第 34 卷 第 5 期 计 算 机 工 程 2008 年 3 月
Vol.34 No.5 Computer Engineering March 2008
·软件技术与数据库· 文章编号：1000—3428(2008)05—0060—03 文献标识码：A 中图分类号：TP301.6

基于散列的关联规则 AprioriTid 改进算法

俞燕燕[1,2]，李绍滋[1]
(1. 厦门大学计算机科学系，厦门 361005；2. 浙江信息工程学校，湖州 313000)

摘 要：发现频繁项集是关联规则挖掘应用的关键，针对采用Apriori类的候选项目集生成-检验方法导致候选项目集产生的代价很高问题，该文提出一种基于散列的快速AprioriTid改进算法，在AprioriTid算法的基础上采用基于候选项L_k地址的哈希映射方法，提高了算法的执行效率。

关键词：关联规则；频繁项目集；AprioriTid 算法；散列

Improvement of AprioriTid Algorithm for Association Rules Based on Hash Technology

YU Yan-yan[1,2], LI Shao-zi[1]
(1. Computer Science Department, Xiamen University, Xiamen 361005; 2. School of Zhejiang Information Engineering, Huzhou 313000)

【Abstract】Finding frequent itemset is a pivotal technology and stage in association rules mining application. Most studies adopt Apriori-like candidate set generation-and-test approach, but candidate set generation is still costly. This paper proposes an improved AprioriTid algorithm to improve the algorithmic executive efficiency, which is based on candidate set L_k address-mapping approach of Hash technology.

【Key words】association rules; frequent item set; AprioriTid; Hash

1 概述

关联规则由Agrawal等人提出[1-2]，其核心是基于两阶段频繁项集思想的递推算法。关联规则挖掘算法的设计可以分解为 2 个子问题：

(1)找到所有支持度大于最小支持度的项集(itemset)，这些项集称为频繁项集(frequent itemset)；

(2)使用第(1)步找到的频繁项集产生期望的规则，这些规则必须满足最小支持度和最小可信度。

发现频繁项集是关联规则挖掘应用中的关键技术和步骤，在众多频繁项目集的算法研究中，以Agrawal等人提出的Apriori算法最为著名，该算法使用逐层搜索的迭代技术，频繁k-项集用于探索频繁(k+1)-项集，并使用Apriori性质压缩搜索空间，巧妙地解决了算法的效率问题。其后的数据挖掘算法大多建立在Apriori算法基础之上，如AprioriTid, AprioriHybird[3-5]等。如上述算法中，计算项目集的支持数是发现频繁项目集中最耗时的工作，因此，目前关联规则的发现算法主要研究如何快速有效地提取频繁项目集，其焦点为：(1)能对项目集进行充分的剪枝，尽量最小化有用的项目集；(2)尽量少地扫描交易数据库T，从而提高算法的效率。

2 AprioriTid算法[6]

AprioriTid 算法是基本 Apriori 算法的一种扩展，其基本原理是当一个交易不包含长度为 k 的大项集，那么必然不包含长度为 k+1 的大项集，因此，可以将这些项目集移去，减少在下一遍扫描中需要进行扫描的项目集个数。例如，买 AB 或买 BC 的人少于 0.3，则买 ABC 的人必少于 0.3。

该算法仅在第 1 次对事务数据库D扫描时计算候选项目集的支持度，其他各次扫描用其上一次扫描生成的候选事务数据库C_k计算候选频繁项目集的支持度。C_k中每一个成员的形式为<TID,{L_k}>，其中，每个X_k是一个事务中潜在的频繁k-项目集，对应一个标示符TID。当k=1 时，C_1对应于数据库D；当k>1 时，由频繁(k-1)-项集集合L_{k-1}自连接，得L_k，根据基本原理进行剪枝，移去不满足条件的k的项集。这样，当k值较大时，C_k中条目数量比数据库中的事务数少，从而减少了 I/O 操作时间和需要扫描数据库的大小。

3 散列表原理

散列表[7]是线性表中一种重要的存储方式和检索方法。散列表算法的基本思想是：由结点的关键码值决定结点的存储地址，即以关键码值k为自变量，通过一定的函数关系$h(k)$，计算对应结点的存储地址，并将结点存入该地址中。

散列表最重要的一个指标是负载因子，即散列表中结点数目与表中能容纳的总结点数的比值，它描述了散列表的饱和程度，负载因子越接近 1.0，内存的使用效率越高，元素的寻找时间越长，同样，负载因子越接近 0.0，元素的寻找时间越短，但内存的浪费越大。

4 基于散列的 AprioriTid 的改进算法

4.1 AprioriTid_Hash 函数定义

AprioriTid_Hash 算法的 Hash 函数定义为

$$h(\{x, y\})=((\text{order of } x)\cdot t+(\text{order of } y))\bmod c$$

其中，order of x, order of y表示产生候选项L_k的 2 个(k-1)-项目x, y在L_{k-1}中的序列号；t, c为 2 个系数，根据L_{k-1}的规模(k-1-频繁项集的个数)调整，记作 i 。

基金项目：浙江省湖州市级科技计划基金资助项目“基于统计学的社会性网络行为研究”(2006YZ15)

作者简介：俞燕燕(1978－)，女，硕士研究生，主研方向：数据挖掘；李绍滋，教授、博士生导师

收稿日期：2007-04-10 **E-mail：**livelyyyy@sina.com

首先给出L_{k-1}中的频繁项目编序号，使每个频繁项目的序号唯一，然后定义t, c，确保每个产生候选项L_k的地址唯一，解决冲突的方法如下：

若 i<10，则 c=(i-1)·10+ i, t=10；

若 i<100，则 c=(i-1)·100+ i, t=100；

若 i<1000，则 c=(i-1)·1000 + i, t=1000；

……

4.2 AprioriTid_Hash 算法

本文采用基于候选项L_k的地址的哈希映射方法，由C_{k-1}中的候选(k-1)-项集产生频繁k项集L_k时，依次对C_{k-1}每个事务集内的L_{k-1}进行连接join操作，产生所有的k-项集，通过$h(\{x,y\})$计算，并将具有相同地址的候选项L_k散列到散列表中相应的存储队列中，如果存储队列的计数器最终数值小于最小支持数，那么此队列中的k-项目集不会成为频繁项目集，可从候选k项集中删除。具体描述如下：

输入 交易数据库 T，最小支持度 minsup

输出 T 中的频繁项目集 L

L_1={frequent 1-itemset}; //生成长度为 1 的频繁项目集

C_1=database D; //将事务集转换为以TID为索引的候选事务集

for each 所有条目 entries $t \in C_1$ { //确定事务 t, TID 中包含的 //候选

C_t={c $\in L_1$ |(c-c[k]) $\in$ t.set-of-itemset $\wedge$ ((c-c[k-1]) $\in$ t.set-of-itemset)};

if ($C_t \neq \phi$) then $C_1^{'}$+=<t.TID, C_t >}

$C_1 = C_1^{'}$

for (k=2, $L_{k-1} \neq \phi$;k++){

L_k=Apriori_gen(L_{k-1}，C_{k-1}); //产生新的频繁k-项目集

$C_k^{'} = \phi$;

for each 所有条目 entries $t \in C_{k-1}$ { //确定事务 t,TID 中包含的 //候选

C_t={c $\in L_k$ |(c-c[k]) $\in$ t.set-of-itemset $\wedge$ ((c-c[k-1]) $\in$ t.set-of-itemset)};

if ($C_t \neq \phi$) then $C_k^{'}$ +=<t.TID, C_t >}

$C_k = C_k^{'}$ }

return $L = \cup_k L_k$

Apriori_gen(L_{k-1}:frequent (k-1)-itemsets; C_{k-1}:候选事务数据库; minsup: support)

定义 hashset={} //hashset 表，初值为 0，累计相同地址的 //候选项个数

定义L_k={} //存放频繁k-候选项目集合，包括支持度

for each itemset $l_i \in L_{k-1}$ {order of l_i=i //记录l_i在L_{k-1}中的序号}

for each 所有条目 entries $t \in C_{k-1}$ { //分别对事务 t 中包括的 //项目连接

for each itemset $c_1 \in C_t$ {

for each itemset $c_2 \in C_t$ {

if $(c_1[1]=c_2[1]) \wedge (c_1[2]=c_2[2]) \wedge \cdots \wedge (c_1[k-2]=c_2[k-2]) \wedge (c_1[k-1]<c_2[k-1])$

then{ x=order of c_1

y=order of c_2

order= h({x,y})= (x*t+y)%c //哈希函数，由 i 确定 h({x,y}) //的参数 c,t

hashset[order]+=1 //根据 order 值给 hashset[order]赋值

$c = c_1 \times c_2$; //连接

L_k [c]= hashset[order]}}}}

for each itemset c $\in L_k$ {if L_k [c]< support then delete c; //剪枝(pruning)}

return L_k;

4.3 AprioriTid_Hash 算法实例分析

设 T={100,200,300,400}，I={1,2,3,4,5}，$minsup$=50%，TID 表示交易标识。

首先，把事务集转换为以TID为索引的候选事务集C_1，最小支持度，为 2 生成长度为 1 的频繁项目集L_1，删除C_1中小于最小支持度的项目，记作候选事务集 $C_1^{'}$。过程如下：

C_1

TID	项目集
100	1,3,4
200	2,3,5
300	1,2,3,5
400	2,5

L_1

项目集	支持度
{1}	2
{2}	3
{3}	3
{5}	3

$C_1^{'}(C_1)$

TID	项目集
100	{{1},{3}}
200	{{2},{3},{5}}
300	{{1},{2},{3},{5}}
400	{{2},{5}}

第 1 次扫描 $C_1^{'}(C_1)$，分别对事务TID内的项目进行自连接操作，给L_1中的项目编号得L_1中项目序列表，建立hashset表。过程如下：

L_1中项目序列

项目集	序列号
{1}	1
{2}	2
{3}	4
{5}	3

hashset表

100 {1 3}

200 {2 3},{2 5},{3 5}

300 {1 2},{1 3},{1 5},{2 3},{2 5},{3 5}

400 {2 5}

由于i<10，得t=10, c=34；Hash函数：

$h(x,y)$=((or der of x)·10+(order of y))mod 34

根据Hash函数建立如下hashset表：

{2 5}

{3 5} {1 5} {1 3} {2 5} {2 3}

{3 5} {1 2} {1 5} {1 3} {2 5} {2 3}

2	1	2	2	3	2	0

hashset 表

0 …12 … 13 … 14 … 23 … 24 … 33 Hash地址

hashset表中的值表示散列到hashset[order]的项目的数目，同时能得到频繁 2-项集集合L_2。删除C_2中小于最小支持度的项目，记作候选事务集C_2' ，过程如下：

L_2

项目集	支持度
{1 3}	2
{2 3}	2
{2 5}	3
{3 5}	2

$C_2^{'}(C_2)$

TID	项目集
100	{{1 3}}
200	{{2 3},{2 5},{3 5}}
300	{{1 3},{2 3},{2 5},{3 5}}
400	{{2 5}}

第 2 次扫描C_2'(C_2)，同样对L_2中的项目编号，分别对事务TID内的项目进行自连接操作，建立如下hashset表：

L_2中项目序列

项目集	序列号
{2 5}	1
{1 3}	2
{2 3}	4
{3 5}	3

因为{2 3},{2 5}产生{2 3 5}，hashset 表如下：

200 {2 3 5}

300 {2 3 5}

由于 i<10，因此 t=10, c=34；Hash 函数为

$h(x,y)$=((order of x)·10+(order of y))mod 34

hashset 表建立如下：

{2 3 5}

{2 3 5}

0	0	0	0	2	0	0

hashset 表

0 1 2 3 … 31 32 33 Hash 地址

最后得到如下频繁 3-项集集合L_3：

L_3

项目集	支持度
{2 3 5}	2

⇨

$C_3'(C_3)$

TID	项目集
200	{{2 3 5}}
300	{{2 3 5}}

结束。

4.4 AprioriTid_Hash 算法时间复杂度分析

算法AprioriTid是利用 $L_{k-1}\cdot L_{k-1}$ 来产生候选集C_k，当L_{k-1}很大时，得到的C_k会非常大，无论怎样剪枝优化，效率都不会得到很大提高，时间复杂度为$O(n^2)$。之后扫描 C'_{k-1} 中的事务，并计算C_k中每个候选项目集的支持度，找出其支持度不小于最小支持度的频繁项目集L_k，时间复杂度为$O(mn)=O(n^2)$。依此类推，随着 C'_{k-1} 中的事务及L_{k-1}的减少，时间复杂度随之降低。

而基于Hash的AprioriTid算法能降低时间复杂度的原因是它分别依次对C_{k-1}每个事务集内的L_{k-1}进行连接join操作，大大减少了n的值，于是时间复杂度$O(n^2)$也降低了。在散列的同时得到了频繁k-项集集合L_k，省去了找出其支持度不小于最小支持度的频繁项目集L_k所要花费的时间。

5 算法实现与比较

文本以 SQL Server 2000 系统的 NorthWind 数据库中的产品销售数据进行模拟实验，把 Order Detail 表中的记录导入 MYSQL 中，将 OrderID 和 ProductID 字段记为 *TID* 和 *IID*，总共 2 155 条记录，用 Ruby(http://www.rubycentral.com/ref/ref_c_array.html#sort)实现了 AprioriTid 与 AprioriTid_Hash。本文对两者的性能进行如下比较。

5.1 频繁集的数目与最小支持度的关系

测试最小支持度对产生频繁集的数目的影响，分别取不同最小支持度：0.002, 0.003, 0.005, 0.008, 0.010, 0.015, …, 0.050。2 种算法所得的频繁集是一致的。图 1 表明，频繁集的数目随着最小支持度的增加而减少。

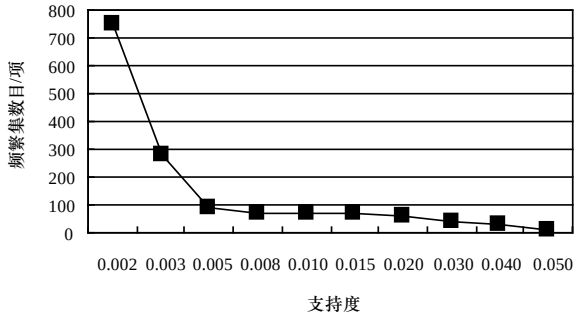

图 1　频繁集的数目与最小支持度的关系

5.2 最小支持度与运行时间、空间复杂度的关系

实验取不同的最小支持度，比较 AprioriTid 与 AprioriTid_Hash 算法的时间开销和占用内存空间大小情况。

(1)系统的时间开销随着最小支持度的增加而减少。因为最小支持度越高，产生的候选项将越少，扫描数据库的次数的时间开销减少。如图 2 所示。

(2)2 种算法的时间开销，AprioriTid_Hash 算法在时间的提高上非常明显。以支持度 0.002 为例，散列表的负载因子分别为

2-频繁项集的负载因子≥630/7677 · 100%=8.2%；

3-频繁项集的负载因子≥ 39/629630 · 100%=0.1%；

4-频繁项集的负载因子≥2/3839 · 100%=0.05%。

负载因子越低速度越快，改进算法快了 52 s，速度提高了 83.87%，如图 2 所示。

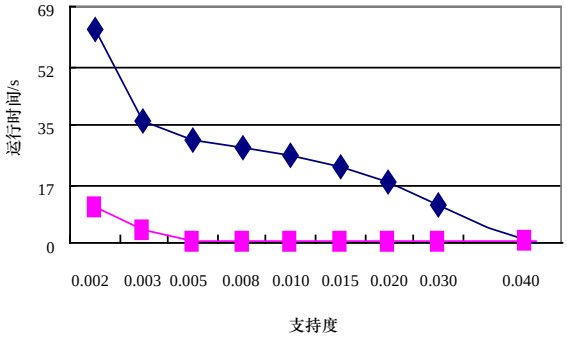

图 2　AprioriTid 与 AprioriTid_Hash 算法性能比较

(3)算法的内存开销。实验表明，2 种算法运行时，内存消耗平均都在 8 200 KB 左右，在支持度增大时，内存占有稍有减少，递减量在 200 KB 左右，内存开销比较稳定。AprioriTid_Hash 算法没有因为负载因子较小而占用更多的内存。

6 结束语

关联规则挖掘问题分为 2 个子问题：发现频繁项集和生成关联规则。其中发现频繁项集是关联规则挖掘的关键。本文提出了一种基于散列方法的AprioriTid算法，能较为快捷地求解频繁项目集，由于分别依次对C_{k-1}每个事务集内的L_{k-1}进行连接join操作，大大减少了n的值，降低了时间复杂度$O(n^2)$，在散列的同时就能得到频繁k-项集集合L_k，省去了找出其支持度不小于最小支持度的频繁项目集L_k所要花费的时间。最后与AprioriTid进行性能比较，实验证明了AprioriTid_Hash算法是有效的。

参考文献

[1] Agrawal R, Srikant R. Fast Algorithms for Mining Association Rules in Large Database[R]. IBM Almaden Research Center, Technical Report: FJ9839, 1994.

[2] Han J, Kamber M. Data Mining: Concepts and Techniques[M]. Beijing: High Education Press, 2001.

[3] Agrawal R, Imielinski T, Swami A. Mining Association Rules Between Sets of Items in Large Databases[C]//Proceedings of the ACM SIGMOD Conference on Management of Data. [S. l.]: ACM Press, 1993.

[4] Fan Ming, Meng Xiaofeng. Data Mining: Concepts and Techniques[M]. Beijing: Mechanical Industrial Press, 2001.

[5] Agrawal R, Srikant R. Fast Algorithm for Mining Association Rules[C]//Proceedings of the 20th International Conference on VLDB. Santiago: [s. n.], 1994.

[6] 马力军. 关联规则算法性能分析与仿真研究[D]. 西安: 西安交通大学, 2000.

[7] 严蔚敏, 吴伟民. 数据结构(C 语言版)[M]. 北京: 清华大学出版社, 2002-9-1.