Sp-IEclat: 一种大数据并行关联规则挖掘算法

李成严， 辛 雪， 赵 帅， 冯世祥
(哈尔滨理工大学 计算机科学与技术学院，哈尔滨 150080)

摘 要: 针对大数据环境下关联规则数据挖掘效率不高的问题，采用 Eclat 算法使用垂直数据库将事务的合并转换成集合操作的方法。研究了一种大数据并行关联规则挖掘算法 - Sp-IEclat (Improved Eclat algorithm on Spark Framework)，该算法基于内存计算的 Spark 框架，减少磁盘输入输出降低 I/O 负载，使用位图运算降低交集的时间代价并减少 CPU 占用，采用前缀划分的剪枝技术减少求交集运算的数据量，降低运算时间。使用 mushroom 数据集和 webdocs 数据集在两种大数据平台下实验，结果表明，Sp-IEclat 算法的时间效率优于 MapReduce 框架下的 Eclat 算法及 Spark 框架下的 FP-Growth 算法和 Eclat 算法。从对集群的性能监控得到的数值表明，同 Spark 框架下的 FP-Growth 算法和 Eclat 算法相比，Sp-IEclat 算法的 CPU 占用和 I/O 集群负载都较小。

关键词: 大数据; 关联规则挖掘; 频繁项集; Spark 弹性分布式数据集; MapReduce 框架
DOI: 10.15938/j.jhust.2021.04.015
中图分类号: TP399 **文献标志码:** A **文章编号:** 1007-2683(2021)04-0109-10

Sp-IEclat: A Big Data Parallel Association Rule Mining Algorithm

LI Cheng-yan， *XIN Xue*， *ZHAO Shuai*， *FENG Shi-xiang*
(School of Computer Science and Technology, Harbin University of Science and Technology, Harbin 150080, China)

Abstract: Aiming at the problem of inefficient data mining of association rules in a big data environment, the Eclat algorithm is used to use a vertical database to convert the merging of transactions into collective operations. We researched a big data parallel association rule mining algorithm-Sp-IEclat (Improved Eclat algorithm on Spark Framework). The algorithm is based on the Spark framework of memory computing, reduces disk input and output, reduces I/O load, and uses bitmap operations to reduce the time of intersection and CPU usage. The pruning technique of prefix division is used to reduce the amount of data in the intersection operation to reduce the operation time. The mushroom dataset and the webdocs dataset are used to test under two big data platforms. The experimental results show that the time efficiency of the Sp-IEclat algorithm is better than the Eclat algorithm under the MapReduce framework and the FP-Growth algorithm and the Eclat algorithm under the Spark framework. The value obtained from the performance monitoring of the cluster shows that, compared with the FP-Growth algorithm and the Eclat algorithm under the Spark framework, the CPU usage and I/O cluster load of Sp-IEclat are smaller.

Keywords: big data; association rule data mining; frequent itemset; Spark resilient distributed dataset (RDD); MapReduce framework

收稿日期: 2020-05-11
基金项目: 黑龙江省教育厅科学技术研究项目(12541142).
作者简介: 李成严(1972—)，男，教授，硕士研究生导师;
赵 帅(1997—)，男，硕士研究生.
通信作者: 辛 雪(1998—)，女，硕士研究生，E-mail: 1043620988@qq.com.

(C)1994-2022 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

0 引 言

关联规则挖掘技术是数据挖掘中的重要组成部分,广泛的应用在金融行业[1],零售业市场营销[2]及医疗[3]等领域。

关联规则挖掘的经典算法有 Apriori[4] 算法,FP-Growth[5] 算法及 Eclat[6] 算法。Apriori 在其进行迭代的过程将会产生大量的候选集并且放在内存中,当处理大数据集时会导致内存不足,而且 Apriori 需要重复的读取数据库,将会给系统 I/O 造成巨大压力; FP-Growth 算法将数据库的扫描次数压缩到了 2 次,但是在生成树结构的时候会有额外的开销,当数据集的支持度较低时,将会产生大量的节点,导致内存不足; Eclat 算法只读取一次数据库,但是取交集时会有大量的候选集存储在内存中,会耗费大量时间。

针对 Eclat 算法在数据集较大时求交集的时间代价会很大的问题,很多学者对 Eclat 算法进行了改进。文[7]提出了一种改进的 Eclat 算法 - Eclat$^+$ 算法。Eclat$^+$ 算法在计算候选集的支持度之前,首先检测支持度,当候选集是潜在频繁项集时,才执行交集操作; 文[8]提出了一种快速的 Eclat 算法,该算法可以通过使用 Minwise 散列和估计量来快速计算多个项目集的交集大小; 文[9]基于递增的搜索策略提出了一种改进的 Eclat 算法,称为 Eclat_growth。以上算法都在一定程度上提高了 Eclat 算法的运行效率,但是在求交集时仍会占用很多时间以及整个算法的 CPU 占用仍很高。

传统的关联规则算法无法处理大数据环境下的数据挖掘问题,所以有学者将算法在 MapReduce 框架下实现。文[10]将 Apriori 算法转移到 MapReduce 框架上实现; 文[11]介绍了两种算法,分别是基于 MapReduce 平台的 Dist-Eclat 算法和 BigFIM。Dist-Eclat 通过使用基于 k-fis 的简单负载平衡方案来关注速度。BigFIM 重点是利用混合方法挖掘非常大的数据库,优化为在真正大的数据集上运行。文[12]将 Eclat 算法转移到 MapReduce 框架上并进行了改进。文[13]提出了一种基于 MapReduce 的等长划分数据库,并使用位图来计算的关联规则挖掘算法。文[14]提出了一种按照等长切割数据集后在每一块数据集上使用 MapReduce 的 Apriori 算法或者 FP-Growth 算法的组合方法。文[15]提出了一种基于前缀共享树设计的关联规则挖掘算法,这种方法通过将共有的前缀树进行合并,从而达到减少内存占用和节省运算时间。文[16]提出了一种并行的 FP-Growth 算法,将传统的 FP-Growth 再加上前缀树的生成模式,使用了消息传递机制将规则按照前缀分配到各个 reduce 中。以上算法选用了 MapReduce 框架来解决大数据挖掘的问题,但由于 MapReduce 是基于非循环的数据流模型,在计算过程中,不同计算节点之间保持高度并行,导致需要反复使用一个特定数据集的迭代算法无法高效地运行。在内存占用方面,如果一个节点运行失败,需要将这个节点的任务重复运行多次甚至交给其他运算能力更高的节点重新计算,从而导致巨大的内存损耗。

Spark 框架不仅克服了 MapReduce 框架的上述缺点,还具有迭代运算效率高,集群 I/O 负载低等优势。文[17]针对提出了一种基于 Spark 框架的并行 Apriori 算法 FAFIM,并证明该算法的运行效率要远远高于文[10]提出的算法。文[18]针对 Apriori 算法在生成候选项集的大量开销问题,提出了 R-Apriori 算法,通过消除候选生成步骤并避免了代价高昂的比较从而降低计算复杂性。文[19]提出了一种基于 Apriori 增量并行算法。该算法随着数据集的增加,不需要从头开始计算整个数据库,而是根据以前的频繁项集更新频繁的项集。文[20]提出了基于 Spark 的分布式 FP-Growth 算法叫做 DFPS 算法,该算法的运行效率远远高于 FP-Growth 算法在 MapReduce 框架下的运行效率。文[21]提出了基于 Spark 实现的可扩展的并行 FP-Growth。文[22]提出了一种改进的 FP-Growth 算法,该算法修改了支持计数并在 Spark 下实现。

以上算法都是基于 Spark 框架下对 Apriori 算法和 FP-Growth 算法的改进,由于都是基于水平数据库的算法,在速度,I/O 负载和 CPU 占用方面仍然存在问题,所以选择在基于内存的 Spark 框架下对基于垂直数据库的 Eclat 算法进行改进,从而降低集群的 I/O 负载。根据文[7]中提到的对数据库使用预处理技术进行数据压缩,减少问题规模,并根据文[9]及文[13]中提出的使用位图的方法来进行计算,减少求交集的运算时间并降低 CPU 占用。根据文[13]及文[14]提出的方法对频繁项集进行划分,根据文[15]及文[16]提出的方法以前缀为划分条件对频繁项集进行划分,即对求得的频繁项集使用前缀策略进行划分并提交给不同的计算节点进行计

(C)1994-2022 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

算,这样能够减少需要求交集的数据量从而减少集群的运算量,从而提高了算法的运行速度。

本文的主要工作如下:

1) 将 Eclat 算法使用基于位图的计算策略并采用前缀划分策略对其进行改进,提高了运行效率,减少了 CPU 占用。

2) 将改进的 Eclat 算法在 Spark 框架下进行实现,降低了集群的 I/O 负载,提出了基于 Spark 框架的关联规则挖掘算法—Sp-IEclat 算法。

3) 通过运行相关的实验,与基于 MapReduce 下的 Eclat 算法和现有的一些基于 Spark 的关联规则挖掘算法进行实验比较。比较的内容为公共数据集下,不同支持度的挖掘时间性能表现。

本文其余部分构成如下: 第 2 部分为介绍相关概念; 第 3 部分介绍 Sp-IEclat 算法; 第 4 部分描述本算法的具体实现,并做复杂度分析; 第 5 部分进行数值实验并对结果分析; 最后给出结论。

1　相关概念

1.1　关联规则

设 $D=\{T_1,T_2,T_3,\cdots,T_n\}$ 是一个事务数据集,该数据包含 n 个事务项集,其中每个事务项集包含 m 个不同的项,$I=\{I_1,I_2,I_3,\cdots,I_m\}$。包含 K 个事务项的项集被称为 $K-$项集,K 为项集的长度。项集 X 在 D 中出现的次数叫做项集 X 的支持度。

如果项集的支持度不小于规定的最小支持度,则为频繁项集。反之,为非频繁项集。

前缀共享树: 设两个项集 $X=\{i_1,i_2,\cdots,i_k,\cdots,i_m\}$,$Y=\{i_1,i_2,\cdots,i_k,\cdots,i_n\}$ 它们的前 k 项相同,则$\{i_1,i_2,\cdots,i_k\}$称为项集 X 和 Y 的 $K-$前缀。项集{A,B,C,D} 和项集{A,B,E,F} 具有相同前缀{A,B}。

1.2　SPARK 框架

Spark 是一种基于内存的分布式计算框架,不仅包含 MapReduce 分布式设计的优点,而且将中间处理数据放入内存中以减少磁盘 I/O 从而提高性能。Spark 是基于 Spark RDD 编程,提供转换算子和行动算子 2 种算子。2 种算子都将中间结果存放在内存中,所以会有较快的运行效率。相比 MapReduce 框架,Spark 具有更高效、充分利用内存、更适合迭代计算和交互式处理的优点。

2　Sp-IEclat 算法

2.1　ECLAT 算法

Eclat 算法采用的数据结构是垂直数型数据结构,即数据形式为{item: TIDSet} 的形式。如此转换后,关联规则的挖掘实际上转换成了使用集合运算的方式。对于小数据集,集合运算的速度将会很快。但当数据集变大的时候,取交集的运算的代价会变得比较大,也会产生比较大的中间数据量。Eclat 算法对于大型数据集数据的关联规则挖掘时时间效率不是很理想,所以将 Eclat 算法进行优化使其更加适用于挖掘大型数据集。

2.2　SP-IECLAT 算法概述

针对 Eclat 算法求交集运算代价大的问题,采用位图交集运算来代替集合交集运算使用前缀划分策略将频繁项集进行划分。本文提出了 Sp-Eclat 算法,一种基于 Spark 框架的关联规则数据挖掘算法,在 Spark 框架下编程运行。Sp-Eclat 算法共分成 2 个部分,分别是数据预处理和计算频繁 $K-$项集。

首先是数据预处理。第一步要读取数据,Spark 的读取数据分为从本地文件系统中加载数据和从分布式文件系统(HDFS) 中读取数据,本文采用的是读取 HDFS 中数据。然后将得到的数据进行数据库格式的转换及非频繁项集的过滤。将水平数据库转换成垂直数据库,转换后将项集的支持度和给定的最小支持度进行比较,如果小于,则将该项集是非频繁项集,将其过滤掉; 如果大于,则得到了频繁 1 - 项集。最后位图存放数据,将得到的频繁 1 - 项集采用位图保存 TID,位图中 1 的个数就是该项集的支持度。

然后为计算频繁 $K-$项集。包含计算频繁 2 - 项集及根据频繁 2 - 项集求出频繁 $K(K>2)$ - 项集。在求频繁 2 - 项集时,对 itemID 求并集并对 TID 求交集,保留 TID 交集的长度大于等于最小支持度的作为频繁 2 - 项集。使用前缀划分策略对频繁 2 - 项集进行划分并分发给计算节点。计算得到的频繁 2 - 项集的大小是远小于整个事务的大小,对频繁 $K-$项集使用前缀划分策略进行划分需要触发到 shuffle 计算,而频繁 $K-$项集的大小同样远小于频繁 2 - 项集的大小,所以 Sp-Eclat 的网络通信开销会很小。Sp-IEclat 算法流程图如图 1 所示。

(C)1994-2022 China Academic Journal Electronic Publishing House. All rights reserved.　http://www.cnki.net

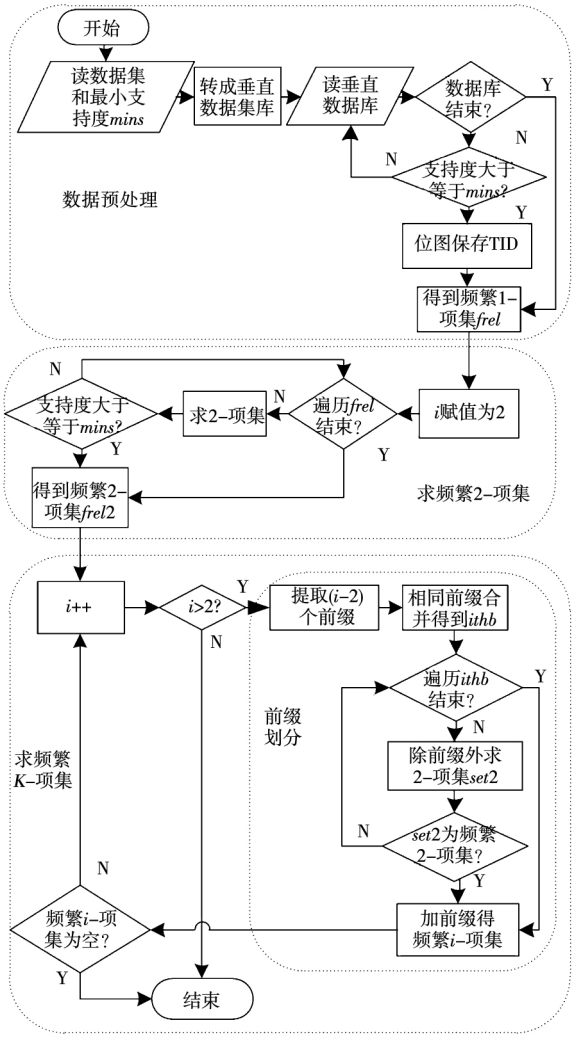

图 1 Sp-IEclat 算法流程图

Fig. 1 The flowchart of Sp-IEclat algorithm

2.3 数据预处理

Eclat 算法在生成 K – 项集的时候总是需要使用到$(K-1)$ –项集作为生成的前缀，但随着数据量的增加或者最小支持度的减小，生成 K – 项集的前缀数量会很大，求交集时的时间代价和 CPU 占用会很大从而导致该方法不可用。这种趋势在分布式框架上也变得非常明显，即当一台机器的性能不足以完成分配到的任务时，往往是需要系统将这个任务分配到性能更强的机器上，而这个调度代价是非常巨大的。

为了降低调度代价，通过数据预处理将读取的数据转换成垂直数据库并过滤掉支持度小于最小支持度的数据，从而减小了整个算法中生成的中间候选集。如图 2 所示，给出了一个示例说明当最小支持度为 2 时，水平数据库转换成垂直数据库并得到频繁 1 – 项集的过程。

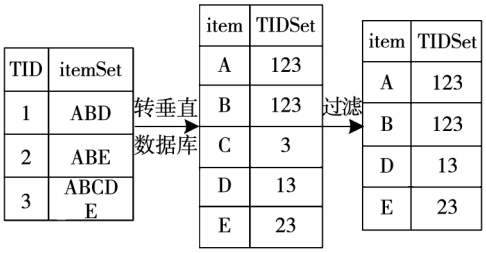

图 2 数据库形式转化得到频繁 1 – 项集示例

Fig. 2 Examples of frequent 1-item sets for database-form conversions

每一个事务 TID 都对应了一个项集 itemSet，表示为在该事务中分别出现的项。将每一个项拿出来并将该项出现的全部 TID 放入到 TIDSet 中就得到了垂直数据库。如图 2 所示，对于项 A 分别出现在 TID 为 1，2，3 的事务中，所以项 A 所对应的 TIDSet 为{1，2，3}。其次，对得到的垂直数据库中支持度小于最小支持度的数据进行删除。如图 2 所示，假设给定最小支持度为 2，转换成垂直数据库形式后，项 C 对应的 TIDSet 为{3}，该集合中只有一个元素，表明项 C 的支持度为 1，小于给定的最小支持度，所以将项 C 从垂直数据库中删除，这个过程使用 Spark 对于 RDD 提供的转换算子 filter 算子来完成，对于其他项 A，B，C，E，支持度都大于最小支持度，所以都保留。上述过程结束后，就得到了频繁 1 – 项集。

对于得到的频繁 1 – 项集，为了降低后续计算频繁 K – 项集的时间代价和 CPU 占用，在数据的导出中直接采用位图的形式存放 TID，位图是位操作的对象，值只有 0 或 1，TID 的值对应于位图中的标号设置为 1。BitSet 最小规模是 64 位，随着存储的元素越来越多，BitSet 内部会自动扩充，每次扩充 64 位，位图的大小是 TIDSet 中最大的 TID 向上取整 64 位。当数据集很大时，位图求交集的时间效率远远高于集合求交集时间效率。对于图 2 得到的频繁 1 – 项集，项集{A} 和项集{D} 的位图内部存放形式如图 3 所示。

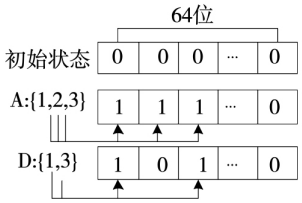

图 3 TIDSet 的位图存放形式

Fig. 3 The BitSet storage form of TIDSet

(C)1994-2022 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

2.4　计算频繁 2 – 项集

预处理中将水平数据库转换成了垂直数据库，并且得到了所有频繁 1 – 项集。在计算频繁 2 – 项集中，Sp-IEclat 算法使用 Eclat 算法取交集的思想，进行频繁 2 – 项集的计算。当数据量巨大的时候，Eclat 算法取交集的成本将会变得非常的高。所以对保存 TID 的位图求交集，位图作为基于内存的数据结构，即使对于很大的数据集，求交集的效率仍然很高。如图 4 所示，给出了用图 2 的频繁 1 – 项集来计算频繁 2 – 项集的过程。

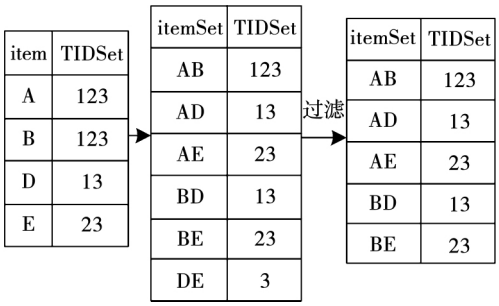

图 4　频繁 2 – 项集获取示例

Fig. 4　The example of getting Frequent 2-itemset

图 4 中，使用了图 2 的频繁 1 – 项集得到的 2 – 项集有{A,B},{A,D},{A,E},{B,D},{B,E},{D,E}，但 2 – 项集{D,E}对应的 TIDSet 为{3}，支持度为 1，小于给定的最小支持度，不满足频繁 2 – 项集要求，因此不将{D,E}加入到频繁 2 – 项集中。

将获得的频繁 2 – 项集使用前缀划分策略进行划分，具体前缀划分策略工作在 2.5 中说明，Sp-IEclat 算法是在 Spark 分布式框架下并行执行的，首先要 Dirver Program 触发集群开始作业，也就是在文件读取时应用已经被提交，然后 Cluster Manager 作为主节点控制着整个集群，将获得的频繁 2 – 项集分发到计算节点进行计算，每个结算节点接收到来自主节点的命令之后负责任务的执行。

2.5　前缀划分策略

根据文 [14] 提出的数据集划分技术，提出了前缀划分策略。根据文 [14] 中的引理 1，可以得到以下推论:

引理 1: 将数据集 D 划分为 S 个相等大小的数据块，记为 $D=\{D_1,D_2,\ldots,D_S\}$，将这些块上的本地频繁项目集分别表示为 $FI_1,FI_2,\ldots,FI_S$。并将数据集 D 上的频繁项目集表示为 FI。FI 是所有块本地频繁项集的并集的子集，即 $FI\subseteq FI_1\cup FI_2\cup\cdots FI_S$。

推论: 将频繁 K – 项集提取前 $(K-1)$ 个项作为前缀，将频繁 K – 项集按照具有相同前缀原则进行划分，划分的块数为前缀的个数。将得到的频繁 K – 项集 $(fre\text{-}k)$ 进行划分，假设不同前缀个数为 s 个，记为 $fre\text{-}k=\{fre\text{-}k_1,fre\text{-}k_2,\cdots,fre\text{-}k_s\}$，在划分后的每个块中除前缀外计算频繁 2 – 项集，每个块中得到的频繁 2 – 项集分别为 $fre\text{-}k_1-2,fre\text{-}k_2-2,\cdots,fre\text{-}k_s-2$，将得到的每个频繁 2 – 项集加上该块的前缀取并集就是所有的频繁 $(K+1)$ – 项集，并且该频繁 2 – 项集的支持度就是频繁 – $(K+1)$ 项集的支持度。

证明: 每一个频繁项集中的项都是顺序存放的，所以前缀是唯一的，对于每个块中的频 2 – 项集会存在重复的情况，但是加上前缀后的频繁 K – 项集就是唯一的。在任何一个频繁项集块 $fre\text{-}k_i$ 中，其中 $1\leqslant i\leqslant s$，如果存在频繁 2 – 项集，那么 $fre\text{-}k_i-2$ 的支持度一定大于最小支持度，所以加上前缀后一定是频繁 K – 项集。因为每个块得到的频繁 K – 项集都是唯一的，所以对于非频繁 2 – 项集，他也不会对应频繁 K – 项集。

在 Spark 框架的具体实现为首先调用 map( ) 将所有的前缀提取得到一个 RDD，也就是将每个频繁项中的第 $(K-1)$ 个元素提取出来得到一个集合。该 RDD 是由两部分组成，第一部分是提取出的前缀，第二部分是一个哈希表，包含剩余的前缀以及该频繁 2 – 项集对应的项集 ID。然后调用 reduceByKey( ) 将相同的前缀进行合并，这里的前缀就是要合并的 *key* 值，合并时相同前缀的 RDD 被合并成一个 RDD，合并后的 RDD 包含两个部分，分别是相同的 *key* 值和所有 value 中的哈希表拼接成一个大的哈希表。

下面给出前缀划分策略的具体示例，图 5 为图 4 中得到的频繁 2 – 项集使用前缀划分策略进行划分的过程示例。

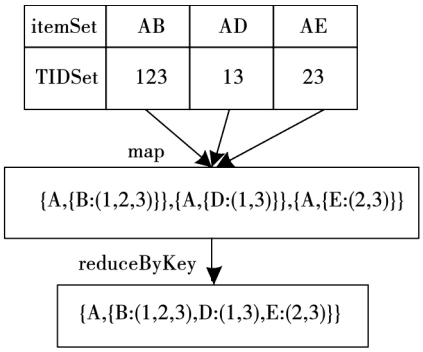

图 5　前缀划分示例

Fig. 5　The example of prefix division

(C)1994-2022 China Academic Journal Electronic Publishing House. All rights reserved.　http://www.cnki.net

对于图 4 所示获得的频繁 2 – 项集中前缀为 A 的 2 – 项集{{A,B}:(1,2,3),{A,D}:(1,3),{A,E}:(2,3)}进行前缀提取得到{A,{B:(1,2,3)}},{A,{D:(1,3)}},{A,{E:(2,3}}},这个过程调用 Spark 提供的转换算子 map()。然后进行规约得到{A,{B:(1,2,3),D:(1,3),E:(2,3)}},这个过程调用 Spark 提供的行动算子 reduceByKey()进行合并。

2.6 计算频繁 $K(K>2)$ – 项集

获取频繁 K – 项集首先要判断获得的频繁项集是否为空,若为空,则结束运行。若不为空,继续进行前缀划分,将在各个计算节点中针对具有相同前缀的项集,并行地以自底向上的形式进行迭代,用频繁 K – 项集产生频繁$(K+1)$ – 项集。即在每个节点中,对分配到该节点的项集进行前缀划分计算,产生不同前缀对应的所有项集,之后对除前缀外的所有项集进行计算频繁 – 2 项集,将满足条件的频繁 2 – 项集添加前缀得到频繁 K – 项集,并将计算结果保存到内存中。

获取频繁 3 – 项集,首先对频繁 2 – 项集进行前缀划分,划分后提取前缀,将除前缀外剩余的部分称为后缀,对后缀计算频繁 2 – 项集,将得到的频繁 2 – 项集加上前缀得到频繁 3 – 项集。对于图 5 中得到的前缀划分后的结果,将前缀为 A 的项集除去前缀的部分得到{B:(1,2,3),D:(1,3),E:(2,3)}再次进行 2 – 项集的计算分别得到{{B,D}:(1,3),{B,E}:(2,3),{D,E}:(3)}。然而对于获得的 2 – 项集中{D,E}的支持度小于给定的最小支持度 2,所以将其过滤掉。所以得到的频繁 2 – 项集为{{B,D}:(1,3),{B,E}:(2,3)},所以只要将前缀 A 分别加入到{B,D},{B,E}中就可以得到频繁 3 – 项集{{A,B,D}:(1,3),{A,B,E}:(2,3)}。

同理,对于频繁 K – 项集的计算,首先提取频繁$(K-1)$ – 项集前缀,这时前缀的个数为$(K-2)$个,提取前缀后对剩余部分计算频繁 2 – 项集,将前缀添加到频繁 2 – 项集中得到频繁 K – 项集。

3 算法实现

3.1 数据预处理

数据预处理将原始数据的储存从水平型数据库转换成垂直型数据库,并用位图保存 TIDSet。针对水平型数据库的数据集,若数据集本身就是以垂直型数据进行储存,计算每行中存在的 TID 即可。这个过程是将数据集进行存储方式的转换,同时过滤掉支持度小于最小支持度的数据,需要对整个数据库进行读取,所以该过程中网络传输量会增大。数据预处理的伪代码如算法 1 所示。

算法 1 数据预处理算法

输入: 原始数据集路径 *path*

输入: 最小支持度 *minsup*

输出: 满足最小支持度并以位图储存的频繁 1 – 项集 *fre_1*

```
1. javaRDD←sc. textFile(path)
2. map←new HashMap
3. for row ∈ javaRDD do
4. count←1 //设置行号
5. set1←row. split(" ")
6. for i ∈ set. length do
7. if set1[i] ∈ map. key then//item 已存在
8. map. put(set1[i],set1[i]. values + count)
9. else //item 不存在
10. map. put(set1[i],count)
11. count + +
12. for i ∈ map do
13. s←s + i. key + i. value + "\\n"
14. fre_1←new HashMap()
15. for i in map. size() do
16. if s[i:]. size > minsup do //频繁 1 – 项集
17. fre_1. add(s[0],BitSet(s[1:]). size) //转换为位图形式
18. return fre_1
```

3.2 计算频繁 2 – 项集

计算频繁 2 – 项集中求交集是采用基于位图计算的方式,提高了算法的效率,并将中间结果保存在内存中,方便后续频繁 K – 项集的计算。计算频繁 2 – 项集的伪代码如算法 2 所示。

算法 2 计算频繁 2 – 项集算法

输入: 预处理得到的频繁 1 – 项集 *fre_1*

输入: 最小支持度 *minsup*

输出: 满足最小支持度并以位图储存的频繁 2 – 项集 *fre_2*

```
1. fre_2←new HashMap()
2. while i ∈ fre_1 do
3. bitset2← i. values()
4. while j ∈ fre_1 and i < j //i 项的后面项
```

(C)1994-2022 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

5. *bitset*3←*j*. *values*( )

6. *bitset*4←*bitset*2&&*bitset*3

7. if *bitset*4. *size*( ) ≥*minsup* do //频繁 2 - 项集

8. *fre*_2. put(*fre*_1. *get*(*i*) + ";" +*fre*_1. *get*(*j*) , *bs*4)

9. return *fre*_2

3.3　计算频繁 K - 项集($K>2$)

计算频繁 K - 项集需要对频繁($K-1$) - 项集进行前缀划分操作,该操作会导致网络开销,因为在调用 reduceByKey 算子时会触发 shuffle,这个过程无法避免。但是因为求得的频繁($K-1$) - 项集都是保存在内存中的,所以后续划分时不需要再次从数据库或者 HDFS 中读取,这样减少很多网络开销。计算频繁 K - 项集的伪代码如算法 3 所示。

算法 3　计算频繁 K - 项集($K>2$) 算法

输入: 计算得到的频繁($K-1$) - 项集 *fre_k* - 1

输入: 最小支持度 *minsup*

输出: 满足最小支持度的频繁 K - 项集 *fre_k*

1. *map* = new HashMap( )

2. while *fre_k* - 1. *key* hasNext

3. *pre* =*fre_k* - 1. *key* [0: $K-2$] //提取 $K-2$ 个前缀

4. *suf* = new HashMap( )

5. *suf*. put(*fre_k* - 1. *key* - *pre*,*fre_k* - 1. get(*fre_k* - 1. *key*)) //除前缀项外都作为后缀

6. *map*. put(*pre*,*suf*)

7. *javaRDD* = sc. parallelizePairs((List) *map*)

8. *map*1 = new HashMap( )

9. *mappreadd* ← *javaRDD*. reduceByKey (*i*. *suf* + *j*. *suf*) //前缀相同时后缀合并

10. *fre*2 ← Getfre _2 (*mappreadd*. *value*. *key*, *minsup*))

11. *fre_k* = *mappreadd*. *key* +*fre_2*

12. return *fre_k*

3.4　算法复杂度分析

3.4.1　I/O 开销

在 Spark 计算框架中,I/O 消耗主要包含网络 I/O 消耗和磁盘 I/O 消耗。在本算法中,主要的 I/O 和网络消耗的步骤发生在计算频繁 K($K>2$) - 项集。

计算频繁 K - 项集用到前缀划分的剪枝技术,而前缀划分阶段会调用到 reduceByKey 算子触发 shuffle 过程,从而产生磁盘 I/O 和网络开销。对于 Spark 的 shuffle 而言,map 端需要把不同的 key 值及其对应的 value 值发送到不同机器上,reduce 端接收到 map 的数据后采用 Aggregator 机制将键值对保存到 HashMap 中,而 Aggregator 的操作是在磁盘上进行的,所以会产生大量的磁盘 I/O。由于对数据进行了清洗并且得到的频繁 K - 项集中随着 K 值的增大,项集大小逐渐变小,磁盘写入需要的 I/O 会逐渐变小,后面实验得到证明。

3.4.2　时间开销

为了描述可能的时间的开销,定义符号及含义如表 1 所示。

表 1　定义符号及含义

Tab. 1　Define symbols and meanings

符号	含义
$\|C_k\|$	数据集中每行数据的长度
M	集群中所拥有的机器数量
K:($K-1$)	表示由第 $K-1$ 项集所产生的数据量
P	计算节点的数量
NET_{speed}	节点之间的网络速度
$DISK_{speed}$	节点的磁盘 I/O 性能

时间开销主要包含位运算消耗,Spark 自身框架的维护消耗,I/O 网络消耗。由于位运算消耗的时间很短所以忽略不计,所以时间代价主要与当前集群的网络质量和 I/O 所使用的存储介质有关。

时间代价如公式(1) 所示。

$$Time_{cost}(K)=\frac{CPU_{cost}(K)}{CPU_{speed}(K)}+\frac{NET_{cost}(K)}{NET_{speed}(K)}+\frac{DISK_{cost}(K)}{DISK_{speed}(K)} \tag{1}$$

数据预处理是整个算法最主要的 *CPU* 消耗。*CPU* 的开销如公式(2) 所示,这个公式表示的是 *CPU* 平均的消耗情况。由于每个节点所分配到的数据量难以确定,但是在分发过后的数据规模都是确定的。

$$CPU_{cost}(K)=\frac{|C_k|\times[K:(K-1)]}{M\times P} \tag{2}$$

I/O 产生的最主要的开销是前缀划分中对前缀进行合并触发 shuffle 导致的,Spark 的 shuffle 过程既会产生网络开销也会产生磁盘开销。网络 I/O 开销如公式 3 所示,磁盘 I/O 开销如公式 4 所示。

$$NET_{cost}(K)=\frac{|K:(K-1)}{NET_{speed}} \tag{3}$$

$$DISK_{cost}(K)=\frac{|K:(K-1)}{DISK_{speed}} \tag{4}$$

(C)1994-2022 China Academic Journal Electronic Publishing House. All rights reserved.　http://www.cnki.net

4 数值实验与结果分析

4.1 实验环境

使用 2 台服务器，配置均为 3 T 硬盘，128 G 物理内存，第六代 Intel 处理器。操作系统是 Ubuntu16.04，集群参数中 Hadoop 的堆大小设置为 25 G。Spark 的 executor 内存设置为 25 G。开发语言采用 Java 语言。开发工具采用 IntelliJ IDEA(COMMUNITY 2018.2) 进行开发、编译和运行，Hadoop 采用 hadoop-2.5.1 版本，JDK 采用 jdk1.7.0_71，Scala 采用 scala-2.11.8，Spark 采用 spark-2.1.0-bin-hadoop2.7 版本。

4.2 数据集

在本实验中，使用了 FIMI 仓库的 Mushroom [21] 和 Webdocs [9] 数据集，是两个被广泛用于关联规则挖掘的数据集。数据集的参数如表 2 所示，常用于比较算法的性能。

表 2 数据集的参数

Tab. 2 Parameters of dataset

数据集	Mushroom	webdocs
事务平均长度	23	177.2
项个数	119	5 268 000
事务总数	8 124	1 690 000
大小/kB	570.4	1 468 006.4

4.3 算法效率对比

这里进行了 2 种比较，为了保证挖掘出的所有频繁项集都不为空，2 种比较都选择最大关联规则长度为 5 的情况。首先是本文提出的 Sp-IEclat 算法和 MapReduce 框架下的 Eclat 算法之间的比较，选择比较的数据集为 Mushroom，比较结果如图 6 所示。

由图 6 可以看出，在数据集较大时，本文提出的算法运行的效率要远高于 Eclat 算法在 MapReduce 框架下的运行效率。当支持度越来越小时，挖掘出的频繁项集越来越多，Sp-IEclat 算法会将中间结果存放在内存中，而且位图求交运算代替集合求交集运算，会节省大量时间。

其次，在 Spark 框架下将本文提出的 Sp-IEclat 算法和 FP-Growth 算法及 Eclat 算法进行比较。选择比较的数据集是 Webdocs 数据集。Spark 中包含了一个可以提供机器学习的功能的程序库，其中算法 FP-Growth 就是调用了 Spark MLlib 中的 FP-Growth 算法的接口。实验结果如图 7 所示。

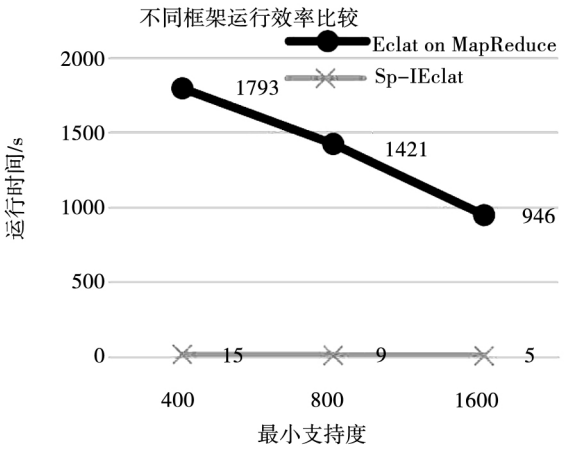

图 6 不同框架下效率对比

Fig. 6 efficiency comparison on different frameworks

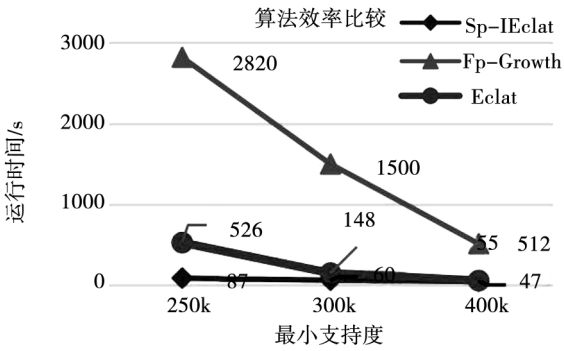

图 7 Spark 下算法效率比较

Fig. 7 Algorithm's efficiency comparison on Spark

在数据集较大的环境下，可以看到本文提出的 Sp-IEclat 算法的运行速度远大于 FP-Growth 算法和 Eclat 算法。随着支持度越来越小，产生的频繁项集越来越多时，Sp-IEclat 算法效率要更明显一点。对于 FP-Growth 算法来说，Sp-IEclat 算法不需要产生中间的树型数据结构，可节省大量时间。对于 Eclat 算法来说，Sp-IEclat 算法求交集时采用位图计算，并且使用前缀划分使求频繁 $K-$ 项集时遍历项集数目变少，提高了算法效率。

4.4 集群性能分析

在这里使用 webdocs.dat 数据集，在支持度为 250 K 的情况下对不同算法进行测试，如图 8 ~ 10 为使用集群监控软件 Ganglia 监测集群 CPU 占用率情况。

(C)1994-2022 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

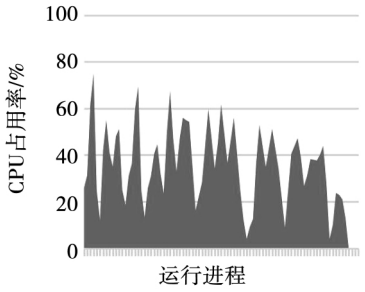

图 8　FP-Growth 算法 CPU 占用率

Fig. 8　CPU utilization of FP-Growth algorithm

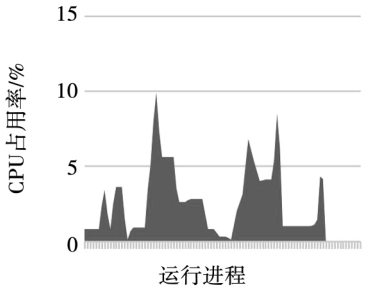

图 9　Eclat 算法 CPU 占用率

Fig. 9　CPU utilization of Eclat algorithm

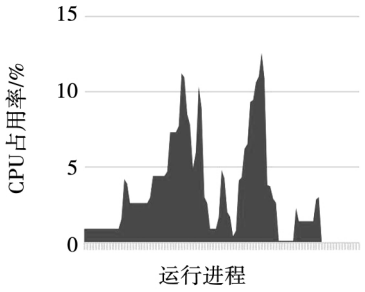

图 10　Sp-IEclat 算法 CPU 占用率

Fig. 10　CPU utilization of Sp-IEclat algorithm

以上 3 幅图片给出了不同算法的 CPU 占用率情况。对于 CPU 占用率，此集群环境下，Sp-IEclat 算法和 Eclat 算法的 CPU 占用率比较低，相对于 FP-Growth 算法来说，FP-Growth 算法 CPU 的占用率最大将近 80%，而 Sp-IEclat 算法和 Eclat 算法的 CPU 占用量最大不到 15%。原因为 Sp-IEclat 算法和 Eclat 算法都采用垂直数据结构，不需要多次扫描数据库，并且 Sp-IEclat 算法中采用的位图计算方法可以有效的减少 CPU 运行压力; 而 FP-Growth 算法采用水平数据库，并且需要构建生成树，在算法初始化的时候造成较大的 CPU 负载压力。

如图 11 ~ 13 为使用集群监控软件 Ganglia 监测集群 I/O 占用情况。

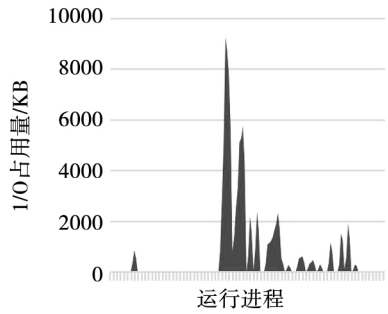

图 11　FP-Growth 算法 I/O 占用量

Fig. 11　I/O utilization of FP-Growth algorithm

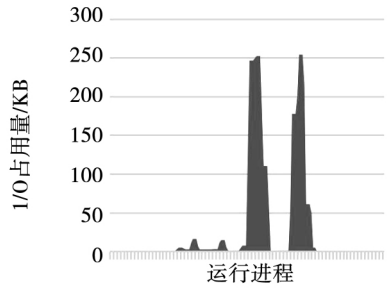

图 12　Eclat 算法 I/O 占用量

Fig. 12　I/O utilization of Eclat algorithm

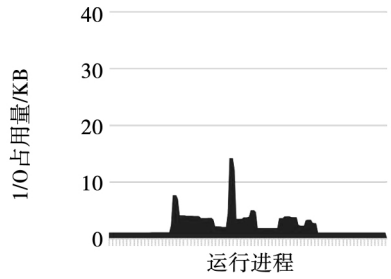

图 13　Sp-IEclat 算法 I/O 占用量

Fig. 13　I/O utilization of Sp-IEclat algorithm

以上 3 幅图片给出了不同算法的 I/O 占用情况。对于集群的 I/O 负载，FP - Growth 算法 I/O 负载压力主要出现在生成规则树中，最大达到 9 000 kB，当支持度低的时候生成树可能会非常大，此时需要大量的进行磁盘交换，从而导致 I/O 负载增大。Eclat 算法有两次负载峰值，达到 250 kB，因为频繁 4 - 项集和频繁 5 - 项集的传输过程 TIDSet 长度比较长，传输数据大，I/O 负载增大。Sp-IEclat 算法在整个算法中有一次负载峰值，仅仅不到 20 kB，因为要进行频繁 2 - 项集的前缀划分，即使后面也要进行前缀划分操作，但是占用量肯定要小于频繁 2 - 项集的前缀划分的 I/O 负载，因此集合的 I/O 负载也显著的降低。

(C)1994-2022 China Academic Journal Electronic Publishing House. All rights reserved.　http://www.cnki.net

5 结 论

本文提出了一种基于 Spark 框架的关联规则挖掘算法。本算法只需进行一次数据库遍历,并基于位图进行计算以及采用了前缀划分的剪枝方法,从而提高了运行效率,减少了集群的 CPU 占用率和 I/O负载压力。通过实验可得出在大数据下,较低支持度中也能保持算法以较快的速度和较低的 CPU 占用量及集群 I/O 负载来执行关联规则。而在较高的支持度下也能保持算法的高效。此算法使用范围比较广,可以用于大数据挖掘环境中。下一步考虑将该算法推广到不确定数据挖掘环境下。

参 考 文 献:

[1] MASUM, HOSEYNI Z. Mining Stock Category Association on Tehran Stock Market [J]. Soft Computing, 2019, 23(4): 1165.

[2] LEUNG K H, LUK C C, CHOY K L, et al. A B2B Flexible Pricing Decision Support System for Managing the Request for Quotation Process under Ecommerce Business Environment [J]. International Journal of Production Research, 2019, 57(20): 6528.

[3] GUAN V X, PROBST Y C, NEALE E P, et al. Identifying Usual Food Choices at Meals in Overweight and Obese Study Volunteers: Implications for Dietary Advice [J]. British Journal of Nutrition, 2018, 120(4): 472.

[4] HAN J, PEI J. Mining Frequent Patterns without Candidate Generation [C] // Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16 - 18, 2000, Dallas, Texas, USA. ACM, 2000: 1.

[5] ZAKI M J, PARTHASARATHY S, OGIHARA M, et al. New Algorithms for Fast Discovery of Association rules [C] // International Conference on Knowledge Discovery and Data Mining. AAAI Press, 1997: 283.

[6] PHILIPPE F V, JERRY C L, BAY V, et al. A Survey of Itemset Mining [J]. Wiley Interdisciplinary Reviews Data Mining & Knowledge Discovery, 2017, 7(4): 1.

[7] LI Z F, LIU X F, CAO X. A Study on Improved Eclat Data Mining Algorithm [C] //Advanced Materials Research. Trans Tech Publications Ltd, 2011, 328: 1896.

[8] ZHABG C K, TIAN P, ZHANG X D, et al. Fast Eclat Algorithms Based on Minwise Hashing for Large Scale Transactions [J]. IEEE Internet Of Things Journal, 2019, 6(2): 3948.

[9] MA Z Y, YANG J C, ZHANG T X, et al. An Improved Eclat Algorithm for Mining Association Rules Based on Increased Search Strategy [J]. International Journal of Database Theory and Application, 2016, 9(5): 251.

[10] VERMA N, SINGH J. An Intelligent Approach to Big Data Analytics for Sustainable Retail Environment using Apriori - map Reduce Framework [J]. Industrial Management & Data Systems, 2017, 117(7): 1503.

[11] CHAVAN K, KULKARNI P, GHODEKAR P, et al. Frequent Itemset Mining for Big Data [C] // International Conference on Green Computing and Internet of Things (ICGCIoT), Noida, IEEE, 2015: 1365.

[12] SUVALKA B, KHANDELWAL S, PATEL C. Revised ECLAT Algorithm for Frequent Itemset Mining [M] // Information Systems Design and Intelligent Applications. Springer India. 2016: 219.

[13] CHON K W, KIM M S. BIGMiner: A Fast and Scalable Distributed Frequent Pattern Miner for Big Data [J]. Cluster Computing, 2018, 21(3): 1507.

[14] WANG L. An Efficient Algorithm of Frequent Itemsets Mining Based on Map Reduce [J]. Journal of Information & Computational Science, 2014, 11(8): 2809.

[15] 丁勇, 朱长水, 武玉艳. 一种基于 Hadoop 的关联规则挖掘算法 [J]. 计算机科学, 2018, 45(S2): 419.
DING Yong, ZHU Changshui, WU Yuyan. Association Rule Mining Algorithm Based on Hadoop [J]. Computer Science. 2018, 45(S2): 409.

[16] LI H, WANG Y, ZHANG D, et al. Pfp: Parallel Fp-growth for Query Recommendation [C] // Acm Conference on Recommender Systems. ACM, 2008: 107.

[17] QIU H, GU R, YUAN C, et al. YAFIM: A Parallel Frequent Itemset Mining Algorithm with Spark [C] // Parallel & Distributed Processing Symposium Workshops. IEEE, 2014: 1664.

[18] RATHEE S, KASHYAP A, KAUL M. R-Apriori: An Efficient Apriori based Algorithm on Spark [C] // Proceedings of the 8th Workshop on Ph. D. Workshop in Information and Knowledge Management. ACM, 2015: 27.

[19] WEN H, KOU M, HE H, et al. A Spark-based Incremental Algorithm for Frequent Itemset Mining [C] // Proceedings of the 2018 2nd International Conference on Big Data and Internet of Things October 2018: 53.

[20] SHI X, CHEN S, YANG H. DFPS: Distributed FP-growth Algorithm Based on Spark [C] // 2017 IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC). IEEE, 2017: 1725.

[21] GASSAMA A D D, CAMARA F, NDIAYE S. S-FPG: A Parallel Version of FP-Growth Algorithm Under Apache Spark™ [C] // IEEE International Conference on Cloud Computing & Big Data Analysis. IEEE, 2017: 98.

[22] LI C, HUANG X. Research on FP-Growth Algorithm for Massive Telecommunication Network Alarm Data Based on Spark [C] // IEEE International Conference on Software Engineering & Service Science. IEEE, 2017: 857.

(编辑: 温泽宇)

(C)1994-2022 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net