IABS:一个基于 Spark 的 Apriori 改进算法*

闫梦洁†, 罗 军, 刘建英, 侯传旺
(国防科学技术大学 计算机学院, 长沙 410073)

摘 要: Apriori 算法是关联规则挖掘中最经典的算法之一,其核心问题是频繁项集的获取。针对经典 Apriori 算法存在的需多次遍历事务数据库及需产生候选项集等问题,首先通过转换存储结构、消除候选集产生过程等方法对 Apriori 算法进行优化;同时,随着大数据时代的到来,数据量与日俱增,传统算法面临巨大挑战,将优化的 Apriori 与 Spark 相结合,充分利用 Spark 的内存计算、弹性分布式数据集等优势,提出了 IABS(Improved Apriori algorithm based on Spark)。通过与已有的同类算法进行比较,IABS 的数据可扩展性和节点可扩展性得以验证,并且在多种数据集上平均获得了 23.88% 的性能提升,尤其随着数据量的增长,性能提升更加明显。

关键词: Apriori 算法; 频繁项集; 存储结构转换; Spark; 内存计算
中图分类号: TP301.6 **文献标志码**: A **文章编号**: 1001-3695(2017)08-2274-04
doi:10.3969/j.issn.1001-3695.2017.08.007

IABS: a parallel improved Apriori algorithm based on Spark

Yan Mengjie†, Luo Jun, Liu Jianying, Hou Chuanwang
(*School of Computer, National University of Defense Technology, Changsha* 410073, *China*)

Abstract: Apriori algorithm is one of the most classical algorithm in association rule mining, the core problem is the generation process of frequent itemsets. Firstly, aimed at the existing problems of classical Apriori algorithm, such as it needsed to scan the transaction global database for several times and needed to generate candidate itemsets, this paper optimized it by transforming storage structure and eliminating the process of candidate itemsets generation. Then, with the advent of the era of big data, data volume rises with the day, classical Apriori algorithm faces severe challenge. Based on the improved Apriori algorithm and combined with Spark platform, this paper proposed the IABS algorithm, which made full use of Spark, such as in-memory computation, resilient distributed datasets. Compared with already existing similar algorithms, the sizeup and node salability of IABS are validated, as well as, IABS achieves 23.88% performance improvement in average for various benchmarks. Especially, as the growth of data, its performance improvement is more obvious.
Key words: Apriori algorithm; frequent itemset; storage structure transformation; Spark; in-memory computation

0 引言

关联规则挖掘是数据挖掘的一个重要研究领域,关联规则挖掘算法也因此备受关注,已广泛地应用于商业、金融等多个领域。当应用于商业时,能够分析顾客经常一起购买的商品,通过把这些商品放在货架相邻位置来提升销量。例如,沃尔玛曾利用关联规则算法对其顾客的购物行为进行了购物篮分析,找出顾客经常同时购买的商品,使商品销量同时增加;应用于犯罪检测时,基于包含大量犯罪活动的大规模数据库,能够预测城市中最容易出现犯罪行为的区域或者可能重复犯罪的人[1]。

Apriori 算法是关联规则挖掘的经典算法,其首要目标是寻找数据库中同时出现频率较高的项集,即频繁项集,进而发现各项之间的关联规则。但是 Apriori 算法也存在性能瓶颈,包括需要多次遍历数据库、需通过自连接和剪枝产生候选项集及产生频繁项集的过程中无法避免无用的比较等。与此同时,在当前大数据背景下,面对海量数据,经典的 Apriori 算法已经无法应对,因此对算法进行并行化改造成为了研究的热点。对于分布式应用而言,MapReduce 是一个成熟的编程模式,但 MapReduce 对 Apriori 之类高迭代算法的实现效率并不高。而 Spark 从一开始便是为应对迭代式应用的高性能需求而设计的,由于计算是在内存中进行的,所以它能够明显地提高实时数据处理能力,同时,它也能确保集群的高容错性和可扩展性。

1 相关研究

1993 年 Agrawal 等人[2]首次提出了关联规则挖掘概念,并于 1994 年提出经典的 Apriori 算法[3],自此以后,Apriori 算法就被学术界广泛和深入地研究并获得了长足的发展。Tsay 等人[4]提出了基于集群的关联算法 CBAR,CBAR 通过一次遍历数据库来构造集群表,并把长度为 k 的事务记录在第 k 个集群表中。这样只需与部分集群表作比较来得到频繁项集,明显地减少了执行数据浏览和作比较的时间,但仍无法避免产生候选项集这个耗时的过程。Han 等人[5]于 2004 年提出了频繁模式增长算法(FP-Growth),FP-Growth 是从 Apriori 算法产生候选

收稿日期: 2016-05-19; **修回日期**: 2016-07-04 **基金项目**: 国家"863"计划资助项目(2014AA01A302)
作者简介:闫梦洁(1993-),女(通信作者),山西夏县人,硕士研究生,主要研究方向为大数据、数据挖掘(15570864501@163.com);罗军(1964-),男,研究员,硕士,主要研究方向为大数据、云计算;刘建英(1992-),女,硕士研究生,主要研究方向为大数据、文本挖掘;侯传旺(1991-),男,硕士研究生,主要研究方向为计算机系统安全.

项集入手来进行改进,采用分治策略,先根据第一次遍历数据库得到的降序排列的频繁 1 项表来构造 FP-tree 树,再通过一定策略遍历该树生成高维频繁项集。

在数据量比较小并且维度不是很高的情况下,上述的算法都能很好地工作。但当数据量越来越大、维度越来越高时,这些算法性能下降明显。自 Hadoop 推出以来,基于 Hadoop 的 Apriori 算法并行化实现层出不穷,Lin 等人[6]提出了三个基于 Hadoop 的实现:SPC、FPC 和 DPC。其中,SPC 直接将 Apriori 算法搬到 MapReduce 框架中,随着迭代轮数增大,SPC 最后几个迭代的候选项集数量变得越来越小,以至于工作节点的利用率不高;FPC 将 SPC 中 m 个迭代候选项集处理集中到一个迭代中,提高了工作节点利用率,但由于 m 是一个固定值,就可能因一次迭代要处理的候选项集数量太大而使工作节点负载过重,导致算法性能下降;DPC 是 SPC 和 FPC 的结合体,可以动态调整 m 的大小,但仍因 I/O 频率较高而无法达到较好的性能。

由于 Apriori 算法本身迭代次数多、结构分层明显以及 MapReduce 自身的工作原理,使得通过 MapReduce 框架并行化 Apriori 算法时,每次迭代都需要一个新的 MapReduce 工作来完成,并且每次迭代都要从 HDFS 重新读取上一阶段的数据,产生额外的负载,所以 MapReduce 并不适合 Apriori。而 Spark 从一开始便是为应对迭代式应用的高性能需求而设计的,因此 Qiu 等人[7]于 2014 年提出了 YAFIM,它将经典的 Apriori 算法基于 Spark 实现。实验结果表明 YAFIM 性能明显优于基于 Hadoop 的 MRApriori。YAFIM 解决了基于 Hadoop 并行化存在的编程模式问题,但它是在经典 Apriori 的基础上进行并行化的,因此也存在经典 Apriori 算法本身的一些问题。Rathee 等人[8]在 2015 年提出了一个在 Spark 平台上并行化的 Apriori 优化算法 R-Apriori。但 R-Apriori 仅将 YAFIM 在第二迭代的实现进行了优化并提高了 YAFIM 的效率。本文在已有研究基础上,提出了更加优化的 Apriori 算法并基于 Spark 实现并行化。

2　相关概念

2.1　频繁项集

假定 $I=(I_1,I_2,I_3,\cdots,I_n)$ 为一个事务数据库 D 中所有项的集合,那么一个事务可用 $T=\{\mathrm{TID},A\mid A\subseteq I\}$ 来表示,其中,TID 为事务 T 的编号,A 是 I 的子集。对于任意一个项集 $B(\subseteq I)$,B 的支持度 $\sup(B)$ 是指在事务数据库中,包含 B 的事务的个数,形式化表示为

$$\sup(B)=|\{\mathrm{TID}\mid(B\subseteq A),(\mathrm{TID},A)\in D\}|$$

如果 B 的支持度不小于最小支持度(最小支持度 min_sup 是用户自定义的数值),那么称 B 是频繁项集。如果 B 中包含 k 个项,则称 B 为频繁 k 项集。

2.2　Apriori 算法

Apriori 算法的目标是寻找事务数据库中的频繁项集,其主要分为两个阶段。

a)遍历事务数据库 D 记录各个项的出现情况,并统计出现次数不小于最小支持度的项,即为频繁 1 项集。

b)迭代地执行以下过程:由频繁 $k-1$ 项集 L_{k-1} 自连接、剪枝生成候选 k 项集。剪枝主要利用“频繁项集的所有非空子集也必频繁”的特征过滤自连接的结果产生候选 k 项集 C_k,得到 C_k 后,再次遍历数据库记录各个候选项集的出现情况,最后统计出现次数不小于最小支持度的候选项集,得到频繁 k 项集。

2.3　Spark 框架

Spark 是由 Berkeley AMP 实验室发展起来的基于内存的并行计算框架。相较于 MapReduce 框架的 Hadoop,Spark 的两大优势在于内存计算和 RDD 的持久化和容错。

Spark 会将数据加载到节点内存中,然后在内存中完成计算,即内存计算,这是 Spark 计算速率快的一个重要原因。Spark 的内存计算是基于一种新的分布式内存抽象弹性分布式数据集(RDD)。对于 RDD,Spark 有许多内置操作,可以将一个 RDD 转换为另一个 RDD。内存计算就是由这一系列的 RDD 操作构成的。特别是 RDD 的持久化操作,可以将 RDD 缓存在工作节点的内存中,这样,后续操作重用数据时,就可以直接从内存中读取了,这也是 Spark 速率快的另一个因素。另外,Spark 的容错方式也与 Hadoop 有很大不同,Hadoop 是通过数据多份复制进行容错的;但 Spark 不需要备份数据,它会记录 RDD 上执行的一系列操作,构建一个有向非循环图(DAG),如果数据有出错或丢失的情况,就根据 DAG 追溯重算。

3　IABS 算法

本文提出的 IABS 是在改进的 Apriori 基础上结合 Spark 编程模型实现的,其实现细节较 YAFIM 有明显区别。第一阶段,从 HDFS 读入待处理数据集的同时进行数据结构的改变,并过滤得到频繁 1 项集,将最终得到的数据集以 RDD 的形式存储于集群各节点内存中;第二阶段,根据频繁 $k-1$ 项集直接生成频繁 k 项集。重复这个过程直到再无频繁项集生成。

这里通过与经典 Apriori 基于 Spark 的实现相比较来具体分析 IABS 的实现。

3.1　IABS 的实现

3.1.1　第一阶段

在经典的 Apriori 基于 Spark 实现的 YAFIM 中,第一阶段直接读入 HDFS 中的待处理数据集并以 RDD 形式存储于集群各节点内存中,接着每一个 map 任务读入并处理若干行,以 1 为 value 发射这些行中包含的所有项,reducer 则将每项对应的所有 value 值求和并过滤得到出现次数不小于最小支持度的项。由于数据集直接从 HDFS 中读入,其在内存中的组织方式保持不变:每行代表一个事务 T,T 由 TID 和该事务 T 包含的所有项组成。因此,需要计算某项集的出现次数,只能整体遍历数据集来统计结果,每个迭代都重复这个过程,是相当大的时间消耗。

为了解决上述问题,IABS 采用了数据结构转换方法,从 HDFS 中读取数据集的同时实现数据结构转换,每个 map 任务读入并处理若干行,处理方式不同于 YAFIM,而是对事务中包含的所有项发射该项及对应事务编号的键值对,reducer 将每项对应的事务编号合并起来;之后对事务编号计数,过滤得到只包含频繁 1 项集的转换后的数据集 F。数据集的结构转换过程如图 1 所示。

F 中 a_{xy} 代表 TID_y 中是否包含 I_x,若包含,a_{xy} 为 1,否则为 0。此时,如果需要计算某 k 项集在整个数据集中的出现次数,只需将 F 中这 k 个项对应的 value 求与,操作结果就是所有包含这个 k 项集的事务编号,计数即可得到其出现次数。

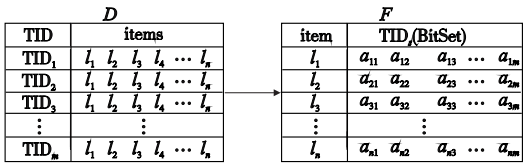

图1 结构转换图

图 2 是 IABS 在第一阶段的 RDD 血统关系图。首先通过 flatMap( )将数据集的每一行转换为多个(item,TID)键值对,接着 reduceByKey( )将相同 key 的 TID 连接成字符串并同时使用 filter( )过滤掉字符串中包含的事务编号个数小于最小支持度的键值对,再采用 map( )将每个 TID 连接成的字符串构造成 BitSet,得到以(item,BitSet)键值对为存储形式的频繁 1 项集存储结构 F。

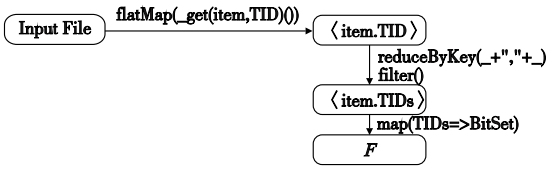

图2 IABS第一阶段RDD血统关系图

IABS 在第一阶段转换数据结构、获取频繁 1 项集采用的算法如下:

```
Algorithm IABS, stage I
foreach transaction t in D_i
  flatMap(line offset,t)
    foreach item I in t
      out(I,line offset)
    end foreach
  end flatMap
end foreach
reduceByKey(I,TIDs)
  ATIDs = ""
  while(item in partition)
    ATIDs = ATIDs + "," + TIDs
  end while
  if(ATIDs.split(",").length≥min_sup)
    out(I,ATIDs)
  end if
end reduceByKey
map(I,ATIDs)
  roreach TID in ATIDs.split(",")
    BitSet + =(TID)
  end foreach
  out (I,BitSet)
end map
```

3.1.2 第二阶段

在经典的 Apriori 基于 Spark 的实现中,第二阶段是迭代地由频繁 $k-1$ 项集获得频繁 k 项集的过程。首先,由频繁 $k-1$ 项集自连接和剪枝得到候选 k 项集,为使查找候选项集速度更快,YAFIM 将候选 k 项集存储于哈希树中。接着开始 map 任务,每个 map 任务处理若干个事务,获取事务中所有的 k 项对并搜索哈希树判断是否为候选项集,若为 k 候选项集,则将其作为 key 发射(key,1)键值对,reducer 则计数并过滤得到频繁 k 项集的部分。这是最经典的 Apriori 算法的执行过程,但也往往因为这个过程中最耗时的生成候选项集过程使得算法的效率受到制约。

在第一阶段中,本文提出的数据结构已经可以使求项集的出现次数从遍历整个数据集简化到将对应项的 BitSet 求与。在第二阶段中作了进一步优化,舍去候选项集的产生过程,进一步提升算法效率。

在 IABS 的第二阶段中,直接将存储于 F_{k-1} 中的频繁 $k-1$ 项集同 YAFIM 一样两两自连接,若可连接,则将它们对应的 BitSet 求与,BitSet 的与操作常数时间就可完成。结果 BitSet 中记录了所有包含连接后的 k 项集的事务编号,接着判断 BitSet 中事务编号数是否大于最小支持度,若大于,则连接后的项集为频繁 k 项集,将其与结果 BitSet 作为键值对存入 F_k 中。IABS 在第二阶段由频繁 $k-1$ 项集获取频繁 k 项集采用的算法如下:

```
Algorithm IABS, stage Ⅱ
getFrequentItem(F_{k-1}:frequent k-1 itemsets,min_sup:minimal support)
output:frequent k itemsets
f_i,f_j:frequent k-1 itemset
f_k: frequent k itemset
begin
F_k←∅
foreach f_i in F_{k-1}
  foreach f_j in F_{k-1}
  begin
    if((f_i[1] = =f_j[1]) and (f_i[2] = =f_j[2]) and…and
      (f_i[k-2] = =f_j[k-2]) and (f_i[k-1] <f_j[k-1])) then
      begin
        BitSet←f_i.BitSet∩f_j.BitSet
        if(|BitSet| > =min_sup) then
        begin
          f_k.item←{f_i[1],f_i[2],…,f_i[k-2],f_i[k-1],f_j[k-1]}
          f_k.BitSet←BitSet
          F_k←F_k∪f_k
        end
      end
  end
end
```

3.2 时间复杂度

本文需要对一些数值进行假设,然后用数学表达式的形式表示本文的分析结果。假设 T 代表待处理数据集中事务个数,M 代表工作中 map 任务的个数,f 代表频繁 1 项集的个数。YAFIM 的时间复杂度为 $O(f^2+\frac{T}{M}\times a^2)$。IABS 的时间复杂度为 $O(\frac{T}{M}\times f)$。

3.3 RDD 和内存计算

如上所述,IABS 算法主要有两个阶段,第一阶段完成频繁 1 项集的获取,第二阶段迭代地由频繁 $k-1$ 项集生成频繁 k 项集。IABS 充分利用 Spark RDD 抽象的优势,将所有数据以 RDD 形式存储于集群各个节点中以实现内存计算,获得更快的速度。第一阶段中,首先通过 textFile( )加载事务数据库以 RDD 形式存储,之后就可基于这个 RDD 进行一系列的内存计算。后续的每个迭代,也都会将得到的频繁 k 项集以 RDD 存储,再进行计算。同时,算法的每个迭代都会重用上一个迭代得到的频繁项集,因此可以对所有的频繁项集应用 RDD 的 cache( )方法使其持久保存于内存中而尽量避免保存于磁盘中,以便快速重用。

4 性能评估

在这部分中,通过与已有的同类算法 YAFIM[7]、R-Aprio-

ri[8] 进行比较完成 IABS 的性能评估。

实验采用的数据集是 Apriori 算法标准测试数据集,所有测试都进行了多次并取平均值作为实验结果。算法运行在 Spark 1.3.1 上,所有数据都存储在同样的 Hadoop 分布式文件系统上。实验环境如下:由八个节点搭建的集群,每个节点由四个单核 CPU 组成,运行于 2.4 GHz 的主频上,内存为 7.8 GB,磁盘大小接近 18 GB。节点运行于 Ubuntu 14.04 版本的操作系统上,Java 版本为 1.8.0。

并行算法的评价指标主要有可扩展性、加速比等,其中,可扩展性又分为数据可扩展性和节点可扩展性,本实验采用数据可扩展性和节点可扩展性来评价算法性能。数据可扩展性是指保持工作节点数不变,改变待处理数据量情形下获得的测试结果;节点可扩展性是指保持待处理数据量不变,改变节点个数情形下获得的测试结果。IABS 的正确性也通过与 YAFIM 的结果进行一致性比较得以证明。

4.1　数据集

本实验的数据集是从频繁项集挖掘数据集知识库[9] 中获得的,本文用到的是三个具有不同特征的大规模数据集,即 Chess、Mushroom 和 T10I4D100K,如表 1 所示。

表 1　数据集特征

数据集	项个数	事物个数
Chess	75	3 196
Mushroom	119	8 124
T10I4D100K	870	100 000

4.2　IABS 与 YAFIM 性能分析

为了测量数据可扩展性,保持节点数目为 8 不改变,将两种数据集的数据量由原数据集增长为其 2 倍、3 倍、4 倍、5 倍,在这五个数据量下分别测量 YAFIM 和 IABS 算法的执行时间,结果如图 3(a)(b)所示。其中 x 轴代表数据集的增长倍数,y 轴为执行时间。从图中可以看出,随着数据量成倍增加,YAFIM 的图形蜿蜒缓慢上升,而 IABS 的上升趋势则更加缓慢,几乎接近水平线,表明 IABS 较 YAFIM 有更高的数据可扩展性。同时可以看出,随着数据量成倍增长,相较于 YAFIM,IABS 的效率提升更加明显。对于 Chess 数据集而言,在原数据量时,YAFIM 的执行时间为 29 s,IABS 为 25 s,IABS 达到了 13.8% 的效率提升;当数据量增长至原数据量的 5 倍时,YAFIM 为 39.4 s,IABS 为 26.6 s,达到了 32.5% 的效率提升。对于 Mushroom 数据集同样有 17.9% ~36% 的效率提升。

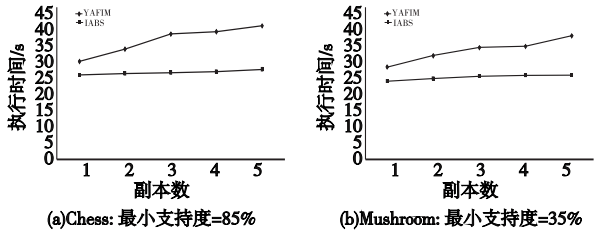

图3　不同数据集的数据可扩展性性能评估

为了测量节点可扩展性,固定数据集大小不变,改变集群节点数目,分别测量节点数为 1、2、4、8 时的 IABS 算法执行时间。结果如图 4(a)(b)所示,横轴代表节点数,纵轴代表执行时间,从图中可以看出,随着节点数目增长,算法执行时间不断缩短,并且接近直线下降。结果表明了 IABS 可达到良好的节点可扩展性。并且可以看出,不同的数据集会有不同的执行时间,并且所成图形的下降趋势也会有所不同,这可能是与数据集本身的特征相关的。

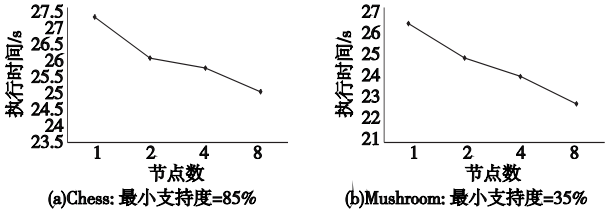

图4　不同数据集的节点可扩展性性能评估

4.3　IABS 与 R-Apriori 性能分析

R-Apriori 算法通过采用布隆过滤器存储候选项集,将 YAFIM 在第二个迭代的实现进行了优化,其余迭代保持不变。实验结果表明,在同等实验环境下,对于 T10I4D100K 数据集,R-Apriori 将第二迭代的时间由 YAFIM 的 76 s 左右降低至 30 s,对于第二阶段而言获得了 60.5% 的提升,但对于算法的总执行时间而言,R-Apriori 的性能提升为 11.5% 左右[8]。本文提出的 IABS,在同等实验环境下对于 T10I4D100K 数据集,比 YAFIM 性能提升了 19.2%,并且随着数据量增加,性能提升更加明显。因此,IABS 的优化程度更高于 R-Apriori。

综上所述,相对于已有的同类算法,本文提出的 IABS 是一个更高效的改进的 Apriori 基于 Spark 的实现。

5　结束语

本文简单总结了经典 Apriori 算法存在的性能瓶颈,并针对这几个方面尤其是候选项集生成过程进行改进,得到了更优化的算法;接着依托 Spark 平台对迭代算法的高效支持,产生了将改进的 Apriori 算法在 Spark 上并行化的思路并进行实现。随后将已存在的经典 Apriori 的 Spark 实现 YAFIM 与改进的 Apriori 算法的 Spark 实现 IABS 进行了详细的分析与比较,描述了如何改进算法,并最终从理论与实验上充分证明了 IABS 的高效性。尤其当数据量不断增大时,IABS 性能提升会更加明显。

参考文献:

[1] Buczak A L, Gifford C M. Fuzzy association rule mining for community crime pattern discovery[C]//Proc of ACM SIGKDD Workshop on Intelligence and Security Informatics. New York:ACM Press,2010.

[2] Agrawal R, Imieliński T, Swami A. Mining association rule between sets of items in large databases[J]. ACM Sigmod Record, 1993, 22(2): 207-216.

[3] Agrawal R, Srikant R. Fast algorithm for mining association rules [C]//Proc of the 20th International Conference on Very Large Data Bases. 1994: 487-499.

[4] Tsay Y J, Chiang J Y. CBAR: an efficient method for mining association rules[J]. Knowledge-Based Systems,2005,18(2):99-105.

[5] Han Jiawei, Pei Jian, Yin Yiwen, *et al*. Mining frequent patterns without candidate generation: a frequent-pattern tree approach[J]. Data Mining and Knowledge Discovery,2004,8(1):53-87.

[6] Lin M Y, Lee P Y, Hsueh S C. Apriori-based frequent itemset mining algorithms on MapReduce[C]//Proc of the 6th International Conference on Ubiquitous Information Management and Communication. New York:ACM Press,2012:76.

[7] Qiu Hongjian, Gu Rong, Yuan Chunfeng, *et al*. YAFIM: a parallel frequent itemset mining algorithm with Spark[C]//Proc of Parallel and Distributed Processing Symposium. 2014:1664-1671.

[8] Rathee S, Kaul M, Kashyap A. R-Apriori: an efficient apriori based algorithm on Spark[C]//Proc of the 8th Workshop on Information and Knowledge Management. 2015:27-34.

[9] Frequent itemset mining dataset repository[EB/OL]. (2012-06-04). http://fimi.ua.ac.be/data/.