◎数据库与信息处理◎

Apriori 挖掘频繁项目集算法的改进

柴华昕,王 勇
CHAI Hua-xin,WANG Yong

桂林电子科技大学 网络中心,广西 桂林 541004
NIC of Guilin University of Electronic Technology,Guilin,Guangxi 541004,China

CHAI Hua-xin,WANG Yong.Improvement of Apriori algorithm.Computer Engineering and Applications,2007,43(24):158-161.

Abstract: In this study,it proposes a new optimization algorithm called Napriori based on the insufficient of Apriori.Napriori algorithm presents optimizations on 2-items generation,transactions compression,items compression and join optimization.Napriori uses hash structure to generate 1-items and 2-items while compression and join optimization to generate k-items (k>3).The performance study shows that Napriori is much faster than Apriori.

Key words: association rule;Apriori algorithm;transaction compression;hash structure

摘 要:针对 Apriori 算法的不足,提出了一种新的优化算法 Napriori。算法从优化产生 2-项目集、事务压缩、项目压缩、优化连接等几个方面对 Apriori 算法进行优化,将散列技术应用于产生 1-项目集和 2-项目集,将压缩优化和连接优化应用于 k-项目集。实验结果表明,Napriori 算法运行速度比 Apriori 算法有了明显的提高。

关键词:关联规则;Apriori 算法;事务压缩;散列结构
文章编号:1002-8331(2007)24-0158-04 **文献标识码**:A **中图分类号**:TP301

关联规则挖掘是近年来数据挖掘研究的一个热门领域。关联规则挖掘发现大量数据中项集之间有趣的关联或相关联系,进而可以帮助许多商务决策的制定,如顾客购物分析、目录设计、商品广告邮寄分析等。

关联规则挖掘的一个典型例子是购物篮分析。该过程通过发现顾客放入其购物篮中不同商品之间的联系,分析顾客的购买习惯。通过了解哪些商品频繁地被顾客同时购买,这种关联的发现可以帮助零售商制定营销策略。一个关联规则的例子就是“80%的顾客在购买面包的顾客同时也会购买牛奶”。

Arpiori 算法是关联规则挖掘中常用的算法,在本文中,针对 Apriori 算法固有的一些不足,提出了一种新的优化算法 NApriori 算法。本算法对很多 Apriori 优化算法对产生 2-频繁项目集没有办法优化的缺点,通过使用 HASH 表的方法,可以高效的生成 2-频繁项目集,解决了很多 Apriori 优化算法性能提高的瓶颈;同时,该算法也对其他几个方面进行优化。

1 相关工作

自从 Apriori 算法于 1994 年被提出后,大量的相关关联规则挖掘算法被提出。算法根据是否产生候选项目集分为两类:第一类是以关联规则算法的提出者 R.Agrawal[1]提出的 Apriori 算法为代表,包括很多对 Apriori 算法的改进算法。这类算法优点是比较直观,算法实现起来容易,缺点是需要产生大量的候选项目集和多次扫描数据库;第二类算法是以 Han[2]提出的 fp-growth 算法以及其改进算法。这类算法的优点是不用产生候选项目集,所以效率比较高,缺点是使用模式树来产生关联规则,算法实现复杂。

针对第一类算法的缺点,很多学者提出了改进的优化算法,主要的优化算法有:基于散列优化[3]、基于事务压缩优化[4]、基于划分优化[5]、基于采样优化基于动态项目集计数优化[6]等。但是,各种 Apriori 的改进算法都不能改进产生候选 2-项目集,因为按照经典 Apriori 的思想,从 L_1 到 C_2 是经过连接运算生成,各种优化算法中的剪枝操作对 2-项目集不起作用,因而产生了大量的候选 2-项目集,而这些项目集大部分的时候又是非频繁的,所以本文针对文献[3]所提出的 DHP 算法做了改进,提出了一个无冲突的散列函数的构造方法,使用本函数,可以直接从 1-频繁项目集中经过散列后,直接生成 2-频繁项目集。当 k>2 时,DHP 算法性能并不是很好,因为随着 k 的增大,冲突也增大。所以,本文只将基于散列函数的优化应用于产生 1-项目集和 2-项目集,而对产生 k-项目集(k>3)时,使用事务压缩优化方法。在采用事务压缩优化算法时,参照了文[7]中提到的性质,并根据对 Apriori 算法的研究,提出了另外的几个性质。

本文提出的算法是结合基于散列的优化算法和基于事务压缩的优化算法,该方法在文献[3][8]中有所提及,但是文献[8]提出的散列函数构造稍显复杂,不容易实现,所以本文提出了一个易于实现的没有冲突的散列函数的构造算法。同时,在事务压缩方面,本文提出了更多的优化策略,在连接运算过程中,

基金项目:广西科技项目(No.桂科能 0537027-2)。
作者简介:柴华昕(1978-),男,硕士研究生,主要研究方向为网络安全;王勇(1964-),男,教授,博士,主要研究方向为网络信息安全。

也使用了连接优化策略。本文在产生 1-频繁项目集和 2-频繁项目集时,采用了基于散列的优化算法;在产生频繁项目集 Lk (k>3)时,采用了事务压缩优化算法结合 Apriori 算法,从而大大减少了数据库的大小,提高了算法的效率。

2　问题描述

关联规则是在交易数据、关系数据或其他信息载体中,查找存在于项目集合或对象集合之间的频繁模式、关联、相关性、或因果结构,通过分析数据或记录间的关系,决定哪些事情将一起发生。

设 $I=\{i_1,i_2,\cdots,i_n\}$是项的集合,其中的元素称为项,S 为 T 的集合,这里 T 是项的集合,并且 $T\subseteq$I 。如果 $X\subseteq T$,那么称 T 包含 X。

一个关联规则是形如 $X\Rightarrow Y$ 的蕴涵式,这里 $X\subset I$,$Y\subset I$,并且 $X\cap Y=\Phi$。规则 $X\Rightarrow Y$ 在集合 S 中的支持度(support)是 S 集中包含 X 和 Y 的数与所有项数之比,记为 support($X\Rightarrow Y$),即:

support($X\Rightarrow Y$)={T:$X\cup Y\subseteq T$,$T\in S$}/S

规则 $X\Rightarrow Y$ 的可信度是指包含 X 和 Y 的数与包含 X 的数之比,记为 confidence($X\Rightarrow Y$),即:

confidence($X\Rightarrow Y$)={T:$X\cup Y\subseteq T$,$T\in S$}/{T:$X\subseteq T$,$T\in S$}

关联规则挖掘的任务是:给定一个集 S,求出所有满足最小支持度和最小可信度的关联规则。

这个任务可以被分解成两个子问题:

(1)寻找频繁项目集;

(2)从频繁项目集中产生关联规则。

由于第(2)个子问题已经有了很好的解决方法,所以不必再讨论。现在大部分的研究者都把精力集中在如何提高查找频繁项目集上。

3　Apriori 算法的不足及优化策略

3.1　典型的 apriori 算法

典型的 Apriori 算法如下:

(1)L_1={large 1-itemsets};
(2)for(k=2;$L_{k-1}\neq\varnothing$;k++)do begin
(3)C_k=Apriori_gen(L_{k-1},min_sup); //由 L_{k-1} 生成新的候选集 L_k
(4)for each transaction $t\in D${//扫描数据库计算支持度
(5)　　C_t=subset(C_k,t); //获取事务 t 的属于候选集的子集
(6)　　for each candidate $c\in C_t$ do
(7)　　c.count++;
(8)　　}
(9)　　L_k={$c\in C_k$ |c.count≥minsup}
(10)　}
(11)　return $L=\cup_k L_k$;

Apriori_gen (L_{k-1}:frequent (k-1)-itemsets;min_sup:minimum support threshold)

(1) for each itemset $l_1\in L_{k-1}$
(2)　for each itemset $l_2\in L_{k-1}$
(3)　if ($l_1[1]=l_2[1]\wedge(l_1[2]=l_2[2])\wedge\cdots\wedge(l_1[k-2]=l_2[k-2])\wedge(l_1[k-1]<l_2[k-1]$)then{
(4)　　$c=l_1\infty l_2$;//连接生成候选项集
(5)　　if has_infrequent_subset(c,L_{k-1})then
(6)　　delete c;
　　//若 c 中有非频繁 k-1 项子集,则本身非候选项集
(7)　　else add c to C_k;
(8)　　}
(9) return C_k;

Apriori_gen 主要通过 L_{K-1} 内部连接后再进行剪枝以获取候选项集 C_k,主算法中的(4)~(7)行主要用于对 C_k 各项进行支持度更新。

3.2　算法的不足

(1)通过分析经典的 Apriori 算法还有一些改进算法的分析,可以看出,不管是哪一种算法,都没有解决候选 2-项目集产生过多的问题,而剪枝操作在 2-项目集又不起作用。如果 1-频繁项目集的数目为 n,则在产生 C_n^2 个候选 2-项目集。大量的试验表明,这些候选 2-项目集中,大部分又不是频繁的,所以,如何减少候选 2-项目集的产生将是提高整个算法性能的关键。

(2)更新支持度的时候,需要扫描数据库,而此时,数据库中有些项目已经被证明是非频繁的,可以不必扫描;有些事务根本不包括要寻找的项目,可以删除。所以,减少事务的数目和修剪每次交易里的项目数也是提高算法性能的关键。

(3)Aprior 算法在从 k-项目集生成候选 k+1 项目集时,采用的是连接操作,该操作要判断是否前 k-1 项相同而第 k 项不同。这个操作占用了比较多的程序运行时间,如果能减少比较次数,也可以提高算法的性能。

3.3　几个性质

性质 1　任何频繁项目集的所有非空子集也是频繁的,非频繁项目集的超集是非频繁的。证明见文献[7]。

性质 2　在生成 k+1 频繁项目集时,k-频繁项目集的项目个数必须大于 k。

证明　这个性质是显然的,因为在生成 k+1 项目集时,如果 k-频繁项目集的项目个数小于或者等于 k,则不可能参与构成 k+1 项目集。

性质 3　如果 k 频繁项目集 L_k 中包含的单个项目 I 的个数小于 k,则 I 不可能包含在频繁 k+1 项目集中。

性质 4　设 M_k 为所有 k-频繁项目集的项目组成的集合,则对于事务中包含的 $i\in(M_{k-1}-M_k)$ 在更新 k+1 频繁项目集的时候,是不需要扫描的。

证明　对于任意的 $i\in(M_{k-1}-M_k)$,表明 I 不在 k-频繁项目集中,根据性质 1 可以知道,它也不在 k+1 频繁项目集中。

3.4　优化策略

针对以上不足,本文提出了以下优化策略:

策略 1　在产生 1-项目集和 2-项目集的时候,使用 hash 结构。在使用 hash 结构的时候,通过策略 2.3 压缩数据库。这一点是不同于文献[3]的地方。

策略 2　在产生 k-项目集(k>3)时,使用以下几种方法进行优化:

策略 2.1 产生一个原数据库 D 的一个拷贝,在拷贝数据库 D_k,在 D_k 中加入一列 Item_count,用来统计事务 t 中所包含的项目集个数;在每次剪枝操作过后,同时更新 Item_count 的值;当产生更新 k 项目集的支持度时,利用性质 2,先删除 D_k 中所有 Item_count 值小于 k 的事务 t,这样可以减少扫描数据库时的事务数。

策略 2.2　在由 k-频繁项目集生成 k+1 候选项目集时,我们可以对该集中出现的项目个数进行计数,如果它的计数小于 k 的话,可以事先删除所有包含该项目的 k-频繁项目集,从而减少不必要的连接比较,提高算法的性能。

策略 2.3 在更新 k+1 频繁项目集的支持度时,先删除掉事务 t 中所有在 $M_{k-1}-M_k$ 中出现的项目，同时更新事务 t 中的项目计数。

策略 3 将每个事务及事务中的项目集按字典顺序排序。对于两个 k-1 频繁项目集 Ix 和 Iy，如果 Ix 和 Iy 不能连接,则 Ix 和 Iy 之后的所有项目集都不满足连接条件，不需要进行连接判断。

4 NAprior 算法的实现

4.1 优化策略 1 的实现

在产生 1-频繁项目集和 2-频繁项目集时，使用 hash 技术,根据文献[2],使用 hash 技术可以大大提高 apriroi 算法的性能,尤其是提高产生 2-项目集的效率。根据文献[3],使用 hash 技术并不能很好的改善 k-项目集(k>2)产生的效率尤其是当 k 很大的时候,很容易产生冲突。所以本文只将该技术应用在产生 1 和 2 项目集的时候。具体算法如下：

```
s=minimum support;
set all the buckets of H2 to zero
forall transaction t∈D do begin
      insert and count 1-itemsoccurrences in a hash tree
      use algori2.3 compress D
      forall 2-subsets x of t do
            H2[h2(x)]++
end
L1={c|c.count≥s,c exists in the leaf node of the hash tree}
```

在文献[3]中,作者并没有给出如何构造一个有效的 hash 函数才能减少冲突,因为使用 hash 技术一个突出的问题是冲突。本文根据文献[3][8],考虑到减少实现的复杂度,给出一个构造 hash 函数的算法。算法如下：

$$h(x,y)=|L_1|*(order(x)-1)\frac{order(x)*(order(x)-1)}{2}+order(y)-order(x)$$

其中$|L_1|$表示频繁 1-项目集中 L_1 的项目数,$order(x)$表示 x 在 L_1 项目中的索引。散列表长度为 $C^2_{|L_1|}$。

在生成 L_1 后，通过对散列表 H_2 使用最小支持度过滤,可以直接得到 2-频繁项目集 L_2。

4.2 优化策略 2 的实现

优化策略 2 主要是应用在产生 k-项目集(k>2)的时候。

4.2.1 优化策略 2.1 的实现

利用优化策略 2.1,实现算法如下：

```
Dk=D,//原始数据库拷贝
Add one colum item_count to Dk
   forall transaciton t∈Dk do begin
    update Dk set item_count=t.itemcount//更新事务长度
   end
//根据性质 2,压缩数据库,删除不必要的事务
 forall transaction t∈Dk do begin
  if t.item_count<k then do begin
   delete t from Dk//当事务的长度小于 k 时,从备份数据库中删除
  end
end
```

4.2.2 优化策略 2.2 的实现

利用优化策略 2.2,实现算法如下：

```
forall item i∈Lk do begin
   count i occurrences in Lk//统计 i 在 Lk 中出现的次数
   if i.count<k then do begin
     delete all the subset of Lk that contain i
   //如果出现次数小于 k 则删除所有包含该项目的 Lk 的子集
   end
end
```

4.2.3 优化策略 2.3 的实现

```
Nfk=Mk-1-Mk//计算 Mk-1-Mk
forall transaction t∈Dk do begin
   delete t where t.item∈Nfk//删除所有包含 Nfk 中项目的事务 t
   update t.item_count//更新事务 t 的长度
   if t.item_count<k delete t from Dk//利用性质 3,删除事务
end
```

4.3 优化策略 3 的实现

优化策略 3 是一个优化的连接操作，在进行此步操作前,要求数据库中的事务及项目集按字典顺序排好序,这一步可以在数据预处理中进行。以下算法是假设数据库中的内容已经做过预处理。

```
forall Ix∈Lk-1 do begin
 forall Iy∈Ix.next do begin
 if(Ix[1]=Iy[1] and Ix[2]=Iy[2]…Ix[k-2]=Iy[k-2] and
       Ix[k-1]≠Iy[k-1])
     {
     c=Ix join Iy
     Ck=Ck∪{c}
     }
     else break
     end
end
```

5 NAprior 算法示例

下面举例说明 NAprior 算法中的优化策略的使用。例子采用经典的 Apriori 算法[1]中的例子,以便进行对比。

假设有如下数据库,并设最小的支持度为 2：

D

TID	Items
100	1 3 4
200	2 3 5
300	1 2 3 5
400	2 5

C_1	count	L_1
{1}	2	{1}
{2}	3	{2}
{3}	3	{3}
{4}	1	

利用优化策略 2.3,$M_{k-1}-M_k$={{1,2,3,4,5}-{1,2,3,5}}={4},所以删除数据库 D 中 TID 是 100 的事务中的{4},数据库现在为：

TID	Items
100	1 3
200	2 3 5
300	1 2 3 5
400	2 5

然后利用策略 1 及实现算法,构造 hash 表。根据本文提到的构造函数：

$$h(x,y)=|L_1|*(order(x)-1)-\frac{order(x)*(order(x)-1)}{2}+order(y)-order(x)$$

可以得到 hash 函数为：

$$h(x,y)=4*(order(x)-1)-\frac{order(x)*(order(x)-1)}{2}+$$

$order(y)-order(x)$

hash 表长度为 $C_4^2=6$。

计算索引 order(x)

x	1	2	3	5
Order	1	2	3	4

hash 表如下:

{1,2}	{1,3}	{1,5}	{2,3}	{2,5}	{3,5}
1	2	1	2	3	2
1	2	3	4	5	6

直接产生频繁项目集 L_2:

{1,2}	1	C_2
{1,3}	2	{1,3}
{1,5}	1	{2,3}
{2,3}	2	{2,5}
{2,5}	3	{3,5}
{3,5}	2	

使用连接优化策略 3,进行连接操作生成 C_3 因为{1,3}和其后的{2,3}不满足连接条件,跳出,进入下轮比较。{2,3}和{2,5}满足条件,共进行 2 次比较,生成{2,3,5}。随后的{2,5}和{3,5}经过一次比较,不满足条件,退出连接。使用优化连接策略从 L2 生成 C3,共需比较 1+3+1=5 次,而原有的 Apriori 算法由于需要循环比较,所以连接操作要进行 3+4+4+3=14 次。由此可见优化的连接策略可以大大提高连接操作的效率。

生成 3-频繁项目集 L_3。

使用优化策略2.1:

D_k

TID	Items	Item_count
100	1 3 4	2
200	2 3 5	3
300	1 2 3 5	4
400	2 5	2

由于要更新 3-项目集的支持度,所以,所有 item_count 小于 3 的都要被删除。

TID	Items	Item_count
200	2 3 5	3
300	1 2 3 5	4

使用优化策略 2.3,$M_{k-1}-M_k=\{\{1,2,3,5\}-\{2,3,5\}\}=\{1\}$,所以删除 D_k 中 TID 是 300 的事务中的{1},D_k 变为如下:

TID	Items	Item_count
200	2 3 5	3
300	2 3 5	3

由于每个事务的 item_count 都大于 3,开始更新 C_3 的支持度。得到 L_3 如下:

L_3	count
{2,3,5}	2

生成 C_4 的时候,结果为空,算法结束。

6　实验结果

为了比较本改进算法的效率,本文进行了一系列的实验。实验环境为:操作系统 Windows XP,内存 512 M,CPU C4 1.7 G。实验数据采用 SQLServer 2000 自带的 Northwind 数据库,之所以使用本数据库是因为该数据库的 1-项目集有 77 个,生成候选2-项目集时有 $C_{77}^2=2\,926$,特别适合检验对 2-项目集的优化算法。本文实现了 Apriori 算法和 Napriori 算法,下图为改进后算法和原 Apriori 算法在不同支持度情况下执行所需时间以及 Napriori 算法在不同支持度下的运行时间。

通过图 1 可以看出,改进后的算法在很大程度上提高了算法的性能。主要原因在于经典的 Apriori 算法没有办法对 2-项目集的生成进行优化,而这也是频繁项目集挖掘算法的瓶颈所在。从图 2 可以看出,Napriori 算法在不同支持度下,运行时间差别不大,从这一点也可以看出,在频繁项目集的挖掘中,2-项目集的挖掘占据了程序运行的大部分时间,从而也表明产生 2-项目集是挖掘频繁项目集算法的瓶颈所在。从实验结果可以看出,本文提出的优化算法 Napriori 使用 hash 结构来存储 2-项目集的优化策略有效的解决了这个瓶颈问题;本文提出的其他的优化策略有效的减少了数据库中的事务数目和长度,减少了在更新支持度时的比较操作,提高了算法的性能。

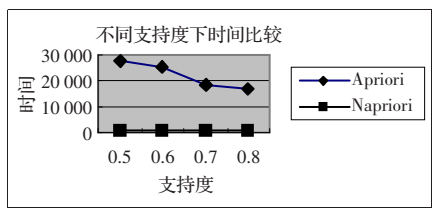

图 1　Napriori **与** Apriori **运行时间比较**

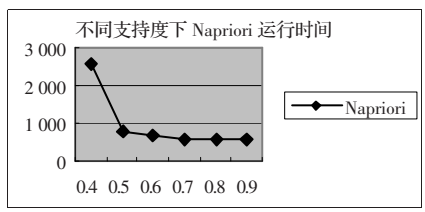

图 2　Napriori **不同支持度下运行时间**

7　结论

本文参照文献[3]提出的 hash 结构和文献[4][7][8]中的性质,提出了四个性质,并给出了证明。利用这些性质,提出并实现了一系列的优化策略。通过无冲突的 hash 结构,解决了传统的 Apriori 算法在产生 2-项目集的瓶颈;通过对事务及数据库的压缩,大大减少了数据库中事务的数目及事务的长度;通过优化的连接算法,大大减少了 Apriori 算法中连接步骤中的比较次数;实验表明,改进后的 Napriori 算法在 2-项目集特别多的情况下,性能较 Apriori 算法有了很大的提高。

(收稿日期:2007 年 1 月)

参考文献:

[1] Agrawal R,Srikant R.Fast algorithms for mining association rules in large databases[C]//Proc 20th Int'l Conf Very Large Data Bases, Sept 1994,1994:478-499.

[2] Han J,Pei J,Yin Y.Mining frequent patterns without candidate generation [C]//Proc 2000 ACM-SIGMOD Int Conf Management of Data(SIGMOD'00),Dalas,TX,May 2000.

[3] Park J S,Chen Ming-Syan,Yu Philip S.Using a hash-based method

(下转 171 页)