

一种改进的并行关联规则增量更新算法研究

王 诚,赵申屹

(南京邮电大学 通信与信息工程学院,江苏 南京 210003)

摘 要:传统的基于频繁模式增长的并行关联规则算法在处理动态更新的数据集时,需要把更新后的数据集全部压缩到频繁模式树中,消耗了大量时间和存储空间,且没有充分考虑头表分组过程中组间负载量不同的问题。针对在关联规则的实际挖掘过程中,数据集快速增长所造成的增量更新问题,基于并行频繁模式增长 PFP-tree 算法,结合 Spark 分布式并行处理框架,提出一种改进的并行关联规则增量更新算法。在增量更新过程中,为了减少挖掘时间和存储空间,利用已有挖掘结果对新增数据集构建频繁模式树。通过改进头表分组策略,实现了并行挖掘节点之间的负载均衡。实验分析表明,相较于传统的关联增量更新算法,该算法是可行的且具备较高的挖掘效率和可扩展性,适用于动态增长的大数据环境。

关键词:Spark;关联规则;增量更新;并行计算;FP-tree

中图分类号:TP311

文献标识码:A

文章编号:1673-629X(2018)07-0048-05

doi:10.3969/j.issn.1673-629X.2018.07.011

Research on an Improved Incremental Updated Algorithm for Parallel Association Rule

WANG Cheng,ZHAO Shen-yi

(School of Telecommunications & Information Engineering,Nanjing University of
Posts and Telecommunications,Nanjing 210003,China)

Abstract:Traditional parallel association rule algorithm based on frequent pattern growth has to compress the whole updated dataset into the frequent pattern tree when processing a dynamically updated dataset,expending much time and storage space. Moreover,it neglects the load-balancing problem during the grouping stage. Aimed at the incremental updating problem caused by the rapid increasing of data in actual association rules mining,we propose an improved incremental updated algorithm for parallel association rule based on parallel frequent pattern-tree algorithm and the Spark distributed processing framework. During the updating process,in order to reduce the mining time and storage space,existing mining results are used to construct frequent pattern trees for the adding datasets. The grouping strategy for header-table is improved to ensure load-balancing between the nodes. The experiment demonstrates that compared with the traditional associative incremental updating algorithm,the proposed algorithm is feasible with high mining efficiency and scalability and suitable for large data environment with dynamic growth.

Key words:Spark;association rule;incremental updating;parallel computing;FP-tree

0 引 言

在现实的挖掘过程中通常存在增量更新^[1]问题。挖掘对象的数据集和支持度会发生变化,使用静态关联挖掘方法需要反复对更新后的数据集进行扫描,原有的挖掘结果失去作用,在面对大规模数据集时挖掘效率低下。将并行化计算框架与增量关联算法相结合,不仅能够提高挖掘效率,而且在海量数据挖掘中有实际意义。

增量关联规则^[2]出现至今,已有了众多的研究成果。Agrawal 提出了关联规则挖掘中最著名的 Apriori 算法。如今的大部分增量关联规则算法都是基于 Apriori 算法的改进或扩展。Cheung 等阐述了数据集变大的增量关联规则挖掘问题,提出了在数据集增加情况下的 FUP 算法^[3]。该算法通过比较原始事务数据库和新增数据库的项集之间的频繁和非频繁关系,对频繁项集进行增量更新得到更新后事务数据库的频

收稿日期:2017-08-22

修回日期:2017-12-28

网络出版时间:2018-03-07

基金项目:江苏省自然科学基金(BK20150861)

作者简介:王 诚(1970-),男,副教授,硕导,研究方向为数据挖掘、嵌入式技术;赵申屹(1992-),男,硕士研究生,研究方向为数据挖掘。

网络出版地址:<http://www.cnki.net/kcms/detail/61.1450.TP.20180307.1431.070.html>

频繁项集。由于 FUP 是基于 Apriori 算法产生的,需要多次扫描原始数据集并且生成大量候选项集,该算法仍存在挖掘效率低下的问题。文献[4]基于 FUP 算法,提出了改进的 UWEP 算法。文献[5]将 FUP 算法思想应用到 FP-Growth 算法中,提出了 FUFUP-tree 算法。该算法在数据更新时,只需扫描原始事务数据库中的变动部分,无须扫描整个数据库,从而大幅提高了挖掘效率。

针对传统单机关联挖掘方法在海量数据环境下挖掘效率低的问题,引入分布式并行处理框架^[6-8],如 Hadoop、Storm、Spark 等。文献[9]在 FUFUP-tree 算法基础上,提出了一种基于 MapReduce 编程框架的并行化模式算法-PFUFUP-tree。文献[10]结合 MapReduce 和 FUP 算法提出了一种并行的增量关联规则算法-MRFUP。文献[11]在 Apriori 算法的基础上,提出了一种基于 Spark 框架的并行化算法-AMRDD。

Spark^[12]作为一个新兴的大数据处理引擎,允许用户将数据加载至内存后重复使用。其基于内存的计算特性使其大数据计算性能大大超过 MapReduce。针对海量数据下的关联规则挖掘和增量更新问题,基于 PFP-tree 算法^[13]的思想,提出一种改进的关联规则增量更新算法(parallel updated frequent pattern growth algorithm on Spark,SPUFP)。该算法优化了频繁模式树结构和并行计算分组策略^[14],减小了时间和空间复杂度,并结合 Spark 编程框架^[15]进行实现,使其能够高效运行于 Spark 平台,大幅提高了海量数据环境下的挖掘效率。

1 增量更新算法

增量关联规则挖掘是指数据集变化或者支持度变化时的关联规则挖掘。Cheung 等提出的快速更新算法 FUP,是基于 Apriori 的增量更新算法,把增量更新后的项集分成 4 种类型,针对数据集增大的情况进行研究。FUP 算法的第 k 次循环仅需扫描数据库一次,候选项集生成新的频繁项集前会根据在 d 上的支持度先进行修剪,挖掘效率相比更新后的数据集中直接使用 Apriori 算法高很多。但是在对候选项集支持度进行比较时,该算法仍需多次重复扫描数据集来找出所有的频繁项集,会因计算量的增大导致计算速度缓慢,内存消耗过大。

PFP-tree 算法是基于 FP-Growth 的并行算法,在数据输入前,将原始数据集划分且存储到不同的节点上,通过扫描数据库,能够并行地计算各个节点上项的支持度计数,产生频繁 1 项集,降序排序生成 FList。之后,将 FList 分成多个小组,每组包含若干个项,组成 GList,再对每个组的事务组按照传统的 FP-Growth 创

建 FP-tree,根据 GList 中的项对 FP-tree 进行递归的频繁挖掘。然而 PFP-tree 算法在更新树结构的过程中需要把更新后的数据集压缩到频繁模式树中,会消耗大量的时间和存储空间,当数据增量较大时,树的规模会非常大,导致挖掘效率低下。其次,PFP-tree 算法在 FList 的分组步骤中,将 FList 根据并行计算的节点数平均分为 g 个组,没有充分考虑不同事务组计算的负载量不同的问题。

2 改进的关联规则增量更新算法

SPUFP 算法的主要思想如下:在增量更新过程中,利用已挖掘的原始数据集 DB 的中间结果建立更新后数据集 S 的头表,仅需要将新增数据集 db 压缩到频繁模式树中,排除了大量在 db 中频繁而在 S 中非频繁的项集,减少了挖掘时间,大幅减小了树的规模,释放出大量内存空间。针对并行计算中的分组问题,提出了一种优化的分组策略,令靠前的分组对更多的项进行挖掘,从而实现分组间的负载均衡。

2.1 算法描述

设定原始事务数据库为 DB,事务集 $T = \{T_1, T_2, \dots, T_n\}$,候选项集为 C_d ,频繁项集为 L_d ,原始事务数据库大小为 $|DB|$;新增数据库为 db ,频繁项集为 L_d ,新增数据库大小为 $|db|$;更新后的数据库为 S , $S = DB \cup db$,频繁项集为 L_s ,支持度为 \sup 。

输入:原始事务数据集 DB;新增事务数据集 db ;

输出:更新后的数据集 S 的频繁项集 L_s 。

步骤 1:扫描原始数据集,得到 DB 的候选 1 项集 C_{d_1} ,根据支持度得出频繁 1 项集 L_{d_1} ,排序后建立 FList,按照分组策略进行分组,构造 DB 的频繁模式树,对频繁模式树进行频繁项集的挖掘,得到 DB 的频繁项集 L_d 。

步骤 2:扫描新增数据集,得到 db 的候选 1 项集 C_{d_1} ,读取步骤 1 中的 C_{d_1} ,合并得到更新后数据集 S 的候选 1 项集 C_s ,根据支持度得出 S 的频繁 1 项集 S_{d_1} ,排序后建立 FList',按照分组策略进行分组,构造 db 的频繁模式树。

步骤 3:对频繁模式树进行频繁项集的挖掘,同时读取步骤 1 中 DB 的频繁项集 L_d ,对生成的每个频繁模式 k 和 L_d 做比较:若 k 属于 L_d ,则 k 是原数据集的频繁项,将 k 的支持度与在 L_d 对应的支持度计数相加可得 k 在 S 中的支持度计数,如果总计数大于 S 的最小支持度计数,则加入 S 的部分频繁项集 L' ,并从 L_d 中删除该项;若 k 不属于 L_d ,则 k 是新增数据集的频繁项,不能确定在 S 中是否频繁,将其加入 S 的候选集 C_s 。判别 L_d 中剩余的项,如果其支持度计数大于 S 的最小支持度计数,则加入 L' 。

步骤 4:扫描原始数据集,读取步骤 3 中 S 的候选集 C_s ,判断是否为 S 的频繁项集,将频繁项集与部分频繁项集 L 合并,便得到更新后数据集 S 的频繁项集 L_s 。

2.2 分组策略

在利用频繁 1 项集排序建立头表 FList 后,为执行并行计算,需要将 FList 中的项目分成 g 个小组,把每个组分得的项目存入名为 GList 的表中,再分发至各个节点展开并行计算。文中分组方法如下:设定 FList 中的项目数量为 M ,GList 中小组数量为 g ,指针 F_s 和 F_e 分别指向 FList 的表头和末尾,指针 G_s 和 G_e 分别指向 GList 的表头和末尾。

- (1) 如果 $2^g \leq M$,则将 FList 中 F_s 到 $F_s + M/2$ 全部的项目分到 GList 中的 G_s , F_s 指向 $F_s + M/2 + 1$, G_s 指向 $G_s + 1$,令 $M = M/2$, $g = g - 1$,转向步骤 4;
- (2) 如果 $2^g > M$,则将 FList 中 F_e 指向的项目分到 GList 中的 G_e , F_e 指向 $F_e - 1$, G_e 指向 $G_e + 1$,令 $M = M - 1$, $g = g - 1$,转向步骤 4;
- (3) 如果 $g > 1$,返回步骤 1、2;
- (4) 如果 $g = 1$,则将 FList 中剩余的项目全部分到 GList 剩余的最后一组中,分组结束。

3 基于 Spark 并行计算实现

3.1 Spark 编程模型

针对传统的增量关联规则算法在面对海量数据集时挖掘效率低的问题,借助 Spark 并行计算框架对算法进行改进,提高增量更新效率。

Spark 是一个通用的大规模数据快速处理引擎,将分布式数据抽象为弹性分布式数据集 RDD。RDD 是 Spark 计算的基础,支持并行操作,允许用户将数据加载至内存中重复地使用,一个 RDD 代表一个分区里的数据集。每个 RDD 的数据都以 Block 的形式存储于多台机器上。Spark 的 RDD 存储架构如图 1 所示。

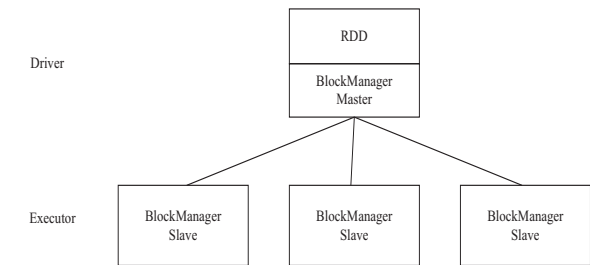


图 1 RDD 存储架构

Spark 支持两种 RDD 的基本操作(见表 1):转换 Transformation 和动作 Action。转换操作属于延迟计算,将一个数据集转换成新的数据集,当一个 RDD 转换成另一个 RDD 时并没有立即进行转换,仅仅是记住了数据集的逻辑操作。动作操作对数据集进行计算并

将计算结果返回给驱动程序,触发 Spark 作业的运行,真正触发转换算子的计算。Spark 中基于 RDD 的所有转换并不会即刻被执行,而是直到出现动作操作请求返回结果时,转换操作才被执行,减少了不必要的计算和返回。另外,RDD 支持内容的持久化,把内容保存在各节点的内存中,不会被覆盖或者删除,这样下次调用 RDD 时就不必创建新的 RDD,加快了计算速度。

表 1 RDD 基本操作

操作类型	函数名
Transformation	map, flatmap, filter, distinct, union,
	intersection, subtract, cartesian,
	reduceByKey, join
Action	collect, count, countByValue, reduce,
	foreach

Spark 的结构与 MapReduce 类似,由一个 Master 节点和若干个 worker 组成。用户通过编写程序 Driver 与 Master 进行交互,将所有定义好的 RDD 操作提交到 Master,Master 再把接受的 RDD 操作分发给各个 worker。worker 收到操作后,根据数据分块的信息,选择相应数据进行操作,生成新的 RDD。

3.2 基于 Spark 的增量频繁模式树算法

文中的关联增量更新算法基于 Spark 并行计算框架实现,算法主要分为两个阶段。第一阶段,对原始数据集 DB 进行并行挖掘并保留计算结果;第二阶段,在新增数据库 db 中,通过使用原数据库 DB 的挖掘结果完成更新后数据库 S 的频繁项挖掘。

3.2.1 对原始数据库的挖掘

设定原始事务数据库为 DB,事务集 $T = \{T_1, T_2, \dots, T_n\}$,候选项集为 C_d ,频繁项集为 L_d ,原始事务数据库大小为 $|DB|$;新增数据库为 db,频繁项集为 L_d ,新增数据库大小为 $|db|$;更新后的数据库为 S , $S = DB \cup db$,频繁项集为 L_s ,支持度为 sup。

输入:原始数据集 DB;

输出:原始数据集 DB 的频繁项集 L_d 。

步骤 1:将原始数据集 DB 存储到分布式文件系统 HDFS 中,数据集会被分割成若干个数据块,分发到各个工作节点上,每个数据分块都分别由多个事务 $T_i = \{item1, item2, \dots, item_n\}$ 组成。通过 textfile 读取数据并存入 RDD 中,对每个数据分块进行 flatmap 操作,过程中遍历所有事务 T_i 中的项,以 $item_i$ 为键,1 为值,输出 (item, 1) 形式的键值对。再利用 reduceByKey 操作累计项目数,将拥有相同键的值进行累加,输出 (item, count) 的键值对, count 表示对应项的总计数,得到原始数据集 DB 的候选 1 项集 C_{d_1} 。

步骤 2:以步骤 1 的结果作为输入,通过 filter 操

作,过滤掉总计数低于最小支持度的项,剩下的项即为频繁1项集 L_{D_1} 。读取 L_{D_1} ,把所有项根据计数 count 降序排列存入 FList,再根据并行计算的节点数量分成 m 个组,得到 GList。通过 broadcast 操作将 GList 转换成全局共享变量,以缓存形式分发到各个节点。

步骤3:GList 以哈希表的形式存储,以项为键,项所在的分组号 g 为值。读取每个事务 T_i ,然后对事务中的每个项 item 取出 Glist 中的分组号 g ,输出 $(g, (item_1, item_2, \dots, item_n))$ 的键值对。通过 group-ByKey 操作,将拥有相同分组号的键值对合并,发送到同一个工作节点,输出得到 (g, D_g) , D_g 为各分组号对应的事务组。利用 foreach 操作,对每个分组 g ,读取其在 GList 中对应的部分,扫描事务组 D_g ,创建根节点为 null 的 FP-tree。最后调用 growth 方法,扫描各个节点对应的 GList 部分,输出以 pattern 为键,支持度为值的键值对,得到原始数据集的频繁项集 L_D ,保存到 HDFS 中。

3.2.2 增量更新

输入:原始数据集 DB 的频繁项集 L_D ;新增数据集 db;

输出:更新后数据集 S 的频繁项集 L_S 。

步骤1:通过 textfile 将新增数据集 db 构成 RDD,db 同样被分布在各工作节点中。类似 3.1 中的步骤,通过 flatmap 操作遍历事务中所有项得到 $(item, 1)$ 形式的键值对,再通过 reduceByKey 操作累计项目数得到 $(item, count)$ 的键值对,即为新增数据集 db 的候选1项集 C_{d_1} 。读取 DB 的候选1项集 C_{D_1} ,将结果与之合并,得到更新后数据集 S 的候选1项集 C_{S_1} 。

步骤2:以 S 的候选1项集 C_{S_1} 作为输入,通过 filter 操作,过滤掉总计数低于最小支持度的项,剩下的项即为 S 的频繁1项集 S_{D_1} ,按支持度计数降序排序后生成 FList',按计算节点数分组得到 GList'。读取 DB 的分组 GList,将在 DB 中非频繁而更新后频繁的项加入分组;将在 DB 中频繁而更新后非频繁的项从分组中删除,得到处理后的 GList*。通过 broadcast 操作将 GList* 转换成全局共享变量。通过 foreach,对每个分组 g ,读取原始数据集 DB 在该分组的频繁项集 L_D ,扫描新增数据集在分组中的事务组 d_g ,创建 FP-tree。调用 growth 方法进行挖掘,过程中扫描 DB 相应分组的频繁项集 L_D ,对产生的每个键值对 k ,若 k 属于 L_D ,则 k 是原数据集的频繁项,将 k 的支持度与在 L_D 对应的支持度计数相加可得 k 在 S 中的支持度计数,通过 filter 操作过滤掉总计数低于最小支持度的项,剩下的项为 S 的部分频繁项集 L' ;若 k 不属于 L_D ,则 k 是新增数据集的频繁项,不能确定在 S 中是否频繁,将其加入 S 的候选集 C_{S_1} 。

步骤3:读取原始数据集 DB,对每个数据分块,根据 GList* 将 DB 发送到相应的分组中。对于每个分组 g ,读取该分组的候选项集 C_S ,扫描原始数据集 DB 在该分组的事务组,根据最终生成的支持度计数,与最小支持度进行比较,得到候选项集 C_S 中包含的 S 的频繁项集。将结果与步骤2中的部分频繁项集 L' 相加,即为更新后数据集 S 的频繁项集 L_S 。

4 实验

实验所用主机为 Intel(R) Core(TM) i3,主频 2 GHz,内存 2 GB,操作系统为 Ubuntu16.04 32 位,1 台为 Master 节点,4 台为 worker 节点,每个节点的配置差异很小。软件环境为 Hadoop2.7.0, JDK1.8, Spark2.1.0, scala2.12。

实验数据来自 <http://fimi.ua.ac.be/data/> 的 web-docs.dat.gz 数据集,容量约为 1 450 MB。

4.1 单机环境下的算法性能分析

实验仅在 Master 节点上将数据集平均分成大小相同的 10 组,每组包括 145 MB 数据。分别将 2 组、4 组、6 组、8 组数据作为原始数据集 DB,标记为 D_1 、 D_2 、 D_3 、 D_4 ,2 组数据作为更新数据集 db,最小支持度设为 10%,对 SPUFP 算法和 PFP-tree 算法、传统 FUP 算法的运行时间进行比较,如图 2 所示。

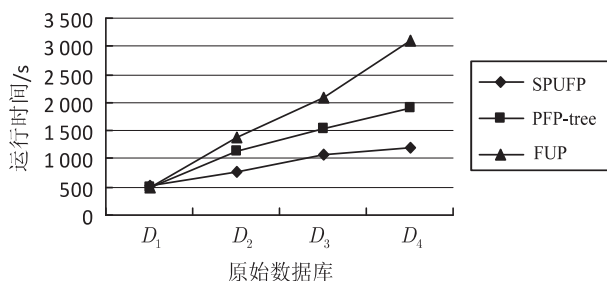


图2 不同算法运行时间比较

从图2可以看出,当数据量较小时,基于并行计算框架的算法和传统的单机算法的耗时差别不大。随着更新数据集的增加,两者计算的时间差开始增大,并行算法的挖掘效率明显提高且保持稳定。当更新数据量继续增长后,SPUFP 算法比基于 MapReduce^[16-17] 的 PFP-tree 算法有了较大的优势。这是因为在处理数据量较小时,并行算法调度操作的时间占用算法运行总时间的比例较大;当更新数据量逐步增大,此比例开始减小,并行算法效率开始优于传统算法;而随着数据集进一步增大,由于 Spark 能够将计算的中间结果缓存在内存中,节省了大量 I/O 操作消耗的时间,所以效率较 MapReduce 算法又有了明显提升。

4.2 分布式集群环境下的算法性能分析

用多个参数相同的节点搭建 Spark 集群环境测试算法的可扩展性,将数据集平均分成大小相同的 10

组,每组包括 145 MB 数据。分别将 2 组、4 组、6 组、8 组数据作为原始数据集 DB,标记为 D_1 、 D_2 、 D_3 、 D_4 ,2 组数据作为更新数据集 db,最小支持度设置为 10%。分别使用 1 到 4 个 worker 节点测试 SPUP 算法的运行时间,如图 3 所示。

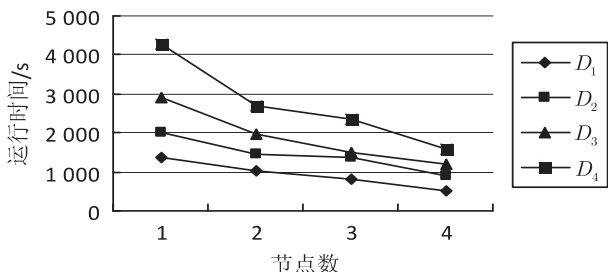


图 3 不同节点数运行时间比较

从图 3 可以看出,在数据量相同条件下,文中并行算法的运行时间随着 worker 节点数量的增加不断减少,且更新后的数据总量越大,时间减少的幅度越大,说明 SPUP 算法在数据集较大的环境下有良好的可扩展性。另一方面,随着节点个数的增加,不同容量的数据集的运行时间差别逐渐变小,这是因为节点增加会导致集群内节点间通信开销增大。

5 结束语

基于频繁模式树的思想,提出了一种改进的并行关联规则增量更新算法,优化了频繁模式树结构和并行计算分组策略,有效减少了挖掘时间和存储空间。实验结果表明,Spark 灵活的并行计算框架具备良好的可扩展性,其基于内存的计算特点在大规模数据环境下效率高于 Hadoop。在今后的研究中,可以通过 Spark 等云计算平台对更多传统数据挖掘算法进行改进,提高海量数据下的挖掘效率。

参考文献:

- [1] SOLANKI S K, PATEL J T. A survey on association rule mining [C]//Fifth international conference on advanced computing & communication technologies. [s. l.]: IEEE, 2015:212-216.
- [2] 张步忠,江克勤,张玉州.增量关联规则挖掘研究综述[J].小型微型计算机系统,2016,37(1):18-23.
- [3] CHEUNG D, HAN Jiawei, VINCENT T N, et al. Maintenance of discovered association rules in large databases; an

incremental updating technique [C]//12th international conference on data engineering. New Orleans, LA, USA: IEEE, 1996:106-114.

- [4] AYAN N F, TANSEL A U, ARKUN E. An efficient algorithm to update large itemsets with early pruning [C]//Proceedings of the 5th ACM SIGKDD international conference on knowledge discovery and data mining. San Diego, California, USA: ACM, 1999:287-291.
- [5] HONG T P, LIN Junwei, WU Y L. A fast updated frequent pattern tree [C]//Proceedings of IEEE international conference on systems, man and cybernetics. Taipei, Taiwan: IEEE, 2006:2167-2172.
- [6] 杨泽民.云计算模型中关联规则增量更新方法[J].计算机工程与设计,2014,35(2):504-508.
- [7] 谢欢.大数据挖掘中的并行算法研究及应用[D].成都:电子科技大学,2015.
- [8] 刘木林,朱庆华.基于 Hadoop 的关联规则挖掘算法研究—以 Apriori 算法为例[J].计算机技术与发展,2016,26(7):1-5.
- [9] 杨勇,高松松.基于 MapReduce 的关联规则并行增量更新算法[J].重庆邮电大学学报:自然科学版,2014,26(5):670-678.
- [10] 朱晓峰,李玲娟,徐小龙,等.基于 MapReduce 的关联规则增量更新算法[J].计算机技术与发展,2012,22(4):115-118.
- [11] 牛海玲,鲁慧民,刘振杰.基于 Spark 的 Apriori 算法的改进[J].东北师大学报:自然科学版,2016,48(1):84-89.
- [12] 曹博,倪建成,李淋淋,等.基于 Spark 的并行频繁模式挖掘算法[J].计算机工程与应用,2016,52(20):86-91.
- [13] LI Haoyuan, WANG Yi, ZHANG Dong, et al. PFP: parallel FP-growth for query recommendation [C]//Proceedings of the 2008 ACM conference on recommender systems. Lausanne, Switzerland: ACM, 2008:107-114.
- [14] 师金钢,郑艳,孙焕良,等.云环境中海量数据的并行分组密码体制研究[J].计算机科学与探索,2014,8(2):161-170.
- [15] 樊嘉麒.基于大数据的数据挖掘引擎[D].北京:北京邮电大学,2015.
- [16] 程广,王晓峰.基于 MapReduce 的并行关联规则增量更新算法[J].计算机工程,2016,42(2):21-25.
- [17] 秦军,郝天曙,董倩倩.基于 MapReduce 的 Apriori 算法并行化改进[J].计算机技术与发展,2017,27(4):64-68.