# 一种基于线性链表的关联规则挖掘算法*)

李晓虹[1] 杨 有[1,2]

(重庆师范大学数学与计算科学学院 重庆 400047)[1]

(北京航空航天大学计算机学院数字媒体室 北京 100083)[2]

**摘 要** 关联规则挖掘是数据挖掘的一个重要研究方向,其算法主要有 Apriori 算法和 FP-growth 算法,它们需要多次扫描事务数据库,严重影响算法的效率。为了减少扫描事务数据库的次数,本文提出一种基于线性链表(Linear Linker)的 LL 算法,它只需扫描事务数据库一次,把事务数据库转换为线性链表 LL,进而对 LL 进行关联规则挖掘。实验表明,LL 算法的时间开销明显优于 Apriori 算法和 FP-growth 算法,且 LL 算法通过定义备用候选频繁项目集,有效地支持了关联规则的更新挖掘。

**关键词** 关联规则挖掘,更新挖掘,线性链表,事务数据库

## An Association Rules Mining Algorithm Based on Linear Linker

LI Xiao-Hong[1] YANG You[1,2]

(College of Mathematics and Computer Science, Chongqing Normal University, Chongqing 400047)[1]

(Digital Media Laboratory, School of Computer Science and Engineering, Beihang University, Beijing 100083)[2]

**Abstract** Association rules mining is an important research aspect of data mining. Apriori algorithm and FP-growth algorithm are the main mining algorithms of this aspect. Because of multiple scanning of transaction database, these algorithms' performances are restricted heavily. An algorithm which was based on linear linker, we called LL algorithm, was presented for the reducing of scanning. LL algorithm scanned the transaction database only once. It translated the transaction database to a linear linker and mined the linear linker further more. The Experiments showed that LL algorithm's time cost is less than Apriori's and FP-growth's evidently. Furthermore, LL algorithm could support incremental updating effectively by employing the concept of backup candidate frequent itemset.

**Keywords** Association rules mining, Incremental updating, Linear linker, Transaction DB

## 1 前言

频繁项目集的产生是关联规则挖掘任务的基本步骤,长期以来,对频繁项目集的挖掘主要采用 Apriori 算法[1]及其改进形式,然而,这些算法都需要产生大量的候选频繁项目集,并且反复地扫描事务数据库,从而严重影响了算法的效率。Jiawei Han 等人提出的 FP-growth 算法[2]是本质上不同于 Apriori 算法的另外一种关联规则挖掘算法,它需要两次扫描事务数据库,第一次扫描事务数据库时获得频繁 1 项目集,第二次扫描事务数据库时建立频繁模式树,关联规则的挖掘则在频繁模式树上进行。无论是 Apriori 算法还是 FP-growth 算法,多次扫描事务数据库都会导致算法的时间开销增大。

针对上述情况,我们提出一种基于线性链表(Linear Linker)的关联规则挖掘算法,简称 LL 算法,它首先扫描事务数据库,将数据库转换为一组带有头结点的线性链表,其次对线性链表进行关联规则挖掘,找出所有频繁项目集。LL 算法只需扫描事务数据库一次,减少了算法的时间开销,且由于 LL 算法保留了备用候选频繁项目集,使得该算法能够有效地支持更新挖掘。

## 2 LL 算法

### 2.1 算法的相关定义

**定义 1** 事务数据库 $D$ 是具体事务的集合,可表示为:$D=\{x_1,x_2,\cdots,x_n\}$,其中 $x_l(1\leqslant l\leqslant n)$ 代表某个具体的事务,$n$ 是数据库中事务总量。具体事务 $x_l$ 由一个唯一的事务标识和一组有序项目集组成。

如表 1 所示,事务标识 Tid 值为 1 的事务 $x_1$ 包含的有序项目集为 I1、I2、I5。

表 1 事务数据库 D

| Tid | Items | Tid | Items |
|---|---|---|---|
| 1 | I1,I2,I5 | 6 | I2,I3 |
| 2 | I2,I4 | 7 | I1,I2 |
| 3 | I2,I3 | 8 | I1,I2,I3,I5 |
| 4 | I1,I2,I4 | 9 | I1,I2,I3 |
| 5 | I1,I3 | | |

**定义 2** 头结点数组 TL 是由所有单链表的头结点组成的一维数组,每个元素指向一个单链表。

**定义 3** 线性链表 LL 由若干个单链表组成,并且所有头结点组成头结点数组 TL。单链表的每个结点包含两个域:项目名和指向下一个结点的 next 指针,如图 1 所示。线性链表 LL 通过结点及指针可完整表示事务项目集。

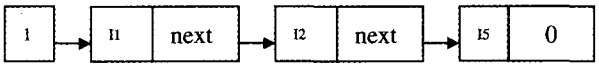


图 1 以 1 为头结点的单链表

*)本文受重庆师范大学科研项目资助(编号:06XLY026)。**李晓虹** 硕士生,讲师,主要研究方向是系统理论和数据挖掘。

**定义 4** 单链表路径的结点组合 $C$ 由单链表中所含项目的任意一个组合及其组合的计数值 Count 组成。例如，在扫描表 1 所示事务数据库 $D$ 的第一条记录时，所产生的结点组合有$\{\{I_1,I_2\}_{(1)},\{I_1,I_5\}_{(1)},\{I_2,I_5\}_{(1)},\{I_1,I_2,I_5\}_{(1)}\}$。

**定义 5** 候选频繁项目集 CF 用于存放挖掘过程中产生的所有候选频繁项目集。

**定义 6** 备用候选频繁项目集 BCF 用于存放 CF 中的项目组合的 Count 值小于 Minsup 的那些元素，即 $BCF=\{C\in CF|C.Count<Minsup\}$。BCF 用于 Minsup 和 $D$ 改变时的更新挖掘。

### 2.2. 算法的基本思想

LL 算法分为构造线性链表 LL 和挖掘线性链表 LL 两个步骤：

第一步，首先以 $D$ 的记录数作为 TL 数组的大小，创建线性链表 LL 的头结点数组 TL，然后取出事务数据库 $D$ 中的每条记录，插入到线性链表 LL 中。具体方法是：取 $D$ 中第一条记录，以 TL 数组中的第一个元素 TL[1]为单链表的头结点，以事务记录中包含的项目为结点形成一个单向链表，最后一个结点的 next 为空，即将 next 置为 0；再取事务数据库 $D$ 的第二条记录，以 TL 数组中的第二个元素 TL[2]为单链表的头结点，用同样的方法形成第二个单链表；以此类推，直到事务数据库的全部记录插入到线性链表 LL 为止。这样，只需要扫描数据库 $D$ 一次，就可以完成数据库 $D$ 到线性链表 LL 的转换，同时，LL 中指针的指向保留了项目的顺序关系。按上述步骤，表 1 所示的事务数据库可转换为图 2 所示的线性链表 LL，图中的“→”表示 NEXT 指针。

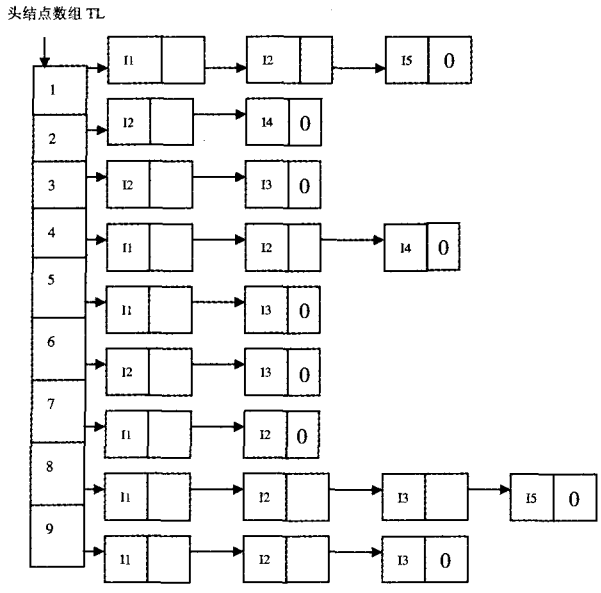


图 2 线性链表 LL 示例

第二步是对线性链表 LL 进行挖掘。首先取出头结点数组 TL 的第一个元素 TL[1]，对该指针所指向的单链表路径中包含的结点进行组合，并按定义 4 设置每个组合的表示和计数值，然后将所有的组合加到候选频繁项目集 CF 中，加入方法是：如果 CF 已经包含该项目组合，则对该组合进行计数累计，否则，该组合直接进入 CF 中。再取出头结点数组 TL 中的第二个元素 TL[2]，重复上述步骤，直到取完数组中的所有元素并处理完。最后，用给定的最小支持数 Minsup 的值，将 CF 按定义 6 划分出 BCF，用于关联规则的更新挖掘，CF 中剩下的元素即为所求的频繁项目集。

对于图 2 所示的线性链表 LL，TL[1]对应的单链表包含项目 I1、I2、和 I5，对三结点进行组合，得到的组合 C 为：$\{\{I1、I2\}_{(1)},\{I1、I5\}_{(1)},\{I2、I5\}_{(1)},\{I1、I2、I5\}_{(1)}\}$，由于此时的 CF 为空，所以将初次得到的这四种组合直接赋予 CF；再根据 TL[2]，得到：$C=\{\{I2、I4\}_{(1)}\}$，由于该组合没有包含在 CF 中，所以将 C 并入 CF 中；以同样的方法，可以完成 TL[3]的相关处理；当处理到 TL[4]时，由于{I1，I2}组合已经包含在 CF 中，所以只需对 CF 中{I1，I2}组合的 Count 值加 1；最后完成 TL 中所有元素的处理，得到 $CF=\{\{I1、I2\}_{(5)},\{I1、I4\}_{(1)},\{I1、I5\}_{(2)},\{I2、I5\}_{(2)},\{I1、I2、I4\}_{(1)},\{I1、I2、I5\}_{(2)},\{I2、I4\}_{(2)},\{I2、I3\}_{(4)},\{I1、I2、I3\}_{(2)},\{I1、I3\}_{(3)},\{I3、I5\}_{(1)},\{I1、I2、I3、I5\}_{(1)}\}$。当取 Minsup＝2，得到频繁项目集：$CF=\{\{I1、I2\}_{(5)},\{I1、I5\}_{(2)},\{I2、I5\}_{(2)},\{I1、I2、I5\}_{(2)},\{I2、I4\}_{(2)},\{I2、I3\}_{(4)},\{I1、I2、I3\}_{(2)},\{I1、I3\}_{(3)}\}$，备用候选项目集 $BCF=\{\{I1、I4\}_{(1)},\{I1、I2、I4\}_{(1)},\{I3、I5\}_{(1)},\{I2、I3、I5\}_{(1)},\{I1、I2、I3、I5\}_{(1)}\}$。可以验证，该结果与用 Apriori 算法和 FP-growth 算法产生的结果是一致的。

### 2.3. LL 算法描述

基于上述思想，LL 算法用伪代码描述如下：

输入：事务数据库 $D$，和最小支持数 Minsup；

输出：频繁项目集合 CF；

(1)建立线性链表 LL

```
1)根据事务数据库中事务个数 n,创建有 n 个元素的数组 TL[n];
2)打开事务数据库 D;
3)for 数据库 D 中的每一个事务记录 Ri do begin
4)while Ri≠Null do begin
5)C=Ri 中第 k 个项目;
6)创建一个结点 a, a.item=C, a.next=0;并 将结点 a 插入到 TL[i]
所指链表中;
7)if C∈CF
8)   C.count++;
9)else
10)   CF=CF∪C;
11) end
12)end
```

(2)挖掘线性链表 LL

```
1)for (i=1; i<=n; i++)do begin
2)  for 每个组合 C∈TL[i]单链表 do
3)  if
4)   CF.C.Count +=C.Count;
5)  else
6)   CF=CF∪C;
7)end
8)for 每个组合 C∈CF do
9)   if C.Count<Minsup
10)      begin
11)        BCF=BCF∪C;
12)        CF=CF−C;
13)      end
14)end
15)输出 CF 中的元素,即频繁项目集;
```

## 3 算法特点与性能

### 3.1 LL 算法支持更新挖掘

当如下三种情况发生时，需要进行更新挖掘。一是最小支持数发生改变，二是一个事务数据库 $d$ 添加到事务数据库 $D$ 中，三是一个事务数据库 $d$ 从事务数据库 $D$ 中删除。常见的更新挖掘算法有 FUP[3]、FUP2[4] 和 IUA[5] 等算法，它们都需要在不同程度上对 $D$ 和 $d$ 进行多次扫描，才能得到更新的频繁项目集，这对海量事务数据库而言开销是巨大的。本文的 LL 算法在充分利用已有挖掘结果的基础上，0 次或 1 次扫描数据库 $d$ 则可以得到更新的频繁项目集。

当最小支持数(newMinsup)增大时，对已有的 CF 扫描一次，判断每个组合的 C.Count，若 C.Count<newMinsup，就将

该组合 C 移入到 BCF 中；当最小支持数(newMinsup)减小时，对已有的 BCF 扫描一次，判断每个组合 C. Count，若 C. Count≥newMinsup，将该组合 C 移入 CF 中；最后 CF 中的组合则为更新的频繁项目集。在此，不涉及对原事务数据库 $D$ 的扫描即可得到更新的频繁项目集。

当最小支持数不变而数据库事务记录增加或减少事务数据库 $d$ 时，先使用 LL 算法求出事务数据库 $d$ 的候选频繁项目集 $CF_d$，再扫描 $CF_d$ 中的每个元素，并对原候选频繁项目集 CF 和备选候选项目集 BCF 进行修正。此方法中，扫描事务数据库 $d$ 一次，不需对原事务数据库 $D$ 的扫描即可求得更新的频繁项目集。

增加事务数据库 d 的算法伪代码描述如下：

输入：原事务数据库的 BF 和 BCF，新的最小支持数 NewMinsup

输出：更新的频繁项目集 CF

```
1)用 LL 算法求出增加事务数据库 d 的 CFd 和 BCFd;
2)for 每一个 C∈CFd do begin
3)if C∈CF
4)CF. C. Count + =C. Count;
5)else if C∈BCF do begin
6)BCF. C. Count +=C. Count;
7)if BCF. C. Count≥NewMinsup
8)   begin
9)     CF = CF∪C;
10)     BCF=BCF−C;
11)   end
12)end
13)else if C. Count≥NewMinsup
14)CF = CF∪C;
15)else
16)BCF =BCF∪C
17)end
18)for 每一个 C∈BCFd do begin
19)   if C∈CF do begin
20)   CF. C. Count + =C. Count;
21)   if CF. C. Count<NewMinsup
22)   begin
23)     BCF=BCF∪C;
24)     CF=CF−C;
25)end
26)end
27)else if C∈BCF do begin
28)BCF. C. Count +=C. Count;
29)else
30)BCF =BCF∪C;
31)end
32)输出更新的频繁项目集 CF;
```

减少事务数据库 $d$ 的算法伪代码描述如下：

输入：源事务数据库的 CF 和 BCF，新的最小支持数 NewMinsup;

输出：更新的频繁项目集 CF

```
1)用 LL 算法求出减少的事务数据库 d 的 CFd 和 BCFd;
2)for 每一个 C∈CFd∪BCFd do begin
3)if C∈BCF
4)CF. C. Count−=C. Count;
5)else if C∈CF do begin
6)CF. C. Count−=C. Count;
7)if CF. C. Count<NewMinsup
8)     begin
9)       CF=BCF∪C;
10)       CF=CF−C;
11)     end
12)end
13)end
14)输出更新的频繁项目集 CF;
```

### 3.2 算法性能与实验

假设 N 表示事务数据库 $D$ 的事务个数，M 表示事务数据库中包含的项目个数，T 表示事务记录平均长度。则：在时间方面，扫描事务数据库 D 建立链表 LL 的时间开销为 $O(N\times M)$，挖掘链表 LL 的时间开销是 $O(N\times M)+O(N\times(2^T-T)\times(2^M-1))+O(2^M-1)$。空间方面，除了对基本事务数据库的存储外，LL 算法保存了链表 LL 的空间 $O(N\times T)$，频繁项目集存储的空间 $O(2^M-1)$，总的空间开销为 $O(2^M-1)+O(N\times T)$。

为进一步验证算法性能，我们进行了仿真实验。测试环境为：Windows 2000 Prefessional、PIII CPU、512MB 内存。实验数据来自真实环境下的模拟交易数据，每一个项目用一个数字表示，交易数据库的项目数是 200，每笔交易项目数的平均长度为 25，最大潜在的频繁项目集的长度是 20，交易的总笔数为 10 万条记录，算法用 Vc6.0 实现，实验结果如图 3 所示。

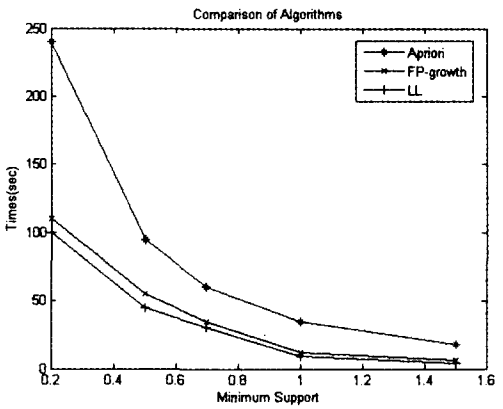


图 3 算法的性能比较

实验结果表明，三个算法所得到的频繁项目集是相同的；LL 算法的挖掘速度比 FP-growth 算法约提高 20%，比 Apriori 算法约提高 60%。由 LL 算法的工作原理可知，当事务数据库的记录数增多时，LL 算法的性能还会进一步提高。特别是在更新挖掘中，LL 算法直接利用 CF 和 BCF 中的项目集，不对原事务数据库进行重复扫描就能产生更新的频繁项目集，这相对 Apriori 算法和 FP-growth 算法而言，会节省大量时间。

**结束语** 挖掘关联规则算法的任务就是在给定的交易集合上找出可信度大于等于用户指定的最小可信度，并且支持度大于等于用户指定的最小支持度的关联规则。其关键步骤是找出满足最小支持度的频繁项目集。交易集合通常都来自于数据仓库中的数据，往往包含了一年甚至数年的数据，数据量非常大的。显然，无论哪种关联规则挖掘算法，都至少需要遍历交易集合一次。LL 算法达到了这种遍历的底限，有效地提高了算法的效率。同时，LL 算法保留了备用候选频繁项目集，以适度的空间开销换取了挖掘性能的提升。

## 参 考 文 献


1 Agrawal R, Ramakrishnan Srikant. Fast algorithms for mining association rules[C]. In: Proc. of the 20th Int'l Conf. VLDB's94, Santiago, Chile, Sept. 1994. 487～499

2 Han Jia-wei, Pei Jian, Yin Yi-wen. Mining Frequent Patterns Without Candidate Generation[C]. In: Proceedings of the 2000 ACM SIGMOD Internal Conderence on Management of Data. Dallas, Texas: ACM Press, 2000. 1～12

3 Cheung D, Han J, Ng V. Maintenance of Discovered Association Rules in Large Database: An Incremental Updating Technique [A]. In: Proceedings of the 12th International Conference on Data Engineering[C]. New Orleans, Louisiana, 1996, 2: 106～114

4 Cheung D, LEE S D, Kao B. A general Incremental Technique for Maintaining Discovered Association Rules[A]. In: Preceedings of the 5thInternational Conference on Database Systems for Advanced Application[C]. Melbourne, Australia: World Scientific, 1997, 4: 185～194

5 冯玉才，冯剑琳. 关联规则的增量式更新算法[J]. 软件学报，1998，(4)：301～306