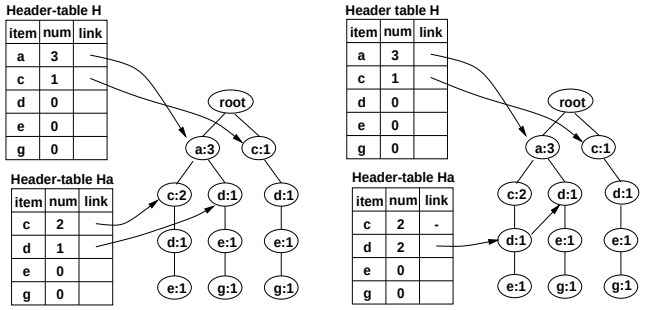
The $PP$-Mine algorithm is illustrated in Algorithm 1. The procedure is to check all the items in the header table $H$ passed (line 1-10). In line 2-3, we check if the corresponding count (num) for $a_i$ is greater than or equal to the minimum support, $\tau$. Recall that counts are accumulated through pushing-right. If num for $a_i$ is greater than or equal to $\tau$, we output the pattern as represented by the path. Then, at line 4, a sub header table is created by removing all the entries before $a_i$ (including $a_i$). Pushing-down $a_i$ (line 5) is outlined below. Because the coverage of $a_i$-prefix has already linked through the link field in the header-table $H$ (by the previous push-rights), all $a_i$'s $j$-th children on the link are pushed-down (chained) into the corresponding $j$-th entry in the sub header table ($H_{\alpha.a_i}$). Line 6 calls $PP$-Mine recursively to check (k+1)-itemset if the length of the path is $k$. After returning, the sub header table will be deleted. Irrelevant with the minimum support, pushing-right $a_i$ (line 9) is described below: a) the coverage of $a_i$'s left siblings are pushed-right from $a_i$ to its right siblings, b) all $a_i$'s $j$-th children on the link are pushed-right (chained) into the corresponding entry in the header table $H$.

Consider the mining process using the constructed $PP_M$-tree (Figure 2(a)). Here, the initial header table $H$ includes all single items in $PP_M$-tree. Only the children of the root are linked from the header-table, and their counts are copied into the corresponding num fields in the header-table. Other

**Algorithm 1** $PP$-Mine($\alpha$, $H$)

**Input**: A constructed $PP_M$-tree identified by the prefix-path, $\alpha$, and the header table $H$.

```
1:  for all a_i in the header table H do
2:    if a_i's support ≥ τ then
3:      output α.a_i and a_i's support;
4:      generate a header-table, H_{α.a_i}, for the subtree
        rooted at α.a_i, based on H;
5:      push-down(a_i);
6:      PP-Mine(α.a_i, H_{α.a_i});
7:      delete H_{α.a_i};
8:    end if
9:    push-right(a_i);
10: end for
```




(a) The header table $H_a$ and its $PP_M$-tree rooted at $a$-prefix

(b) The header table $H_a$ after mining the $PP_M$-subtree rooted at $a.c$-prefix

**Figure 4. An Example**

links/nums in the header-table are initialized as null and zero. (The initial header $H$ is shown in Figure 4 (a).)

1. Call $PP$-Mine(root, $H$). Item $a$ is first to be processed, as the first entry in $H$. The support of $a$ is 3. It is the exact total support for the item $a$, because $a$ does not have any left siblings. Next, the subtree $a$-prefix is to be mined.

   The second header table, $H_a$, consists of all items in $H$ except for $a$. Only the children nodes of $a$ are pushed-down into $H_a$ (Figure 4 (a)). In $H_a$, $c$ and $d$ counts are copied from the node $a.c$ and $a.d$, in the $PP_M$-tree. Their values are 2 and 1.
2. Call $PP$-Mine($a$-prefix, $H_a$). Item $c$ is picked up as the first entry in $H_a$. Because $c$'s count (num) is 2 (frequent), we output $a.c$. Next, the subtree $a.c$-prefix is to be mined.

   The third header-table is constructed for the subtree of $a.c$-prefix, denoted as $H_{ac}$, in which $d$'s num is 1 and $d$'s link points to the node $a.c.d$. Other fields for $e$ and $g$ are set as zero/null.

3. Call $PP$-Mine($a.c$-prefix, $H_{ac}$). Item $d$ is picked up. Because $d$'s `num` is 1 (infrequent), return.
4. Backtrack to the subtree $a$-prefix. Here, the header-table $H_a$ is reset (Figure 4 (b)). First, the entry $c$ in $H_a$ becomes null (done). Second, $a.c$'s child, $d$, is pushed-right into $d$'s entry in the header-table $H_a$. In other words, the `link` of the entry $d$ in $H_a$ is linked to the node $a.d$ through the node $a.c.d$. The $d$'s count (`num`) in $H_a$ is accumulated to 2, which indicates $\{a, d\}$ occurs 2 times.

The correctness of $PP$-Mine can be showed as follows in brief. A $PP_M$-tree of rank $N$ has $N$ subtrees. First, we mine patterns in a subtree following a depth-first traversal order. All patterns in a subtree will be mined (vertically). Second, the $k$-th subtree is mined by linking all required subtrees in its left siblings (horizontally). Linking to those subtrees will be completed at the time when the $k$-subtree is to be mined. Third, the above holds for any subtrees in the $PP_M$-tree of rank $N$ (recursively).

## 5. Efficient Loading

In this section, we assume that a $PP_D$-tree is available on disk with $\tau_m$, and discuss how to process any of the three primary types of mining queries (frequent itemsets mining, frequent superitemsets mining and frequent subitemsets mining) with a threshold $\tau$ and an itemset $V$. We emphasize on two things: a) loading a sub $PP_D$-tree from disk, and b) constructing a *minimum* $PP_M$-tree in memory. Here, the minimum $PP_M$-tree is a $PP_M$-tree such that it cannot process the mining query correctly if any node in the tree is removed. It is important to note that, here, a) is to reduce I/O costs for loading, and b) is for further reducing CPU costs for mining in memory.

We studied three primary loading algorithms: $PP_\tau$-load, $PP_\supseteq$-load and $PP_\subseteq$-load. These algorithms load subtrees of a $PP_D$-tree from disk and construct a $PP_M$-tree in memory. The $PP_\tau$-load algorithm supports loading for frequent itemsets mining. The integration of $PP_\supseteq$-load with $PP_\tau$-load supports loading for frequent superitemsets mining. The integration of $PP_\subseteq$-load with $PP_\tau$-load supports loading for frequent subitemsets mining. Due to space limit, we only present our $PP_\tau$-load algorithm in this paper.

The loading algorithm, $PP_\tau$-load, is outlined in Algorithm 2. Four parameters will be passed, the code of a root node $p$ of a prefix-path tree of rank $N$, the reading position $d$, and a new rank $M$. The new rank is computed by the given $\tau$ as follows. Suppose the prefix-path tree on disk is based on frequency order. $M$ is the total number of frequent 1-itemsets stored that are greater than $\tau$. If a given threshold $\tau$ is larger, the computed $M$ will be smaller. Therefore, the $PP$-tree to loaded into memory will be smaller. The newly computed $M$ reduce the number of page accesses.

Initially, when loading, we call $PP_\tau$-load(0, $M$, $N$, 0), where the first zero is the code of the root of the $PP$-tree of rank $N$, and the second zero is the reading position of the $PP_D$-tree on disk. Algorithm 2 is a recursive algorithm. The $PP_D$-tree, represented by $p$, to be loaded has $N$ children at most. Line 1-3 reads the page where $d$ exists, if $d$ has not been read-in. Line 4-5 calculate the code of children nodes. Here, $c_i$ is the code in terms of the $PP_D$-tree, passed by the parameter $p$, and $a_i$ is the code in terms of the whole $PP_D$-tree on disk. Line 7-12 attempt to jump to a page and find the next page to read if the code in the reading position is less than the $i$-th child ($a_i$). The `readPage` function will use the index to load a page in which at least a $d$ exists whose code is greater than or equal to $a_i$. If the code of $d$ matches $a_i$ (line 13), a new child node is constructed in memory, and $PP_\tau$-load will be recursively called. Note, $d$ is called by-reference. The coding scheme and the index allows us to reduce the I/O cost to minimum.

**Algorithm 2** $PP_\tau$-load($p$, $M$, $N$, $d$)

**Input**: the code of root ($p$), the required rank ($M$), the rank of the $PP_D$-tree ($N$), and the current reading position on disk ($d$) (call by reference).
**Output**: a $PP_M$-tree.

```
1:  if page(d) does not exist in memory then
2:     readPage(d) using the index;
3:  end if
4:  let c_i be the code of the i-th child of p (of rank N);
5:  a_i ← c_i + p;
6:  while i < M do
7:     if code(d) < a_i then
8:        d ← readPage(a_i);
9:        while code(d) < a_i do
10:          d++;
11:       end while
12:    end if
13:    if a_i = code(d) then
14:       build the new child node for d in memory as the
          child of p;
15:       d++;
16:       PP_τ-load(a_i, M − i, N − i, d);
17:    end if
18: end while
```

## 6. Performance Study

We conducted performance studies to analyze the efficiency of $PP$-Mine in comparison of FP-tree [8] and H-Mine [9]. We did not compare $PP$-Mine with TreeProjection [2], because, as reported in [8], FP-growth outperforms TreeProjection.

All the three algorithms were implemented using Visual C++ 6.0. The synthetic data sets were generated using the procedure described in [3]. All our experimental studies were conducted on a 900MHz Pentium PC, with 128MB main memory and a 20GB hard disk, running Microsoft Windows/NT.

Given a database $TDB$. We reemphasize the differences between $PP$-Mine and FP-tree/H-Mine for the mining task with a minimum support $\tau$ below.

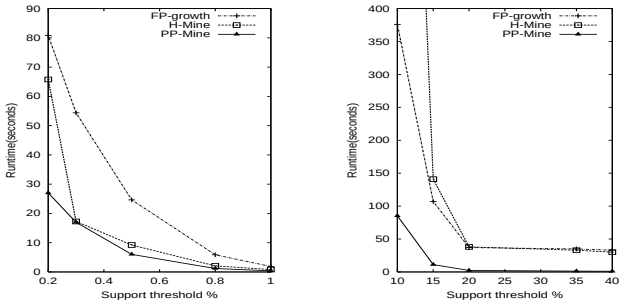
- In our framework, A $PP_D$-tree is possibly stored on disk with a materialized threshold $\tau_m$. For a mining task with a minimum support $\tau \geq \tau_m$ with/without ($\subseteq V$, $\supseteq V$), a loading algorithm loads a subtree from disk and constructs a $PP_M$-tree in memory. The conditions ($\tau$, $\supseteq$, $\subseteq$) are pushed into the loading. Upon the prefix-path tree is constructed in memory, $PP$-Mine further mines $PP_M$-tree using $\tau$ only. Otherwise, when $PP_D$-tree is not available or $\tau < \tau_m$, a $PP_M$-tree is constructed from the transactional database in memory to be mined.
- Both FP-growth and H-Mine consists of two phases, *constructing* and *mining*. In the constructing phase, they scan $TDB$ and construct a FP-tree/H-struct in memory using a minimum support $\tau$. In the mining-phase, they conduct the mining task further using the minimum support $\tau$.

### 6.1 $PP$-Mine, FP-growth, H-Mine

In this section, we focus on the mining task with a minimum support only. We assume that no $PP_D$-tree exist on disk. For a given minimum support $\tau$, we assume that we have to construct $PP_M$-tree, FP-tree and H-struct in memory from scratch. The constructing time for both H-struct and $PP_M$-tree is marginally better than FP-tree construction. To give a fair view on this three algorithms, here we only compare the mining-phase of the three algorithms.

We have conducted experimental studies using the same datasets as reported in [8]. We report our results using one of them, T25.I20.D100K with 10K items, as representative. In this dataset, the average transaction size and average maximal potentially frequent itemset size are set to be 25 and 20, respectively, while the number of transactions in the dataset is 100K. There are exponentially numerous frequent itemsets in this dataset, when the minimum support is small. The frequent patterns include long frequent itemsets as well as a large number of short frequent itemsets.

The scalability of the three algorithms, $PP$-Mine, FP-tree and H-Mine, is shown in Figure 5 (a). While the support threshold decreases, the number as well as the length of frequent itemsets increases. High overhead incurs for handling projected transactions. FP-growth needs to construct conditional FP-trees using extra memory space repeatedly. H-Mine needs to count every projected transactions. $PP$-Mine does not need to construct conditional trees and uses accumulation technique, which avoids unnecessary counting. From Figure 5 (a), we can see $PP$-Mine significantly outperforms FP-growth and H-Mine. $PP$-Mine scales much better than both FP-tree and H-Mine.




(a) small threshold ($\tau$) (b) large threshold ($\tau$)

**Figure 5. Scalability**

We also compared the mining phase of the three algorithms using a very dense dataset. The dataset was generated with 101 distinct items and 1K transactions. The average transaction size and average maximal potentially frequent itemset size are set to 40 and 10. When the minimum support is 40%, the number of frequent patterns is 65,540. When the minimum support becomes 10%, the number of frequent patterns is up to 3,453,240. As shown in Figure 5 (b), $PP$-Mine outperforms both FP-growth and H-Mine significantly. $PP$-Mine has the best scalability while the threshold decreases.

For sparse datasets and small datasets, $PP$-Mine marginally outperforms H-Mine, because both use the similar dynamic link adjusting technique. The effectiveness of $PP$-Mine's accumulation (or non-counting) techniques becomes weaker. Both $PP$-Mine and $H$-Mine outperform FP-growth.

### 6.2 $PP$-Mine Analysis

In this section, we further analyze the effectiveness of $PP$-Mine (and $PP$-tree) in terms of loading/constructing/mining, using a very large tree. Such a large tree was generated using T40I10D100K. Its average transaction size and average maximal potentially frequent itemset size are 40 and 10, respectively. The number of distinct items generated was 59. We chose a minimum support (50%) to build a $PP_D$-tree on disk for this dataset. The minimum support was chosen, because the resulting number of frequent patterns is large enough for our testing purposes, 138,272,944. The $PP_D$-tree we built on disk has 51,982 nodes, which is considerably small.

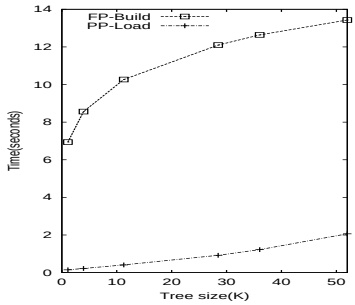



**Figure 6. Scalability with the tree size**

Figure 6 compares the cost for FP-growth to construct a FP-tree in memory with the cost for $PP_\tau$-load to load a sub $PP_D$-tree and construct a rather small $PP_M$-tree. The intension of the figure is to show the necessity of $PP_D$-tree. In Figure 6, we use tree size rather than threshold, because a threshold does not precisely indicate the tree size. Different thresholds may end up the same tree size. The tree sizes and the corresponding thresholds used in this figure are listed below, as a pair of tree size and threshold, (1,100, 90%), (3,943, 80%), (11,281, 77%), (28,474, 76%), (36,038, 75%), (51,982, 50%). The tree sizes are the same for the threshold in the range of 75-50%. Note: a smaller threshold results in a larger tree.

As shown in Figure 6, $PP_\tau$-loading time is much smaller than FP-growth constructing time (constructing an initial FP-tree in memory), as expected. Saving $PP_D$-tree on disk can significantly reduce both the time to construct a tree in memory and the memory space. It is worth noting that the loading time for a tree is proportional to the size of the $PP_D$-tree size. That suggests that, if we only need a small portion of the data, with the help of $PP_D$-tree, we do not need to load the whole dataset.

## 7. Conclusion

In this paper, we propose a new framework for mining frequent patterns from large transactional databases in a multiuser environment. With this framework, we propose a novel coded prefix-path tree with two representations, a memory-based prefix-path tree and a disk-based prefix-path tree. The coding scheme is based on a depth-first traversal order. Its unique features include easy identifying of the location in a prefix-path tree, and easy identifying of the itemsets. The loading scheme makes the disk-based prefix-path tree node-link-free. With help of a simple index, several new loading algorithms are proposed which can further push constraints into the loading process, and, therefore, reduce both I/O cost and CPU cost, because the prefix-path tree constructed in memory becomes smaller. In terms of mining in memory, $PP$-Mine algorithm outperforms FP-tree significantly, because $PP$-Mine does not need to construct any conditional FP-trees for handling projected databases. Instead, dynamic link adjusting are used. Both $PP$-Mine and H-Mine adopt dynamic link adjusting technique. In addition, $PP$-Mine further minimizes counting cost. Accumulation technique is used, and therefore, unnecessary counting is avoided. $PP$-Mine outperforms H-Mine significantly when dataset is dense, and outperforms H-Mine marginally when dataset is sparse and is small.

**Acknowledgment**: The work described in this paper was supported by grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (CUHK4229/01E, DAG01/02.EG14).

## References

[1] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. Depth first generation of long patterns. In *Proc. 6th ACM SIGKDD Int. Conf. on Knowledge discovery and data mining*, pages 108–118. ACM Press, 2001.

[2] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent item sets. *Journal of Parallel and Distributed Computing*, 61:350–371, 2001.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 12–15 1994.

[4] R. J. Bayardo. Efficiently mining long patterns from databases. In *1998 ACM SIGMOD Intl. Conference on Management of Data*, pages 85–93. ACM Press, 05 1998.

[5] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: A maximal frequent itemset algorithm for transactional databases. In *2001 Intl. Conference on Data Engineering,ICDE*, pages 443–452, 04 2001.

[6] B. Dunkel and N. Soparkar. Data organization and access for efficient data mining. In *Proc. of 15th IEEE Intl. Conf. on Data Engineering*, pages 522–529, 03 1999.

[7] J. Han and J. Pei. Mining frequent patterns by pattern-growth: Methodology and implications. In *ACM SIGKDD Explorations*. ACM Press, 12 2001.

[8] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In W. Chen, J. Naughton, and P. A. Bernstein, editors, *2000 ACM SIGMOD Intl. Conference on Management of Data*, pages 1–12. ACM Press, 05 2000.

[9] J. Pei, J. Han, H. Lu, S. Nishio, and D. Y. S hiwei Tang. H-mine:hyper-structure mining of frequent patterns in large databases. In *2001 IEEE Conference on Data Mining*. IEEE, 11 2001.

[10] P. Shenoy, J. R. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In *2000 ACM SIGMOD Intl. Conference on Management of Data*, pages 22–33. ACM Press, 05 2000.

[11] K. Wang, L. Tang, J. Han, and J. Liu. Top down fp-growth for association rule mining. In *Proc. of 6th Pacific-Asia conference on Knowledge Discovery and Data Mining*, 2002.

[12] M. J. Zaki. Scalable algorithms for association mining. *Knowledge and Data Engineering*, 12(2):372–390, 2000.