# 不产生候选的快速投影频繁模式树挖掘算法

Mining Project Frequent Patterns without Candidate Generation

何炎祥　向剑文　朱骁峰　孔维强

（武汉大学计算机学院，软件工程国家重点实验室　武汉430072）

**Abstract**　Frequent Pattern mining plays an essential role in data mining. 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 introduce a novel frequent pattern growth (FP-growth)method, which is efficient and scalable for mining both long and short frequent patterns without candidate generation. And build a new project frequent pattern growth (PFP-tree)algorithm on this study, which not only heirs all the advantages in the FP-growth method, but also avoids it's bottleneck in database size dependence. So increase algorithm's scalability efficiently.

**Keywords**　Data mining, Frequent patterns-tree, Frequent patterns-growth, Project frequent pattern-tree

## 1. 概述

近年来，对事务数据库、时序数据库和各种其它类型数据库中的频繁模式挖掘的研究越来越普及。许多先前的研究都是采用 Apriori 或类似的候选产生－检查迭代算法[1~9]，使用候选项集来找频繁项集。这些算法都基于一种重要的反单调的 Apriori 性质：任何非频繁的(k－1)-项集都不可能是频繁 k-项集的子集。因此，如果一个候选 k-项集的(k－1)-子集不在频繁(k－1)-项集中，则该候选也不可能是频繁的，从而可以在候选集的集合中删除。它的作用就在于有效地压缩搜索空间，提高频繁项集逐层产生的效率。

在许多情况下，Apriori 性质大幅度压缩了候选项集的大小，并导致很好的性能。然而，它有两种开销可能是十分巨大的：

·它可能需要产生大量的候选项集。例如，如果有$10^4$个频繁1-项集，则 Apriori 算法需要产生多达$10^7$个候选2-项集，并累计和检查它们的频繁性。此外，为发现长度为100的频繁模式，如$\{a_1, a_2, \cdots, a_{100}\}$，它必须产生多达$2^{100} \approx 10^{30}$个候选。而且，无论采用什么样的技术，都很难减少这种候选产生的固有开销。

·它可能需要重复地扫描数据库，通过模式匹配检查一个很大的候选项集合。对于挖掘长模式尤其如此。

“有没有可能设计一种方法，挖掘全部频繁项集而不产生候选？”一种有效的方法称作频繁模式增长(frequent-pattern growth)，或简称 FP-growth[10,11]，它采取如下分而治之的策略，整个过程分为两步：首先，将提供频繁项集的数据库压缩到一颗频繁模式树(或 FP-tree)，但仍保留项集关联信息；然后，将这种压缩后的数据库分成一组条件数据库(一种特殊类型的投影数据库)，每个关联一个频繁数据库。FP-growth 方法将发现长频繁模式的问题转换成递归地发现一些短模式，然后连接后缀。它使用最不频繁的项作为后缀，提供了良好的选择性。该方法大大降低了搜索开销。

对 FP-tree 方法的性能研究表明：对于挖掘长的和短的频繁模式，它都是有效的和可伸缩的，并且大约比 Apriori 算法快一个数量级。它也比树-投影算法快，树-投影算法是递归地将数据库投影为投影数据库树。

但是，通过我们对 FP-growth 方法的仔细研究表明，在构造 FP-tree 的过程中，必须遍历三次数据库，第一次用来收集频繁1-项的集合 F，构造频繁项表；第二次则根据频繁项表对数据库事务中的所有频繁项进行排序；第三次则依次逐个扫描排序后的频繁项，构造 FP-tree(第二次和第三次遍历其实是交叉进行的，即对于数据库中的每一个事务，首先根据频繁项表对其频繁项进行排序，然后再逐个扫描每一个频繁项，构造 FP-tree。直至扫描完所有的事务)。算法效率的瓶颈主要集中在第三次遍历，因为在构造 FP-tree 的过程中，必须对数据库中每个事务的每个频繁项逐个进行判断，决定如何加入到 FP-tree 的结点中。也就是说，假设数据库中每个事务包含 $\lambda$ 个频繁1-项，事务个数为 $\theta$(数据库大小的重要指标)，那么系统执行计算(用来构造 FP-tree)的次数为：

$$\omega = \sum_{i=1}^{\theta} \lambda \qquad (1.1)$$

系统开销主要取决于 $\theta$ 的大小(而实际应用中事务个数，即数据库中记录的个数通常都是很大的，导致 $\omega$ 值十分庞大)。而且由于在 FP-tree 构造算法中，这是一个串行的过程，后一个项的计算必须严格基于前一个项的计算结果，不能并行计算。如果 $\omega$ 比较大的话，那么系统的开销将会变得十分昂贵，效率也将大大降低。

为此，我们提出了一种新的改进的投影频繁模式树(Project Frequent Pattern-tree)构造算法，或简称为 PFP-tree 构造方法，这是一种基于投影技术的 FP-tree 构造方法。在构造 FP-tree 的构造过程中，系统开销主要取决于数据库中包含频繁1-项最多的事务频繁长度 $\zeta$(事务的频繁长度即指事务包含频繁项的个数)。而在实际应用中，$\zeta$ 取值一般都比较小，相比 $\omega$ 一般都相差好几个数量级，因此系统的性能也将大大提高，算法也因此具有更好的可伸缩性。

本文详细介绍传统的 FP-growth 方法和改进的 PFP-

何炎祥　教授，博士生导师，主要研究方向为分布信息处理、电子商务等。向剑文　博士生，主要研究方向为电子商务、数据挖掘、软件工程与软件可靠性工程。

tree 方法，最后总结了我们的工作并指出了进一步研究的方向。

## 2. 传统的 FP-growth 算法

设 $I=\{\alpha_1,\alpha_2,\cdots,\alpha_m\}$是项的集合，事务数据库 $DB=\langle T_1,T_2,\cdots,T_n\rangle$，其中每个事务 $T_i(i\in[1,\cdots,n])$是 $I$ 中项的集合，使得 $T\subseteq I$。每一个事务有一个标识符，称作 TID。设模式 $A$ 是一个项集，$A$ 的支持度(注意在本文中支持度定义为绝对的发生频率，而在有的文献中定义为相对值，即 $DB$ 中包含 $A$ 的事务的百分比)(或者发生频率)就是 $DB$ 中包含 $A$ 的事务的个数。如果 $A$ 的支持度不小于预先设定的最小支持度阈值 $\xi$，则称它为频繁模式(frequent pattern)。

给定一个事务数据库 $DB$ 和一个最小支持度阈值 $\xi$，挖掘所有的频繁模式的问题就称之为频繁模式挖掘问题。

### 2.1 频繁模式树

为了构造一个压缩的树结构用来有效地挖掘频繁模式，首先让我们来考察一个例子。

**例1** 给定一个事务数据库 DB(表1的头两列)，$\xi=3$。

表1 一个事务数据库样例

| TID | 项 | (排序后的)频繁项/PDB | 投影修改后的 PDB |
|---|---|---|---|
| 100 | f,a,c,d,g,i,m,p | f,c,a,m,p | f,c,a,m,p |
| 200 | a,b,c,f,l,o | f,c,a,b,o | f,c,a,b,o |
| 300 | b,f,h,j,m,p | f,b,m,p | $f,b_f,m_{bf},p$ |
| 400 | b,c,k,m,o,s | c,b,m,o | $c,b_c,m_{bc},o_{mbc}$ |
| 500 | a,f,c,e,l,n,o,p | f,c,a,o,p | $f,c,a,o_a,p$ |

首先，我们给出 FP-tree 的定义如下：

1. 由一个根结点(用“null”标记)，一组根结点的项前缀子树和一个项头表组成。

2. 项前缀子树的每个结点由项名、结点计数和结点链三部分组成。其中，结点计数表示通过该结点的事务的个数，结点链指向 FP-tree 下一个拥有相同项名的结点(当没有时，为空)。

3. 项头表的每个表目由项名和结点链头两个字段组成，其中结点链头指向 FP-tree 中第一个该项的结点。

根据该定义，可以给出 FP-tree 的构造算法：

**算法1**(FP-tree 构造算法)

**输入**：事务数据库 DB；最小支持度阈值 $\xi$。
**输出**：FP-tree
**方法**：

1. 扫描事务数据库 DB 一次，收集频繁项的集合 F 和它们的支持度。对 F 按支持度降序排序，结果为频繁项表 L。
2. 创建 FP-tree 的根结点，以“null”标记它。对于 DB 中的每个事务执行以下操作：
   对事务中的频繁项按照 L 中的顺序进行排序，排序后的频繁项表记为[p|P]，其中 p 是第一个元素，而 P 是剩余元素的表。调用 insert_tree([p|P],T)。该过程执行情况如下。如果 T 有子女 N 使得 N.项名=p.项名，则 N 的计数增加1；否则创建一个新结点 N，将其计数设置为1，链接到它的父结点 T，并且通过结点链结构将其链接到具有相同项名的结点。如果 P 非空，递归地调用 insert_tree(P,N)。

根据算法1，我们来构造例1的 FP-tree。首先，第1次扫描数据库，导出频繁项(1-项集)的集合 L=[f:4, c:4, a:4, b:3, m:3, o:3, p:3](“:”后的数字代表支持度)。排序后频繁项见表1第3列。

然后，创造树的根结点，用“null”标记。第2次扫描数据库 DB。对每一个事务创建一个分枝。例如，第一个事务“T100: f, a, c, d, g, i, m, p”按 L 的次序包含5个项{f, c, a, m, p}，导致构造树的第一个分枝⟨(f:1), (c:1), (a:1), (m:1), (p:1)⟩。对于第二个事务，由于其排序后的频繁项表⟨f, c, a, b, o⟩与已有的分枝路径⟨f, c, a, m, p⟩共享前缀⟨f, c, a⟩，因此前缀中的每个结点计数加1，只创建两个新的结点⟨(b:1), (o:1)⟩链接在⟨(f:2), (c:2), (a:2)⟩后。依此类推，得到 FP-tree 如图1。

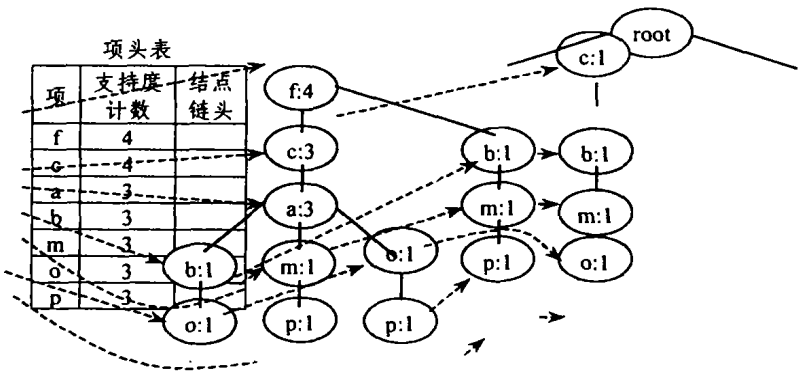


图1 例1中存放压缩的频繁模式信息的 FP-tree/PFP-tree

分析：从 FP-tree 的构造过程，我们可以看出整个过程只需要扫描三遍数据库 DB：第一遍收集频繁项的集合，第二遍用来对频繁项进行排序，第三遍逐个判断频繁项来构造 FP-tree。我们可以看出，在第三遍扫描中，假设数据库中一共有 $\omega$ 个频繁1-项，那么仅仅将所有事务的频繁项插入到 FP-tree 的代价就是 $O(\omega)$。如果是一个大型数据库或者 $\omega$ 比较大的话，也会造成算法的性能瓶颈。而且由于在树的构造过程中，项的判断和插入是有严格先后顺序的，是一个串行的计算过程，很难用并行或其他方法来改善性能。

### 2.2 用 FP-tree 来挖掘频繁模式

FP-tree 挖掘进行如下。由长度为1的频繁模式(初始后缀模式)开始，构造它的条件模式基(一个“子数据库”，由 FP-tree 中与后缀模式一起出现的前缀路径集组成)。然后，构造它的(条件)FP-tree，并递归地在该树上进行挖掘。模式增长通过后缀模式与由条件 FP-tree 产生的频繁模式连接实现。

**算法2**(FP-growth：在 FP-tree 中通过频繁模式增长挖掘频繁模式)

**输入**：基于算法1构造的 FP-tree；事务数据库 DB；最小支持度阈值 $\xi$。
**输出**：频繁模式的完全集。
**方法**：Call FP-growth (FP-tree, null)

```
Procedure FP-growth (Tree, α)
{
  if Tree 只含单个路径 P
  then for 路径 P 中结点的每个组合(记作 β) do
       产生模式 β∪α，其支持度 support=β 中结点的最小支持度;
```

```
else for each $a_i$ 在 FP-tree 的项头表(倒序)do {
    产生一个模式 β=$a_i$∪α,其支持度 support=$a_i$.support;
    构造 β 的条件模式基,然后构造 β 的条件 FP-tree $Tree_\beta$;
    if $Tree_\beta \neq \varnothing$
    then call FP-growth ($Tree_\beta$,β)}
}
```

根据算法2,对例1产生的FP-tree(图1)进行挖掘。由于初始参数α=null,没有单通路,因此在项头表中找到最后一项p(倒序搜索项头表)。P出现在图1中的FP树中的三个分枝(p的出现容易通过沿它的结点链找到):〈f:4,c:3,a:3,m:1,p:1〉,〈f:4,c:3,a:3,o:1,p:1〉和〈c:1,b:1,m:1,p:1〉。第一个路径显示项集"〈f,c,a,m,p〉"在数据库中出现一次。注意尽管项集〈f,c,a〉出现三次,而且〈f〉本身甚至出现四次,但它们与p一起只出现一次。因此在研究与p一起出现的项集时,只统计p的前缀路径〈fcam:1〉。同样地,p的第二个和第三个前缀路径分别为〈fcao:1〉和〈cbm:1〉。它们形成p的条件模式基。因此,p的条件FP-tree只包含单结点〈c:3〉;不包含{f,a,m,b,o},因为它们的支持度计数小于最小支持度计数3。导出的频繁模式为:{(cp:3)}。

依此类推,挖掘过程和结果总结在表2中。

表2 通过创建条件(子)模式基挖掘FP-tree

| 项 | 条件模式基 | 条件FP-tree | 产生的频繁模式 |
|---|---|---|---|
| p | {(fcam:1),(fcao:1),(cbm:1)} | 〈c:3〉 | cp:3 |
| o | {(fcab:1),(fca:1),(cbm:1)} | 〈c:3〉 | co:3 |
| m | {(fca:1),(fb:1),(cb:1)} | ∅ | ∅ |
| b | {(fca:1),(f:1),(c:1)} | ∅ | ∅ |
| a | {(fc:3)} | 〈fc:3〉 | fa:3,ca:3,fca:3 |
| c | {(f:3)} | 〈f:3〉 | fc:3 |
| f | ∅ | ∅ | ∅ |

分析:对于每一个频繁项$a_i$,FP-growth算法扫描一次FP-tree,产生一个小的条件模式基$B_{a_i}$,并在$B_{a_i}$的基础上形成一个更小的条件FP-tree,其中包含了所有后缀为$a_i$的频繁模式的信息。频繁模式的挖掘就变成对$a_i$的条件FP-tree的递归挖掘。正如算法1的分析中指出的那样,一个FP-tree的规模远远小于原有数据库的大小。同样地,基于$a_i$的条件FP-tree是在$B_{a_i}$的基础上构造的,所以它的规模也远远小于$B_{a_i}$,不可能大于$B_{a_i}$。而且一个条件模式基$B_{a_i}$的规模也通常远远小于原始的FP-tree,因为它是由只与一个频繁项$a_i$相关的前缀路径(条件模式基)所组成的。因此,系统代价远远小于Apriori算法迭代检查大量候选项集的开销。

从算法1、2中,我们可以看出FP-growth挖掘过程是一种分而治之的过程。而且,即使数据库可能产生指数级大量的频繁模式,FP-tree的规模还是很小,不会呈指数级增长。例如,对于一个长度为100的频繁模式"$a_1,\cdots,a_{100}$",FP-tree只会生成唯一一条长度为100的路径,如"$a_1\rightarrow\cdots\rightarrow a_{100}$"。而且,FP-growth算法可以导出所有的大约$10^{30}$个频繁模式(如果时间允许的话),如"$a_1,a_2,\cdots,a_1a_2,\cdots,a_1a_2a_3,\cdots,a_1\cdots a_{100}$"。

## 3. 投影频繁模式树(PFP-tree)的设计与构造

正如算法1和2.1节分析中所指出的那样,在构造FP-tree的过程中,将事务中的频繁项插入到FP-tree的过程是一个严格串行计算的过程。如果DB的数据量很大(即θ值很大导致ω值巨大时)时,会造成性能瓶颈。

也许有人提出采用分布并行计算的技术或者多CPU,一次并行地计算多个事务中的频繁项。但是这种方案很明显地提高了系统用于交换、合并控制信息的代价,算法的复杂度也大大增高,无法有效地解决问题;即使采用多CPU技术,提高硬件要求,对性能的改善也很有限。

这样就提出了一个问题:是否存在其它的算法可以减少这种FP-tree构造过程中的昂贵开销,有没有其它的方法或者技术可以有效地提高系统性能?

经过仔细研究和实验表明,我们发现运用以下巧妙的思想和方法,系统性能可以有效地得以改善:

首先,我们可以创建一个临时数据库用以存放所有排序后的频繁项(频繁1-项集)。这个临时数据库我们称之为投影数据库(Project Database,PDB),可以被用来运用投影技术来快速构造树的结点,而不是像传统的FP-tree那样逐个地计算每个频繁项。

然后,我们对PDB进行投影计算,一次投影两列,(j-1)和j列(投影过程执行如下:第一次投影PDB的第1列;然后每次投影两列,第(j-1)的第j列,将j值加1后再循环投影(j-1)和j列,直接投影完PDB所有的列)。第j列称之为当前结点列,用来统计所有不同频繁项的出现次数,第(j-1)列称之为父结点列,用以判断当前结点的父结点。这样,我们一次就可以计算出树的一层结点并把它们链接到对应的父结点上,不必要逐个地计算每个频繁项。因此,算法的性能只与树的深度有关,而不是数据库中所有频繁项的个数。

由于在对PDB进行投影的过程中,每次我们只投影两列,只保存了当前结点和它们的上一层结点信息,即第j列和第(j-1)列结点。这样就有可能出现一种异常情况:即当前几个结点(第j列结点)相同而且本属于树的不同分枝,但由于它们的父结点(对应的第(j-1)列结点)相同,系统将会导致错误的判断,认为它们在树中是同一个结点而简单地将出现计数累加。例如假设数据库中有两个事务[f,b,a]和[c,b,a],那么当投影第3列时就会得出[b,a:2],误认为结点a的父结点是b而且在同一个分枝中出现两次。实际上这两个a分属于两条不同的分枝,而且在各自的分枝上都只出现一次。因为虽然它们的父结点都是b(不同分枝上的b),但结点b的父结点则分别对应两个不同的结点f和c。为了解决这个问题,我们巧妙地引入了临时父结点下标(Temp Parent Index,TPI)的概念,当发现几个当前结点相同而它们的父结点不同时,我们就把父结点名作为临时下标加到当前结点上,以示区别这些当前结点隶属于树的不同分枝。这样当下一次投影时,就不会出现异常情况。

### 3.1 PFP-tree构造算法

投影频繁模式树(Project Frequent Pattern-tree,PFP-tree)的构造算法如下:

**算法3**(PFP-tree构造算法)

**输入**:事务数据库DB;最小支持度阈值ξ。
**输出**:PFP-tree
**方法**:
1. 扫描事务数据库DB一次,收集频繁项的集合F和它们的支持度。对F按支持度降序排序,结果为频繁项表L。
2. 根据L,对DB的每个事务中的频繁项按L中的顺序进行排序,排序后的结果存放在临时投影数据库PDB中。
3. 创建FP-tree的根结点,以"null"标记它。假设j为列号,对PDB中的每一列顺序执行以下操作,直至投影完PDB中所有的列:
   **If** j=1
   **Then** 对第1列作投影,并计算该列中每个频繁项的出现次数,结果形为[root,q:n](其中q为频繁项名,n为累计出现次数,root代表该结点的父结点是根结点),然后将这些频繁项[q:n]作

为 root 的子女结点加入到 PFP-tree 中；
Else do ｛①对第(j-1)和第 j 列两列作投影，统计所有父结点和当前列结点相同的二元频繁项组的出现次数，结果形为二元频繁项组[$p_x$，q:n]（其中 p 为第(j-1)列的父结点；x 为 p 的 TPI，如果 p 没有 TPI，则 x 为空；q 为当前列结点，n 为累计出现次数。）；
②对上面的计算结果[$p_x$，q:n]（所有的二元频繁项组）进行比较，如果两个二元频繁项组 q 相同，则将各自对应的 $p_x$ 作为 q 的 TPI 并写入到 PDB 中，结果形为[$p_x$，$q_y$:n]（y=px），否则不执行任何操作。
③然后将结点[$q_y$:n]作为对应父结点 $p_x$ 的子女结点加入到 PFP-tree 中。｝
j:= j+1；
4. 清除树中所有结点的临时数组下标和临时数据库 PDB（此步可以省略）。

我们根据算法3，再来对例1构造 PFP 树。第一步与算法1相同，导出频繁项(1-项集)的集合 L=[f:4，c:4，a:4，b:3，m:3，o:3，p:3]。第二步则对 DB 中的事务根据 L 重新排列频繁项，结果存放在 PDB 中（其实就是表1中的第三列）。

然后，开始创建 PFP-tree。首先，创建根结点，用"null"标记；第一次投影 PDB 的第一列，得出两个结点[root，f:4]和[root，c:1]，将它们作为根结点的子女结点加入到 PFP-tree 中（即树的第二层结点），这样我们一次投影就能计算出树的一层所有的结点；第二次投影 PDB 的第一和第二列，得出[f，c:3]，[f，b:1]和[c，b:1]，同时由于[f，b:1]和[c，b:1]的父结点不同，因此根据算法的第②步，将结点改写成[f，$b_f$:1]和[c，$b_c$:1]，并用 $b_f$ 和 $b_c$ 分别替换临时数据库 PDB 中对应的频繁项 b，然后再将这三个结点作为对应父结点的子女结点加入到 PFP-tree 中；第三次投影 PDB 的第二和第三列，得出[c，a:3]，[$b_f$，m:1]和[$b_c$，m:1]（注意：这里由于我们引入了 TPI 的概念，所以很容易区分[$b_f$，m:1]和[$b_c$，m:1]属于两条不同的分枝），同样根据算法第②步，将结点改写成[$b_f$，$m_{bf}$:1]和[$b_c$，$m_{bc}$:1]，依此类推，直至投影计算完最后一列（投影修改后的 PDB 见表1的第4列）。最后再清除所有的临时数组下标和临时数据库 PDB。（结果同图1）。

需要值得注意的是在投影第三和第四列时，得出[a，m:1]，[a，b:1]，[$m_{bf}$，p:1]，[$m_{bc}$，o:1]和[a，o:1]。结果结点[$m_{bc}$，o:1]和[a，o:1]满足算法第②步的条件，改写成[$m_{bc}$，$o_{mbc}$:1]和[a，$o_c$:1]；而对于结果结点[$m_{bf}$，p:1]和[$m_{bc}$，o:1]，原有的两个 TPI，{bf}和{bc}由于分属于两个不同的当前结点 o 和 p，不再符合算法第②步的条件，因此不再传递给当前结点，自动消失。而这也正是算法的巧妙之处，即 TPI 的生成和传递是有严格条件限制的，当条件不再满足，系统不再需要借助 TPI 区分父结点时，它们会自动消失，从而大大减少系统额外开销。

另外一个有趣的现象就是 TPI 实际上就是结点的前缀路径信息，而且它只保留了必要的前缀路径信息。

分析：从 PFP-tree 的构造过程我们可以看出整个过程也只需要扫描三遍数据库 DB：第一遍收集频繁项的集合，第二遍用来对 DB 的所有事务按照 L 重新排列频繁项，得出临时数据库 PDB；第三遍则使用投影技术快速地构造 PFP-tree，将数据库中所有的频繁项插入到 PFP-tree 的代价是 $O(\zeta)$，其中 $\zeta$ 是数据库中包含频繁项(1-项集)最多的事务的频繁长度，也就是生成树的层数，$\zeta \leqslant |L|$。这样，利用投影技术，在构造树的过程中，系统代价与数据库的大小和事务的个数无关，只与 PDB 的列数有关，大大降低了系统开销，提高了系统性能。唯一增加的开销是增加了一个临时数据库 PDB 的存储空间。

### 3.2 PFP-tree 的完整性、紧致性和有效性证明

我们可以从 PFP-tree 树的构造过程中得出几条重要的性质：

**引理3.1** 给定一个事务数据库 DB 和它的最小支持度阈值 $\xi$，它对应的 PFP-tree 包含了数据库中用于挖掘的所有的频繁模式信息。

证明：在 PFP-tree 的构造过程中，通过投影，PDB 的每一列的频繁项被变换成了 PFP-tree 中的一层结点。而且由于我们每次同时投影两列，保存了频繁项的父结点信息，同时通过引入 TPI，避免了数据不一致和无法判断发生的可能性，可以容易地将结点加入到 PFP-tree 中。因此，数据库中的所有频繁模式信息被完整地保存到了对应的 PFP-tree 中。

从另一个方面来理解，PFP-tree 中的一条路径/分枝就代表了一个事务，而且这个路径中的频繁项集有可能为多个事务所共享。这是因为所有的事务路径都必须从 root 结点开始，即第一列结点的父结点都是 root 结点，因此它们有可能共享前缀路径。由此我们可以得出以下引理。

**引理3.2** PFP-tree 的大小远小于原始数据库的大小，而且树的高度/层数就等于数据库中包含频繁项最多的事务的频繁长度。

证明：在 PFP-tree 的构造过程中，每次投影我们都将项名相同而且它们的父结点项名和 TPI 也相同的频繁项看作同一个结点，只是将它们的出现次数累加。这样，每次创建的结点数都小于该列中频繁项的个数，惟一没有根据频繁项创建的结点就是 root 结点，因此相对于原始数据库中的海量数据和事务个数，PFP-tree 是一种高度压缩的数据结构。它存储了挖掘频繁模式所必需的所有信息。由于它的一个单一路径"$a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_n$"就注册了所有形如"$a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_k$"的事务($1 \leqslant k \leqslant n$)，因此 PFP-tree 的规模远远小于原始数据库和用 Apriori 算法挖掘关联规则时所产生的候选项集。而且由于 PDB 事务中的频繁项是按照支持度降序排序，越频繁发生的项离根结点越近，这样，它们也就更可能被共享，这更加说明了 PFP-tree 结构的高度压缩特性。

树的高度其实就是作投影的次数，从算法的执行过程我们可以看出，树的高度就等于包含频繁项最多的事务的频繁长度。

**引理3.3** 相对于传统的 FP-tree 构造算法，PFP-tree 构造算法性能上有了本质的提高并具有更好的可伸缩性。

证明：从 FP-tree 的构造过程我们可以看出，它是基于对每个频繁项逐个进行判断来构造树的结点，即横向地对每个频繁项进行判断，每次生成一个新的结点或者将已有结点的出现次数加1，严格串行的计算过程不但使大量的机器内存和计算性能处于空闲状态，而且使得算法性能严重依赖于数据库的大小 $\omega$。而 PFP-tree 的构造算法则巧妙地运用投影技术，一次性地纵向读入 PDB 中的一列所有的频繁项进行比较，然后生成树的一层结点。充分利用了机器内存和计算能力。它与数据库的大小没有直接联系，主要只和投影的次数有关，根据引理3.2，投影的次数等于树的高度，也就是包含频繁项最多的事务的频繁长度。因此，算法性能上有了本质上的提高。

而且，在实际应用中，数据库事务个数的可伸缩性很大，甚至有可能是海量记录个数，而每个事务的长度则一般是个常量，相比于事务个数，两者往往相差好几个数量级。因此，PFP-tree 构造算法具有更好的可伸缩性。

另外，仔细研究 PFP-tree 构造算法，在每一次投影中，系统还增加了一个判断是否需要增加 TPI 的开销。但是，由于

此步骤是在投影合并完相同频繁项后，在具有相同当前项名的二元频繁项组之间进行操作，是一个简单的分类问题。而且在投影后，得出的二元频繁项组的个数本身就大大小于投影前频繁项的个数(事务个数)，因此，相对于 FP-tree 构造算法，这部分开销是值得的和可以忽略的。

**总结** 通过以上的分析和研究表明：如果我们将 PFP-tree 构造算法(算法3)和 FP-growth 算法(算法2)结合起来进行频繁模式挖掘，可以有效地提高数据挖掘的性能，具有良好的可伸缩性。这主要体现在以下三个方面：

1. 运用投影和树结构技术，可以将一个大型数据库压缩到一个高密度的、更小的数据结构(PFP-tree)中，能有效地避免重复扫描数据库的昂贵代价；而且在构造 PFP-tree 的过程中，算法的性能不受数据库大小的影响。

2. 基于 PFP-tree 的挖掘采用频繁模式增长的方法，避免了迭代产生大量候选项集的代价。

3. 采用对频繁项进行分割和分而治之的方法来挖掘频繁模式，将繁重的挖掘任务分解为一组小的多的任务——对各个频繁项的条件模式树的挖掘，能有效地减少搜索空间。

目前，对基于 FP-tree 挖掘的研究包括基于 SQL、高伸缩性的 FP 树结构的研究与应用，基于约束的频繁模式挖掘，以及运用该项技术来挖掘序列模式[12]、最大模式[13]和部分频率[14]等相关领域。如何将改进的 PFP-tree 方法运用到这些研究领域以及如何与实际的工业、商业大型数据相结合，则是我们进一步研究的方向。

## 参 考 文 献


1 Agrawal R, Srikant B. Fast algorithms for mining association rules. In VLDB'1994. 487～499

2 Grahne G, Lakshmanan L, Wang X. Efficient mining of constrained correlated sets. In ICDE'2000

3 Klemettinen M, et al. Finding interesting rules from large sets of discovered association rules. In CIKM'1994. 401～408

4 Lent B, Swami A, Widom J. Clustering association rules. In ICDE'1997. 220～231

5 Ng E, Lakshmanan L V S, Han J, Peng A. Exploratory mining and pruning optimizations of constrained associations rules. In SIGMOD'1998. 13～24

6 Park J S, Chen M S, Yu P S. An effective hash-based algorithm for mining association rules. In SIGMOD'1995. 175～186

7 Sarawagi S, Thomas S, Agrawal R. Integrating association rule mining with relational database systems. Alternatives and implications. In SIGMOD'1998. 343～354

8 Savasere A, Omiecinski E, Navathe S. An efficient algorithm for mining association rules in large database. In VLDB'1995. 432～443

9 Srikant R, Vu Q, Agrawal R. Mining association rules with item constraints. In KDD'1997. 594～605

10 Han Jiawei, Kamber M(加). 数据挖掘概念与技术. 机械工业出版社，2001. 158～161

11 Han Jiawei, Pei Jian, Yin Yiwen. Mining Frequent Patterns without Candidate Generation. http://www.cs.sfu.ca/～peijian/personal/publications

12 Agrawal R, Srikant S. Fast algorithms for mining association rules. In VLDB'1994. 487～499

13 Bayardo R J. Efficiently mining long patterns from databases. In SIGMOD'1998. 85～93

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


(上接第125页)

评价一个签名认证系统的性能需要分析两种错误率：FRR (False Reject Rate)——真实签名被系统错误拒绝的比例和 FAR(False Accept Rate)——伪造签名被系统错误接受的比例[2]。通常，一个系统的性能用 EER(Equal Error Rate)——FRR 与 FAR 相等时的值来评价。本系统测试了3345个伪笔签名和584个亲笔签名，得到图3所示的测试结果。

从测试结果看，本系统的 ERR 在4%之下，FAR 和 FRR 的最小值均接近2%，性能是令人较为满意的。

## 5. 进一步的工作

签名者的签名风格是随时间变化的。如果参考集保持不变，系统性能会随时间下降，因此有必要对参考集更新。一种办法是用户主动地更新，但这样安全性较差。最好是系统自动更新，最简单的更新策略是把最早的参考签名替换掉，但这不是最佳策略。参考集的更新策略是一个值得进一步研究的有趣的课题。

另外什么是最佳的特征集，全局特征与局部特征怎样结合都需要进一步的研究。

## 参 考 文 献


1 Nalwa V S. Automatic on-line signature verification. Proceedings of the IEEE, 1997, 85: 215～239

2 Griess F D. Project Report: On-line Signature verification. May 2000

3 Dullink H, et al. Implementing a DSP Kernel for Online Dynamic Handwritten Signature Verification Using the TMS320 DSP Family. Dec. 1995

4 Lee L L, et al. Reliable on-line human signature verification systems. IEEE Transactions on Pattern Analysis and Machine Intelligence, 1996, 18: 643～647

5 Yang L, et al. Application of hidden markov models for signature verification. Pattern Recognition, 1995, 28(2): 161～170

6 Wirtz B. Stroke-based time warping for signature verification. In: Proc. of the Intl. Conf. on Document Analysis and Recognition, 1995, 1: 179～182

7 Dolfing J, et al. On-line signature verification with hidden markov models. In: Intl. Conf. on Pattern Recognition, vol. 2, IEEE, 1998, 2: 1309～1312

8 Hastie t, et al. A model for signature verification: [tech. rep]. AT&T Laboratories, Feb. 1992