一种基于模式树的频繁项集快速挖掘算法

战立强,刘大昕,张健沛
ZHAN Li-qiang,LIU Da-xin,ZHANG Jian-pei

哈尔滨工程大学 计算机学院,哈尔滨 150001
Department of Computer Science,Harbin Engineering University,Harbin 150001,China
E-mail:zhanlq_66@yahoo.com.cn

ZHAN Li-qiang,LIU Da-xin,ZHANG Jian-pei.Fast algorithm for frequent item-set mining based on pattern tree.Computer Engineering and Applications,2007,43(11):15-16.

Abstract: Pattern tree is the most frequently used data structure in frequent item-set mining.By using pattern tree,database can be effectively compressed into main memory,and the subsequence mining task can be completed in main memory.To make further improvement on the scalability of the algorithm,we make a further study on the pattern tree,and propose a new algorithm called FP-DFS based on the study.FP-DFS simplifies the mining processes through applying various operations on pattern tree. The experiments show that FP-DFS has good efficiency in frequent item-set mining.

Key words: association rule;frequent item-set mining;scalability;pattern tree

摘 要:模式树是目前频繁项集挖掘最常用的数据结构,使用模式树可以有效地将数据库压缩于内存,并在内存中完成对频繁项集的挖掘。为了进一步提高频繁项集挖掘算法的可扩展性,对模式树进行了细致的研究,在此基础上提出了一种挖掘频繁项集的新算法,FP-DFS 算法。该算法通过对模式树的各种操作简化了对频繁项集的搜索过程。实验表明,该算法对于频繁项集挖掘具有比较高的效率。

关键词:关联规则;频繁项集挖掘;可扩展性;模式树
文章编号:1002-8331(2007)11-0015-02 **文献标识码**:A **中图分类号**:TP18;TP311;TP391

频繁项集挖掘是数据挖掘中的一项基础性的工作,它是关联规则挖掘的一个关键步骤,也可以应用于诸如分类、聚类、预测等数据挖掘任务中。为了降低频繁项集挖掘算法的时间和空间复杂性,J.Han 等提出了著名的 FP-growth 算法,该算法是基于内存算法的典型代表,它采用一种特殊的数据结构 FP-tree(又称为模式树)将数据库压缩于内存,然后在内存中完成频繁项集的挖掘[1]。FP-growth 算法的效率优于一般的类 Apriori 算法,它的缺点是需要使用条件模式基递归地构造 FP-tree,不仅占用大量的内存空间,而且一次迭代过程结束后通常只能得到几个频繁模式,因此算法的效率有待进一步提高。在 FP-growth 之后出现了大量的基于 FP-tree 的算法,其中一些算法不再利用条件模式基递归构建 FP-tree,然而这些算法的过程都很复杂,过多的操作产生较大的时间开销[2,3]。本文提出了一种新的挖掘频繁模式算法,FP-DFS(Searching on FP-tree with Deep First Strategy)算法。该算法使用 FP-tree,但不再使用条件模式基递归地构造 FP-tree,而是尽可能使用遍历和递增构建操作,通过使用这些操作实现对频繁模式的高效率的搜索,并且减少了对内存的占用。算法过程十分简单,实验表明,对于稠密数据库,FP-DFS 算法的时空复杂性远低于已有的同类算法。

1 问题的提出

频繁项集的概念可描述如下:设 $I=\{a_1,a_2,\cdots,a_n\}$是一个项目集,一个事务 T 是由 I 中的项目组成的项目集。一个事务数据库 D 是一个事务集。令 X 代表一个项目集。事务 T 支持 X,当且仅当 $X\subseteq T$。D 中支持 X 的事务的数目称为 X 的支持度,记为 $support(X)$。如果 $support(X)$不低于预先设定的最小支持度 $\sigma_{\min}$,则称 X 为频繁项集。

许多频繁项集挖掘算法都以 FP-tree 作为其基本的数据结构。FP-tree 的每一个结点代表一个项目,结点之间有结点链相连,形成一种树状的结构。在 FP-tree 上,除根结点外的每个结点均含有两个域:label 域和 count 域,分别记录结点所代表项目的名称和支持度值。为了方便挖掘,FP-tree 中有一个头表,表中的每一行代表一个项目,由头表出发的结点链将 FP-tree 中所有相同标号的结点连接成一条结点链。关于 FP-tree 的构建方法详见文献[1],对于例 1 中的数据库,得到 FP-tree 如图 1 所示。

例 1 假设事务数据库 D 包含的事务有{A,B,C}、{A,B,D,E}、{B,C,D}、{B,C,D,E}、{C,D},D 的 FP-tree 如图 1 所示。

关于 FP-tree 有如下定义:

定义 1 FP-tree 上 Root 的子结点称为子根结点。

基金项目:国家自然科学基金(the National Natural Science Foundation of China under Grant No.60673131)。

作者简介:战立强(1966-),男,讲师,博士研究生,主要研究方向为数据挖掘;刘大昕(1941-),男,教授,博士生导师,主要研究方向为数据库技术、数据挖掘;张健沛(1956-),男,教授,博士生导师,主要研究方向为数据库技术、数据挖掘。

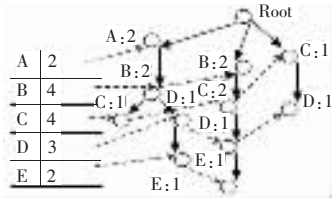

图 1 FP-tree

定义 2 对于 FP-tree 上的结点 a_i，如果 a_i 的子结点的 *count* 值小于 a_i.*count*，或者 a_i 是叶结点，则称由子根结点到 a_i 的路径为一个单一模式，称 a_i.*count* 为此单一模式的计数值。

2 FP-DFS 算法

频繁项集的挖掘问题可以转化为对频繁项集的搜索问题，FP-DFS 算法每次迭代得到以某个项目开头的所有频繁项集。假设 $I=\{a_1,a_2,\cdots,a_n\}$已按项目出现频率的降序排序，且 I 中的项目都是频繁的，令 L 代表事务数据库中所有频繁项集的集合，L_i 代表以 I 中的项目 a_i 开头的所有频繁项集的集合，则 $L=\sum L_i$。假设已经得到$\{L_{i+1},L_{i+2},\cdots,L_n\}$，将 a_i 加到其中每个频繁项集的前面，得到 L_i 的候选项集集合 C_i。由 Apriori 性质容易得到如下性质[4]。

性质 1 $\forall p\in L_i$，有 $p\in C_i$，即 $L_i\subseteq C_i$。

FP-DFS 算法以 FP-tree 作为其基本的数据结构，它的基本思想是首先将事务数据库 D 压缩为内存中的 FP-tree，即 T-tree。此后按照相反次序逐个处理 I 中的项目，每次迭代得到以某个项目开头的所有频繁项集。假设经过若干次迭代后得到集合 $L_{\succ i}=\{L_{i+1},L_{i+2},\cdots,L_n\}$，用其中的项集构建一棵 FP-tree，称为 C-tree。对于 C-tree 上每一个从子根结点出发的路径，如果将 a_i 加到路径的前面，则得到候选集 C_i，即关于 C-tree 有如下性质：

性质 2 令 C-tree 由 $L_{\succ i}$ 构建，它的所有由子根结点出发的前缀构成集合 *prefix-set*，$\forall p\in$*prefix-set*，令 $p'=a_i\cup p$，则 $p'\subseteq C_i$。

为了节省内存空间，C-tree 不需要头表和结点链。在用 $L_{\succ i}$ 构建 C-tree 后，接下来用 T-tree 上的模式对 C-tree 上的前缀计数。方法是沿着头表中的 a_i 结点链遍历 T-tree，得到所有以 a_i 为根的子树，然后统计这些子树对 C-tree 上的前缀的计数。为了减少 FP-tree 的构建操作，在构建 C-tree 时采用增量式构建方法。令 C_{i+1}-tree 代表得到 L_{i+1} 后 C-tree 的状态，显然将 L_i 添加到 C_{i+1}-tree 即可得到 C_i-tree。算法对 C_{i+1}-tree 计数得到 L_i，然后将 L_i 添加到 C_{i+1}-tree 得到 C_i-tree。下面是 FP-DFS 算法的完整过程：

FP-DFS 算法在第一次扫描数据库时统计长度为 1 的频繁模式，存入 L，将 I 中的非频繁项集去除，并按项目出现频率的降序排序。然后第二次扫描数据库，将事务中的项目排序后构建 T-tree，按照与 I 中的项目相反的顺序，逐个处理 I 中的项目，即算法在第 $n-i+1$ 次迭代中得到所有以 a_i 开头的频繁项集。算法在第一次迭代时，构建一棵只含有结点 a_n 的 C-tree，如果在 T-tree 上存在以 a_n 为子根结点的子树，则将该子树删除。在第二次迭代时，先将 C-tree 中结点 a_n 的 *count* 域置 0，沿着头表中的 a_{n-1} 结点链遍历 T-tree，用得到的 a_{n-1} 子树对 C-tree 中结点 a_n 计数，如果 $a_n.count\geqslant\sigma_{\min}$，则将频繁项集 $a_{n-1}a_n$ 存入 L，同时将 $a_{n-1}a_n$ 插入 C-tree，如果在 T-tree 上存在以 a_{n-1} 为子根结点的子树，则将该子树删除。算法的第 $n-i+1$ 次迭代过程与此类似，即先将 C-tree 中各结点的 *count* 域置 0，沿着头表中的 a_i 结点链遍历 T-tree，用得到的以 a_i 为根结点的子树对 C-tree 计数，如果 p 是由 C-tree 得到的频繁项集，则将频繁项集 $a_i\cup p$ 存入 L，同时将 $a_i p$ 插入 C-tree，最后将 C-tree 上以 a_i 为子根结点的子树删除。

在 C-tree 上得到频繁项集的方法是，如果某结点 a_i 的 $a_i.count\geqslant\sigma_{\min}$，则由子根结点到 a_i 的路径构成频繁项集，支持度为 $a_i.count$。

```
Input:D,σmin
Output:L
for each t∈D
    统计长度为 l 的频繁项集;
除去 I 中的非频繁项目并排序;
for each t∈D
    Make-tree(T-tree,t);//*构建 T-tree *//
for each 项目 ai in I{//*按照与 I 中项目相反的顺序处理 *//
    将 C-tree 上所有结点的 count 值置为 0;
    沿着头表中的 ai 结点链遍历 T-tree;
    得到以所有 ai 为根结点的子树 ai-tree;
    for each ai-tree
        使 ai-tree 对 C-tree 中所有前缀计数;
    for each C-tree 上 count 值大于 σmin 的路径 P {
        P'=ai P;
        将 P'添加到 L;
        Make-tree(C-tree,P');}
    从 T-tree 上删除以 ai 为子根结点的子树 sub(ai)-tree;
}
```

3 实验结果分析

为了验证 FP-DFS 算法的有效性，通过实验将 FP-DFS 算法的性能与目前已有的频繁项集挖掘高效算法进行比较。算法使用 VC++实现，实验环境为：Genuine Intel T2050 1.6 GHz，1 G RAM，Windows XP Professional。实验使用的数据集为 mushroom、chess、connect，均来自 UCI Machine Learning Database Repository，为频繁项集挖掘研究中常用的测试数据集，3 个数据集的参数如表 1 所示。

表 1 数据集

事务数据库名称	事务数目	项目数目	事务平均长度(最大长度)
connect	67 557	129	43(43)
mushroom	8 124	120	23(23)
chess	3 197	75	37(37)

通过实验将 FP-DFS 算法的性能与 FP-growth 算法以及 AFOPT 算法的性能进行对比。AFOPT 算法是由文献[3]提出的频繁模式挖掘的高效算法，曾获 fimi04(Frequent Itemset Mining Implementations 2004)的优胜奖，是基于 FP-tree 的频繁模式挖掘的代表算法。FP-growth 算法实现来自“C++Frequent Itemset Mining Template Library”，AFOPT 算法实现来自 fimi04。

图 2 为 chess 数据集下各算法的实验结果，在本实验环境下，FP-growth 算法在支持度低于 30%的情况下由于内存空间不足而无法运行，AFOPT 算法在支持度低于 20%的情况下也由于运行时间过长而失去意义。图 3 为 mushroom 数据集下各算法的实验结果，在本实验环境下，FP-growth 算法在支持度低

(下转 207 页)