Computer Science and Application 计算机科学与应用, 2021, 11(6), 1706-1716
Published Online June 2021 in Hans. http://www.hanspub.org/journal/csa
https://doi.org/10.12677/csa.2021.116176

Hans汉斯

基于改进的Apriori算法的关联规则分析

汪　敏，朱习军

青岛科技大学信息科学技术学院，山东　青岛
Email: 2334158648@qq.com

收稿日期：2021年5月18日；录用日期：2021年6月15日；发布日期：2021年6月22日

摘　要

关联规则反映事物与其他事物之间的关联性，是数据挖掘领域研究的一个重要方面，关键概念包括支持度，置信度，提升度。在关联规则中，Apriori算法是其重要组成部分。传统的Apriori算法存在如多次扫描数据库，需要很大的I/O负载，以及产生大量冗余性的候选项集等瓶颈问题。因此，对Apriori算法进行改进，通过布尔矩阵进行行列压缩来减少扫描数据的规模，通过引用索引表的形式来替代生成候选项集，并且以Tried树的形式来对最后所生成的所有频繁项集进行查找，从而加快了计算置信度的时间，以此来解决其瓶颈问题。最终实验结果表明，改进后的算法相比于传统的算法，大大提高了Apriori算法的时间及空间效率。

关键词

关联规则，Apriori改进算法，频繁项集，Tried树，索引表

Analysis of Association Rules Based on Improved Apriori Algorithm

Min Wang, Xijun Zhu

School of Information Science and Technology, Qingdao University of Science and Technology, Qingdao Shandong
Email: 2334158648@qq.com

Received: May 18th, 2021; accepted: Jun. 15th, 2021; published: Jun. 22nd, 2021

Abstract

Association rules reflect the association between things and other things, which is an important aspect of data mining research. The key concepts include support, confidence and promotion. Apriori algorithm is an important part of association rules. The traditional Apriori algorithm has

文章引用: 汪敏, 朱习军. 基于改进的 Apriori 算法的关联规则分析[J]. 计算机科学与应用, 2021, 11(6): 1706-1716.
DOI: 10.12677/csa.2021.116176

some bottleneck problems, such as scanning the database many times, requiring a lot of I/O load, and producing a large number of redundant candidate itemsets. Therefore, the Apriori algorithm is improved. The scale of scanning data is reduced by row and column compression of Boolean matrix. The candidate itemsets are generated by using index table instead. All frequent itemsets are searched in the form of tried tree, which speeds up the calculation time of confidence, so as to solve the bottleneck problem. The final experimental results show that the improved algorithm greatly improves the time and space efficiency of Apriori algorithm compared with the traditional algorithm.

Keywords

Association Rules, Improved Apriori Algorithm, Frequent Itemsets, Tried Tree, Index Table

1. 引言

当今社会，是一个大数据的社会，数据挖掘技术的应用越来越广泛。数据挖掘是指从大量不完整的且有噪声的随机数据中，提取隐含在其中人们事先不知道但可能有潜在作用的信息和知识。在第十一届国际联合人工智能学术会议上“从数据库中发现知识(KDD)”一词首次出现，这标志着数据挖掘的初始形成。20 世纪 60 年代，还停留在数据的搜集阶段，主要以存储在计算机，磁带或者磁盘里的静态的历史数据为主；到了 20 世纪 80 年代，可以在关系型数据库中通过结构化查询语言对历史性的数据进行动态访问；20 世纪 90 年代，可以在多维的数据库及数据仓库中通过联机分析处理等技术对数据进行动态访问；21 世纪，通过一些高级的算法，多处理器计算机，达到对于数据的海量存储，并把这些数据进行数据预处理，除噪等操作，从海量的数据中获取有价值的数据信息。到目前为止，数据挖掘技术规模逐渐变大，也逐渐成熟。而关联规则[1]反映了一个事物与其他事物之间的相互依存性与关联性关系，通过关联规则，可以发现大型数据集之间的频繁项集，由频繁项集产生强关联规则，进而找到数据间的相关性。在如此的核心思想下，Apriori 算法应运而生。

传统的 Apriori 算法通过迭代的方法扫描数据库，先将项集进行连接操作，找到不满足支持度的项集进行剪枝，然后将符合要求的频繁项集进行再次连接，依次迭代进而找到满足最小支持度阈值的频繁项集，再由频繁项集找到满足最小支持度阈值与最小置信度阈值的强关联规则。在对传统的 Apriori 算法使用过程中，越来越发现其存在的不足之处。对于每次循环，候选集都要扫描数据库，如果说一个频繁的大项目包含 10 个项的话，那么就至少需要扫描事务数据库 10 遍，并且在这个过程中产生了庞大的候选项集，这对算法运行的时间及主存的空间都是一种挑战[2]。基于以上问题，本文提出了一种通过布尔矩阵压缩，生成索引表，再将索引表转换成 Tried 树的方式来对 Apriori 算法进行改进。改进后的算法将事务数据库转换为布尔矩阵，并对该矩阵进行行压缩与列压缩，从而有效减少了所需扫描数据的规模，减少了扫描次数。并且通过数学运算的方式代替事务查找，即通过按位与运算，从布尔矩阵中找到项集的索引表，在这个过程中不用再生成候选项集，免去了产生庞大的候选项集的麻烦。最后将索引表转换为 Trie 树的形式来查找频繁项集进而计算置信度找到强关联规则。以上改进解决了传统 Apriori 算法中存在的性能问题，提高了该算法的运行效率。

2. Apriori 算法

2.1. Apriori 算法概述

1993 年，Rakesh Agrawal [3]等首次提出了顾客交易数据库中项集间关联规则挖掘问题，随后大批科研人员对于该问题进行了深入研究。1994 年，Rakesh Agrawal 和 Ramakrishnan Srikant 正式提出 Apriori 算法用于挖掘数据库中的频繁项集。Apriori 在拉丁语中的意思为“来自以前”，也就是先验或者假设的理论，Apriori 算法的名字正是基于这样的一个事实：算法使用频繁项集的先验性质，即频繁项集的所有非空子集也一定是频繁项集。Apriori 算法作为第一个关联规则挖掘算法，也是数据挖掘中最经典的关联规则挖掘算法，其主要是找到数据集间的关系来帮助人们做一些决策，现阶段已经被广泛应用到商业，农业，医学等各个领域。

2.2. Apriori 算法核心思想

Apriori 算法通过迭代[4]，检索事务数据库中所有的频繁项集，即支持度不低于用户设定阈值的项集，然后再利用频繁项集构造出满足用户最小置信度的关联规则。首先通过扫描事务数据库，生成候选项集 C_1，并计算它的支持度，通过筛选去掉支持度低于阈值的候选 1 项集，得到频繁 1 项集，将该集合记作 L_1，然后再连接频繁 1 项集得到候选 2 项集并计算支持度，通过筛选去掉不满足支持度的候选 2 项集，得到频繁 2 项集，将该集合记作 L_2，L_2 找 L_3，以此类推，不断循环，直到不能再找到任何的频繁 K 项集。最后，在所有的频繁项集中查找并计算置信度，同时满足最小支持度阈值与最小置信度阈值的规则即为强关联规则。

Apriori 算法的先验性质是其一大特点，所有频繁项集的子集必是频繁项集。从而也可以得到一个推论，非频繁项集的超集必是非频繁的[5]。依据这一性质一推论，挖掘出满足支持度和可信度阈值的所有级别的频繁项集。

3. 改进的 Apriori 算法

3.1. 改进算法的思想

改进该算法的主要思想是通过布尔矩阵进行行列压缩，减少事务数据库的扫描次数[6]-[12]；在扫描过程中以索引表的形式替代候选项集，免去了生成大量候选项集的麻烦；在查找频繁项集并计算置信度时以 Tried 树的形式加快查找速度。

1) 将事务数据库 D 转换为矩阵 Mat，其中事务按照列顺序排序，项集按照行顺序排序，该矩阵的表示如下：

$$\mathrm{Mat}=\begin{array}{c}\begin{matrix}\mathrm{T}_1 & \mathrm{T}_2 & \cdots & \mathrm{T}_n\end{matrix}\\ \begin{bmatrix} d_{11} & d_{12} & \cdots & d_{1n} \\ d_{21} & d_{22} & \cdots & d_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ d_{m1} & d_{m2} & \cdots & d_{mn} \end{bmatrix}\end{array}\begin{matrix} I_1 \\ I_2 \\ \vdots \\ I_m \end{matrix} \tag{1}$$

2) 如果第 i 个项集在第 j 个事务中，则矩阵第 i 行，第 j 列的值 d_{ij} 为 1，否则为 0，由此得到布尔矩阵。

3) 由上一步得到的布尔矩阵可以计算该矩阵中某一行形成的项集的支持度。支持度由各行向量通过按位与运算得到。

$$\mathrm{support\_count}\left(\backslash\mathrm{mathrm}\left\{\mathrm{C}_k\right\}\right)=\sum_{j=1}^{n}\left(d_{i_1 j}\wedge d_{i_2 j}\wedge d_{i_3 j}\wedge\cdots\wedge d_{i_k j}\right) \tag{2}$$

4) 由布尔矩阵以及支持度的计算方法，得到各项集的支持度，依此得到项集索引表，再与所设置的最小支持度比较得到频繁项集。

5) 依据频繁项集的一性质一推论，如果一个项集是非频繁的，那么所有包含该项集的项集也是非频繁的，可以将其直接进行删除，也就是进行行压缩。

6) 由于布尔矩阵每个事务对应一个列向量，所以如果一个事务的长度小于 K，那么不可能包含 K-频繁项集 L_k，在搜索时可直接将该事务删除，也就是进行列压缩。

7) 将压缩后的布尔矩阵再次进行扫描，计算支持度，创建索引表。重复上述步骤，直到不能再产生 K-频繁项集，最终以索引表的形式得到所有频繁项集。

8) 最后引用 Tried 树的形式对所有的频繁项集进行查找，计算置信度，从而产生强关联规则，即产生用户感兴趣的关联规则。

3.2. 改进算法实例说明

以一实例来对该算法进行简单说明，如表 1 为事务数据库 D：

Table 1. Database D
表 1. 数据库 D

TID	项集
T_1	{1, 2, 5}
T_2	{2, 4}
T_3	{1, 2, 3, 4}
T_4	{1, 2, 4}
T_5	{2, 3}
T_6	{1, 3}

将该数据库转换为布尔矩阵，该矩阵的表示如下：

$$\begin{array}{r} \\ 1 \\ 2 \\ \text{Mat} = 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} T_1 & T_2 & T_3 & T_4 & T_5 & T_6 \end{array} \\ \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \tag{3}$$

假设该数据库最小支持度阈值(min_support)为 2，通过按位与运算得到各行向量的支持度，形成索引表，如表 2。因为 min_support = 2，所以 Count 需要大于等于 2，即可把不满足要求的直接删除，得到如表 3。

Table 2. Indextable-L_1
表 2. 索引表-L_1

Item	Count
{1}	4
{2}	5
{3}	3
{4}	3
{5}	1

Table 3. Frequent itemsets-L_1
表 3. 频繁项集-L_1

Item	Count
{1}	4
{2}	5
{3}	3
{4}	3

论将频繁项集-L_1转换为矩阵形式，得到 M_1，该矩阵的表示如下：

$$M_1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccccc} T_1 & T_2 & T_3 & T_4 & T_5 & T_6 \end{array} \\ \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \end{bmatrix} \end{array} \tag{4}$$

由 M_1 得到索引表-L_2 索与频繁项集-L_2，如表 4，表 5。

Table 4. Indextable-L_2
表 4. 索引表-L_2

Item	Count
{1, 2}	3
{1, 3}	2
{1, 4}	2
{2, 3}	2
{2, 4}	3

Table 5. Frequent itemsets-L_2
表 5. 频繁项集-L_2

Item	Count
{1, 2}	3
{1, 3}	2
{1, 4}	2
{2, 3}	2

再由 M_1 删除少于 3 项的事务，即进行列压缩，得到 M_2，该矩阵的表示如下：

$$M_2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cc} T_3 & T_4 \end{array} \\ \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \end{array} \tag{5}$$

由 M_2 得到索引表-L_3 索与频繁项集-L_3，如表 6，表 7。

Table 6. Indextable-L_3
表 6. 索引表-L_3

Item	Count
{1, 2, 3}	1
{1, 2, 4}	2
{1, 3, 4}	1
{2, 3, 4}	1

Table 7. Frequent itemsets-L_3
表 7. 频繁项集-L_3

Item	Count
{1, 2, 4}	2

将所得所有频繁项集存入索引表中，然后将频繁项集转换为 Tried 树的形式进行查找，并计算得到所有满足最小置信度的规则，产生的规则既为强关联规则。

3.3. 改进算法的实现

1) 将数据集转换为矩阵

```
public void addData(String[] projects) {
    BitSet bitSet = new BitSet();
    for (String temp : projects) {
        ProjectAttr projectAttr = projectMap.get(temp);
        if (Objects.isNull(projectAttr)) {
            projectAttr = new ProjectAttr();
            projectAttr.index = projectMap.size();
            projectMap.put(temp, projectAttr);
            this.maxProjectSize = projectMap.size();
            projectIndexMap.put(projectAttr.index, temp);
        }
            projectAttr.count++;
            bitSet.set(projectAttr.index);
}}
```

1) 行压缩

```
public Map<String, Integer> supportCountOneProjectAndCompressProjects(int threshold) {
Map<String, Integer> map = new HashMap<>();
List<String> willRemoveProjects = new LinkedList<>();
for (Entry<String, ProjectAttr> entry : projectMap.entrySet()){
    if (entry.getValue().count >= threshold) {
        map.put(entry.getKey(), entry.getValue().count);
    } else {
```

```
            willRemoveProjects.add(entry.getKey());
        }
    }
```

2) 列压缩

```
public void compressAffair(int threshold) {
Iterator<BitSet> iterator = matrix.iterator();
while (iterator.hasNext()) {
    BitSet next = iterator.next();
    if (next.cardinality() < threshold) {
        iterator.remove();
        for (int i = 0; i < next.size(); i++) {
            if (next.get(i)) {
                String projectKey = projectIndexMap.get(i);
                ProjectAttr projectAttr = this.projectMap.get(projectKey);
                if (Objects.nonNull(projectAttr)) {
                    projectAttr.count--;
                }
            }
        }
    }
}}
```

3) 索引表查找与 Tried 树查找

```
public class IndexRecord implements Comparable<IndexRecord> {
    public static IndexRecord of(Collection<String> item, int count) {
        return new IndexRecord(item, count);}
public IndexRecord inspectTakeIndexRecord(final Collection<String> item) {
        if (Objects.isNull(rootMatcher)) {
            this.resetMatcher();}
        TriedNode parent = this.rootMatcher;
        for (String temp : item) {
            parent = parent.map.get(temp);}
        return parent.item;}}
private int getCount(Collection<String> collection) {
        long start = System.nanoTime();
        int count = 0;
        if (triedFlag) {
            IndexRecordinspectTakeIndexRecord = triedHallows.inspectTakeIndexRecord(collection);
            if (Objects.nonNull(inspectTakeIndexRecord)) {
                count = inspectTakeIndexRecord.getCount();}
```

```
} else {
    for (IndexRecord indexRecord : indexTable) {
        if (indexRecord.collectionEquals(collection)) {
            count = indexRecord.getCount();
            break;}}}
long end = System.nanoTime();
sum += end - start;
return count;}
```

4. 算法性能验证及结果分析

为了更好的证明改进后的算法相比于改进前的算法在性能及效率上的提升，因此进行了对比实验。在不同事务数以及不同最小支持度阈值的情况下，对比改进前的算法与改进后的算法在运行后产生频繁项集所需的时间；在查找频繁项集计算置信度时对以索引表的形式查找和以 Tried 树的形式进行查找进行对比实验。从而更加清晰直观有效的证明了改进后的算法在效率上有了相当大的提升。

4.1. 对产生频繁项集所需时间的对比

本实验所采用的数据为模拟随机生成的数据集，共 7 组数据，分别为 1000，2000，5000，10,000，20,000，50,000 以及 80,000 条事务，并将其存入.txt 文件中。

如图 1，为相同最小支持度阈值，不同事务数的情况下得到的实验结果转化而成的折线图。实验结果为：事务数据量为 1000 时，算法优化前所需时间为 258 毫秒，优化后所需时间为 59 毫秒；事务数

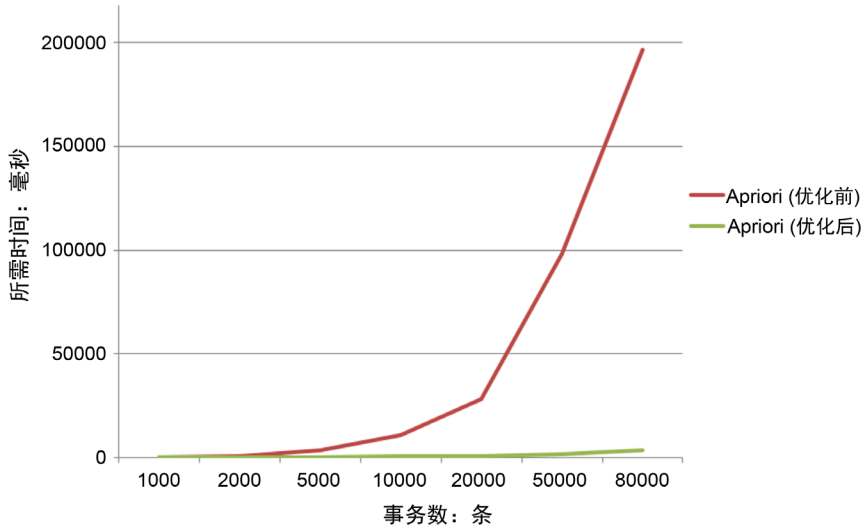

Figure 1. Under different number of transactions
图 1. 不同事务数情况下

据量为 2000 时，算法优化前所需时间为 682 毫秒，优化后所需时间为 82 毫秒；事务数据量为 5000 时，算法优化前所需时间为 3316 毫秒，优化后所需时间为 184 毫秒；事务数据量为 10,000 时，算法优化前所需时间为 10,750 毫秒，优化后所需时间为 466 毫秒；事务数据量为 20,000 时，算法优化前所需时间为 28,160 毫秒，优化后所需时间为 649 毫秒；事务数据量为 50,000 时，算法优化前所需时间为 98,016 毫秒，优化后所需时间为 1731 毫秒；事务数据量为 80,000 时，算法优化前所需时间为 196,606 毫秒，优化后所

需时间为 3516 毫秒；由图 1 可知：在相同最小支持度阈值的情况下，随着事务数的不断增大，改进前的算法与改进后的算法运行所花费时间均增长；但整体上看，各事务数相同的情况下，改进后的算法要比改进前的算法快的多，这充分证明了改进后的算法在 apriori 算法性能优化方面是高效的。

2) 如图 2，为相同事务数，不同最小支持度阈值的情况下得到的实验结果转化而成的折线图。由于在实验过程中产生大量的偶发情况，如某一支持度下运行时网络突然不好则运行结果会相当大，故图 2 展示的实验结果为进行大批量实验后，所有实验结果的平均值。选择事务数为 80,000 情况下的实验结果进行描述，80,000 情况下的实验结果为：最小支持度阈值为 0.01 时，算法优化前所需的时间为 386,369 毫秒，优化后所需时间为 153,255 毫秒；最小支持度阈值为 0.02 时，算法优化前所需的时间为 309,468 毫秒，优化后所需时间为 5740 毫秒；最小支持度阈值为 0.03 时，算法优化前所需的时间为 248,277 毫秒，优化后所需时间为 3901 毫秒；最小支持度阈值为 0.04 时，算法优化前所需的时间为 207,720 毫秒，优化后所需时间为 2783 毫秒；最小支持度阈值为 0.05 时，算法优化前所需的时间为 193,114 毫秒，优化后所需时间为 1781 毫秒；最小支持度阈值为 0.06 时，算法优化前所需的时间为 192,231 毫秒，优化后所需时间为 1585 毫秒；由图 2 知，在相同事务数的情况下，随着最小支持度阈值的不断增大，改进前的算法与改进后的算法运行所花费的时间均不断减小，并且改进后的算法相比于改进前的算法，在不同最小支持度下均是改进后的算法所需的时间少，效率比改进之前的有明显提高。由该图也可分析出，如果当最小支持度阈值达到一定程度，运行所需要的时间可能会达到 0，且增大到一定程度时，也有可能改进前与改进后的算法运行所需时间相同；如果最小支持度阈值越小，则改进前算法与改进后算法所需时间差距也就越大，因为设置的最小支持度越小，则满足条件的频繁项集就越多，在数据集多的情况下，更能体现出改进后的算法在运行时间性能等上的优越性。

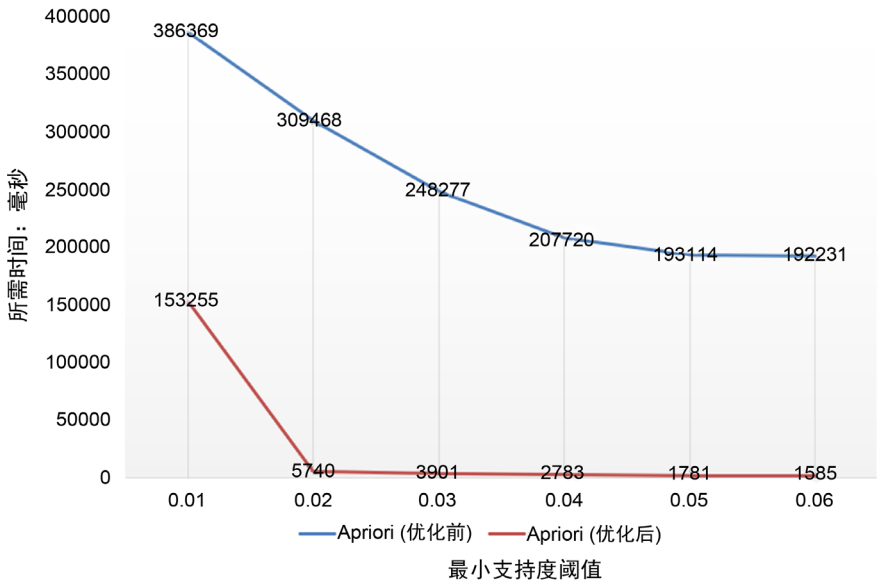

Figure 2. In the case of different minimum support thresholds
图 2. 不同最小支持度阈值情况下

4.2. 对查找频繁项集生成强关联规则所需时间的对比

将上一步中查找出来的频繁项集存在索引表中，随后进行再次查找并计算置信度，从而得到强关联规则。为了更快的得到强关联规则，所以决定把索引表的形式转化为 Tried 树的形式进行查找。通过实验，

对比以索引表的形式查找和以 Tried 树的形式进行查找所需的时间，有力的证明了以 Tried 树的形式查找要比以索引表的形式查找快的多。

如图 3，为相同最小支持度以及事务数的情况下，以索引表形式进行查找和以 Tried 树进行查找所需时间的对比。在进行多次实验后选取平均值。由该图可知：最小支持度为 0.01 时，以索引表形式查找所需时间为 184 毫秒，以 Tried 树形式查找所需时间为 73 毫秒；最小支持度为 0.02 时，以索引表形式查找所需时间为 142 毫秒，以 Tried 树形式查找所需时间为 56 毫秒；最小支持度为 0.03 时，以索引表形式查找所需时间为 137 毫秒，以 Tried 树形式查找所需时间为 55 毫秒；最小支持度为 0.04 时，以索引表形式查找所需时间为 135 毫秒，以 Tried 树形式查找所需时间为 53 毫秒；最小支持度为 0.05 时，以索引表形式查找所需时间为 135 毫秒，以 Tried 树形式查找所需时间为 50 毫秒；最小支持度为 0.06 时，以索引表形式查找所需时间为 130 毫秒，以 Tried 树形式查找所需时间为 49 毫秒。

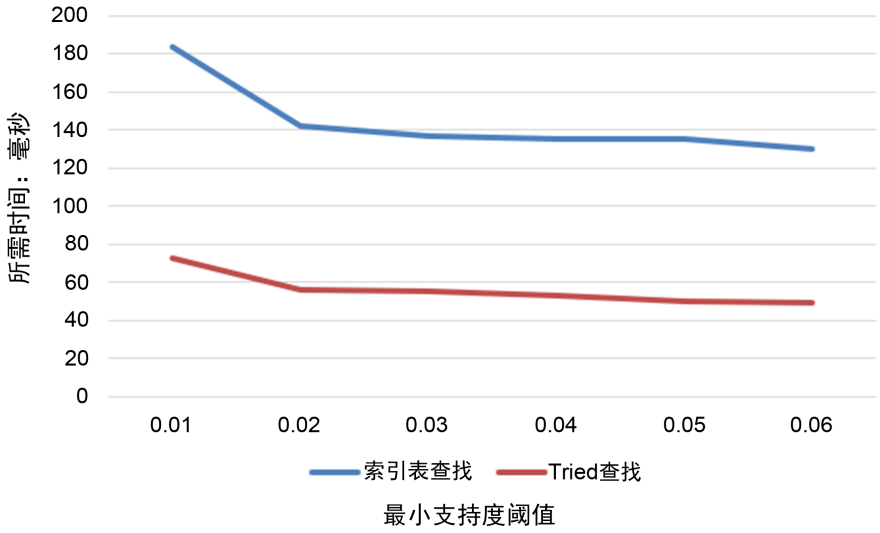

Figure 3. Different search results
图 3. 不同查找结果的情况下

5. 结语

本文通过分析传统的 Apriori 算法存在的缺点，因此对其进行了改进。改进后的算法通过运用布尔矩阵进行行列压缩，通过运用索引表替代事务查找，通过 Tried 树减少查找计算置信度时所需的时间。在多次实验中证明了改进后算法的优越性，同时从实验结果中也可知此改进的算法更适用于海量数据的处理。总体上解决了传统 Apriori 算法中存在的那些性能问题，有效减少了挖掘时间。

参考文献

[1] Zhong, R. and Wang, H. (2011) Research of Commonly Used Association Rules Mining Algorithm in Data Mining. 2011 *International Conference on Internet Computing and Information Services*, Hong Kong, 17-18 September 2011, 219-222. https://doi.org/10.1109/ICICIS.2011.63

[2] Li, G-Z., Wang, H.C. and Li, S.-G. (2013) An Improved Apriori Algorithm for Association Rules. *Telkomnika Indonesian Journal of Electrical Engineering*, **11**, 942-946. https://doi.org/10.11591/telkomnika.v11i11.3491

[3] Agrawal, R., Imieliński, T. and Swami, A.N. (1993) Mining Association Rules between Sets of Items in Large Databases. *ACM SIGMOD Record*, **22**, 207-216. https://doi.org/10.1145/170035.170072

[4] Oruganti, S., Ding, Q. and Tabrizi, N. (2013) Exploring HADOOP as a Platform for Distributed Association Rule Mining.

[5] 刘芳, 吴广潮. 一种基于压缩矩阵的改进 Apriori 算法[J]. 山东大学学报(工学版), 2018, 48(6): 82-88.

[6] 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.

[7] 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.

[8] 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.

[9] 周凯, 顾洪博, 李爱国. 基于关联规则挖掘 Apriori 算法的改进算法[J]. 陕西理工大学学报(自然科学版), 2018, 34(5): 40-44.

[10] 李龙, 刘澎, 张可佳, 等. 改进的 Apriori 算法的研究与应用[J]. 计算机与数字工程, 2019, 47(6): 1293-1297.

[11] 曲睿, 张天娇. 基于矩阵压缩的 Apriori 改进算法[J]. 计算机工程与设计, 2017, 38(8): 2127-2131.

[12] 苗苗苗, 王玉英. 基于矩阵压缩的 Apriori 算法改进的研究[J]. 计算机工程与应用, 2013, 49(1): 159-162.