第35卷 第15期 计 算 机 工 程 2009年8月
Vol.35 No.15 Computer Engineering August 2009
·软件技术与数据库· 文章编号：1000—3428(2009)15—0097—03 文献标识码：A 中图分类号：TP311.52

基于事务树的最大频繁项集挖掘算法

张忠平，郑为夷
（燕山大学信息科学与工程学院，秦皇岛 066004）

摘 要：针对 Apriori 算法在寻找频繁项集的过程中需多次扫描数据库、侯选项集过多、支持度计算过于复杂等问题，提出 TT-Apriori 算法。该算法将事务数据库转化成事务树，通过遍历事务树能直接快速地找到最大频繁项目集。简化支持度的计算，避免对整个数据库的扫描和大量的连接步骤，从而提高挖掘效率。

关键词：最大频繁项集；TT-Apriori 算法；事务树；向量内积

Maximal Frequent Itemsets Mining Algorithm Based on Transaction Tree

ZHANG Zhong-ping, ZHENG Wei-yi
(College of Information Science and Engineering, Yanshan University, Qinhuangdao 066004)

【Abstract】Aiming at the shortage of Apriori algorithm in find of frequent items such as numerous search-designate database set too many times and gennerate too many candidate itemsets, this paper proposes the TT-Apriori algorithm. This algorithm maps the tings-datebase into transaction-tree. Using the transaction tree can quickly find the maximal frequent itemsets. In the meamwhile it can simplify the calculation of support and avoid the scanning of the entire database and a large number of connecting steps to improve the efficiency of the mining.

【Key words】maximal frequent itemsets; TT-Apriori algorithm; transaction tree; vector in plot

1 概述

关联规则挖掘是从大量的数据中挖掘出有价值描述数据项之间相互关系的有关知识。在众多关联规则挖掘算法中，文献[1]提出的 Apriori 算法是最基本和常见的算法。在 Apriori 算法中，对频繁项集的挖掘是基本步骤，也是关键步骤[2]。在频繁项集的挖掘中会产生大量的候选项集，根据频繁项集向上封闭的性质，最大频繁项集已包含所有的频繁项集[3]。此外，很多应用中只须挖掘出最大频繁项集即可。本文从 Apriori 算法寻找频繁项集的过程出发，提出一种高效动态的基于事务树最大频繁项集挖掘算法：TT-Apriori(Transaction Tree-Apriori)算法。该算法扫描事务数据库一次，首先将事务数据库映射成布尔型矩阵，然后由布尔型矩阵组建事务树，通过对事务树节点的遍历处理，得到最大频繁项集。

2 TT-Apriori算法

2.1 性质和定义

定义 1 (最大频繁项集)[4] 若频繁项集 X 的所有超集都是非频繁项集，则称 X 为最大频繁项集。

性质 1 Apriori 性质:频繁项集的任何子集都是频繁项集。

性质 2 频繁(k+1)-项集只能产生于包含项目数大于 k 的事务中。

证明：频繁(k+1)-项集包含 k+1 个项目，不可能出现在包含 k 个项目的事务中。

定义 2(向量内积) 对于任意 2 个 m 维向量 $\boldsymbol{\alpha}=(x_1, x_2,\cdots, x_m)$, $\boldsymbol{\beta}=(y_1, y_2, \cdots, y_m)$，则 $\boldsymbol{\alpha}$ 和 $\boldsymbol{\beta}$ 的内积定义为

$$\langle \boldsymbol{\alpha}, \boldsymbol{\beta} \rangle = \sum_{i=1,\, j=1}^{m} x_i y_j$$

定义 3(事务树) 事务树为完全二叉树，事务树每个节点存储与某个事务相关的信息，表示形式为如图 1 所示。

S	*N*	*T*	*L*	*R*	*P*	*C*

图 1 事务树表示形式

节点数据类型定义如下：

```
Typrde struct Node{
int S;//事务支持度
int N;//事务编号
trans T//事务项内容
Struct Node *L, *R, *P;
int C; //事务包含项数}//事务树节点；
```

节点在事务树中的位置由该节点表示的事务项在经由事务数据库预处理后得到的布尔型矩阵中的位置决定。

2.2 事务数据库预处理

首先将事务数据库映射成布尔型矩阵。对于任意事务数据库 D，可以将其映射成布尔型矩阵 $\boldsymbol{R}$。其中，$\boldsymbol{R}$ 每一行代表一个事务，每一列代表一个项目。如果第 i 个事务中包括项目 j，则矩阵相应位置的值 r_{ij}=1，否则 r_{ij}=0。扫描一次事务数据库后就可以得到相应的布尔型矩阵。其次剔除非频繁项列，对矩阵的列元素求和，计算每个项目的支持度，并剔除布尔矩阵中支持度小于 min_*sup* 的非频繁项列。

基金项目：国家自然科学基金资助项目(60773100)；河北省教育厅科研计划基金资助项目(2006143)

作者简介：张忠平(1972－)，男，副教授、博士后，主研方向：数据挖掘，XML 数据库，网格技术；郑为夷，硕士研究生

收稿日期：2008-12-10 **E-mail：**zhengweiyi321_2005@163.com

假设最小支持度为 2，以文献[5]中的稀疏事务数据库为例，如表 1 所示，经过预处理后可以得到如表 2 所示的布尔型矩阵 ***R***。

表 1　事务数据库 *D*

Tid	*Items*
100	*A, B, C, E*
200	*A, B, D*
300	*B, C, D, E, F*
400	*B, D*
500	*C, D, E*

表 2　预处理后的布尔型矩阵 *R*

Tid	*A*	*B*	*C*	*D*	*E*
100	1	1	1	0	1
200	1	1	0	1	0
300	0	1	1	1	1
400	0	1	0	1	0
500	0	0	1	1	1

3　算法思想及描述

3.1　项集支持度的计算

由算法涉及的性质和相关工作的过程可知每个频繁项集中包含的项目完全出现在事务数据库 *D* 中的某些事务中。假设事务数据库 *D* 对应的布尔型矩阵 ***R***，记 ***R***=(α_1, α_2, …, α_n)$^{\mathrm{T}}$，$\boldsymbol{\alpha}_i$ 为 ***R*** 在实数域上的 *m* 维行向量，*i*=1, 2, …,*n*。每个向量对应一个事务。在 ***R*** 中，计算 *c* 值为 *k* 的行向量(事务)$\boldsymbol{\alpha}_i$ 的支持度计数时，即计算该行向量在 ***R*** 中出现的次数。由内积定义，$\boldsymbol{\alpha}_i$ 在 R 中重复出现的条件是 ***R*** 中存在行向量 $\boldsymbol{\alpha}_j$，使$<\boldsymbol{\alpha}_i, \boldsymbol{\alpha}_j>=<\boldsymbol{\alpha}_i, \boldsymbol{\alpha}_j>=k$。而由性质 2 可知，*c* 值为 *k* 的行向量 $\boldsymbol{\alpha}_i$ 只能会在 *c* 值为与其相等的或 *c* 值大于 *k* 的事务中出现。因此，在判定某向量(事务)的支持度时，只须将 $\boldsymbol{\alpha}_i$ 与 *c* 值大于和等于 *k* 的向量作内积计算判定即可。在事务树中，*c* 值为 *k* 的节点在遍历事务树时只需与 *c* 值不小于 *k* 的节点做内积即可。在遍历事务树前可根据各个节点的 *c* 值和编号来得出所有需要进行内积运算的节点组合，以避免重复的运算。但是，若某一事务不是频繁项集，但其子集在 ***R*** 中出现的次数满足用户设定的最小支持度计数，且该子集并不构成某一事务。即该子集不是某一事务树节点，按上述支持度计算的方法则可能遗漏频繁项集。为避免这种情况，可再开辟一段空间，记为 new space，来记录 2 个向量做内积时各项累加前所表示的向量，即某一子集。例如，向量(11101)与向量(01111)做内积时累加前所表示的向量为(01101)，即项集{*B, C, E*}。这时，若满足判断条件，则将向量(01101)加入到新开辟的空间中。对于这种新生成的向量所表示的项集，可采用以下思路计算支持度。

为每个项目建立一个比特向量，与项目 *i* 相关的比特向量记为 $\boldsymbol{BV}_i$，若某项目在第 *n* 个事务中出现，则将比特向量的第 *n* 个位置记为 1，否则记为 0。本例中 $\boldsymbol{BV}_A$=(11000), $\boldsymbol{BV}_B$=(11110), $\boldsymbol{BV}_C$=(10101), $\boldsymbol{BV}_D$=(01111), $\boldsymbol{BV}_E$=(10101), $\boldsymbol{BV}_F$=(00100)。由比特向量 $\boldsymbol{BV}_i$ 和支持度的定义可知，项目集{$i_1, i_2, \cdots, i_m$}的支持度就是同时包含项目 $i_1, i_2, \cdots, i_m$ 的事务数，即 $\boldsymbol{BV}_1, \boldsymbol{BV}_2, \cdots, \boldsymbol{BV}_m$ 连续作内积的结果。

3.2　算法处理基本思想

TT-Apriori 算法首先从事务中搜索最大频繁项集，再从事务作内积生成的子集中搜索最大频繁项集。算法的第 1 步产生事务树；第 2 步以每个节点为起始节点遍历事务树搜索最大频繁项集；第 3 步在 new space 中搜索最大频繁项集。本文以表 1 所示的布尔型矩阵 ***R*** 为例，来说明算法各步的处理思想。假设最小支持度计数为 2。

(1)由表 1 所示的布尔型矩阵 ***R***，按定义 3，得到事务树如图 2 所示。为简略起见，在图中省略指针项和支持度项。

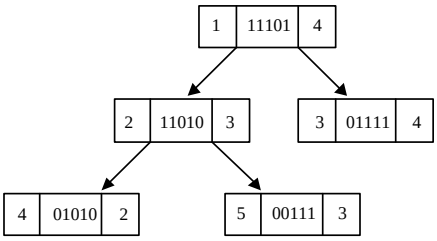

图 2　事务树

(2)从各个节点出发遍历事务树，搜索频最大繁项集。根据各个节点的 *c* 值和编号由 3.1 节中提到的原则得出需要进行内积计算的节点组合，图 3 所示的事务树中需要进行内积运算的节点组合有(1,3) (2,1) (2,3) (2,5) (4,1) (4,2) (4,3) (4,5) (5,1) (5,3)。每个节点初始支持度计数为 1(自身计一次)。根据性质 1，若某事务所代表的项集是频繁项集，其任意子集都是频繁项集。那么，在某节点满足最小支持度计数后，则停止遍历，输出该节点表示的项集，同时不再将其生成的子集插入到 new space 中。

从节点 1 出发遍历，根据上述支持度计算方法，节点 1 只需与节点 3 进行内积运算：(11101)⊙(01111)=3 小于节点 1 自身的 *c* 值(用⊙表示内积运算)，因此，节点 1 所代表的事务不是频繁项集，但是生成了向量(01101)，且节点 1 所表示的项集不是频繁项集，将其插入到 new space 中。

从节点 2 开始遍历事务树，须与节点 2 做内积的有节点 1，节点 3 和节点 5。节点 2 与节点 1：(11010)⊙(11101)=2，小于节点 2 自身的 *c* 值，可知节点 2 所代表的事务目前不是频繁项集，但是生成了向量(11000)。节点 2 与节点 3：(11010)⊙(01111)=2，同理节点 2 目前不是频繁项集，但是生成了项集(01010)。节点 2 与节点 5：(11010)⊙(00111)=1，可知节点 2 不是频繁项集。生成了项集(00010)，该项集表示单一项目 *D*，因为在事务数据库预处理的过程后已得到了所有的频繁 1-项集，所以项集(00010)不插入到 new space 中。由于节点 2 所代表事务不是频繁项集，因此将其遍历事务树时生成的子集(11000)，(01010)插入到 new space 中。

从节点 4 出发遍历事务树，节点 4 与节点 1：(01010)⊙(11101)=1，小于节点 4 自身的 *c* 值，可知节点 4 所代表的事务目前不是频繁项集，生成向量(01000)，该项集表示单一项目 *B*。同节点 2 的情况相同，该子集不插入 new space 中。节点 4 与节点 2：(01010)⊙(11010)=2，等于节点 4 自身的 *c* 值，则节点 4 的支持度计数增加为 2，等于最小支持度计数。其所代表的事务为频繁项集，输出其所代表事务{*B, D*}。

从节点 5 出发遍历事务树，节点 5 须与节点 1 和节点 3 进行内积运算。节点 5 与节点 1：(00111)⊙(11101)=2，小于节点 5 自身的 *c* 值，则节点 5 所代表的事务目前不是频繁项集，生成了项集(00101)。节点 5 与节点 3：(00111)⊙(01111)=3，等于节点 5 自身的 *c* 值，则节点 5 的支持度计数为 2，等于最小支持度计数，其所代表的事物为频繁项集，输出节点 5 向量(00111)所代表事务{*C, D, E*}。由于节点 5 所代表事务是频繁项集，因此，将其遍历时生成的子集(00101)不插入到 new space 中。

至此，在事务树中搜索最大频繁项集的过程结束。产生

的频繁项集为{*B, D*}与{*C, D, E*}。

(3)在 new space 中搜索频繁项集。

根据在事务树中搜索的过程，可得 new space 中的向量为(01101), (11000)和(01010)。分别计算这 3 个项集的支持度计数。项集(01101)的支持度计数为：$\boldsymbol{BV}_B \odot \boldsymbol{BV}_C \odot \boldsymbol{BV}_E$ = (11110)⊙(10101)⊙(10101)=2，等于最小支持度计数，则可得向量(01101)所代表的项集是频繁项集，输出向量(01101)所代表事务{*B, C, E* }。同理输出向量(11000)所代表的事务{*A, B*}与向量(11000)所代表的事务{*B, D*}。

至此，在 new space 中搜索最大频繁项集的过程结束，产生的频繁项集为{*B, C, E* }, {*A, B* }, {*B, D* }。综上所示，得到事务数据库 *D* 中包含的最大频繁项集为：{*C, D, E*}, {*B, C, E*}, {*A, B*}, {*B, D*}。这与 Apriori 算法得到的结果相同，但由于没有产生大量的候选项集同时无须进行连接操作，因此提高了算法效率。

3.3 算法描述

根据上述的算法思想，提出了 TT-Apriori 算法。

算法 1 基于事务树的最大频繁项集挖掘 TT-Apriori 算法

输入 布尔型矩阵 ***R***，最小支持度 min_*sup*

输出 最大频繁项集

```
TT-Apriori：(R, min_sup)
Begin
(1) int c[i]; //存储节点 c 值
      s[i]; //记录节点代表事务的支持度
    trans t[i];//记录布尔型事务
//计算各节点的 c 值
(2) for(i=1;i≤n;i++)
(3) {for(j=1;j≤n;j++)
(4)   c[i]= c[i]+bool[i][j]}
//由各节点的 c 值生成遍历组合
(5) comp[i][j]=0;//记录遍历组合
(6) for(i=1;i≤n;i++)
(7) {for(j=1;j≤n;j++)
(8)   {if i≠j&c [i]≤c [j];
(9)   comp[i][j]=1}}// comp[i][j]=1 的 i, j 值组合为遍历组合
//以每个节点为起始节点按照遍历组合遍历事务树
(10) PT[k]= ∅ ; k=1//存储内积运算各项累加前所表示的项集
(11) for(i=1;i≤n;i++) //i 为起始节点编号
(12)   {s[i]=1; k=1;j=1; //每个节点的起始支持度计数为 1。
(13)   while(s[i]＜min_sup&j≤n)
{ if comp[i][j]=1; //i, j 为遍历组合
{vect(t[i]，t[j]); //计算向量内积
(14)       swhich vect(t[i], t[j])
(15)       {case c[i] ： s[i]++   break; //增加支持度计数
(16)        case 1     ： break
(17)        default    ： PT[k]=prevect(t[i], t[j]); k++;}
//存储内积运算各项累加前所表示的项集}
j++; }}
(18)      if s[i]≥min_sup
(19)       output(t[i])
(20)       delate(every PT[k]);
(21)      else insert(every PT[k]) ; }
//某事务不是频繁项集时将生成的子集插入 new space 中
//在 new space 中搜索频繁项集
(22)   for (all itemsets in new space)
(23)    {countsup(itemset)
//利用连续内积运算计算项集支持度计数
(24)      if countsup(itemset) ≥min_sup
(25)        output(itemset) }//结束
End
```

3.4 基于事务树的数据库动态更新策略

在频繁项集挖掘中，对频繁项集的管理和维护是非常重要。当数据库发生变化或者支持度更新时，已有的频繁项集可能不再是频繁项集，同时，可能产生新的频繁项集。

本文提出的基于事务树的频繁项集挖掘算法中的事务树可以被实时地构造与修改。当有新的事务加入数据库时，对事务树进行插入更新，此时只要通过再执行一次算法 1 就可以得到新的频繁项集。当最小支持度发生变化时，只要随其变化调整参数 min_*sup* 即可。

4 算法分析及实验

4.1 算法分析

假设事务数据库包含 *n* 个事务 *m* 个项目。时间复杂度分析：整个搜索过程分为 3 个阶段：

(1)扫描事务数据库并生成事务树，时间复杂度为 $O(n)$。

(2)以每个节点为起始节点按照遍历组合访问其他节点，最坏的情况下 *n* 个节点的 *c* 值相等，共需进行 $n(n\text{-}1)/2$ 次内积运算。时间复杂度为 $O(n^2)$。

(3)在 new space 中搜索频繁项集。最坏的情况下每隔几点都与其他节点产生要插入到 new space 中的子集。则 new space 中最多存储 $n(n\text{-}1)/2$ 个项集，利用连续向量内积计算这些项集的支持度计数，时间复杂度为 $O(n^2)$。综上，TT-Apriori 算法时间复杂度为 $O(n^2)$，其时间复杂度远小于传统的 Apriori 算法和类似算法。

空间复杂度分析：算法在空间上的代价主要是存储事务树和 new space。这两者的大小由事务数据库的长度决定。组建事务树所占内存空间为 $O(n)$，new space 在最坏的情况下存储 $n(n\text{-}1)/2$ 个项集，空间复杂度为 $O(n^2)$。综上，TT-Apriori 算法时间复杂度为 $O(n^2)$。相比于传统的 Apriori 算法随最小支持度减小和频繁项集长度增加而产生呈指数级增长的大量候选项集，提高了空间利用率。

4.2 实验分析

为验证算法的性能，用 VC++6.0 编程语言实现了 Apriori 算法和本文的 TT-Apriori 算法。在内存为 512 MB，CPU 主频为 3.00 GHz 的计算机上进行实验。实验的数据是由 IBM 的合成数据产生器生成，事务的最大长度为 20，事务的平均长度为 5，平均最大模式长度为 4，事务数量为 20 000。图 3 为 2 种算法对给定的不同最小支持度的比较结果。通过实验结果可以看出，在相同最小支持度的情况下，本文算法的运行时间要明显低于传统的 Apriori 算法且在支持度发生变化时，TT-Apirori 算法受影响较小。

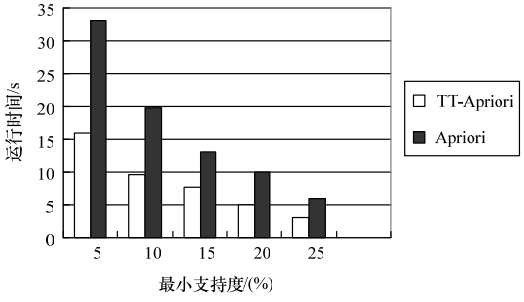

图 3 不同支持度下运行时间比较

(下转第 120 页)