一种基于事务树的快速频繁项集挖掘与更新算法*)

阮幼林　李庆华　杨世达
(华中科技大学计算机科学与技术学院　武汉430074)

摘　要　挖掘频繁项集是数据挖掘研究中的关键问题。基于FP-Tree的挖掘及其更新算法无需生成候选项目集因而效率明显高于Apriori类算法，但FP-Tree结构存在动态维护复杂、必须两次扫描数据库等缺点。因此，本文提出一种基于事务树Trans-Tree的新算法。该算法通过引入一种新结构—事务树Trans-Tree来压缩存放数据的相关信息且易于更新，挖掘算法只需对数据库扫描一次，而且更新算法只需对新增数据扫描一次，无需扫描原始数据，从而大大提高了频繁项集的挖掘和维护效率。

关键词　频繁项集，频繁模式树，事务树，更新

A Fast Algorithm Based on Trans-Tree for Mining and Updating Frequent Itemsets

RUAN You-Lin　LI Qing-Hua　YANG Shi-Da
(Department of Computer Science and Technology, HuaZhong University of Science and Technology, Wuhan 430074)

Abstract　Mining frequent patterns is a key problem in data mining research. Although mining based on FP-Tree achieves better performance and efficiency than Apriori-like algorithms because of avoiding costly candidate generation, it still have problems such as update of FP-Tree and require two scans of the database. Therefore, this paper proposes a new method that designs a new structure called Trans-Tree, which stores all of the information in a highly compact form and updates easily. Thus, mining requires only one scan of the database and updating Trans-Tree needs one scan of the new data only without scanning the existing data.

Keywords　Frequent itemsets, Frequent pattern tree, Trans-tree, Updating

1　引言

关联规则是由Agrawal等人首先提出来的一个重要的KDD研究课题，它反映了大量数据项集之间有趣的关联或相关联系。在挖掘过程中，频繁项集的计算是整个算法的瓶颈。因此发现频繁项集是关联规则挖掘中的关键技术和步骤。近年来，人们对频繁项集的挖掘算法进行了大量的、深入的研究工作。在众多算法中，以Agrawal等人提出的Apriori算法[1]最为著名。该算法采用候选项目集生成-筛选方法，必须耗费大量时间处理规模巨大的候选项目集，同时必须多次扫描数据库，对候选项目集进行筛选。为提高Apriori算法的有效性，人们已经提出了许多Apriori算法的变形[1~3]，融合了许多技术，如散列项集计数、事务压缩、划分、选样和动态项集技术等。众多Apriori类算法虽然大幅度压缩了候选集的大小，但仍然需要产生大量候选集，并可能需要重复扫描数据库。因此，针对Apriori类算法存在的问题，Han等人提出了FP-Tree和相应的FP-Growth[4]算法。该算法无需生成候选项目集，将挖掘长频繁项目集的问题转换成递归挖掘一些短频繁项目集，然后连接后缀。它使用最不频繁的项目作后缀，提供了好的选择性。因此，该方法显著地缩小了搜索空间，有效地避免了组合爆炸，挖掘效率明显提高。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。但FP-Tree结构存在动态维护复杂、不利于频繁项集的更新、必须两次扫描数据库等缺点。为此，本文提出一种基于Trans-Tree的新算法。该算法通过引入一种中间结构—事务树Trans-Tree来压缩存放数据的相关信息而且易于更新，FP-Tree完全可以基于Trans-Tree建立，而无需再对数据库进行扫描。因此挖掘算法只需对数据库扫描一次，而且更新算法只需对新增数据扫描一次而无需扫描原始数据，大大提高了频繁项集的挖掘和维护效率。

2　相关概念

2.1　频繁项目集

设$I=\{i_1,i_2,\cdots,i_m\}$是m个不同项目的集合。给定事务数据库DB，对于项目集$X\subseteq I$，X在DB中的支持数是指D中包含X的事务数，记为$X.count$。X在DB中的支持度是指DB中包含X事务的百分比，记为$X.sup$。如果X的支持度不小于用户给定的最小支持度阈值$minsup$，则称X为DB中的频繁项目集。如果X包含k个项目，那么又称为频繁k-项目集。项目集中项目的个数称为项目集的维数或长度，频繁1-项目集简称频繁项目。

2.2　事务树Trans-Tree

事务树Trans-Tree与频繁模式树FP-Tree非常类似，只不过Trans-Tree记录的是DB中的每个事务Trans中的所有项，而FP-Tree只记录DB中的每个事务Trans中的频繁项。频繁模式树FP-Tree中事务中的项按支持度计数降序排列，而事务树Trans-Tree中事务中的项按大小顺序排列。因此，

*)基金项目：国家自然科学基金项目(60273075)。阮幼林　博士研究生，主要研究方向为数据挖掘，并行与分布式计算。李庆华　教授，博导，主要研究方向为并行与分布式计算，高性能算法。杨世达　博士研究生，主要研究方向为演化计算，高性能算法。

两者只是记录的项及其顺序不同。在 Trans-Tree 中，每个节点仍由4个域组成：节点名称 *itemname*、节点计数 *count*、节点链 *nodelink* 及父节点指针 *parent*。

性质1 事务数据库 DB 的每次事务记录在事务树 Trans-Tree 的一条路径中。

性质2 事务树 Trans-Tree 中每条路径的计数表示事务数据库 DB 中相同事务的个数。

性质3 Trans-Tree 的深度为相应的事务数据库 DB 中事务的最大长度。

3 频繁项集挖掘算法

通常基于 FP-Tree 的挖掘算法的主要步骤为：(1)扫描 DB 一遍得到各项目的频度，根据最小支持度 *minsup* 得到频繁项目；对频繁项目按其频度由大到小排列成表 L，形成头表；(2)再次扫描 DB，对每一条交易中的所有频繁项目，按表 L 中的次序插入到 FP-Tree 中；(3)调用 FP_Growth 算法对 FP-Tree 进行挖掘。因此，原有基于 FP-Tree 的挖掘算法必须对数据库进行两次扫描。而数据库中通常数据量较大，扫描两次的代价较高。因此，为了提高挖掘效率，必须尽可能减少数据库的扫描次数。因此，本文提出一种只需对数据库扫描一次的挖掘算法 TFP-Growth。该算法通过引入事务树 Trans-Tree 来压缩存放数据库中所有数据的相关信息而不仅仅只有频繁项集的信息，FP-Tree 完全可以基于 Trans-Tree 建立，而无需再对数据库进行扫描。因此，该挖掘算法可分为以下三个步骤：

1. 扫描事务数据库 DB 一次，构造事务树 Trans-Tree，同时产生所有项的集合 F 及其支持数；

2. 基于事务树 Trans-Tree 和最小支持度阈值 *minsup* 构造 FP-Tree；

3. 调用 FP_Growth 算法进行挖掘。

3.1 事务树 Trans-Tree 的构造算法

输入：事务数据库 DB

输出：事务树 Trans-Tree 和项集合 F 及其支持数

事务树 Trans-Tree 的构造算法如下：

(1)创建 Trans-Tree 的根节点，以"*root*"标记它。扫描事务数据库 DB 一次，构造事务树 Trans-Tree。同时产生所有项的集合 F 及其支持数；

(2)对于 DB 中的每个事务 *Trans* 作如下处理：

①事务 *Trans* 中的项按大小次序排列。设排序后的项列表为$[p|P]$，其中 p 是第1个项目，而 P 是剩余项目的列表；

②调用 *insert_tree*$([p|P],T)$：如果 T 有子女 N 使得 $N.node\text{-}name = p.item\text{-}name$，则 N 的计数应增加1；否则创建一个新节点 N，将其名称 *node-name*、计数 *node-count* 应分别设置为 p 和1，由父节点指针 *node-parent* 链接到它的父节点 T，并通过节点链 *node-link* 将其链接到具有相同名称 *node-name* 的节点。如果 P 非空，递归调用 *insert_tree*(P,N)。

3.2 频繁模式树 FP-Tree 的构造算法

本文中 FP-Tree 是在事务树 Trans-Tree 的基础上创建而不是基于事务数据库 DB，由于事务树 Trans-Tree 中的每条路径表示事务数据库 DB 中所有相同事务而不是单个事务，因此需要对 FP_Tree 的构造进行一些相应的修改。

输入：事务树 Trans-Tree 和项集合 F 及其支持数

输出：频繁模式树 FP-Tree

频繁模式树 FP-Tree 的构造算法如下：

(1)创建 FP-Tree 的根节点，以"*null*"标记它。对项集 F 按支持数降序排列，结果为频繁项集 L。

(2)对 Trans-Tree 中的每条路径作如下处理：

①路径的计数应为该路径尾节点的计数；

②选择路径中的频繁项，并按 L 中的次序排列。设排序后的频繁列表为$[p|P]$，其中 p 是第1个项目，而 P 是剩余项目的列表；

③调用 *insert_tree*$([p|P],T)$：如果 T 有子女 N 使得 $N.node\text{-}name = p.item\text{-}name$，则 N 的计数改为 $p.count$(而不是1)；否则创建一个新节点 N，将其名称 *node-name*、计数 *node-count* 应分别设置为 p 和 $p.count$(而不是1)，由父节点指针 *node-parent* 链接到它的父节点 T，并通过节点链 *node-link* 将其链接到具有相同名称 *node-name* 的节点。如果 P 非空，递归调用 *insert_tree*(P,N)。

3.3 FP_Tree 的挖掘算法

FP_Tree 的挖掘算法仍通过调用 *FP_Growth*(*FP_Tree*, *null*)实现。该过程无需修改，实现如下：

```
Procedure FP_Growth(Tree,α)
if Tree 含单个路径 P then
  for 路径 P 中节点的每个组合(记作 β) do
    产生频繁模式 β∪α，其支持度 sup=β 中节点的最小支持度;
else for each αi 在 Tree 的头部 do
    { 产生一个模式 β=αi∪α，其支持度 sup=αi.sup;
      构造 β 的条件模式基;
      构造 β 的条件 FP-树 Treeβ;
      if Treeβ≠Φ then 递归调用 FP_Growth(Treeβ,β);
    }
```

4 应用实例

对表1所示的交易数据库 DB，数据项集 $I=\{I1,I2,I3,I4,I5\}$，设最小支持度计数为2，则由 Trans-Tree 构造算法可得事务树 Trans-Tree，如图1所示，并有频繁项集 $L=[I2:7, I1:6, I3:6, I4:2, I5:2]$；然后根据 Trans-Tree 可建立 FP_Tree，如图2所示；最后基于 FP_Tree 调用 FP_Growth 算法可得所有的频繁模式，如表2所示。

表1 交易数据库 DB

交易	数据项
T1	I1,I2,I5
T2	I2,I4
T3	I2,I3
T4	I1,I2,I4
T5	I1,I3
T6	I2,I3
T7	I1,I3
T8	I1,I2,I3,I5
T9	I1,I2,I3

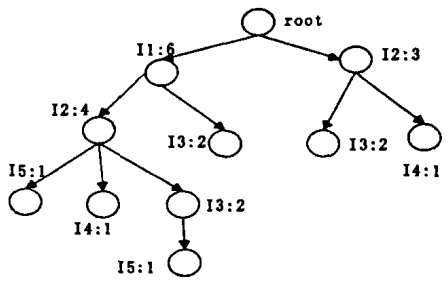


图1 事务树 Trans-Tree

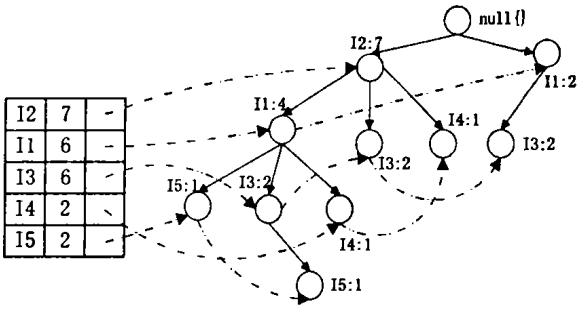


图2 频繁模式树 FP-Tree(最小支持度计数为2)

表2 交易数据库 DB 的频繁模式

项目	频繁模式
I5	I2 I5:2,I1 I5:2,I2 I1 I5:2
I4	I2 I4:2
I3	I2 I3:4,I1 I3:4,I2 I1 I3:2
I1	I2 I1:4

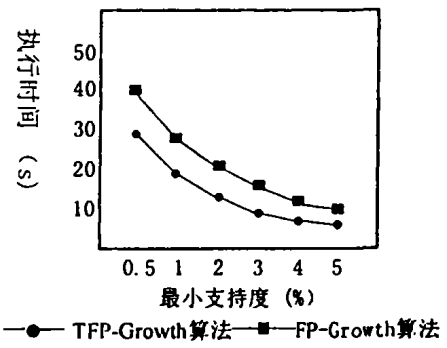
5 频繁项集的增量更新算法

由于 FP-Tree 仅存放频繁项集的信息,当新数据加入或支持度发生改变时,可能原有的频繁项集不再频繁或原有的非频繁项集成为频繁项集。因此,算法必须建立新的 FP-Tree,即使新增数据库很小,而建立新的 FP-Tree 仍必须对原数据库和新增数据库扫描两次。通常原数据库积累了大量数据,扫描两次的代价太大,因而传统的基于 FP-Tree 的更新算法的效率很低。很明显,原有的基于 FP-Tree 的更新算法的最大问题在于必须对原数据库进行两次扫描。针对该问题,为了提高挖掘和更新效率,本文引入事务树 Trans-Tree 来压缩存放数据库中所有数据的相关信息而不仅仅只有频繁项集的信息。由于事务树 Trans-Tree 已存放原数据库中所有数据的相关信息,因此只需对新增数据库扫描一次,把新增数据加入事务树 Trans-Tree 中,然后基于新的事务树 Trans-Tree 构造 FP-Tree 进行更新挖掘。

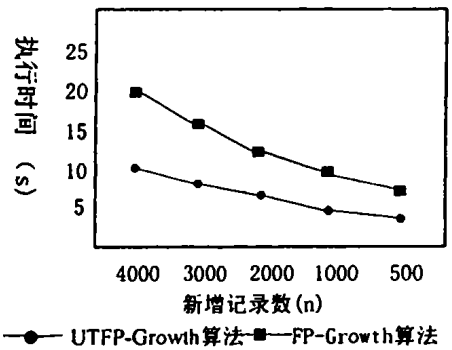
更新算法 UTFP-Growth 也分为三个步骤:第一步为:扫描新增事务数据库 db 一次,每个事务 *Trans* 通过 *insert_tree*(*Trans*,*T*)插入到原事务树 Trans-Tree,并更新项集 *F* 及其支持数。后两步与挖掘算法 TFP-Growth 相同。

6 算法实现与比较

我们用 VC++6.0 在内存256M,CPU 为 Pentium III-733MHZ,操作系统为 Windows 2000的机上实现了 TFP-Growth 和 UTFP-Growth 算法并进行了性能测试。利用 http://www.ics.uci.edu/~mlearn/MLSummary.html 上提供的蘑菇数据库(mushroom database)来进行实验。该数据库有8124条记录,记录了蘑菇的23种属性。图3(a)显示了在不同的最小支持度下算法的性能比较。由于 TFP-Growth 算法只需扫描数据库一次,因此 TFP-Growth 算法的执行时间比 FP-Growth 算法要少,图3(a)也说明了这一点。为了验证增量更新算法 UTFP-Growth 的有效性,我们随机抽取8000个记录,并分为两部分:原始 DB(4000个记录)和新增 db(4000个记录),$minsup^{DB\cup db}==minsup^{DB}=2\%$,从 db 中抽取不同的记录数(500,100,2000,3000,4000)作为 db 的不同增量情况,对更新算法 UTFP-growth 进行测试,结果如图3(b)所示。由于 UTFP-growth 算法只需对新增 db 扫描一次而不需对原始 DB 扫描,因此其效率极大提高。



(a)



(b)

图3 算法的执行时间

结束语 针对 FP-Tree 结构必须两次扫描数据库、动态维护复杂、不利于频繁项集的更新等缺点,本文提出一种快速挖掘与更新算法。该算法引入一种事务树 Trans-Tree 来压缩存放数据的相关信息,不仅只需对数据库扫描一次,而且更新算法只需对新增数据扫描一次而无需扫描原始数据,大大提高了频繁项集的挖掘和维护效率。

参 考 文 献

1 Agrawal R, Srikant R. Fast Algorithm for Mining Association Rules. In VLDB'94, Sep. 1994. 489~499

2 Park J S, Chen M-S, Yu P S. An Effective Hash Based Algorithm for Mining Association Rules. In: Proc. ACM SIGMOD Int. Conf. on Management of Data, 1995. 175~186

3 Chen M Y, Han J, Yu P. Data Mining: An Overview from a Database Perspective. IEEE Transactions on Knowledge and Data Engineering, 1996, 8(6): 866~883

4 Han J, Pei J, Yin Y. Mining Frequent Patterns Without Candidate Generation. In: Proc. ACM-SIGMOD, Dallas, TX, May 2000

5 Agarwal R, Aggrawal C, Prasad V V V. A Tree Projection Algorithm for Generation of Frequent Itemsets. Journal of parallel and Distributed Computing, 2000. 427~434

6 Wang K, Tang L, Han J, Liu J. Top Down FP-Growth for Association Rule Mining. In: Proc. of 6th Pacific-Asia conference on Knowledge Discovery and Data Mining, 2002

7 Cheung W, Zaiane O R. Incremental Mining of Frequent Without Candidate Generation or Support Constraint. In: Proc. of 7th Database Engineering and Applications Symposium(IDEAS'03)

8 Huang H, Wu X, Relue R. Association Analysis with One Scan of Databases. In: Proc. of Pacific-Asia Conf. PAKDD 2002. 334~340