基于哈希表与十字链表存储的 Apriori 算法优化

吴 昊 刘 钊 顾进广
(武汉科技大学计算机科学与技术学院 湖北 武汉 430065)
(武汉科技大学大数据科学与工程研究院 湖北 武汉 430065)
(湖北省智能信息处理与实时工业系统重点实验室 湖北 武汉 430065)

摘 要 Apriori 算法在数据挖掘过程中需要多次扫描数据库,会造成 I/O 上有较大时间开销和负载,影响算法的运行速度,同时在计算频繁项集的过程中,需要进行大量迭代搜索与计算,算法的时间复杂度和空间复杂度较高。基于此,提出一种基于哈希表与十字链表存储的优化算法 HTACL-Apriori。通过理论分析和数据进行实验对比,验证了优化后的 HTACL-Apriori 算法相对于传统的 Apriori 算法在时间效率和空间效率方面有明显的提高,达到了预期效果。

关键词 时间复杂度 空间复杂度 哈希表 十字链表 布尔矩阵

中图分类号 TP311 **文献标志码** A DOI:10.3969/j.issn.1000-386x.2022.07.038

OPTIMIZATION OF APRIORI ALGORITHM BASED ON HASH TABLE AND CROSS-LINKED LIST STORAGE

Wu Hao Liu Zhao Gu Jinguang
(*School of Computer Science and Technology, Wuhan University of Science and Technology, Wuhan 430065, Hubei, China*)
(*Big Data Science and Engineering Research Institute, Wuhan University of Science and Technology, Wuhan 430065, Hubei, China*)
(*Hubei Province Key Laboratory of Intelligent Information Processing and Real-Time Industrial, Wuhan 430065, Hubei, China*)

Abstract The Apriori algorithm needs to scan the database multiple times during the data mining process, which will cause a large time overhead and load on the I/O and affect the speed of the algorithm. In the process of computing frequent itemsets, it needs to perform a large number of iterative searches and calculations, and the time complexity and space complexity of the algorithm are high. Therefore, this paper proposes an optimized algorithm HTACL-Apriori based on Hash table and cross-linked list storage. Through the theoretical analysis and experimental data, it is verified that compared with the traditional Apriorialgorithm, HTACL-Apriori significantly improves the time efficiency and space efficiency, and has achieved the expected results.

Keywords Time complexity Space complexity Hash table Cross linked list Boolean matrix

0 引 言

虽然 Apriori 算法原理简单,算法容易实现,但是该算法的时间复杂度和空间复杂度比较高,导致在计算频繁项集的过程中时间效率和空间效率比较低。针对传统 Apriori 算法在时间复杂度和空间复杂度上的不足,文献[1]提出了使用优化的链表数据结构进行存储,并提高支持计数效率,同时采用了候选生成方法来减少匹配候选项目集。文献[2]提出了一种基于 MapReduce 的频繁项集挖掘方法,在云计算中引入了 MapReduce 模型来实现 Apriori 算法并行化。文献[3]提出了一种基于标记事务压缩改进的 Apriori 算法,该算法优化了关联规则的参数,减少标签比较的次数。

收稿日期:2020-02-27。国家自然科学基金项目(61673304);国家社科基金重大计划项目(11&ZD189)。**吴昊**,硕士生,主研领域:数据挖掘,智能计算。**刘钊**,教授。**顾进广**,教授。

文献[4]提出了编码和适应度函数的方案,建立了关联规则的模式,提高了算法的效率和准确性。廖纪勇等[5]提出了一种基于布尔矩阵约简的 Apriori 改进算法,将矩阵各列元素按照支持度升序排列,使得算法在压缩过程中减少了扫描矩阵各列的次数,缩短了算法的运行时间。李洁等[6]提出一种基于哈希存储与事务加权的并行 Apriori 改进算法,通过哈希存储的去重特性,减少冗余计算,开启多个线程,并行计算候选集的支持度,从而提高 Apriori 算法的运行效率。文武等[7]提出了一种结合遗传算法的 Apriori 算法来挖掘频繁项集,结合 Apriori 算法和遗传算法的特点,利用交叉算子产生候选项集和变异算法筛选频繁项集。赵学健等[8]提出了一种利用正交链表存储数据所对应的关系矩阵,同时优化了传统的 Apriori 算法的自连接和剪枝过程,合理缩小了搜索空间。

1 Apriori 算法

1.1 数据关联规则原理

假定数据集为 D,$Item=\{I_1,I_2,\cdots,I_m\}$ 表示数据库所有数据的集合,$I=\{t_1,t_2,\cdots,t_k\}$ 用来表示不同的事务集,每个 t_i 对应一个事务。i_s、i_t 是 I 中任意一个项集,并且 i_s 中含有事务的数量记为 n,表示 n 项集,根据数据集 D 中生成的关联规则表示 $i_s \geqslant i_t$,同时 $i_s \subseteq I$,$i_t \subseteq I$,$i_s \cap i_t$ 为空集,i_s 与 i_t 是通过最小支持度(Support)和最小置信度(Conf)来对关联规则进行约束的。频繁项集的支持度表示对应关联规则的频度,频繁项集的置信度表示对应的关联规则的强度[9]。

关联规则 $i_s \Rightarrow i_t$ 在 D 中的频繁项集的支持度是指 $Item$ 中同时包含 i_s、i_t 的数量与 $Item$ 的总的数量之比[9],记作:

$$Support(i_s \Rightarrow i_t) = |\{I: i_s \cup i_t \subseteq I, Item \in D\}| / |D|$$

关联规则 $i_s \Rightarrow i_t$ 在数据集 D 中的频繁项集的置信度是指 $Item$ 中同时包含 i_s 项集和 i_t 项集的数量与 $Item$ 中包含 i_s 项集的数量之比,记作:

$$Conf(i_s \Rightarrow i_t) = |\{I: i_s \cup i_t \subseteq I, I \in D\}| / |\{I: i_s \subseteq I, I \in D\}|$$

在数据集 D 中利用算法挖掘有效的关联规则就是为了找出符合用户给定的最小支持度(Min_Support)和最小置信度(Min_Conf)的关联规则,当利用算法挖掘出来的关联规则的置信度和支持度分别大于 Min_Support 与 Min_Conf 时,则认为该关联规则是有效的,称为强关联规则[9]。

1.2 Apriori 算法过程

Apriori 算法计算频繁项集的主要原理:首先利用数据的关联规则计算出候选 1 项集(简称 C_1);然后通过 C_1 自连接的方式计算生成候选 2 项集(简称 C_2);然后由候选 2 项集继续通过自连接的方式来扩展生成候选 3 项集(简称 C_3);接下来按上述方法一直迭代计算下去,直到判断不能继续扩展生成下一项项集时,表示该迭代计算的过程结束,算法结束。

算法具体实现步骤如下:

(1) 首先对数据库中的数据进行预处理,然后扫描数据库中数据中的事务,统计出每个事务对应的支持度频数,即可生成候选 1 项集(简称 C_1),如果候选 1 项集中的事务支持度频数满足最小支持度频数,经过筛选之后就可以生成频繁 1 项集(简称 L_1)。

(2) 将 L_1 中项集进行自连接,由此可以通过 L_1 自连接生成候选 2 项集合(简称 C_2)。

(3) 通过 C_2 继续遍历数据库中每条数据的事务,利用计算每一个候选项集的支持度频数,如果候选项集支持度频数满足最小支持度频数,经过筛选后就生成频繁 2 项集(简称 L_2)。

(4) 将 L_2 中项集进行自连接,由此可以通过 L_2 自连接生成候选 3 项集合(简称 C_3)。

(5) 通过 C_3 继续遍历数据库中每条数据的事务,计算每一个候选项集的支持度频数,如果候选项集支持度频数满足最小支持度频数,经过筛选后就生成频繁 3 项集(简称 L_3)。

(6)将上述计算过程一直迭代进行,直到候选 k 项集为空就停止计算,算法结束,符合最小支持度的项集称为频繁项集[10]。

由上述过程可以看出 Apriori 算法在时间效率和空间效率方面有以下不足之处[9]:

(1) 首先将数据进行预处理之后,每计算一次项集的频数就需要访问数据库,并对数据库每一条数据进行扫描,因此扫描的次数比较多,这样将会造成 I/O 上有比较大负载和时间开销,降低了算法的计算性能。

(2) Apriori 算法计算过程中会产生大量冗余候选项集。例如频繁 n 项集经过自连接扩展生成所有的候选 $n+1$ 项集 C_{n+1},扫描数据库后就需要进行多次自连接产生冗余的项集,同时生成的候选项集与最小支持度比较次数增多。这样造成了内存空间占用和计算时间的浪费。

(3) 在频繁项集进行扩展生成下一项的时候,测试 $C(C \in C_k)$ 里每个 $k-1$ 项集是否是 L_{k-1} 中频繁 $k-1$ 项集,扫描 k 次 L_{k-1},对于 L_{k-1} 扫描的时间复杂度为 $O(k \times |L_{k-1}|/2)$,所以扫描每一个候选 k 项集 C_k 的时间复杂度为 $O(C_k \times k \times |L_{k-1}|/2)$,在计算过程中需要

消耗大量时间[11]。

2　HTACL-Apriori 算法

2.1　十字链表表示方法

十字链表节点的结构如图 1 所示,Item 为项目名,Number 表示 Item 部分节点数目,Down 表示向下指向其他事务部分,Right 表示向右指向其他事务部分[11]。

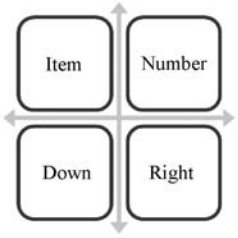

图 1　十字链表结构

2.2　HTACL-Apriori 挖掘算法原理

假定 $D=\{I_1,I_2,\cdots,I_n\}$ 表示是所有数据的集合,$I=\{t_1,t_2,\cdots,t_n\}$ 表示每条数据中的每一个事务。其中 $t_i(i=1,2,\cdots,n)$ 表示每条数据中第 i 个事务,同时 t_i 对应 D 上的一个子集($t_i\subseteq D$)。

定义 1　对于每一个项 P_j 定义为:

$$P_j=(X_{1j},X_{2j},\cdots,X_{mj})$$

数据库 D 中的事务其对应的类型的矩阵为 $\boldsymbol{M}_i=(X_{i1},X_{i2},\cdots,X_{in})^{\mathrm{T}}$,其中 X_{ij} 表示事务存在情况,即 $X_{ij}=\begin{cases}1 & X_{ij}\in D\\ 0 & X_{ij}\notin D\end{cases}$,则项目 P_j 的支持度频数为 $Support\_count(P_j)=\sum_{i\geqslant 1}X_{ij}$。

性质 1　矩阵中存储的每行元素都对应事务数据库中的一个事务,每个事务中在矩阵用 1/0 表示该项目是否存在,这样可以将所有事务都存储在矩阵中(设定为 1),后面就可以利用矩阵计算项集的频数[10]。

定理 1　如果集合 X 的频数小于最小支持度频数,则集合 X 一定不是频繁项集,如果存在 X 集合,且 $Y\subseteq X$,可以判断 Y 集合一定不是频繁项集[12]。

定理 2　如果存在集合 X,且集合 X 支持度频数大于最小支持度频数,则可以判定集合 X 为频繁项集,如果存在 Y 集合,且 $Y\subseteq X$,可以判断 Y 集合同样是频繁项集[12]。

推论 1　假设频繁 k 项集合个数记为 $|L_k|$,如果 $|L_k|<k+1$,说明只能扩展生成频繁 k 项集,一定不存在 $k+1$ 项集,反之,如果 $|L_k|\geqslant k+1$,说明可以继续扩展 $k+1$ 项集。

3　HTACL-Apriori 算法的优化

3.1　算法优化思想

通过 3 个步骤对传统的 Apriori 算法进行改进优化:

(1) 将数据库中的事务映射到布尔矩阵进行存储(0 表示不存在,1 表示存在),这样避免多次访问数据库扫描数据,只需要将数据一次性存储在布尔矩阵中,后面利用布尔矩阵的特性来进行计算项集的频数,极大地减少了计算过程中的 I/O 时间开销与负载,优化了算法的运行速度[13]。

(2) 由于链表具有动态插入和动态删除的优点,因此对每次生成的项集和对应频数作为一个节点插入到链表中,然后对链表节点进行部分选择排序,经过部分选择排序,将不满足最小支持度频数的项集直接删除(满足最小支持度的项集不用排序),在后面的计算过程中减少了一些无效的自连接匹配运算,优化了算法运行速度和内存空间。

(3) 利用布尔矩阵的特性减少匹配时间。十字链表中节点的 Item 部分转换为布尔向量和布尔矩阵中每一行进行“与”操作,然后利用定义 1 来计算项集在数据中出现的频数,其过程所需要的时间小于事务数据之间自连接后再逐一判断每个事务是否匹配所需要时间,通过布尔矩阵的特性来判断,减少了一些无效自连接匹配运算,减少了时间消耗。

(4) 利用十字链表交叉正交存储频繁项集。根据定理 1 和定理 2,利用哈希表对项集进行剪枝和去除重复项集,然后存储在十字链表中,极大地减少了内存占用空间和冗余项集计算频数的时间[13]。

3.2　算法的优化步骤

输入:数据库中数据经过预处理后的数量有 N 条,假定最小支持度为 $Min\_Support$,则最小支持度频数为 $Min\_support\_Count=N\times Min\_Support$。

输出:输出符合条件的频繁项集。

算法优化的具体步骤如下:

(1) 将数据库中的数据经过预处理后,根据性质 1 将每条数据中的事务按照 0/1 分布(0 表示不存在,1 表示存在)映射到布尔矩阵中,遍历矩阵,利用定义 1 计算出候选 1 项集对应的频数。

(2) 将候选 1 项集和对应频数作为节点插入到链表中,然后对链表中项集的频数进行部分选择排序,只需要将不满足最小支持度的项集频数对应的链表节点

筛选出来(不需要链表所有节点全部排序)进行删除,同时利用哈希表 *hash_unfrequent* 存储非频繁项集(后面过程中利用定理 1 可以剪枝优化,减少计算时间),利用哈希表 *hash_ frequent* 存储频繁项集(后面过程利用定理 2 可以剪枝优化,减少计算时间),就生成了频繁 1 项集链表 L_1。

(3) 利用步骤(2)得到的频繁 1 项集的链表进行交叉正交,然后将交叉正交的项集用哈希表 *hash_ unique* 存储(表示该项集已经出现,在后面项集的生成过程中可以去除重复项集,减少计算时间和冗余数据内存空间),将十字链表中节点 Item 部分转换为对应布尔向量(0 表示不存在,1 表示存在),然后与布尔矩阵中每条数据进行“与”运算,如果判断在该条数据中存在该项集,利用定义 1 可以计算出每个项集的对应的频数,然后链表交叉正交生成项集时,首先用哈希表 *hash_unfrequent* 来判断,如果符合定理 1 的情况的非频繁项集进行剪枝优化(不用计算项集支持度频数,不用将节点插入到十字链表中,继续下一步),然后用哈希表 *hash_ frequent* 来判断,如果符合定理 2 情况的频繁项集进行剪枝(直接将项集的频数设置为最小支持度频数,然后将节点直接插入到十字链表中)。接着用哈希表 *hash_unique* 判断是否出现重复项集,如果哈希表 *hash_unique* 判断已经存在,继续下一步,不用将节点插入到十字链表中,反之,将节点插入到十字链表中。这样就生成候选 2 项集的十字链表,遍历十字链表中的每一个节点,将节点插入到新的链表当中。此时就生成候选 2 项集链表 C_2,将链表进行部分选择排序,动态删除小于最小支持度的链表节点,同时将非频繁项集插入到哈希表 *hash_unfrequent* 中,将频繁项集插入到哈希表 *hash_ frequent* 中,最后生成频繁 2 项集链表 L_2。

(4) 利用推论 1 来判断 $|L_k| < k+1$,如果该条件成立,则表明无法继续扩展生成下一项频繁项集,算法计算过程结束;反之,就可以继续进行计算下一项频繁项集。重复步骤(2)和步骤(3)中的计算频繁项集的过程,继续利用第 k 项频繁项集扩展生成第 $k+1$ 项频繁项集。

(5) 循环重复步骤(4)。如果满足 $|L_k| < k+1$ 条件,则无法扩展下一项频繁项集,算法计算过程结束。

HTICL-Apriori 算法优化思想伪代码描述如下:

```
HTICL-Apriori( )
{
//将数据库中的每条数据映射存入布尔矩阵 Boolean_Matrix
//(M×N,其中:N 为数据中事务最多的数量;M 为数据条数)。
//用 0/1 区别,1 表示存在,0 表示不存在
//数组 origin 存储扫描数组后的候选项集 C1,哈希表 hash_
//frequent 用来存储生成的频繁项集,hash_unfrequent 用来存储
//生成的非频繁项集,hash_unique 存储链表交叉正交生成的项
//集,用于去除重复项集
FOR(i=1;i<=N;i++) DO
  FOR(j=1;j<=M;j++) DO
    IF Boolean_Matrix[i][j]>0
      origin[i].count++;
  END FOR
END FOR
//将 origin 中的项集和对应的频数插入 P1 链表中
P1.insert(origin[i](i=1,2,…,n))
HTACL-Apriori_Sort(P1);
//对链表 P1 中每一个项集对应的频数进行部分选择排序
//遍历 Pk+1
IF(P1.number>=Min_support_Count)
  hash_ frequent.insert(P1.frequent);
  //存储满足最小支持度的项集
Else
  hash_unfrequent.insert(P1.unfrequent);
                    //存储不满足最小支持度的项集
  delete(P1.unfrequent);
                    //将链表 P1 中非频繁项集的部分删除
WHILE(Pk.length( )>=k+1&&Pk!=∅)
{
根据定理 1 可知,由频繁 1 项集事务进行转置和频繁 1 项集进
行内积变换,生成二维布尔矩阵
FOR(i=1;z<|Pk|.length( );i++) DO
FOR(j=i+1;y<|Pk|.length( );j++) DO
  Ck=Pk.Item[i]∪Pk.Item[j]          //交叉正交
  //根据定理 1 和定量 2 判断
  IF Ck∈hash_unfrequent
    Pk 同时向右和向下移动
    Continue
  IF Ck∈hash_ frequent
    Pk.Ck.number=Min_Support_Count;
    Pk 同时向右和向下移动
    Continue
  IF Ck∈hash_unique                 //项集已经出现
    Pk 同时向右和向下移动
    Continue
  hash_unique.insert(Ck);
  //hash_unique 插入 Ck,方便在十字链表中去除存储重复项集
  将 Ck 中的项集先转换成布尔向量,用 0/1 区别,1 表示存
在,0 表示不存在
  WHILE 矩阵 Matrix
  Ck 与布尔矩阵 Matrix 每一行对应的列进行“与”运算,如
```

果判断在该条数据中存在该项集,利用定义 1 计算项集频数。

$P_k.C_k.number++$;　　//对该项集计数

Cross_Listk. insert(P_k)　　//将该节点 P_k 插入十字链表中

END FOR

END FOR

将 $Cross\_List_k$ 的向下和向右的尾端设置为 NULL

将十字链表 $Cross\_List_k$ 中生成的候选项集全部插入到链表 P_{k+1} 中。

HTACL-Apriori_Sort(P_{k+1})//对链表 P_{k+1} 中每一个项集对应的

//频数进行部分选择排序遍历 P_{k+1}

IF($P_{k+1}.number >= Min\_support\_Count$)

hash_ frequent. insert($P_{k+1}.frequent$);

//存储频繁项集。

Else

hash_unfrequent. insert($P_{k+1}.unfrequent$);

//存储非频繁项集

Delete($P_{k+1}.unfrequent$);

//将链表 P_{k+1} 中非频繁项集的部分删除

END WHILE

}

HTACL-Apriori_Sort(P_k)、

{

//链表节点中的 *number*,用部分选择排序(把项集按照最小支

//持度分成两个部分)

调用选择排序算法 Sort(P_k)

}}

3.3　算法的优化示例说明

假定关联规则的最小支持度为 0.4,数据量有 10 条,首先将数据进行预处理,将每条数据中的事务按照字母序列递增排序(如表 1 所示),然后可以计算出最小支持度频数 $Min\_Support\_Count = 10 \times 0.4 = 4$,同时按照以下步骤来计算频繁项集。

表 1　数据表

Tid	Item
I_1	d,e,f,g
I_2	b,d,e,h
I_3	d,e,g,h
I_4	b,c,d,e,h
I_5	a,d,e,h
I_6	a,b,e,g,h
I_7	a,b,d,e
I_8	a,d,g,h
I_9	b,d,f
I_{10}	b,c,d,e,g

(1) 将经过预处理的数据根据性质 1 按照 0/1 分布映射到布尔矩阵中对应的位置来表示对应事务是否存在,矩阵中的行表示每条数据,每一列表示每一个事务,转换成对应矩阵 ***Boolean_Matrix***(本示例中以事务 a,b,c,d,e,f,g,h 为行)。

$$\textbf{\textit{Boolean\_Matrix}} = \begin{bmatrix} 0&0&0&1&1&1&1&0 \\ 0&1&0&1&1&0&0&1 \\ 0&0&0&1&1&0&1&1 \\ 0&1&1&1&1&0&0&1 \\ 1&0&0&1&1&0&0&1 \\ 1&1&0&0&1&0&1&1 \\ 1&1&0&1&1&0&1&0 \\ 1&0&0&1&0&0&1&1 \\ 0&1&0&1&0&1&0&0 \\ 0&1&1&1&1&0&1&0 \end{bmatrix}$$

(2) 遍历布尔矩阵中的每一个事务,根据定义 1 可以计算出该数据集中每个事务对应的频数,数据如表 2 所示。

表 2　数据中事务表

Item	number
a	4
b	6
c	2
d	9
e	8
f	2
g	6
h	6

(3) 按照每个事务对应的数量进行部分选择排序,将小于最小支持度的项集排在最后,然后将满足最小支持度的项集插入到链表中,由此可以得到新链表,同时将链表中频繁项集全部插入到哈希表 *hash_frequent* 中,将链表中非频繁项集全部插入到哈希表 *hash_unfrequent* 中。

(4) 由此可以计算出频繁 1 项集链表(如图 1 所示),即 $L_1 = \{\{a\},\{b\},\{d\},\{e\},\{g\},\{h\}\}$。同时计算出频繁 1 项集的个数 $|L_1| = 6 \geq K+1$。根据推论 1 可以得出可以由 L_1 继续向候选 2 项集进行扩展,将链表 L_1 交叉正交生成候选 2 项集,首先根据定理 1 判断该项集是否在哈希表 *hash_unfrequent* 中存在,如果存在则进行剪枝(不必计算项集的频数,不必插入十字链表,继续下一步计算),继续根据定理 2 判断是否在哈希表 *hash_ frequent* 中存在,如果存在(直接设置为

最小支持度,插入十字链表),继续判断该项集是否存在于哈希表 *hash_unique*,如果存在则去重(不必计算项集的频数,不必插入十字链表,继续下一步),将生成候选 2 项集插入哈希表 *hash_unique*(去除重复项集),最后将生成的项集转换为布尔向量(0 表示该事务不存在,1 表示该事务存在),通过定义 1 进行计算对应的频数,如果频数不满足最小支持度,就将项集插入哈希表 *hash_unfrequent* 中,反之插入哈希表 *hash_frequent* 中。经过上述步骤,就可以生成交叉正交后十字链表,如图 2 所示。

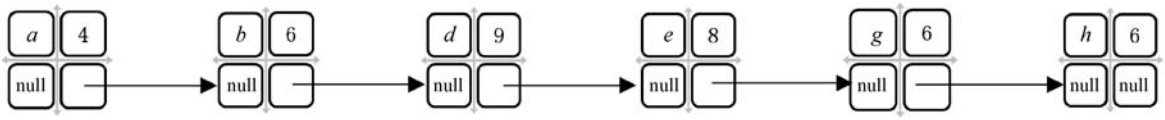

图 2 频繁 1 项集链表

(5) 遍历十字链表中每一个节点,并且插入在新链表中,将新链表节点中的频数按照最小支持度频数进行部分选择排序,将链表部分非频繁项集进行删除,这样就生成新链表(如图 3 所示),将频繁项集全部插入到哈希表 *hash_ frequent*,非频繁项集插入到哈希表 *hash_unfrequent*,由此计算出频繁 2 项集 $L_2 = \{\{bd\},\{be\},\{de\},\{dg\},\{dh\},\{eg\}\}$,同时可以计算出频繁 2 项集的数量为 $|L_2| = 6 \geqslant k+1$,由推论 1 可以继续向 $k+1$ 项集进行扩展,重复步骤(3)中链表交叉正交的步骤,可以生成十字链表(如图 4 所示)。

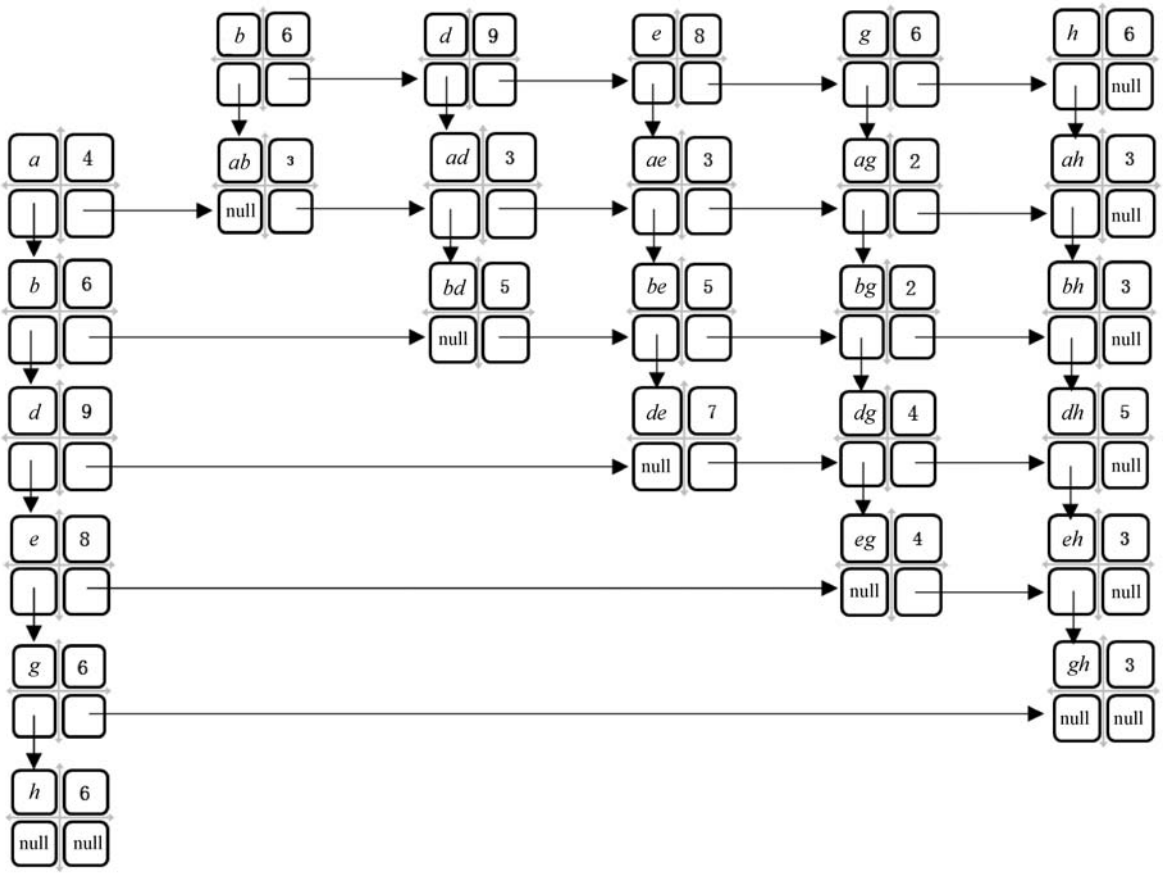

图 3 频繁 1 项集十字链表

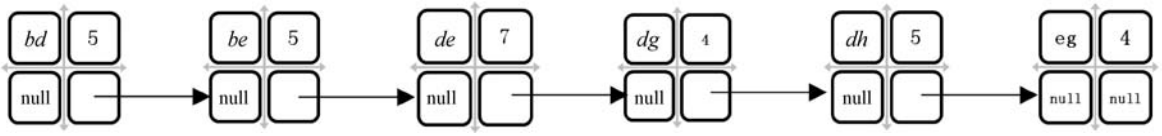

图 4 频繁 2 项集链表

(6) 遍历十字链表中的节点,将每一个节点插入到新的链表中,然后将新链表进行部分选择排序,将链表中非频繁项集进行删除,这样就生成新链表(如图 5 所示[13]),重复步骤(4),由此计算出 $L_3 = \{\{bde\},\{deg\},\{deh\}\}$(如图 6 所示),然后通过推论 1 判断是否能够继续进行扩展。

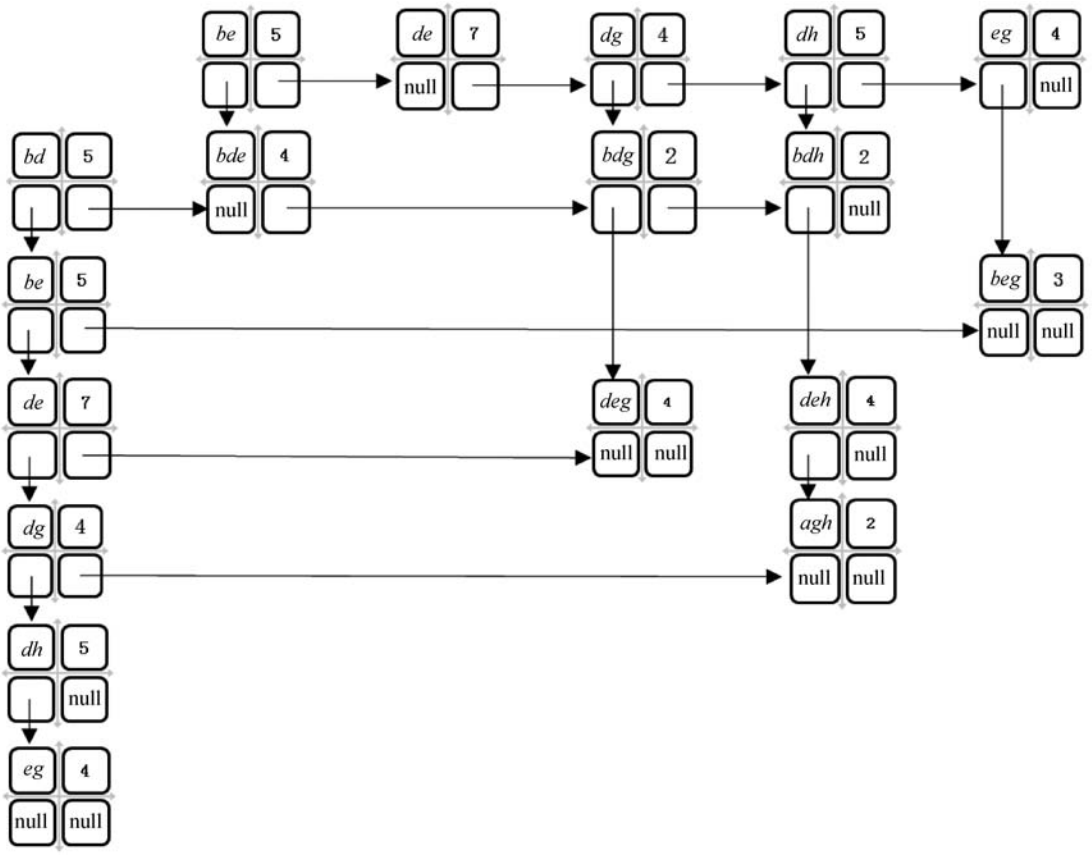

图 5　频繁 2 项集交叉正交十字链表

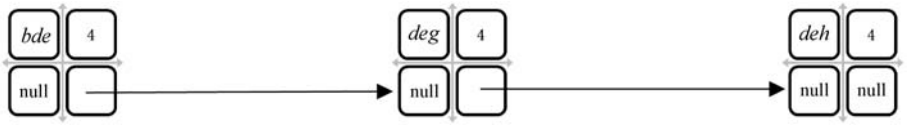

图 6　频繁 3 项集链表

(7) 重复循环步骤(4)和步骤(5)中计算$(k+1)$项的步骤,如果不满足推论 1 条件,表示无法计算出下一项频繁项集,算法结束,输出频繁项集。

3.4　算法的性能分析

评价算法性能的优劣主要从算法的时间复杂度和空间复杂度来进行衡量。

3.4.1　两种算法时间复杂度分析

假定有 M 条数据,每条数据中有事务数为 L。计算频繁项集分为 3 个过程。

步骤 1　每一次计算项集频数扫描数据库数据时间复杂度约为:$T_1 = O(M \times L)$。

步骤 2　扫描数据库中的数据后,进行项集自连接平均的时间复杂度约为:$T_2 = O(|L_{k-1}| \times |L_{k-1}|)$。

步骤 3　在计算频繁项集的过程中,经过动态剪枝判断符合最小支持度的平均的时间复杂度为:$T_3 = O(M \times |C_k|)$。

本文利用了布尔矩阵“与”运算来求项集频数,结合哈希表剪枝和十字链表去重来计算频繁项集,因此主要在布尔矩阵生成以及链表交叉正交及剪枝去重过程中消耗时间。假设布尔矩阵数据为 M 条,每一条数据的事务长度为 N,频繁项集的长度为$|L_k|$。

步骤 1　每计算一次项集需要扫描矩阵中的数据的时间复杂度为:$H_1 = O(M \times N)$。

步骤 2　链表交叉正交和剪枝与去除重后存储在十字链表中的项集的时间复杂度约为 $O(((|L_{k-1}| \times |L_{k-1}|)/2 - |L_{k-1}|) \times M)$,剪枝和去重的时间为$|T_k|$,该过程时间复杂度为:$H_2 = O((((|L_{k-1}| \times |L_{k-1}|)/2 - |L_{k-1}|) - |T_k|) \times M)$。

步骤 3　遍历十字链表生成新链表的时间复杂度为:$H_3 = O(|C_k|)$。

结论:通过前面的分析,本文利用布尔矩阵“与”运算来计算频繁可以优化算法的运行速度,说明时间复杂度 $H_1 < T_1$。结合十字链表交叉正交存储和哈希表剪枝与去重,优化了冗余项集计算过程,说明时间复杂度 $H_2 < T_2$ 和时间复杂度 $H_3 < T_3$。

3.4.2　算法的空间复杂度分析

假定数据量为 M,事务项的长度为 L。Apriori 算法的空间复杂度为:$S_1 = O(|L_{k-1}| \times |L_{k-1}| + |C_k|)$。

本文提出基于哈希表和十字链表来存储频繁项集,通过剪枝和去重的方法有效地减少了非频繁项集和冗余的项集占有的存储空间,所以该算法所占用在空间上面的消耗近似为:$S_1^{(1)} = O(|C_k|)$。

结论:通过分析,本文利用布尔矩阵的"与"运算,同时结合了十字链表和哈希表进行剪枝,有效地优化了空间复杂度,说明空间复杂度 $S_1 < S_1^{(1)}$。

3.5 实验结果分析

对文献[14]的优化方法进行算法复现(简称 VM_Apriori 算法),本文在相同环境下做了大量的实验来比较三种算法的时间效率和空间占用效率,实验的操作系统为 Windows 10,数据库为 MySQL,编程语言为 Python 3.7,开源软件 RGui 自带的 Groceries 数据集作为实验数据集,该数据集中有 9 835 条数据,169 项不同的事务。

(1) 分组设置不同的最小支持度,来比较两种算法在计算时间上的性能[15],三种算法使用相同的数据,均为 9 835 条,如图 7 所示。

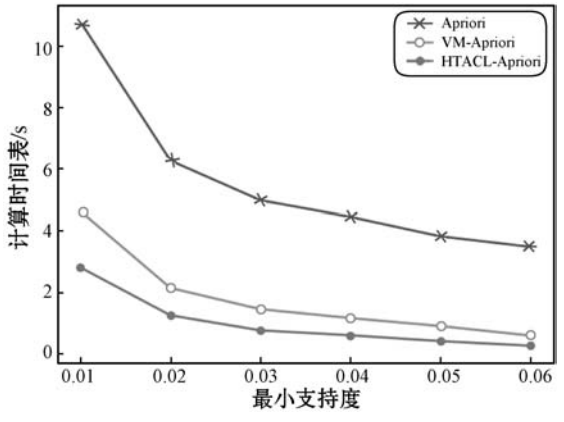

图 7 不同支持度的计算时间

在实验数据的测试的对比之下,经过改进后的 HTACL_Apriori 算法在同样的环境与数据下,耗时明显减少,达到了预期效果。

(2) 分组设置不同的数据量,来比较三种算法的时间效率[16],三种算法的最小支持度均为 0.01,如图 8 所示。

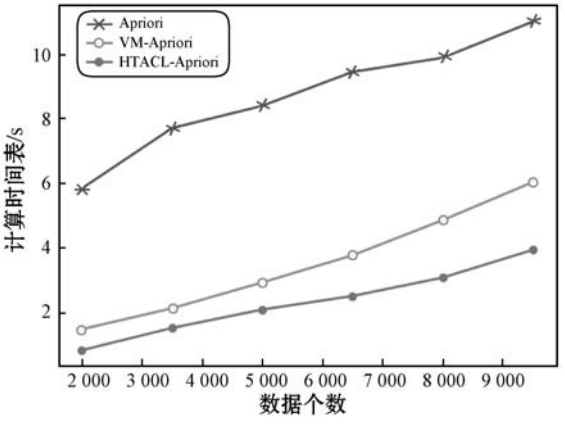

图 8 不同数据量的计算时间

当数据量比较小的时候,算法的耗时也比较少,随着数据量增加,三种算法所消耗的时间也不断地增加。在实验数据的对比之下,经过改进后的 HTACL_Apriori 算法在同样的环境与数据下,耗时明显减少,达到了实验的预期效果。

(3) 分组设置不同的数据量,来比较三种算法的空间效率[17],三种算法的最小支持度均为 0.01,实验对比如图 9 所示。

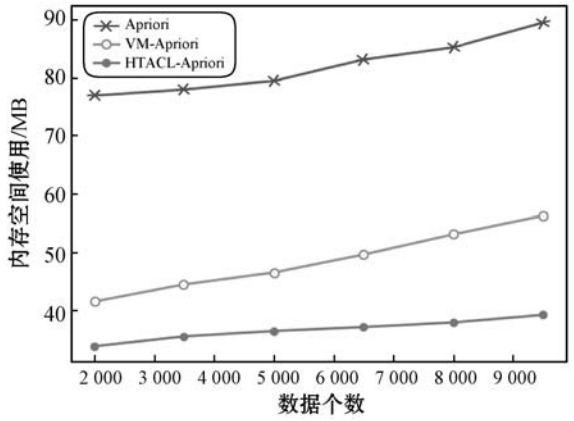

图 9 不同数据量内存空间占用

可以看出,当数据量比较小的时候,算法的空间占用也比较小,随着数据量增加,三种算法的空间内存开销也不断地增加。在实验数据的对比之下,经过改进后的算法在同样的环境与数据下,空间内存开销明显减少,达到了预期效果[18]。

通过上述理论分析和多组数据实验对比了三种算法的性能[19],在环境相同的情况下,算法优化后的时间复杂度和空间复杂度得到了明显的改善,达到了预期效果。

4 结 语

本文提出利用布尔矩阵表示事务,并且结合哈希表和十字链表存储对项集进行剪枝和去除重复值降低时间复杂度和空间复杂度。并且通过对实验数据对比分析,验证了 HTACL_Apriori 算法的时间复杂度和空间复杂度得到了明显的优化。但是当数据量达到较大的规模,所处理的数据会受限于内存计算容量。在后续的研究中将考虑基于结合 Spark 框架,实现大规模数据处理,期望在数据处理方面得到更好的效果。

参 考 文 献

[1] Mlambo M, Gasela N, Esiefarienrhe M, et al. On the optimization of improved apriori algorithm via linked-list trie [C]//2017 International Conference on Big Data Research, 2017:62 - 66.

[2] Wang R. Research on apriori algorithm optimization of cloud computing and big data in software engineering[C]//2018 5th International Conference on Electrical & Electronics Engineering and Computer Science (ICEEECS 2018), 2018: 53 - 56.

[3] Xiao M, Yin Y, Zhou Y, et al. Research on improvement of apriori algorithm based on marked transaction compression [C]//2017 IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC 2017), 2017:1067 - 1071.

[4] Zhou B. A new improved aprior algorithm in big data environment[C]//2018 5th International Conference on Electrical & Electronics Engineering and Computer Science(ICEEECS 2018),2018:89 - 96.

[5] 廖纪勇,吴晟,刘爱莲. 基于布尔矩阵约简的 Apriori 算法改进研究[J]. 计算机工程与科学,2019,41(12):2231 - 2238.

[6] 李洁,朱洪亮,陈玉玲,等. 基于哈希存储与事务加权的并行 Apriori 改进算法[J]. 计算机工程,2020,46(11):109 - 116.

[7] 文武,郭有庆. 结合遗传算法的 Apriori 算法改进[J]. 计算机工程与设计,2019,40(7):1922 - 1926.

[8] 赵学健,孙知信,袁源,等. 一种正交链表存储的改进 Apriori 算法[J]. 小型微型计算机系统,2016,37(10):2291 - 2295.

[9] 杨鹏,赵辉,鲍忠贵. 基于双十字链表存储的共享资源矩阵方法特性研究[J]. 计算机应用,2016,36(3):653 - 656.

[10] 黄剑,李明奇,郭文强,等. 基于 Hadoop 的 Apriori 改进算法研究[J]. 计算机科学,2017,44(7):262 - 266,269.

[11] 阴爱英. 基于线程并行计算的 Apriori 算法[J]. 西安科技大学学报,2014,34(1):71 - 74.

[12] 张敏,姚良威,侯宇,等. 基于向量和矩阵的频繁项集挖掘算法研究[J]. 计算机工程与设计,2013,34(3):939 - 943.

[13] 沈艳,张琦智,刘垠,等. 矩阵压缩 Apriori 算法分析[J]. 计算机应用,2017,37(S2):207 - 209,240.

[14] 曹莹,苗志刚. 基于向量矩阵优化频繁项的改进 Apriori 算法[J]. 吉林大学学报(理学版),2016,54(2):349 - 353.

[15] 王蒙,方睿,邹书蓉. 基于矩阵相乘的 Apriori 改进算法[J]. 计算机与数字工程,2018,46(10):1974 - 1979.

[16] 魏玲,魏永江,高长元,等. 基于 Bigtable 与 MapReduce 的 Apriori 算法改进[J]. 计算机科学,2015,42(10):208 - 210,243.

[17] 李伟,朱赵元. 一种基于并行矩阵目标明确的 Apriori 算法[J]. 浙江工业大学学报,2017,45(5):574 - 579.

[18] 孙学波,石飞达. 基于 Hadoop 的 Apriori 算法研究与优化[J]. 计算机工程与设计,2018,39(1):126 - 133,145.

[19] 刘军煜,贾修一. 一种利用关联规则挖掘的多标记分类算法[J]. 软件学报,2017,28(11):2865 - 2878.

(上接第 234 页)

到有显著灰度差异的图像,所以后续将会试图将本文算法应用于海冰漂移跟踪。

参 考 文 献

[1] Flora D, Julie D, Yann G, et al. SAR-SIFT: A SIFT-Like algorithm for SAR images[J]. IEEE Transactions on Geoscience and Remote Sensing,2015,53(1):453 - 466.

[2] Gui Y, Su A, Du J. Point-pattern matching method using SURF and shape context[J]. Optik, 2013, 124(14): 1869 - 1873.

[3] Ma W, Wen Z, Wu Y, et al. Remote sensing image registration with modified SIFT and enhanced feature matching [J]. IEEE Geoscience and Remote Sensing Letters,2017,14(1):3 - 7.

[4] Alcantarilla P F, Bartoli A, Davison A J. KAZE Features [C]//12th European Conference on Computer Vision (ECCV),2012:214 - 227.

[5] Demchev D, Volkov V, Kazakov E, et al. Sea ice drift tracking from sequential SAR images using Accelerated-KAZE features[J]. IEEE Transactions on Geoscience and Remote Sensing,2017, 55(9):5174 - 5184.

[6] Song M P, Cao Y J, Yu C Y, et al. Solar image matching based on improved freak algorithm[C]//2018 International Conference on Machine Learning and Cybernetics(ICMLC), 2018:126 - 131.

[7] Nakashima S, Morio T, Mu S. AKAZE-Based visual odometry from floor images supported by acceleration models[J]. IEEE Access, 2019, 7:31103 - 31109.

[8] 吴含前,李程超,谢珏. 一种改进的 A-KAZE 算法在图像配准中的应用[J]. 东南大学学报(自然科学版),2017,47(4):667 - 672.

[9] Mikolajczyk K, Schmid C. A performance evaluation of local descriptors[J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2005, 27(10):1615 - 1630.

[10] Yang X, Cheng K T. LDB: An ultra-fast feature for scalable Augmented Reality on mobile devices[C]//2012 IEEE International Symposium on Mixed and Augmented Reality(ISMAR),2012.

[11] Fischler M A, Bolles R C. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography[J]. Communications of the ACM, 1981,24(6):381 - 395.

[12] Lv H, Guo H, An J. Sea ice drift tracking with a MCC method of automatically acquiring features[C]//4th International Conference on Information Science and Control Engineering(ICISCE),2017:655 - 658.