第 5 章　相关性与关联规则

关联规则挖掘是数据挖掘领域中研究较为广泛也较为活跃的方法之一。最初的研究动机是针对购物篮分析(Basket Analysis)问题提出的,目的是为了解决发现交易数据库(Transaction Database)中不同商品之间的联系规则。关联规则最早由 Agrawal 等人在 1993 年提出,随后大量的研究人员对关联规则挖掘问题进行了大量的研究,从最初的挖掘理论的探索、原有算法的改进、新算法的设计,到今天的并行关联规则挖掘(Parallel Association Rule Mining)以及数量关联规则挖掘(Quantitive Association Rule Mining)等方法的应用,关联规则挖掘方法已经日臻成熟,并在很多领域中有了广泛的应用。本章将对相关性和关联规则挖掘的基本概念、方法以及相关算法进行介绍。

5.1　基 本 概 念

相关性与关联规则是对给定数据集中反复出现的联系进行挖掘提取,本节中将对关联规则挖掘的基本概念进行简单介绍。5.1.1 节给出了关联规则潜在的应用;5.1.2 节介绍购物篮分析的例子,这是关联规则频繁模式挖掘的初始形式;5.1.3 节对频繁模式分析、闭项集和关联规则的基本概念进行详细解释。

5.1.1　潜在的应用

在传统的零售商店中顾客购买东西的行为是零散的,但如今,大型超市已经可以满足顾客一次购物即可买到所有自己想要的商品,同时随着网络购物的兴起,很多人选择在网上挑选自己想要的东西,这些商家以及网站很容易将购买记录收集和存储下来。通过对这些数据的智能化分析,可以获得有关顾客购买模式的一般性规则。

早在 20 世纪 90 年代的美国沃尔玛超市中,沃尔玛的超市管理人员分析销售数据时发现了一个令人难以理解的现象,在某些特定情况下,“啤酒”与“尿布”两件看上去毫无关系的商品经常会出现在同一个购买记录里。如果一个年轻的父亲可以很方便地同时购买到两件产品,那么他很可能会经常性地选择在这家超市购买商品。通过对客户购买模式的分析寻找到一般性的规则,从而使顾客能够更加快捷方便地完成购物,进而产生良好的销售记录,这也就产生了最初的关联规则的潜在应用。这些规则刻画了顾客购买行为模式,可以用来指导商家科学地安排进货、库存以及货架商品摆放设计等。图 5.1 描绘了一个简单的关联规则挖掘的潜在应用。

其实,除了上面提到的一些商品间存在的奇特关联现象外,也体现在其他方面。例如医学研究人员希望从已有的成千上万份病历中找到患某种疾病的病人的共同特征,从而为治愈这种疾病提供一些帮助。另外,通过对用户信用卡账单的分析也可以得到用户的消费方式,有助于对相应的商品进行市场推广等。关联规则的挖掘方法已经涉及了生活的很多方面,为人们的生活提供了极大的便利。

图 5.1 一个简单的关联规则挖掘的潜在应用

5.1.2 购物篮问题

通过频繁项集挖掘可以发现大型事务或关系数据集中事物与事物之间有趣的关联。随着大型数据库的建立和不断扩充，很多分析人员已经可以从数据库中挖掘潜在的关联规则，从而发现事物间的相关联系，进而帮助商家进行决策、设计和分析顾客购买习惯。

假设给定一个很大的超市，里面包含任何顾客想购买的任何东西，那么作为超市的管理者，如何找到最常出现在顾客“购物篮”中的东西呢？或者哪些东西是最为常见同时出现在顾客的“购物篮”中的呢？或者某些商品后可能诱发顾客一路购买哪些其他商品？

频繁项集挖掘的典型事例就是购物篮问题。通过发现顾客“购物篮”中不同商品之间的关联，分析顾客的购买习惯，帮助零售商了解哪些商品被频繁地同时购买。例如，如果顾客购买了面包，那么他们很可能也会购买果酱，这种信息可以很好地帮助管理人员选择性地安排货架商品位置，以减少顾客购买所花费的时间以及提高销售量。

例 5-1 购物篮问题。假设商店里有商品{milk,coke,pepsi,beer,juice}，并有以下购物记录：

B_1={milk,coke,beer},B_2={milk,pepsi,juice},B_3={milk,beer},
B_4={coke,juice},B_5={milk,pepsi,beer},B_6={milk,coke,beer,juice},
B_7={coke,beer,juice},B_8={coke,beer}

作为商店的主管，了解什么商品会被顾客经常性地购买，从而预测进货的数量等。为了解决这个问题，可以通过统计商品被购买的次数来进行分析。一般来说，会对支持度较高的一些商品感兴趣，也就是说当支持度达到一定的阈值后，某种(些)商品才有被挖掘的潜力，这个阈值就是最小支持度计数(min_sup)，当某种商品的支持度超过最小支持计数阈值时，这个(些)商品就叫频繁项集。假设设定最小支持度为 3，也就是出现次数最少为 3 的商品(集)，通过简单的计数统计，最终得到的频繁项集为：

{milk},{coke},{beer},{juice},{milk,beer},{coke,beer},{juice, coke}。

5.1.3 频繁模式分析、闭项集和关联规则

一个事务数据库中的关联规则挖掘可以描述如下：

设 $I=\{I_1,I_2,\cdots,I_m\}$是一个项目集合，事务数据库 $D=\{t_1,t_2,\cdots,t_n\}$是由一系列具有唯一标识 T_{ID}的事务组成，每个事务 $t_i(i=1,2,\cdots,n)$都对应 I 上的一个子集。

定义 5-1 设 $I_1 \subseteq I$，项目集(itemset) I_1 在数据集 D 上的支持度(support)是包含 I_1 的事务在 D 中所占的百分比，即

$$\text{support}(I_1) = \frac{||\{t \in D \mid I_1 \subseteq t\}||}{||D||} \tag{5-1}$$

例如在例 5-1 中，milk 的支持度为 62.5%。

定义 5-2 对项目集 I 和事务数据库 D，T 中所有满足用户指定的最小支持度(minsupport)的项目集，即大于或等于 minsupport 的 I 的非空子集称为频繁项目集(frequent itemsets)或大项目集(large itemsets)。在频繁项目集中挑选出所有不被其他元素包含的频繁项目集作为最大频繁项目集(maximum frequency itemsets)或最大项目集(maximum large itemsets)。

定义 5-3 假设 $A \subset I$，$B \subset I$，并且 $A \cap B = \varnothing$，关联规则是形如 $A \Rightarrow B$ 的蕴含式，定义在这个关联规则的置信度是指包含 A 同时包含 B 的事务数之比，即

$$\text{confidence}(A \Rightarrow B) = \frac{\text{Support}(A \cup B)}{\text{Support}(A)} \tag{5-2}$$

在例 5-1 中，milk⇒beer 的置信度是 80%。

定义 5-4 D 在 I 上满足最小支持度和最小置信度(minconfidence)的关联规则称为强关联规则(strong association rule)。

通常意义上所说的关联规则都是指强关联规则。

一般来说，给定一个事务数据库，关联规则挖掘问题就是通过用户指定最小支持度和最小置信度来寻找强关联规则的过程。关联规则挖掘一般可以划分为两个子问题。

1. 发现频繁项目集

通过用户给定的最小支持度，寻找所有频繁项目集，即满足支持度不小于 min-support 的所有项目子集。事实上，这些频繁项目集可能具有包含关系。一般地，只关心那些不被其他频繁项目集所包含的所谓最大频繁项目集的集合。发现所有的频繁项目集是形成关联规则的基础。

2. 由频繁项集产生关联规则

通过用户给定的最小置信度，在每个最大频繁项目集中寻找置信度不小于 min-confidence 的关联规则。

相对于第一个子问题来说，由于第二个子问题相对简单，而且在内存、I/O 以及算法效率上改进余地不大，因此第一个子问题是近几年来关联规则挖掘算法研究的重点。

从大型数据集中挖掘频繁项集的主要挑战是这种挖掘常常产生大量满足最小支持度的项集，当最小支持度设置很低的时候更是如此。这是因为如果一个项集是频繁的，它的每个子集也是频繁的，一个长项集将包含组合个数较短的频繁子项集。这将产生过于庞大的数据开销，尤其当数据量很大的时候。对于任何计算机来说，计算的速度和存储空间都是制约关联挖掘的重要问题。因此，为了解决这个问题，在这里引入闭项集和极大频繁项集的概念。

如果不存在真超项集[①] Y 使得 Y 与 X 在 I 中有相同的支持度计数，则称项集 X 在数据集 I 中是闭的。项集 X 是数据集 I 中的闭频繁项集，如果 X 在 I 中是闭的和频繁的，项集 X 是 I 中的极大频繁项集（或极大项集）。

5.2 频繁项集挖掘方法

本节将介绍最简单也是最常见的频繁项集挖掘的算法。5.2.1 节对 Apriori 算法进行详细介绍，Apriori 是一种发现频繁项集的基本算法；5.2.2 节介绍如何通过频繁项集产生关联规则；5.2.3 节介绍 Apriori 的效率，以及如何提高 Apriori 的效率；5.2.4 节介绍另一种频繁项集挖掘的算法，它与 Apriori 算法不同，并且在算法的效率上有显著提高。

5.2.1 Apriori 算法

Apriori 算法是 R.Agrawal 和 R.Srikant 于 1994 年提出的为布尔关联规则挖掘频繁项集的原创性算法。Apriori 使用一种称作逐层搜索的迭代方法，k 项集用于探索 $k+1$ 项集。

在介绍 Apriori 算法前，首先介绍一种称为 Apriori 性质的重要理论，它主要用于压缩搜索空间，从而更快地找到频繁项集。

Apriori 性质：频繁项集的所有非空子集也必须是频繁的。即如果项集 A 不满足最小支持度阈值 min-support，则 A 不是频繁的，如果将项集 B 添加到项集 A 中，也就是 A∪B 也不可能是频繁的。

该性质是一种反单调性的性质，也就是说如果一个集合不能通过测试，则它的所有超集也都不能通过相同的测试。

Apriori 算法简单来说主要有以下几个步骤：首先通过扫描数据库积累每个项的计数，并收集满足最小支持度的项，找出频繁 1-项集的集合（该集合记为 L_1）。然后 L_1 用于找到频繁 2-项集的集合 L_2，利用 L_2 再找到 L_3，如此下去直到不能再找到频繁 k-项集为止。其算法描述如算法 5.1 所示。

算法 5.1 Apriori 算法

输入：数据集 D；最小支持度计数 min-sup_count。
输出：频繁项目集 L。

```
(1) L1={频繁 1-项集};                    //所有支持度不小于 min-support 的 1-项集
(2) for(k=2;Lk-1≠0;k++)
(3)     Ck=apriori-gen(Lk-1);            //Ck 是 k 个元素的候选集
(4)     for all transaction t∈D
(5)         Ct=subset(Ck,t);
(6)         for all candidates c∈Ct
(7)             c.count++;
(8)         End for
(9)      End for
```

① Y 是 X 的真超项集，如果 X 是 Y 的真子项集，即如果 $X \subset Y$。换言之，X 中的每一项都包含在 Y 中，但是 Y 中至少有一个项不在 X 中。

```
(10)       Lk={c∈Ck|c.count> =minsup_count}
(11) End for
(12) L=∪Lk
```

算法 5.1 中调用了 apriori-gen(L_{k-1})，是为了通过 ($k-1$)-项集产生 k-项集。算法 5.2 对 apriori-gen 过程进行了详细描述。

算法 5.2　apriori-gen(L_{k-1})(候选集产生)

```
输入：(k-1)-项集。
输出：k-候选集 Ck。
(1) for all itemset p∈Lk-1
(2)   for all itemset q∈Lk-1
(3)       if(p.item1=q.item1, p.item2=q.item2,…,p.itemk-2=q.itemk-2,
(4)          p.itemk-1<q.itemk-1)
(5)              c=p∞q;
(6)          if(has_infrequent_subset(c,Lk-1)) delete c;
(7)         else add c to Ck;
(8)    End for
(9) End for
(10) Return Ck
```

在算法 5.2 中调用了 has_infrequent_subset(c,L_{k-1})，是为了判断 c 是否需要加入 k-项集中。根据 Apriori 的性质，含有非频繁项目子集的元素不可能是频繁项目集，因此应该删掉那些含有非频繁项目子集的项目集，以提高效率。对于 has_infrequent_subset(c,L_{k-1})过程，算法 5.3 给出了详细的描述。

算法 5.3　has_infrequent_subset(c,L_{k-1})(判断候选集的元素)

```
输入：一个 k-项集 c,(k-1)-项集 Lk-1。
输出：c 是否从候选集中删除。
(1) for all (k-1)-subsets of c
(2)    if S∉Lk-1
(3)       return true;
(4) return false
```

为了更好地了解 Apriori 算法，下面用一个例子对 Apriori 算法进行详细说明。

例 5-2　假设如表 5.1 所示的样本数据库 D，假设最小支持度为 2。

表 5.1　样本数据库 *D*

ID	样　本
10	A,C,D
20	B,C,E
30	A,B,C,E
40	B,E

产生频繁项集的过程如图 5.2 所示。

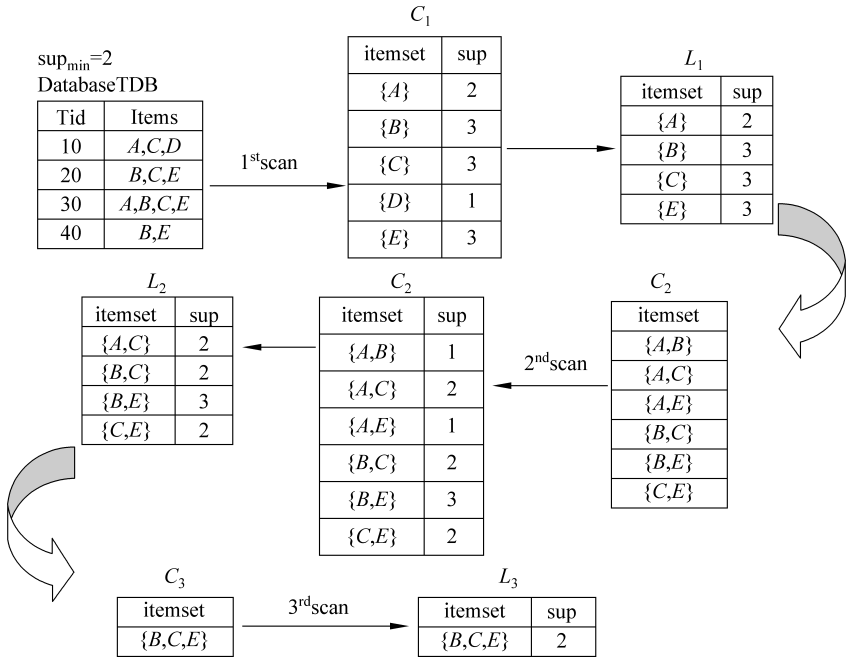

图 5.2 产生频繁项集的过程

那么，最终产生的频繁 1-项集、频繁 2-项集和频繁 3-项集分别是：

$$L_1：\{A\},\{B\},\{C\},\{E\}；$$

$$L_2：\{A,C\},\{B,C\},\{B,E\},\{C,E\}；$$

$$L_3：\{B,C,E\}；$$

所有的频繁项集为$\{A,B,C,E,AC,BC,BE,CE,BCE\}$。

5.2.2 由频繁项集产生关联规则

一旦由数据库 D 中的事务找出频繁项集，可以直接由它们产生强关联规则，即满足最小支持度和最小置信度的规则(最小置信度的公式如式(4-2)所示)。

例 5-3 在例 5-2 产生的频繁项集中，对于频繁项集 $L=\{B,C,E\}$来说，可以通过 L 得到哪些关联规则？L 的非空子集有$\{B\},\{C\},\{E\},\{B,C\},\{C,E\},\{B,E\}$。部分关联结果如下：

$B\Rightarrow C,E$ confidence$=2/3=67\%$

$C\Rightarrow B,E$ confidence$=2/3=67\%$

$E\Rightarrow B,C$ confidence$=2/3=67\%$

$C,E\Rightarrow B$ confidence$=2/2=100\%$

$B,E\Rightarrow C$ confidence$=2/3=67\%$

$B,C\Rightarrow E$ confidence$=2/2=100\%$

如果最小置信度阈值为 80%，那么只有第一个和第三个规则满足条件，即强关联规则。在这里需要注意的一点就是，关联规则的右端可以包含多个合取项。

5.2.3 提高 Apriori 的效率

Apriori 作为经典的频繁项集产生算法，在数据挖掘领域里具有里程碑的作用。但随着应用的深入，它的缺点也逐渐暴露出来，其主要的瓶颈有以下两个：

(1) 多次扫描事务数据库，需要很大的 I/O 负载。

对每次 k 循环，候选集 C_k 中的每个元素都必须通过扫描数据库一次来验证其是否加入 L_k。加入一个频繁大项集包含 10 个项，那么就至少需要扫描事务数据库 10 次。

(2) 可能产生庞大的候选集。

由 L_{k-1} 产生 k-候选集 C_k 是指数增长的，例如 10^4 个频繁 1-项集就有可能产生将近 10^7 个元素的 2-候选集。如此庞大的候选集对时间和主存空间都是一种挑战。因此很多研究人员对 Apriori 算法进行了很多的改进，以提高算法的效率。

1. 基于散列的方法

1995 年，Park 等提出了一种基于散列(Hash)技术产生频繁项集的算法。这种方法把扫描的项目放到不同的 Hash 桶中，每个频繁项最多只可能放在一个特定的桶里，这样可以对每个桶中的频繁项自己进行测试，减少了候选频繁项集产生的代价。

例 5-4 对于表 5.2 中给出的数据，加入使用 Hash 函数“(10×x+y)mod 7”生成{x，y}对应的桶地址，那么扫描数据的同时可以把可能的 2-项集{x，y}放入对应的桶中，并对每个桶内的项目集进行计数，结果如表 5.3 所示。假设最小支持度计数为 3，根据表 5.3 的计数结果，$L_2=\{(I2,I3),(I1,I2),(I1,I3)\}$。

表 5.2 事务数据库示例

ID	事　务	ID	事　务
1	$I1,I2,I5$	6	$I2,I3$
2	$I2,I4$	7	$I1,I3$
3	$I2,I3$	8	$I1,I2,I3,I5$
4	$I1,I2,I4$	9	$I1,I2,I3$
5	$I1,I3$		

表 5.3 2-项集的桶分配示例

桶地址	0	1	2	3	4	5	6
桶计数	2	2	4	2	2	4	4
桶内容	{$I1,I4$} {$I3,I5$}	{$I1,I5$} {$I1,I5$}	{$I2,I3$} {$I2,I3$} {$I2,I3$} {$I2,I3$}	{$I2,I4$} {$I2,I4$}	{$I2,I5$} {$I2,I5$}	{$I1,I2$} {$I1,I2$} {$I1,I2$} {$I1,I2$}	{$I1,I3$} {$I1,I3$} {$I1,I3$} {$I1,I3$}

2. 事务压缩

事务压缩是指压缩未来迭代扫描的事务数。由于不包含任何频繁 k-项集的事务是不可能包含任何频繁($k+1$)-项集的，因此这种事务在后续的考虑中可以加上标记或者直接删除，因此产生 j-项集($j>k$)的数据库扫描不再需要它们。

3. 基于数据划分(Partition)的方法

Apriori 算法在执行过程中首先生成候选集，然后再进行剪枝。可是生成的候选集并不都是有效的，有些候选集根本就不是事务数据的项目集。因此，候选集的产生具有很大的代价。特别是内存空间不够导致数据库与内存之间不断交换数据，会使算法的效率变得很差。

把数据划分应用到关联规则挖掘中，可以改善关联规则挖掘在大容量数据集中的适应性。其基本思想是把大容量数据库从逻辑上分成几个互不相交的块，每块应用挖掘算法(如 Apriori 算法)生成局部的频繁项集，然后把这些局部的频繁项集作为候选的全局频繁项目集，通过测试它们的支持度来得到最终的全局频繁项目集。

4. 基于采样(Sampling)的方法

基于采样的方法是 Toivonen 于 1996 年提出的，这个算法的基本思想是：选取给定数据 D 的随机样本 S，然后在 S 而不是 D 中搜索频繁项集。用这种方法牺牲了一些精度换取有效性。样本 S 的大小选取使得可以在内存搜索 S 中的频繁项集。这样，只需要扫描一次 S 中的事务。由于算法只是搜索 S 中的数据，因此可能会丢失一些全局频繁项集。为了减少这样的情况，使用比最小支持度低的支持度阈值来找出局部于 S 的频繁项集(记作 Ls)。然后，数据库的其余部分用于计算 Ls 中每个项集的实际频率。使用一种机制来确定是否所有的频繁项集都包含在 Ls 中。如果 Ls 实际包含了 D 中的所有频繁项集，则只需扫描一次 D。否则，可以做第二次扫描来找出第一次扫描时遗漏的频繁项集。

5.2.4 挖掘频繁项集的模式增长方法

在很多情况下，Apriori 算法已经能够很好地解决关联规则挖掘的问题，并有很好的性能表现。但同时它也有着很大的缺陷：会产生大量的候选项集；需要重复地扫描数据库。

那么，是否可以设计一种方法挖掘全部频繁项集而不产生候选集？Han 等人于 2000 年提出了一种称为频繁模式增长(Frequent Pattern-growth，FP-growth)的算法。这种算法只需要进行两次数据库扫描，并且它不会产生候选集，直接压缩数据库成为一个频繁模式树，最后通过这棵树生成关联规则。

FP-growth 算法主要采用如下的分治策略：首先将提供频繁项的数据库压缩到一个频繁模式树(FP-tree)，但仍保留相关信息。然后将压缩后的数据库划分成一组条件数据库，每个关联一个频繁项或“模式段”，并分别挖掘每个条件数据库。具体算法如算法 5.4 所示。

算法 5.4　FP-tree 构造算法

输入：事务数据库 DB；最小支持度阈值 Minsupport。

输出：FP-tree 树。

(1) 扫描事务数据库 D 一次。收集频繁项集合 F 以及它们的支持度计数，对 F 按照支持度计数降序

排序，得到频繁项列表 L。

(2) 创建 FP-tree 的根结点，以"null"标记它。对于 D 中的每个事务 T，作如下处理：选择 T 中的频繁项，并按照 L 中的次序进行排序，排序后的频繁项标记为[p|P]，其中 p 是第一个元素，P 是剩余元素的表。调用 insert_tree([p|P],T)将此元组对应的信息加入 T 中。

构造 FP-tree 算法的核心是 insert_tree 过程。Insert_tree 过程是对数据库的一个候选项目集的处理，它对排序后的一个项目集的所有项目进行递归式的处理，直到项目表为空。

算法 5.5　insert_tree([p|P],T)

(1) if(T 有一个子女 N 使得 N.item- name=p.item- name)。

(2) N 的计数加一。

(3) else。

(4) 创建一个新结点 N，将其计数设为 1，链接到它的父结点 T，并通过结点链结构将其链接到具有相同项名的结点。

(5) 如果 P 非空，递归地调用 insert_tree(P,N)。

为了更好地理解 FP-tree 的构建，通过下例来对算法进行说明。

例 5-5　对于一个给定的事务数据库，通过一次扫描后去掉不频繁的项目(本例子中设定最小支持度阈值为 3)，并按照出现的频率降序排列。表 5.4 给出了原始数据以及整理后的数据。

表 5.4　样本数据库/排序后的数据库

Tid	原始项目集	整理后的项目集
100	$\{f, a, c, d, g, i, m, p\}$	$\{f, c, a, m, p\}$
200	$\{a, b, c, f, l, m, o\}$	$\{f, c, a, b, m\}$
300	$\{b, f, h, j, o, w\}$	$\{f, b\}$
400	$\{b, c, k, s, p\}$	$\{c, b, p\}$
500	$\{a, f, c, e, l, p, m, n\}$	$\{f, c, a, m, p\}$

通过一次扫描，可以得到频繁 1-项集 $L=\{f,c,a,b,m,p\}$，如图 5.3 所示。利用得到的频繁 1-项集创建树的根结点，用 Null 标记。第二次扫描数据库 D，并对每一个事务创建一个分支。

(1) 第一个事务 T100 按照 L 的次序包含 5 个项$\{f,c,a,m,p\}$，导致构造树的第一个分支$<f1-c1-a1-m1-p1>$。

(2) 对于第二个事务 T200，由于其排序后的频繁项表为$\{f,c,a,b,m\}$，已经与分支$\{f,c,a,m,p\}$有共同的前缀$\{f,c,a\}$，因此前缀中的每个结点计数加 1，只创建两个新的结点 $m1$ 和 $p1$。形成分支链为$<f2-c2-a2-b1-m1>$。

(3) 按照这种方法处理 T300～T500，并按照要求连接到项头表和把相同的项目连接起来，如图 5.3 所示。最终得到 FP-tree。

前面已经提到了如何构建 FP-tree，接着对如何通过 FP-tree 产生频繁项集进行详细解释。利用 FP-tree 算法分析频繁项集的基本思想，即分而治之，构造过程如算法 5.6 所示。

算法 5.6　利用 FP-tree 挖掘频繁项集

输入：构造好的 FP-tree，事务数据库 D，最小支持度阈值 Minsupport。

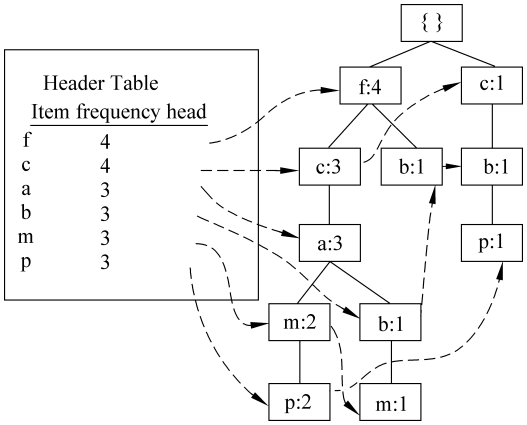

图 5.3 样本数据库对应的 FP-tree

输出：频繁项集。

```
FP-growth(Tree,α)
(1) if(Tree 含单个路径 P)
(2)     for 路径 P 中结点的每个组合(记作 β)
(3)         产生模式 β ∪ α,其支持度 support=β 中结点的最小支持度
(4) else for each ai在 Tree 的头部 {
(5)     产生一个模式 β =ai ∪ α,其支持度 support=ai.support;
(6)     构造 β 的条件模式基,然后构造 β 的条件 FP-树 Treeβ;
(7) if Treeβ≠0 then
(8) 调用 FP_growth (Treeβ, β);
```

通过算法 5.6，可以对例 5-6 得到的 FP-tree 进一步分析，得到挖掘 FP_tree 的过程，如表 5.5 所示。

表 5.5 频繁项集产生过程

项	条件模式基	条件 FP-tree	产生的频繁项集
c	f:3	<f:3>	fc:3
a	fc:3	<fc:3>	fca:3,ca:3,fa:3
b	fca:1, f:1, c:1	∅	∅
m	fca:2, fcab:1	<fca:3>	fcma:3,fm:3,cm:3,am:3,fcm:3,fam:3,cam:3
p	fcam:2, cb:1	<c:3>	cp:3

5.3 多种关联规则挖掘

5.3.1 挖掘多层关联规则

对于许多应用，由于多维数据空间数据的稀疏性，在低层或原始层的数据项之间很难找出强关联规则。在较高的概念层次发现的强关联规则有可能提供具有普遍意义的知识。然而，对一个用户代表普遍意义的知识，对另一个用户可能是新颖的。这样，数据挖掘系统应