

基于 N-list 的并行频繁项集挖掘算法

陈 奇, 张曦煌

(江南大学 物联网工程学院, 江苏 无锡 214122)

摘 要: N-list 是近几年提出的一种新的数据结构, 它在频繁项集挖掘中有很高的效率. 本文基于 N-list 提出了一种新型的并行频繁项集挖掘算法 PPF 算法. 该算法通过扫描数据库创建一颗 PPC-tree 树, 利用 PPC-Tree 树生成一系列 N-list, 将 N-list 数据表项分配到不同的节点进行深度挖掘, 最后汇总所有节点的结果挖掘出所有的频繁项集. 本文在四种不同的数据集上对 PPF 算法进行了测试和分析, 实验结果表明在任何数据集上 PPF 算法的运行速度都是最优的.

关键词: 数据挖掘; 频繁项集挖掘; 并行; N-list

中图分类号: TP301.6

文献标识码: A

文章编号: 1000-7180(2017)05-0040-05

An N-list based Parallel Algorithm for Mining Frequent Itemsets

CHEN Qi, ZHANG Xi-huang

(School of Internet Things Engineering Jiangnan University, Wuxi 214122, China)

Abstract: N-list is a novel data structure proposed in recent years. It has been proven to be very efficient for mining frequent itemsets. In this paper, we present PPF, a new parallel algorithm for mining frequent itemsets. The algorithm directly scans dataset to construct a PPC-Tree. Then, the algorithm uses PPC-Tree to generate a series of N-lists which will be assigned to different nodes to mining frequent itemsets. We have conducted extensive experiments to evaluate PPF against PrePost algorithm on four various real datasets. The experimental results show that PPF algorithm is always the fastest one on all datasets.

Key words: data mining; frequent itemset mining; parallel; N-list

1 引言

随着大数据时代的到来, 如何高效地从大数据中挖掘出具有决策意义的信息已成为业界关注的重点问题之一. 频繁项集挖掘是数据挖掘领域的一种很流行的挖掘技术, 并且是很多数据挖掘的基础和核心问题, 具有广泛的应用. 由于它是数据挖掘过程中最耗时的部分, 因此频繁项集挖掘算法的效率直接影响着整个数据挖掘的效率. 因此对频繁项集挖掘算法的深入研究绝对是很有必要, 也很有意义的一项研究工作. 在频繁项集挖掘方向, 目前为止, 最为经典的两个频繁项集挖掘算法是: Apriori 算法^[1]和 FP-growth 算法^[2]. 其中 Apriori 算法采用逐层

搜索的迭代策略, 它需要多趟扫描事务数据库, 并且在迭代过程中产生大量的候选集. 然而, 当数据量稍大时, Apriori 算法的执行效率很低. 为了弥补以上 Apriori 算法效率低的缺陷, 国内外学者选取了各式技术与 Apriori 算法融合改进, 诸如基于 Map-Reduce、基于布尔矩阵^[3]、基于压缩事务矩阵相乘^[4]、基于十字链表^[5]、基于 HMT 和哈希树的改进算法^[6]等. 然而, 面对大数据, 文献^[3-6]中的算法从不同角度上又增加了新的负荷, 影响了 Apriori 算法效率的大幅提高.

在此基础上, Han 等人提出了一种基于频繁模式树 (FP-tree)^[7] 进行频繁模式挖掘的 FP-growth 算法. 与 Apriori 算法相比, 该算法不需产

生候选集,并且只需要两次扫描事务数据库,大大提高了频繁项集挖掘的效率.然而该算法美中不足的是需要递归构建 FP-tree,并且需要频繁遍历 FP-tree,由此会带来很大的时间和空间的开销,并且当数据量的规模扩大到一定量时,计算资源已经无法满足 FP-growth 算法挖掘的需求.为了弥补上述 Fp-Growth 算法的不足文献[8]分别提出了基于 MPI 编程模型的并行 FP-Growth 算法,取得了较好的结果.因此,在大数据时代的如今,对传统的频繁项集挖掘算法进行并行化改造就显得更有意义.为了解决上述文献[1-6]算法中的性能不足问题,本文提出了一种新型的并行频繁项集挖掘算法 PPF 算法,通过压缩数据库生成一颗 PPC-Tree^[9],利用并行化方式有效地提升了在大数据情况下频繁项集挖掘的效率.

2 相关知识

2.1 项集和 K 项集

设集合 $I = \{i_1, i_2, \dots, i_m\}$, 事务数据库 $DB = \{T_1, T_2, \dots, T_n\}$ 是一系列事务的集合,其中 $T_k (1 \leq k \leq n)$ 表示一个事务并且 $T_k \subseteq I$.

定义 1 (项集和 k 项集) 如果集合 $A \subset I$, 则集合 A 被称为项集. 当 $|A| = k$, 则集合 A 被称为 k 项集.

2.2 频繁项集

定义 2 (支持度) 给定一个集合 $A \subset I$, 则集合 A 的支持度为:

$$sp(A) = |\{T_i \mid T_i \subset T \cap A \subset T_i\}| (1 \leq i \leq n)$$

即事务数据库中包含集合 A 的事务数量之和.

定义 3 (频繁项集) 如果一个集合 $A \subset I$, 并且 $sup(A) \geq minsup$, 则项集 A 称为频繁项集.

3 PPC-Tree

3.1 PPC-Tree 的定义和构造算法

PPC-Tree 是 PPF 算法的基础,它是在 FP-Tree 的基础上进行改进的一种新型的树形结构.它的作用和 FP-Tree 相似,需要扫描事务数据库,将事务数据库压缩成一颗 PPC-Tree.但是 PPC-Tree 比 FP-Tree 更加紧凑,占用的内存更少.下面将详细介绍该数据结构.

(1)它是一个以 null 为根节点,以一系列项前缀子树作为根结点的子节点,从而构成一颗 PPC-Tree.

(2)PPC-Tree 的每个节点包含 5 个部分,分别

是节点名称 ItemName, 节点数量 Count, 前序编号 PreOrder, 后续编号 PostOrder, 孩子节点 children-List. 算法的具体步骤如下:

算法 1: PPC-Tree 构造算法

输入: 事务数据库 DB 和最小支持度 M

输出: PPC-Tree, F_1

(1)产生频繁 1 项集,扫描数据库 DB,根据最小支持度 M 找到所有频繁 1 项集 F_1 并记录它们的支持度.对 F_1 进行排序,按支持度降序排序,得到有序的频繁 1 项集 S_1 .

(2)创建一个根节点 root,将 ItemName 设置为 null. 遍历每一个事务 $T_i (1 \leq i \leq n)$, 删除 T_i 中的非频繁项并按照 S_1 的顺序重新排序. 得到一个新的事务 $T_i[p|P]$, p 为新事务的第一项, P 为剩下的所有项的集合. 调用 $Insert(T_i[p|P], root)$ 方法.

(3)扫描 PPC-Tree 为每个节点产生先序编号 PreOrder 和后续编号 PosOrder 次序.

其中函数 $Insert(T_i[p|P], root)$ 的过程: 如果 root 节点存在孩子节点 N , 并且 $N.ItemName = p$. ItemName 那么令 $N.Count = N.Count + 1$, 否则新建一个新的节点 N , 令 $N.ItemName = p$. ItemName. $N.Count = 1$. 递归调用 $Insert$ 函数, 直到 P 为空函数结束.

3.2 PP-code 和 N-list 的定义

PPC-Tree 和 Fp-Tree 最大的不同就在 PPC-Tree 是为了产生频繁项的 PP-code, 而 Fp-Tree 是直接递归 Fp-Tree 找出频繁项集.

定义 4 (PP-code) 对于 PPC-Tree 中的每个节点 N , 那么 $(N.count, [N.preOrder, N.posOrder])$ 就是一个 PP-code.

定义 5 (PP-code 间的祖先关系) 给定两个 PP-code X_1 和 X_2 , 如果 X_1 是 X_2 的祖先节点, 则一定有 $X_1.preOrder < X_2.preOrder$ 并且 $X_1.posOrder > X_2.posOrder$.

定义 6 (N -List) 频繁项 I_i 在 PPC-Tree 中对应的所有 PP-code 组成的集合叫 N -list.

定义 7 (N -List 的支持度) 给定一个频繁项的 N -list, $\{(z_1, [x_1, y_1]), (z_2, [x_2, y_2]), \dots, (z_n, [x_n, y_n])\}$, 则该频繁项的支持度为支持度为 $z_1 + z_2 + \dots + z_n$.

4 PPF 并行算法

PPF 算法是在 N -list 基础上提出的一种并行算法. 算法分为四个阶段, 第一个阶段扫描事务数据库

构建 PPC-Tree,通过 PPC-Tree 得到各个频繁项对应的 N -list. 第二个阶段利用频繁 1 项集的 N -list 产生频繁 2 项集 N -list. 第三个阶段将第二个阶段产生的频繁 2 项集 N -list 分布到不同的节点中深度挖掘出 K 频繁项集. 第四个阶段将第三个阶段产生的 K 频繁项集汇总. 下面具体介绍算法的具体细节和相关理论.

4.1 基于 N -list 的频繁项集挖掘

4.1.1 频繁 2 项集的挖掘

```

 $d \rightarrow (5, [1, 7])$ 
 $c \rightarrow (2, [3, 5]) \text{ --- } (1, [6, 6])$ 
 $dc \rightarrow (2, [1, 7]) \text{ --- } (1, [1, 7])$ 
 $dc \rightarrow (3, [1, 7])$ 

```

如上所示,频繁 2 项集的过程就是比较 2 个频繁 1 项集的 N -list 中的 PP-code 节点是否有祖先关系. 如果有则加入到新的 2 项集 N -list 中,最后如果 PP-code 中的 [PreOrder, PosOrder] 相同表示同一个节点需要合并. 因此,dc 的支持度为 3.

频繁 2 项集的挖掘算法的具体步骤如下:

算法 2: 频繁 2 项集挖掘算法

输入: 2 个频繁 1 项集的 N -list: NL_1, NL_2

输出: 频繁 2 项集 N -list: NL_3

```

1  $i \leftarrow 1; j \leftarrow 1;$ 
2 While  $i \leq NL_1.size \& \& j \leq NL_2.size$  Do
3 If  $(x_{1i} < x_{2j})$  then
4 if  $(y_{1i} > y_{2j})$  then
5   insert  $(z_{2j}[x_{1i}, y_{1i}])$  into  $NL_3$ 
6    $j++$ 
7 else
8    $i++$ 
9 end if
10 else
11    $j++$ 
12 end if
13 end while
14 合并  $NL_3$  中的相同节点
15 Return  $NL_3$ 

```

4.1.2 频繁 K 项集的挖掘

频繁 k 项集的挖掘和频繁 2 项集的挖掘过程相似,通过遍历 2 个长度为 $(k-1)$ 的 N -list,查找出 2 个 N -list 中满足祖先关系的 PP-code,并且把新的 PP-code 插入到长度为 K 的 N -list 中,最后再合并长度为 k 的 N -list 结合中相同 PP-code. 因此利用 4.1.1 节中计算出的频繁 2 项集的 N -List,PPF 算法可以挖掘出频繁 3 项集,频繁 4 项集... 频繁 K 项集(所有的频繁项集).

频繁 K 项集挖掘算法的具体步骤如下:

算法 3 频繁 K 项集挖掘算法

输入: 最小支持度, 同底频繁 2 项集 L_2 , 以及对应的 N -list NL_2, L_2 按支持度降序排列

输出: 频繁项集 F

```

1 For  $i \leftarrow L_k.length - 1$  to 1 Do
2    $L_{k+1}^i \leftarrow null, NL_{k+1}^i \leftarrow null$ 
3   For  $j \leftarrow i - 1$  to 0 Do
4   假设  $L_k[i] = x_1 x_2, \dots, x_k$  并且  $L_k[j] = y_1 y_2, \dots, y_k$ 
5    $l \leftarrow x_1 y_1 x_2, \dots, x_k$  // 合并两个同项集
6   If  $L_k.count > M$  Then
7      $l_{k+1}^i \leftarrow l_{k+1}^i \cup \{l\}$  &
8      $Nl_{k+1}^i \leftarrow Nl_{k+1}^i \cup \{L_k N-list\}$ 
9   EndIf EndFor
10 If  $l_{k+1}^i \neq null$  Then
11   If  $l_{k+1}^i.length = 1$ 
12      $F = F \cup \{l\}$ 
13 Else
14   mine  $(l_{k+1}^i, NL_{k+1}^i)$ 
15 End If
16 Return  $F$ 

```

4.2 可并行性分析

定义 8 (同底频繁项集) 有频繁项集 $x_1 x, x_2 x, \dots, x_n x$, 则这 n 个频繁项集被称为同底频繁项集, 相同的底为 x .

同底频繁 2 项集经过 4.1 节中的算法进行深度挖掘, 就可以挖掘出所有的同底频繁 k 项集. 不同的同底 2 项集之间是不存在依赖关系的, 因此可以并行地进行挖掘.

例 有 2 组同底频繁 2 项集 $\{dc, bc\}, \{de, ae, ce\}$, 利用 $\{dc, bc\}$ 可以深度挖掘出所有以 c 为底的同底频繁项集 $\{dc, bc, dbc\}$, 利用 $\{de, ae, ce\}$ 可以深度挖掘出所有以 e 为底的频繁项集 $\{de, ae, ce, dae, dce, ace, dace\}$. 并且这 2 组频繁项集之间是不存在依赖关系的, 因此完全可以并行执行.

所以 PPF 并行算法将所有的频繁 2 项集按底划分数据块, 将不同的数据块分配给相应的节点, 进行并行挖掘. 由于每个数据块的长度不同需要的运行时间也不同, 为了提高并行的效率, PPF 算法提出了一种基于 N -list 的数据块分配策略.

4.3 PPF 并行算法的数据划分和分配策略

如图 1 所示, PPF 并行算法的数据划分用在数据的预处理阶段之后, 数据块的分配用在数据划分阶段之后. 数据的划分, PPF 算法经过 4.1.1 节的频繁 2 项集挖掘算法的处理得到一系列的频繁 2 项

集,把这些频繁 2 项集按不同的底进行划分,就得到了图 1 中的 K 个数据块.

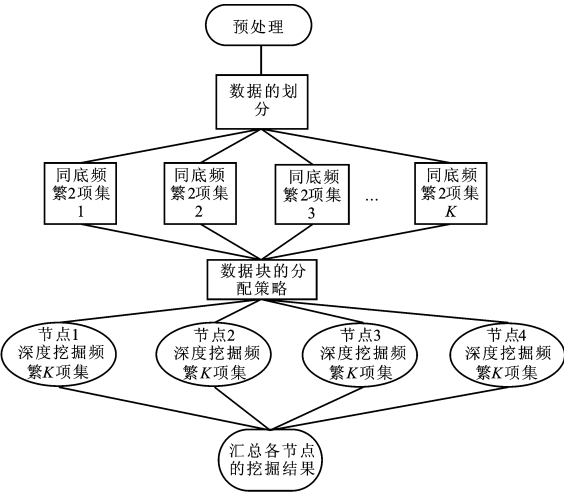


图 1 PPF 算法的数据块的划分和分配

对每个数据块的挖掘的时间消耗和数据块的大小有关,数据块越大需要的时间越多. 因此合理地分配这 K 个数据块,才会使 PPF 算法的效率更高.

分配策略优化目标如下式所示,

$$\min(\max_{1 \leq n \leq N} \sum_{nlist_k \in T_n} t_k) \quad (1)$$

式中, t_k 表示每个数据块的估计耗时, 本文采用 N -List 的规模来近似估计计算耗时, 即 $t_k = |nlist_k|$. 如上可知, 本文的数据分配策略的目的是使计算时间最长的节点的耗时最小化. 然而该问题为 NP 困难问题, 对此本文借鉴了降序最适算法 (Best Fit Decreasing, BFD) 的思想, 提出了一种启发式近似分配策略, 分配策略的具体步骤如下:

(1) 将预处理后的数据块进行耗时估计, $t_k = |nlist_k|$, t_k 表示第 K 个数据块的估计耗时.

(2) 对这 K 个数据块按照 t_k 进行降序排序.

对排好序的这 K 个数据块进行遍历, 依次将这 K 个数据块分配给当前状态下 N 个节点中.

(3) $\text{sum}T_n$ 最小的那个节点. $\text{sum}T_n$ 表示第 n 个节点的总耗时.

(4) 每分配一个数据块就更新当前节点的总耗时, $\text{sum}T_n = \text{sum}T_n + t_k$.

4.4 PPF 并行算法的具体步骤

PPF 算法的具体步骤如下:

(1) 利用 3.1 节中的算法 1 扫描数据库构建 PPC-Tree.

(2) 通过 PPC-Tree 生成频繁 1 项集.

(3) 将第二阶段产生的频繁 1 项集通过算法 2

生成频繁 2 项集.

(4) 对频繁 2 项集进行数据块的划分, 将相同底的频繁 2 项集合并到一起.

(5) 将步骤 4 中的同底频繁 2 项集通过分配策略分配给相应的节点.

(6) 在这 n 个节点中同时调用算法 3 进行频繁 K 项集的深度挖掘.

(7) 最后汇总这 n 个节点中的所有数据.

5 实验对比

为了验证 PPF 并行算法的高效性, 本文采用了 4 个节点, 做了 4 组实验. 每组实验分别在不同的数据集上, 对 PrePost 算法和本文提出的 PPF 算法进行了测试和分析. 数据集分别是 mushroom, connect, pumsb, T10I4D100K. 其中 mushroom, connect, pumsb 为稠密数据集^[10] T10I4D100K 为稀疏数据集. 表 1 给出了这四种数据集的各参数指标.

表 1 数据集参数

数据集	大小/kB	项数	事务数
mushroom	558	119	8 124
connect	9 031	129	67 557
pumsb	16 280	2 113	49 046
T10I4D100K	3 928	870	100 000

注: 最小支持度越小, 表示挖掘的数据量越大, 因此算法所用时间越多.

图 2 是两种算法在 mushroom 数据集上不同 minSupport 下的运行时间对比, 绿色线是 PrePost 算法的运行时间, 从图中我们可以明显看出随着最小支持度的降低, PrePost 算法的运行时间快速增长. 蓝色线是 PPF 并行算法的运行时间, 从图中可以看出随着最小支持度的不断降低 PPF 算法的运行时间明显比 PrePost 算法少. 因为 PPF 算法通过并行化处理随着最小支持度的降低 (频繁项集数据量的增大), PPF 算法的优势就越大, 因此 PPF 算法的时间明显比 PrePost 算法少.

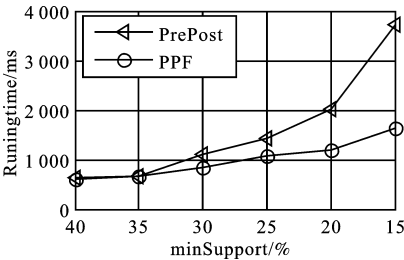


图 2 PrePost 和 PPF 算法在 mushroom 上的运行时间

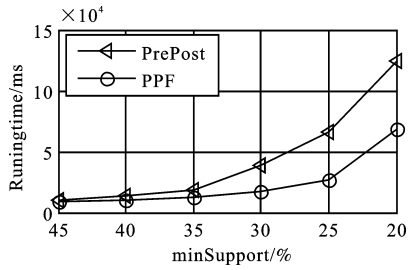


图 3 PrePost 和 PPF 算法在 connect 上的运行时间

图 3 和图 4 分别是 PrePost 算法和 PPF 算法在 connect 和 pumsb 数据集上的时间对比图. 由于 connect 和 pumsb 这两种数据集和 mushroom 数据集一样都属于稠密数据集, 因此图 3 和图 4 的曲线变化和图 1 很相似.

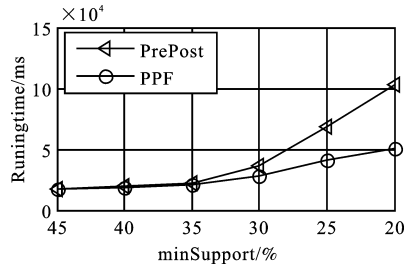


图 4 PrePost 和 PPF 算法在 pumsb 上的运行时间

图 5 是两种算法在 T10I4D100K 数据集上不同 minSupport 下的运行时间对比, 绿色线是 PrePost 算法的运行时间, 从图中我们可以明显看出随着最小支持度的降低, PrePost 算法的运行时间平稳增长. 蓝色线是 PPF 并行算法的运行时间, 从图中可以看出随着最小支持度的不断降低 PPF 算法的运行时间也是平稳增长, PPF 算法略优于 PrePost 算法. 这是由于 T10I4D100K 数据集是一个稀疏数据集, 它的频繁项不是很集中.

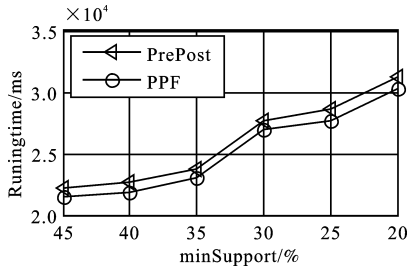


图 5 PrePost 和 PPF 算法在 T10I4D100K 上的运行时间

6 结束语

本文提出的基于 N-List 的并行频繁项集挖掘算法 PPF 算法充分利用了并行计算的优势, 从而有效地解决了传统频繁项集挖掘算法存在的计算能力不足的

问题. 通过实验表明, 改进后的算法较 PrePost 算法有明显提高. 因此, 把该方法运用于大数据的挖掘, 将能为企业更快、更准的决策支持. PPF 算法虽然优于 PrePost 算法, 但是在稀疏数据集上的效果不是很明显, 因此本文下一步将研究稀疏集的挖掘特点, 进一步提高 PPF 算法在稀疏集上的效率.

参考文献:

[1] Zhong S C. Improving algorithm apriori for data mining[C]// Computational Intelligence in Decision & Control international Flins Conference. Madrid, Spain, 2015:17-22.

[2] 李也白, 唐辉, 张淳, 等. 基于改进的 FP-tree 的频繁模式挖掘算法[J]. 计算机应用, 2011, 31(1): 101-103.

[3] Wang B L, Shen Y G. Improvement of apriori algorithm based on boolean matrix[J]. Procedia Engineering, 2011, 15(1):1827-1831.

[4] Li T, Luo D. A new improved apriori algorithm based on compression matrix[M]. Berlin: Springer International Publishing, 2014:1-15.

[5] Wei Y Q, Yang R H, Liu P Y. An improved apriori algorithm for association rules of mining[C]// IEEE International Symposium on It in Medicine & Education. Jinan, China, 2009:942-946.

[6] Grudzinski P, Wojciechowski M. Integration of candidate hash trees in concurrent processing of frequent itemset queries using Apriori. [J]. Control & Cybernetics, 2009, 38(1):47-64.

[7] 吴倩, 罗健旭. 压缩 FP-Tree 的改进搜索算法[J]. 计算机工程与设计, 2015, 35(7):1771-1777.

[8] Salah S, Akbarinia R, Masegla F. Fast parallel mining of maximally informative k-itemsets in big data [C]// IEEE International Conference on Data Mining. Atlantic City, NJ, USA, IEEE, 2015:359-368.

[9] Deng Z H, Wang Z H, Jiang J J. A new algorithm for fast mining frequent itemsets using N-lists[J]. Science China Information Sciences, 2012, 55(9): 2008-2030.

[10] Nguyen G, Le T, Vo B, et al. EIFDD: An efficient approach for erasable itemset mining of very dense datasets [J]. Applied Intelligence, 2015, 43(1):85-94.

作者简介:

陈 奇 男, (1991-), 硕士研究生, 研究方向为数据挖掘. E-mail: 876447590@qq.com.

张曦煌 男, (1962-), 博士, 教授, 研究方向为数据挖掘、嵌入式系统.