doi:10.3969/j.issn.1673-9833.2020.01.013

基于 Spark 平台的 FP-Growth 算法优化与实现

黄 婕[1, 2, 3]

（1. 湖南省飞机维修工程技术研究中心，湖南 长沙 410124；2. 长沙航空职业技术学院 航空电子设备维修学院，湖南 长沙 410124；3. 中南大学 软件学院，湖南 长沙 410075）

摘 要: 针对FP-Growth算法面对海量数据挖掘时串行操作机制出现内存瓶颈或者数据挖掘失效等问题，提出将基于 Spark 平台的 FP-Growth 算法在数据分组策略和项头表结构两方面进行优化。一方面提出一种 S 型的负载权值均衡分组的方式；另一方面，设计出一种新的项头表结构，此结构包含 Hash 查找表，能有效降低查找时间复杂度。实验证明，优化的基于 Spark 平台的 FP-Growth 算法（OptFP-Spark 算法）具有更高的并行运算加速比、更好的并行挖掘效果及更高效的计算效率。

关键词：Spark；关联规则；频繁项集；FP-Growth

中图分类号：TP311 **文献标志码**：A **文章编号**：1673-9833(2020)01-0077-08

引文格式：黄 婕 . 基于 Spark 平台的 FP-Growth 算法优化与实现 [J]. 湖南工业大学学报，2020，34(1)：77-84.

Optimization and Implementation of FP-Growth Algorithm Based on Spark Platform

HUANG Jie[1, 2, 3]

（1. Hunan Provincial Engineering Research Center for Aircraft Maintenance，Changsha 410124，China；2. Department of Aviation Electronic Equipment Maintenance，Changsha Aeronautical Vocational and Technical College，Changsha 410124，China；3. School of Software，Central South University，Changsha 410075，China）

Abstract：In view of the defect of memory bottleneck or data mining failure found in FP growth algorithm when processing massive data mining, a new method has thus been proposed to optimize FP growth algorithm based on spark platform in data grouping strategy and item header table structure. On the one hand, an S-typed grouping method has been proposed, which can realize a balanced grouping of load weights. On the other hand, a new item header table structure of FP-Growth with a hash look-up table has been proposed, which can effectively reduce the complexity of look-up time. Experimental results show that, characterized with a very high computational efficiency, the optimized FP-Growth algorithm, which is based on Spark platform, has a higher speedup of parallel computing and better parallel mining efficiency.

Keywords：Spark；association rule；frequent item set；FP-Growth

1 研究背景

在海量数据产生的今天，传统单机的关联规则挖掘算法在挖掘步骤上耗时多[1]，甚至无法进行关联规则的挖掘。为解决这一问题，将优化的关联算法在 Spark 并行平台上进行海量数据挖掘，提取出有规律

收稿日期：2019-05-22

基金项目：湖南省教育厅科学研究基金资助项目（17C0009）

作者简介：黄 婕（1979-），女，湖南长沙人，长沙航空职业技术学院副教授，硕士，中南大学访问学者，主要从事云计算，大数据方面的教学与研究，E-mail：huangjie918@163.com

有意义的数据信息，能进一步有效提高大数据时代海量数据分析的效率。

Apriori算法是主要针对布尔关联规则的最经典、最有影响力的算法。但是该算法会产生大量的候选集，使得复杂度急剧增大，算法效率大大降低。而且该算法还需要对频繁项集做大量的 IO 扫描，耗时且耗资源，对大数据的操作缺点明显。于是，由韩家炜等 [2] 提出的一种基于迭代 FP 树（frequent patten-tree，FP−Tree）生成频繁项集的关联规则的 FP−Growth（frequent patten-growth，FP−Growth）算法很好地解决了 Apriori 算法的问题。FP−Growth 算法是基于迭代 FP−Tree 生成频繁项集的关联规则算法。此算法仅进行两次数据集扫描，递归迭代构建 FP−Tree（FP 条件树），当 FP−Tree 中只有一个单分支时，递归迭代构建结束，最终得到频繁项集，FP−Growth 算法在时间、空间复杂度和数据挖掘的效率上都有明显改善，对于数据量较小的数据挖掘，FP−Growth 改进算法 [3] 具有一定优势，但随着数据量呈指数级增长时，这种串行的操作机制会出现内存瓶颈或者数据挖掘失效等问题。因此，利用 Spark 平台的并行计算框架能有效解决这一问题。

国内外学者对关联规则并行化操作有一定的成果。如黎丹雨等 [4] 构建了一种多层数据的模型，在不同层次之间挖掘出频繁多维序列模式，经过协同过滤输出 TOP−N 的推荐项目，但多维序列推荐模型并行挖掘的性能需加强。厍向阳等 [5] 在关联规则算法中指出，基于 Hadoop 的负载均衡数据 FP−Growth 并行算法在数据处理方面还有缺陷。M. Adnan 等 [6] 提出了一种 VMM（virtual memory manager）算法，该算法通过虚拟内存的管理，利用虚拟内存之间的通信实现并行化关联操作，但 VMM 算法需要消耗很多资源。

而 Spark 平台是一种高容错性的并行计算框架，速度远超 Hadoop，能快速进行大数据的分析挖掘，因此，分析 Spark 平台的 FP−Growth 并行化算法的优化及该算法的应用尤为重要。

2 Spark 简介

2.1 Spark 系统

Spark 是建立在 Java 虚拟机（Java virtual machine，JVM）上的开源数据平台框架，不需要涉及操作系统的底层细节，只借助 HDFS（Hadoop distributed file system）存储系统，就可以进行基于 Spark 平台的数据分析。因此，Spark 平台是一个通用性较强、成本较低的数据平台。与 Hadoop 平台相比较，Spark 是基于内存的运算框架，是一种基于 MapReduce 的并行分布式计算框架 [7]，具有速度快、效率高等特点。

Spark 的核心可以用 Spark 生态圈伯克利数据分析栈（Berkeley data analytics stack，BDAS）表示，如图 1 所示。

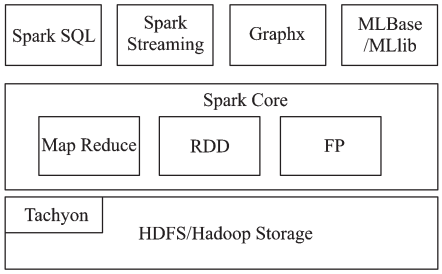


图 1 Spark 生态图

Fig. 1 A diagram of Spark ecosystem

Spark 是一种新兴的、快速处理海量数据的计算框架 [8]，该系统可提供流运算、迭代运算、图运算等解决方案。

2.2 Spark 分布式集群搭建

Spark 可以在本地进行单机模式的运行计算，也可以在分布式集群上并行运算。Spark 集群实现并行运算时，需要搭建分布式集群，常用的运行模式有 Standalone、Yarn−client、Yarn−cluster 3 种。这 3 种运行模式的不同在于有不同的资源分配方式，由不同的任务调度算法来执行计算任务。任意一个 Spark 程序都有对应的 Executor 进程，而每个 Executor 进程内部有多个 Task 线程与之对应。这种并行的资源分配、调度模式有利于不同 Spark 程序间的资源共享，大大提高了执行效率。

图 2 为 Spark 运行构架图。

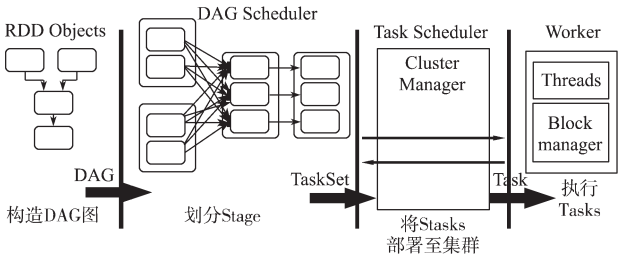


图 2 Spark 运行构架

Fig. 2 Schematic diagram of Spark run-time architecture

图 2 中，程序运行中通过动作触发 job，其中 job 构建 DAG 图是基于 RDD 的依赖关系的，构建好的 DAG 图再由 DAGScheduler 解析、构建成不同的 Stage，并计算 Stage 间的依赖关系。然后，由 TaskScheduler 调度器分配工作集 TaskSet，再划分成多个线程并行计算，同时将计算结果返回给 TaskScheduler，再返回给 DAGScheduler，计算结果全部完成后返回给驱动程序或者保存在外部存储系

统中，并将资源全部释放。

3　基于 Spark 平台的 FP−Growth 算法

3.1　FP−Growth 算法工作流程

Apriori 算法产生大量数据集，造成算法运行效率低下，而 FP−Growth 算法不用反复读取事务集，且不会生成大量的候选项集，既节省了内存资源又提高了读写效率，适合大数据量的数据挖掘 [9]。

FP−Growth 算法整个工作过程只进行两次扫描事务集：进行第一次事务集扫描后，利用支持度次数递减的规则将事务集排序，找出支持度最高的项即频繁 1− 项集，将第一次排序的末次项删除；再进行第二次扫描，过滤后将事务插入构建的 FP−Tree 中；再从 FP−Tree 中挖掘频繁项集后按条件迭代生成条件 FP−Tree，直到 FP−Tree 中只有一个结点时结束，将挖掘出的频繁项集合得到频繁项集。在生成上述频繁项集的同时，按 *A*−*B*=>*B* 的置信度大于最小置信度的原则，生成强关联规则 [10]。

3.2　基于 Spark 的 FP−Growth 算法思想

基于 Spark 平台 FP−Growth 算法直接递归求得每一次排序后支持度最高的频繁项，并将事物数据集分配给各计算节点，这种利用 Spark 平台的 FP−Growth 算法称为 FP−Spark（FP−Growth−based−on−Spark）算法。此算法先将事物集转换成弹性分布式数据集（resilient distributed datasets，RDD），再在各计算节点上对频繁项集进行并行数据挖掘。FP−Spark 算法设计如图 3 所示。

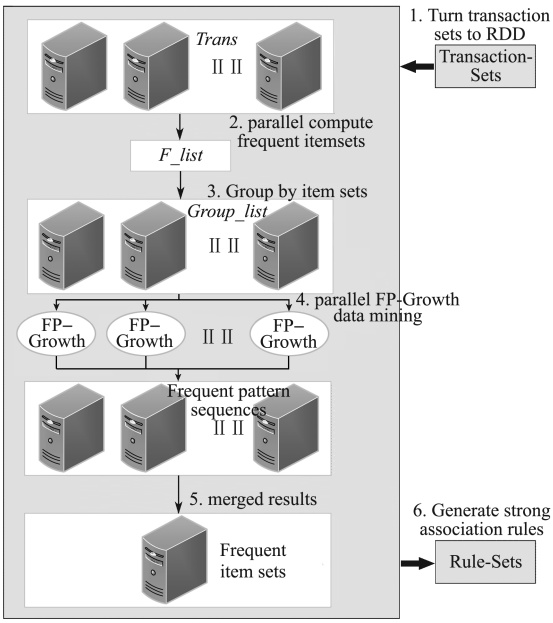


图 3　FP−Spark 算法设计

Fig. 3　FP−Spark algorithm design

FP−Spark 算法步骤可分成如下 6 步。

1）生成 *Trans*。生成弹性分布式数据集 RDD 数据集，利用弹性分布式数据集的 <*map*, *reduceByKey*> 键值对操作事务，构成 *Trans* <*Transactions*, *times*> 键值对，其中 *Transactions* 为事务，*times* 为事务出现的次数。

2）生成频繁 1− 项集 *F_list*。先将 *Trans* 进行 flatMap 操作，把 *Trans* 中的 *Transactions* 分成一个个事务项；再利用 reduceByKey 操作生成 <*item*，*support*> 键值对，事务项由 *item* 表示，支持度计数由 *support* 表示；然后经过 RDD 的 collect、toArray 操作，将 RDD 输出成 list 格式，按照 sortBySupport 降序排序，过滤掉最小支持度项后，得到频繁项集 *F_list*。

3）构建 *Group_list*。根据频繁项集 *F_list*，把 *Trans* 中的 *Transactions* 重新排序，过滤掉非频繁项，再按一定方式分组，得到 *Group_list*<*id*，*super_Trans*>，其中 *id* 为组号，*super_Trans* 为属于该组的事务集。

4）生成频繁模式序列。将 *Group_list* 中的事务分布到各节点，利用 flapMap 调用 FP−Growth 算法，构建 FP−Tree，递归挖掘 FP−Tree，生成频繁模式序列。

5）合并频繁项集。将上一步骤生成的频繁模式序列转码合并，写入 HDFS。

6）生成强关联规则。先产生后项是一项的关联规则，再两两合并后项，由两项的候选关联规则生成后项，其中的强关联规则由最小置信度阈值而得，依次反复逐层生成强关联规则。

3.3　基于 Spark 的 FP−Growth 算法关键方法

基于 Spark 的 FP−Growth 算法（也叫做 FP−Spark 算法）关键点是并行计算项的支持度和将事务集排序过滤后的分组策略。

3.3.1　并行计算项的支持度

按照上述 6 个步骤中的生成频繁项集 *F_list*，按照降序排序支持度和频繁 1− 项集。算法 1 是将原始数据集转换到弹性数据集的过程，算法 2 是计算支持度的过程。

算法 1　原始数据集转换到弹性数据集的过程

input：Source datasets

output：RDD

{

　result=(result.split(“”).toList,1)

}

Return <Transactions：事务列表 , times：1>

{

　将相同事务的 times 相加

```
}
Return Trans< Transactions ：事务列表，times>
```

算法 2 支持度的计算过程

```
input：弹性数据集
output：F-list、support_times
{
 List=List[(String, Int)]（ ）
 for（ i=0;i<list.length;i++）
    result.add(i, list_2)
}
Return result<item：item, support：support_times >
{
  将相同事务的 times 出现次数相加
  按 times 值过滤并降序 sortBySupport
}
Return F_list< item：item，support：support_times>
```

3.3.2 事务分组策略

将事务集 *Trans* 进行挖掘频繁 1 项集过滤得到 *F_list*，映射 map，其中 *Transactions* 表示事务项名称，*times* 表示在频繁 1 项集中出现的次数，并将 *Trans* 集合利用 map 映射到 *F_map* 中。映射过程中，对 *Trans_id* 排序后过滤非频繁项；其中 *Trans* 的每个事务，按 *F_map* 的 <*Transactions*，*times*> 将 *times* 的编码按升序排序，删除非频繁项后得到 *Trans_list*，其中每一条事务项按支持度计数递减排序，且都大于最小支持度计数。图 4 是对事务集分组的示意图。

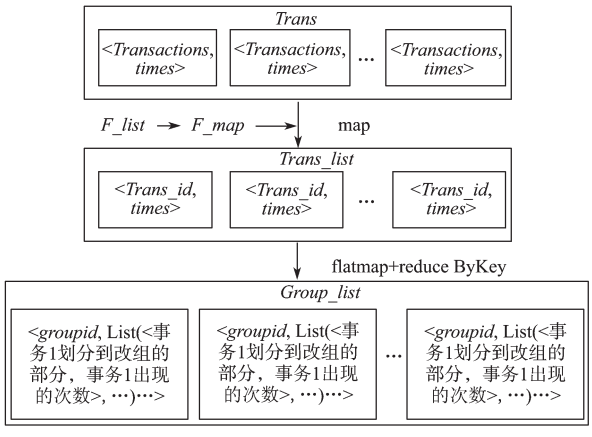


图 4 事务集分组示意图

Fig. 4 Process of transaction grouping

FP-Spark 算法分组策略依据 *F_list* 的分组情况对 *Trans_list* 进行分组。首先求出每个分组的最大个数 *g_size*：利用 *F_list* 的长度除以节点数向下取整后加 1。

$$g\_size=\left\lfloor \frac{F\_list.\text{size}()}{\text{分组数}} \right\rfloor +1。 \quad (1)$$

然后将 *F_list* 中的项依次分到 Spark 集群节点上，集群节点与 *groupid* 一一对应，成为 FP-Spark 算法挖掘时对应的表头项；再对事务集数据分组。算法 3 为对事务集分组的伪代码。

算法 3 划分事务集

```
input：Trans_list
output：Group_list
{
 groupid=-1
  for（ i=T.list_length()-1;i>=0;i--）{ // 从后往前遍历
    if(!(T_list[i]-1/g_size==groupid)){
      groupid=T_list[i]-1/g_size
      T=T_list.dropRight(T_list.length()-i-1, value)
result.add(groupid,T)
        }
    }
}
Return result< groupid，事务集部分 >
{
     统计 result 中的数据集合
}
Return Group_list< groupid，该组的事务集 >
```

从后往前遍历，如果 *T_list*[*i*] 所属的 *groupid* 没有出现过，则此事务还没有划分到该组，将 *T_list*[0] 到 *T_list*[*i*] 划分到该组；若 *groupid* 出现过，则不做任何操作，跳过此项，*i*−1，往前遍历，直到没出现过该组。

4 基于 Spark 的 FP-Growth 算法优化

基于 Spark 的 FP-Growth 算法实现了数据挖掘频繁项集的并行化操作，但这种 FP-Spark 算法的分组策略比较简单，且算法在进行本地挖掘时使用的项头表结构是数组，时间复杂度较高。针对这两个问题，课题组提出一种优化的均衡分组和新的项头表结构的 FP-Spark 算法（称为 OptFP-Spark 算法），大大提高了大数据处理的效率和性能。

4.1 优化分组

为了实现数据的均衡划分，只需将频繁项集在 *F_list* 的序号视为负载权值，在保证负载权重总和相差不大时，即可将数据划分均衡化。假设分组个数是 *m* 个，从 *F_list* 的表尾往前遍历，将 *m* 个项依次划分到第 1~*m* 组，再往前将 *m* 个项划分到第 *m*~1 组，S 形的划分方式将 *F_list* 分组完。优化分组的方式如图 5 所示。

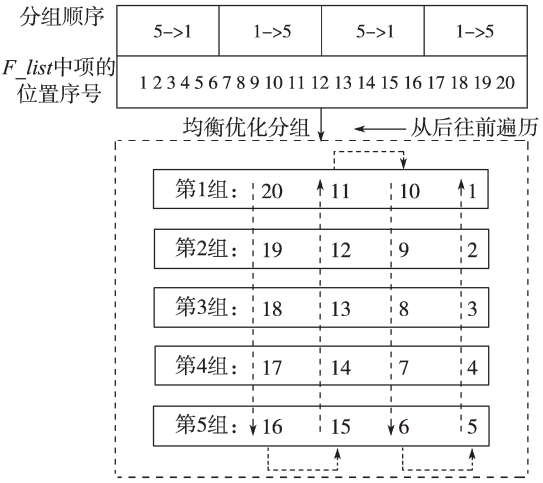


图 5　优化分组方式

Fig. 5　Optimized grouping

图 5 所示的优化分组方式，采用 S 形分组方式，分组后 *F_list* 中负载权值平均分布，从而达到均衡分组，使得 Spark 集群上的节点量保持相对一致，提升了整个集群的运行效率，同时也提高了 FP−Spark 算法的效率。

4.2　优化项头表结构

项头表结构直接影响着算法的运行效率，因此优化项头表结构能提高算法在构造 FP−Tree 的遍历效率，进而提高整个算法的效率。优化项头表结构是增加一个哈希表，即利用 Hash 算法进行优化，如图 6 所示。

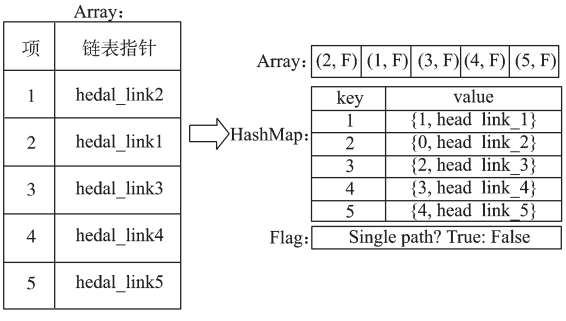


图 6　项头表结构优化图

Fig. 6　Optimized head-table diagram

其优化的过程如下：

1）初始的项头表结点原本是个数组，包含事务项和链表指针，优化后将链表指向布尔值，是一个布尔值的指针。当指向的布尔指针指向 True 时，说明属于此次扫描的事务项，再将此事务插入 FP−Tree，设置项表头结构的布尔值为 True。继续扫描下一事务，直至布尔值为 False 为止。

2）Hash 表结构是（*key*，*value*）的二元组，*key* 为事务项编码，*value* 为对应的链表指针，指向在 FP−Tree 出现的位置。排序时根据项目名查找哈希表，找出下标值，且设置对应的布尔值为 True。

3）Flag 标签有 True、False 两种值。单一路径时频繁模式树是 True，不是单一路径则为 False。单一路径的这种获取频繁项集的方式，无需递归操作，降低了时间和空间的复杂度，提高了算法效率。

4）优化后的项头表结构也需将 FP−Tree 的节点做修改，优化后的 FP−Tree 节点如图 7 所示。

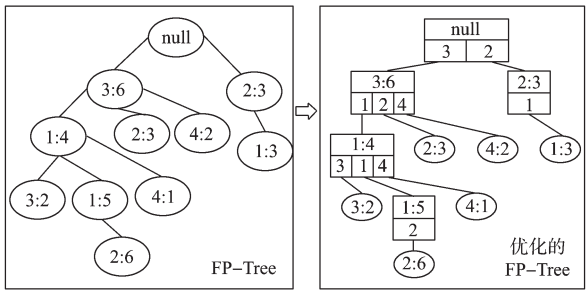


图 7　优化前后的 FP−Tree

Fig. 7　FP−Tree before and after optimization

由图 7 可知，优化后的项头表结构将传统算法的时间复杂度由原来的 $O(n^2)$ 降低到 $O(n)$，算法效率有很大提升；增加的 flag 标志位能减少递归所占用的资源，虽然空间复杂度有所消耗，但是在可以接受的范围内。优化了项头表结构后，大大提高了整个算法的运行效率。

4.3　OptFP−Spark 算法实现步骤

基于 Spark 平台的优化 FP−Growth 算法，也称 OptFP−Spark 算法。它既对分组策略进行了优化，又对项头表结构实现了优化。其优化步骤如下：

1）生成 *Trans*。首先将事物集转换成弹性数据集 RDD，在弹性数据集 RDD 上对事务进行 map 和 reduceByKey 操作，构成 *Trans*<*Transactions*，*times*> 键值对，其中 *Transactions* 是事务，*times* 是事务出现的次数。

2）生成 *F_list* 列表。在 *Trans* 集合上进行下列 flatMap、reduceByKey、collect、toArray 和 map 操作，构成频繁 1− 项集，把 *Trans* 中的 *Transactions* 分成一个个事务项；再利用 reduceByKey 生成键值对 <*item*：事务项，*support*：支持度计数 >；然后经过 RDD 的 collect、toArray 操作，将 RDD 输出成 list 格式，按照 sortBySupport 降序排序，过滤掉最小支持度项后，得到频繁 1− 项集 *F_list*。

3）构建 *Group_list*。根据频繁项集 *F_list*，把 *Trans* 中的 *Transaction* 重新排序，过滤掉非频繁项，再按优化分组的策略将事务集均衡分组。

①在排序过滤后的 *F_list* 分组基础上，根据前面提到的优化分组策略，进行 S 形分组，得到新的分组 *B_list*。

②在 *B_list* 中实现对 *Trans* 事务集分组，得到 *Group_list*：<*groupid*，*super_Trans*>。其中，*groupid* 为组号，*super_Trans* 为属于该组的事务集。

4）并行生成频繁模式序列。在各 *worker* 节点并行频繁模式树挖掘。将 *Group_list* 中的事务分布到各节点，再在各 *worker* 节点利用优化的项头表结构并行挖掘，生成频繁模式序列。

5）合并频繁项集。将上一步骤生成的频繁模式序列转换格式合并后写入到 HDFS。

6）强关联规则的产生。在逐层生成频繁模式序列时，逐层生成强关联规则。

对 OptFP−Spark 算法的优化，重点是对上述步骤中的第 3 步和第 4 步进行，即利用 S 型的分组策略均衡分组后，再实现优化的项表头结构的并行频繁模式树的挖掘，从而得到频繁模式序列集。OptFP−Spark 算法的实现过程如图 8 所示。

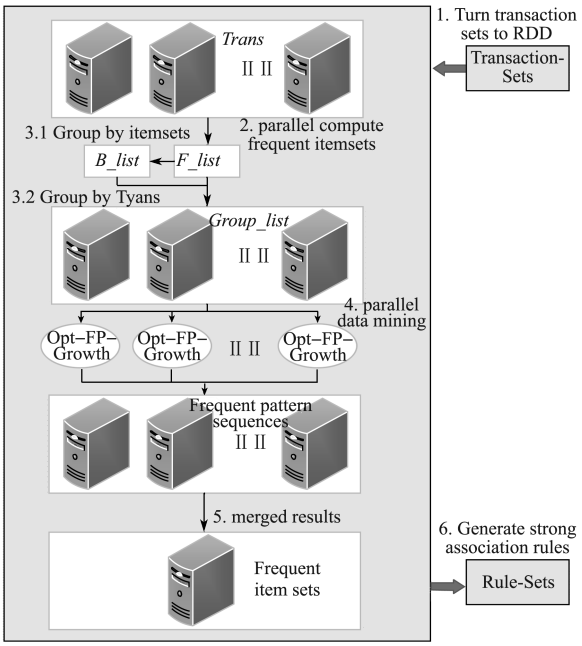


图 8　OptFP−Spark 算法实现图

Fig. 8　Optimized FP−Spark algorithm diagram

利用 S 形的优化分组算法如算法 4 所示，将 *F_list* 转换成 *B_list*。

算法 4　*F_list* 优化分组

输入：*F_list*

输出：*B_list*

```
Begin：
  Len=F_list.length    // 从后往前遍历
  For(i=len;i>0;i-=(2*gnum)) do
 For(j=i,gid=0;j>0 and gid<gnum;j--,gid++) do
  B_list(j)=gid    // 将项划分到对应组
    End
    End
    End
    For(i=len-gnum;i>0;i-=(2*gnum)) do
    For(j=i,gid=gnum-1;j>0 and gid)=0;j--,gid--) do
     B_list(j)=gid    // 将项划分到对应组
      End
    End
    End
```

在输出 *B_list* 后，将对事务集均衡分组。先将 *B_list* 转换成 *Map* 个数，再从后向前遍历，事务集优化分组伪代码如算法 5 所示。

算法 5　事务集优化分组

```
input：B_list ,Trans_list
output：Group_list
{
 hash=B_list.clone()
 result=List[(Int,(list[Int],Int))]( )
 for（ i=T_l.length-1;i>=0;i--）{
   if(hash.contains(t._l(i))){
    result.add(t._l(i))
    for each group  in trans_group(hash(t._l(i))) do
     hash.remove(group)
   }
}
}
Return result,<groupid，Transactions of this group>
{
  统计相同组号的数据信息并保存至 result 中
}
Return Group_list,< 组号，该组的事务集 >
```

在均衡分组输出 *Group_list* 后，利用优化的项表头结构并行频繁模式序列的挖掘，生成的 FP−Tree 的子结点数组增加 hash 表。最终生成强关联规则合并输出结果。

5　对比实验与结果分析

优化的 OptFP−Spark 算法和 FP−Spark 算法在两个大型数据集上做对比实验，考查数据规模和支持度对数据挖掘的影响。

实验采用 Spark 集群，一个 Master 主节点，4 个 Slave 从节点。实验数据源自数据仓库 [11] 中下载，D1 数据集和 D2 数据集的事务项和事务是网站真实交易的数据，如表 1 所示，有 1.48 GB 数据量，包括 160 多万条事务，500 多万事务项。

利用以上数据集特征从数据规模、支持度两个方面对比 FP−Spark 算法和 OptFP−Spark 算法的性能。

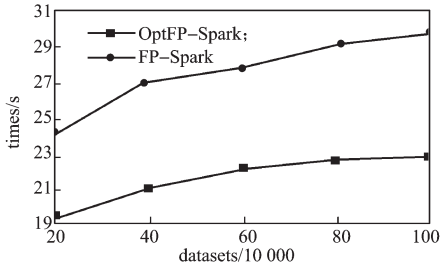
表 1　数据集表

Table 1　Table of data sets

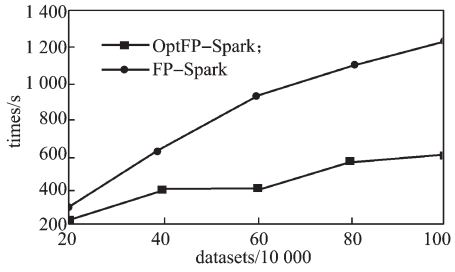
数据集	事务项 / 个	事务 / 个
D1	880	100 180
D2	5 270 090	1 699 016

5.1　数据规模对算法性能的影响

将支持度定为 0.8，节点数量保持不变，测试数据规模对比算法的运行时间。对比实验选取了 D1 和 D2 两个数据集，分别在 20 万 ~100 万事务集上进行比较，由于 D1 数据集较小，增加了 D1 数据集的拷贝来进行实验，实验结果如图 9 所示。



a）D1



b）D2

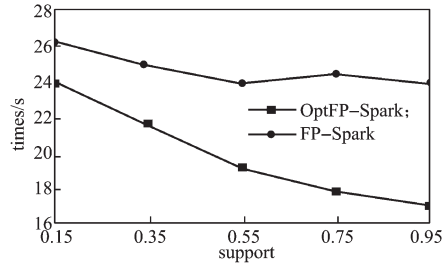
图 9　数据规模对性能的影响

Fig. 9　The impact of data size on performance

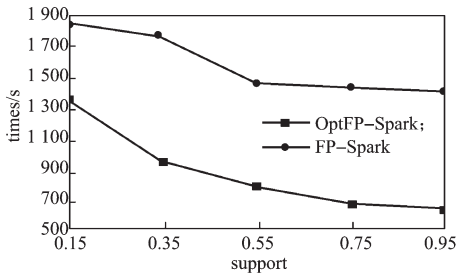
由图 9 所示实验结果可以得知，随着数据量的增大，优化的 OptFP-Spark 算法的性能明显提高。因此均衡的分组策略和项头表结构 hash 表的优化都有利于算法的数据挖掘，OptFP-Spark 算法的挖掘效率有了明显提升。

5.2　支持度对算法性能的影响

在节点数量不变，支持度分别取不同值时，对比两个算法的数据挖掘性能，得到不同的实验结果，如图 10 所示。



a）D1



b）D2

图 10　支持度对算法性能的影响

Fig. 10　The influence of support degree on algorithm performance

由图 10 所示的实验结果可以得知，随着支持度的增加，数据挖掘的时间复杂度减少，但优化的 OptFP-Spark 算法的性能更优，因此均衡的分组策略和项头表结构 hash 表的优化在数据量越大的情况下，性能提升越明显。

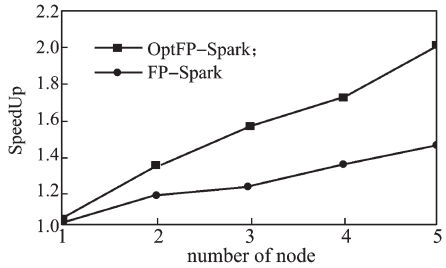
5.3　加速比对算法性能的影响

为了检验算法的并行运行效率，利用加速比（SpeedUp）这一概念进行对比实验 [12]。算法加速比用于测试多节点并行运算与单机运算的效率比，本质上是通过运行时间的比较来实现的。

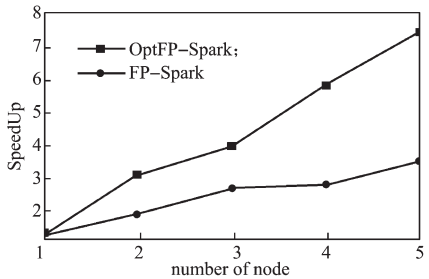
$$SpeedUp = \frac{T}{T_n}。\qquad (2)$$

式中：T 为单机运行时间；T_n 为 n 个节点并行计算的时间。

算法保持支持度为 0.8，D1 和 D2 数据集的两个算法的对比结果如图 11 所示。



a）D1



b）D2

图 11　加速比对算法性能的影响

Fig. 11　The effect of acceleration ratio on algorithm performance

由图 11 所示的实验结果可以得知，随着节点数的增加，加速比逐步增加，但 OptFP−Spark 算法较 FP−Spark 算法加速比增加的效率更明显，说明进行优化分组策略和项头表结构后的 OptFP−Spark 算法性能更优。

对比实验分别从数据规模、支持度和算法的加速比 3 个方面进行比较，结果表明，优化的 OptFP−Spark 算法具备更好的并行挖掘效果和更高的效率。

6 结语

本文首先介绍了研究背景和 Spark 的系统框架，阐述了基于 Spark 平台的 FP−Growth 算法思想及过程，再对该算法在分组策略和项表头结构上进行优化。最后的实验结果证明，优化的 OptFP−Spark 算法具有更高的并行运算加速比、更好的并行挖掘效果及更高的效率。

参考文献：

[1] RAJARAMAN A，ULLMAN J D. 大数据：互联网大规模数据挖掘与分布式处理 [M]. 北京：人民邮电出版社，2012：10−34.
RAJARAMAN A，ULLMAN J D. Big Data：Internet Large-Scale Data Mining and Distributed Processing[M]. Beijing：Posts and Telecommunications Press，2012：10−34.

[2] 李文栋 . 基于 Spark 的大数据挖掘技术的研究与实现 [D]. 济南：山东大学，2015.
LI Wendong. The Research and Implementation of Mining Large Data Based on Spark[D]. Jinan：Shandong University，2015.

[3] 金宗泽，冯亚丽，纪 博，等 . 大数据分析中的关联挖掘 [J]. 计算机与数字工程，2014，42(10)：1924−1928.
JIN Zongze，FENG Yali，JI Bo，et al. Data Mining Association in the Data Analysis[J]. Computer & Digital Engineering，2014，42(10)：1924−1928.

[4] 黎丹雨，陈怡华 . 一种多层多维的关联规则挖掘算法在推荐系统中的应用 [J]. 计算机与现代化，2019(6)：44−48，54.
LI Danyu，CHEN Yihua. Application of Multi-Layer Multi-Dimensional Association Rule Mining Algorithm in Recommendation System[J]. Computer and Modernization，2019(6)：44−48，54.

[5] 厍向阳，张 玲 . 基于 Hadoop 的 FP−Growth 关联规则并行改进算法 [J]. 计算机应用研究，2018，35(1)：109−112.
SHE Xiangyang，ZHANG Ling. Parallel Improved Algorithm of FP-Growth Association Rules Based on Hadoop[J]. Application Research of Computers，2018，35(1)：109−112.

[6] ADNAN M，ALHAJJ R. A Bounded and Adaptive Memory-Based Approach to Mine Frequent Patterns From Very Large Databases[J]. IEEE Transactions on Systems，Man，and Cybernetics，Part B (Cybernetics)，2011，41(1)：154−172.

[7] 马 强，杨金民 . 基于 MapReduce 的频繁项集并行挖掘算法 [J]. 计算机应用与软件，2015，32(9)：13−16，101.
MA Qiang，YANG Jinmin. A Parallel Frequent Itemsets Mining Algorithm Based on Mapreduce[J]. Computer Applications and Software，2015，32(9)：13−16，101.

[8] ENGLE C，LUPHER A，XIN R，et al. Shark：Fast Data Analysis Using Coarse-Grained Distributed Memory[C]//Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. New York：ACM，2012：689−692.

[9] 韩天鹏，王 峰，王 浩 . 基于 FP−Growth 算法构造批量增量的 FP−Tree[J]. 嘉应学院学报，2017，35(8)：21−25.
HAN Tianpeng，WANG Feng，WANG Hao. Constructing FP-Tree of Batch Increment Based on FP-Growth Algorithm[J]. Journal of Jiaying University，2017，35(8)：21−25.

[10] 吕雪骥，李龙澍 . FP−Growth 算法 MapReduce 化研究 [J]. 计算机技术与发展，2012，22(11)：123−126，130.
LÜ Xueji，LI Longshu. Research on Improved FP−Growth Algorithm with MapReduce[J]. Computer Technology and Development，2012，22(11)：123−126，130.

[11] WONG R C W，FU A W C，WANG K. Data Mining for Inventory Item Selection with Cross-Selling Considerations[J]. Data Mining and Knowledge Discovery，2005，11(1)：81−112.

[12] 石陆魁，张 欣，师胜利 . 基于 Spark 的 FP_Growth 算法的并行与优化 [J]. 计算机工程与应用，2018，54(13)：52−58，110.
SHI Lukui，ZHANG Xin，SHI Shengli. Parallelization and Optimization of FP_Growth Algorithm Based on Spark[J]. Computer Engineering and Applications，2018，54(13)：52−58，110.

（责任编辑：申 剑）