软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(5):2446−2464 [doi: 10.13328/j.cnki.jos.006404]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种基于 Spark 的频繁项集快速挖掘算法*

丁家满[1,2], 李海滨[1,2], 邓 斌[1,2], 贾连印[1,2], 游进国[1,2]

[1](昆明理工大学 信息工程与自动化学院, 云南 昆明 650504)

[2](云南省人工智能重点实验室, 云南 昆明 650504)

通信作者: 贾连印, E-mail: JLianyin@163.com

摘 要: 如何在海量数据集中提高频繁项集的挖掘效率是目前研究的热点. 随着数据量的不断增长, 使用传统算法产生频繁项集的计算代价依然很高. 为此, 提出一种基于 Spark 的频繁项集快速挖掘算法 (fast mining algorithm of frequent itemset based on spark, Fmafibs), 利用位运算速度快的特点, 设计了一种新颖的模式增长策略. 该算法首先采用位串表达项集, 利用位运算来快速生成候选项集; 其次, 针对超长位串计算效率低的问题, 考虑将事务垂直分组处理, 将同一事务不同组之间的频繁项集通过连接获得候选项集, 最后进行聚合筛选得到最终频繁项集. 算法在 Spark 环境下, 以频繁项集挖掘领域基准数据集进行实验验证. 实验结果表明所提方法在保证挖掘结果准确的同时, 有效地提高了挖掘效率.

关键词: 频繁项集; 模式增长; 位串; 位运算; 垂直分组; Spark

中图法分类号: TP311

中文引用格式: 丁家满, 李海滨, 邓斌, 贾连印, 游进国. 一种基于Spark的频繁项集快速挖掘算法. 软件学报, 2023, 34(5): 2446–2464. http://www.jos.org.cn/1000-9825/6404.htm

英文引用格式: Ding JM, Li HB, Deng B, Jia LY, You JG. Fast Mining Algorithm of Frequent Itemset Based on Spark. Ruan Jian Xue Bao/Journal of Software, 2023, 34(5): 2446–2464 (in Chinese). http://www.jos.org.cn/1000-9825/6404.htm

Fast Mining Algorithm of Frequent Itemset Based on Spark

DING Jia-Man[1,2], LI Hai-Bin[1,2], DENG Bin[1,2], JIA Lian-Yin[1,2], YOU Jin-Guo[1,2]

[1](Faculty of Information Engineering and Automation, Kunming University of Science and Technology, Kunming 650504, China)

[2](Yunnan Key Laboratory of Artificial Intelligence, Kunming 650504, China)

Abstract: Improving the efficiency of frequent itemset mining in big data is a hot research topic at present. With the continuous growth of data volume, the computing costs of traditional frequent itemset generation algorithms remain high. Therefore, this study proposes a fast mining algorithm of frequent itemset based on Spark (Fmafibs in short). Taking advantage of bit-wise operation, a novel pattern growth strategy is designed. Firstly, the algorithm converts itemset into BitString and exploits bit-wise operation to generate candidate itemset. Secondly, to improve the processing efficiency of long BitString, a vertical grouping strategy is designed and the candidate itemset are obtained by joining the frequent itemset between different groups of same transaction, and then aggregating and filtering them to get the final frequent itemset. Fmafibs is implemented in Spark environment. The experimental results on benchmark datasets show that the proposed method is correct and it can significantly improve the mining efficiency.

Key words: frequent itemset; pattern growth; BitString; bit-wise operation; vertical grouping; Spark

随着大数据时代的到来, 各行各业产生的数据量呈爆炸式增长, 人们迫切希望可以快速获取到有价值的信息[1,2]. 关联规则挖掘作为一种常用的数据挖掘方法, 可以从大量数据中挖掘出未知的但隐藏内在联系的知识和规则, 从而帮助人们更好的决策. 关联规则最早由 Agrawal 等人[3]提出, 以购物篮项目为背景, 通过分析顾客的购买

* 基金项目: 国家自然科学基金 (61562054)
收稿时间: 2020-08-17; 修改时间: 2020-12-13, 2021-06-03; 采用时间: 2021-06-28; jos 在线出版时间: 2022-07-07
CNKI 网络首发时间: 2022-11-15

行为, 挖掘出被人们经常组合购买的商品, 借此指导管理者制定更优的计划管理和销售商品. 正是由于关联规则分析对人们的决策具有良好的指导意义, 近年来, 其被广泛应用于犯罪预测、网络安全、流量统计分析和生物基因信息分析等领域[4]. 文献 [5] 将关联规则应用于动车组运维数据挖掘, 通过对检修数据产生的强关联规则进行分析可以有效提高动车组运维效率. 文献 [6] 基于大量开源代码, 通过挖掘函数调用中错误参数之间的强关联规则, 可以更加精准地发现软件系统中存在的参数错误的缺陷. 关联规则挖掘的核心是频繁项集的获取, 这也是最关键和耗时的过程, 所以多数学者将研究重点放在如何提高频繁项集的挖掘效率上.

Apriori[7]、FP-Growth[8]和 Eclat[9]是最经典的 3 种挖掘频繁项集的算法. 基于这 3 种算法的思想, 可将传统的频繁项集挖掘算法分为 3 类: 基于产生-测试的挖掘算法、基于模式增长的挖掘算法和基于垂直网格的挖掘算法. 其中前两类算法将数据表示为<事务号, 项> 的水平格式, 后一类算法则采用垂直数据格式<项, 事务号>. 基于产生-测试的挖掘算法主要面临候选项集生成速度过慢以及频繁扫描数据库的问题. 基于模式增长的算法通常采用 FP-tree 存储数据, 然后通过构造项集的条件模式树递归挖掘, 但当项集数量较多时, 递归挖掘容易造成内存溢出. 基于垂直网格的算法主要通过对两个项集求交集的方式获取候选项集的支持度计数, 但当数据量增大时, 项集间求交集的计算效率变低. 针对上述问题, 许多改进的算法被提出. 文献 [10] 将图计算技术和 Apriori 算法相结合, 提出了 ANG 算法, 当 k 较小时, 利用 Apriori 算法产生频繁 k 项集, 反之则使用图计算方法. 通过使用图中的顶点和边分别表示事务和事务间的关联, 可以快速得到项集的支持度计数. 文献 [11] 改进了 FP-tree 的节点结构, 新增加 *node-id* 属性, 有效避免使用条件模式树挖掘频繁项集. 文献 [12] 采用 HCFP-tree 来挖掘频繁项集, HCFP-tree 相较于 FP-tree 增加了节点的前缀共享路径, 进一步削减树中节点的数量, 继而减少递归过程的调用次数, 生成更少的条件模式树, 但同时每个节点所占内存空间相对变大. 为了提高项集间求交集的效率, 文献 [13] 提出了一种基于最小哈希的 HashEclat 算法, 其使用 MinHash 来快速估计交集的大小, 该方法虽然在执行时间上有提高, 但结果却存在一定的误差.

随着数据量的不断增长, 传统单机版算法产生频繁项集的效率愈来愈低下. 频繁项集挖掘由于其迭代计算的特点, 加之众多分布式计算框架的出现, 为并行挖掘提供了可能. Spark[14]作为目前主流的分布式计算框架, 在大数据处理方面展现出了一定的优势. 由于其基于内存计算的特性, 非常适于迭代式计算[15], 为此本文提出一种基于 Spark 的频繁项集快速挖掘算法. 首先设计了一种新颖的模式增长策略, 采用位串表达项集, 利用位运算一次性生成候选项集. 其次, 针对超长位串计算效率低的问题, 采用事务垂直分组方式处理, 将同一事务不同组之间的频繁项集通过连接获得组间候选项集, 最后进行聚合筛选得到最终频繁项集. 通过位串表达项集, 可以充分压缩事务集, 而位运算可以减少频繁项集的挖掘时间. 算法在 Spark 环境下, 以频繁项集挖掘领域的基准数据集进行了实验验证. 实验结果表明本文方法在保证挖掘结果准确的同时, 有效地提高了挖掘效率.

本文的主要贡献如下.

(1) 针对候选项集生成速度过慢以及重复扫描数据库统计项集支持度计数的问题, 利用位运算速度快的特点, 设计了一种新颖的模式增长策略, 采用位串表达项集, 通过相关位运算一次性生成候选项集, 项集的计数无需通过扫描数据库来获得, 而是等候选项集生成后对所有项集进行聚合得到其计数值.

(2) 针对项集映射为位串可能出现超长位串的问题, 提出事务垂直分组的策略, 对属于同一条事务不同组包含的位串利用位运算首先生成组内频繁项集, 不同组之间通过笛卡尔积产生组间频繁项集, 最后合并得到全部频繁项集.

(3) 结合 Spark 计算框架, 提出了一种频繁项集的快速挖掘算法——Fmafibs 算法;

(4) 利用频繁项集挖掘领域的基准数据集, 在 Spark 计算环境下进行了详细的实验验证. 验证了本文所提出算法的有效性、运算速率及可扩展性.

1 相关工作

为提高频繁项集的挖掘效率, 一些学者提出用二进制位运算来改善候选项集的生成速率和支持度计数的获取方式. BitTable 是由文献 [16] 提出的一种用来压缩数据库的数据结构, 可以水平和垂直表示数据库, 以分别快速生

成候选项集和统计其相应的支持度计数. BitTableFI[16]是最早提出的基于 BitTable 的算法, 尽管使用了高效的位运算, 但迭代产生-测试的计算代价使得 BitTableFI 的时间和空间复杂度仍较高. 为此, 文献 [17] 提出了 Index-BitTableFI 算法. Index-BitTableFI 仍然使用 BitTable 作为数据存储结构, 同时利用索引数组 index array 和相应的计算方法来降低搜索空间. 通过遍历代表项的索引数组, 使用广度优先搜索策略来快速获得跟代表项共同出现的频繁项集, 对于通过索引数组产生的项集, 再次使用深度优先搜索策略得到全部频繁项集. 文献 [18] 将矩阵算法中项集的计数方法引入 Index-BitTableFI 算法, 在提高候选项集计数效率的同时, 也通过理论证明改进的算法可以直接确定某些项集的支持度而无需单独计数, 显著减少了候选项集的数量, 此外, 该算法使用广度优先搜索策略代替递归方法挖掘所有频繁项集. 上述 3 种算法重点从如何减少项集的搜索空间以及提高项集支持度计算效率两方面入手展开研究, 其中 Index-BitTableFI 算法及其改进算法不采用产生-测试的方式生成频繁项集, 挖掘效率较基于 Apriori 的算法有了显著提高, 但当频繁 1 项集的数量较多时, 需多次遍历 BitTable 求交集, 并且在稀疏数据集中, 大量项集无法直接通过 index array 生成. 文献 [19] 同样用二进制数组存储项的事务索引, 采用深度优先搜索策略递归生成包含频繁 k 项集的频繁 k+1 项集, 在生成频繁 k 项集的过程中同时保存其事务索引, 加快了项集之间求交集的操作, 并利用位运算优化了 BitTableFI 中的项集支持度计数方法, 实验证明所提出的 DF-FIMBII 算法优于 BitTableFI 及 Index-BitTableFI 算法. 考虑到定长 BitTable 中包含大量稀疏向量, 文献 [20] 使用动态位向量 *DBV* 表示数据, 通过将位向量前后两端的 0 移除来降低内存消耗, 同时为了加快项集计数, 使用 lookup table 存储两个 *DBV* 求交集后向量中 1 的个数, 根据 lookup table 的索引可以直接获取该值. 文献 [21] 使用带标签的有向图代替树结构存储事务, 图中各边的标签利用二进制字符串表示, 不同事务则根据标签的“按位与”来抽取, 高度压缩的节点结构使得图中节点个数远小于 FP-tree 中的节点个数, 提高了遍历效率. 文献 [22] 提出了一种基于矩阵分解的频繁项集挖掘算法, 使用布尔矩阵存储整个数据库, 然后对其进行初等变换, 在满足一定条件的情况下将其分解为若干个子矩阵, 并重复上述过程, 直至矩阵不能被分解, 该算法只需要扫描一遍数据库, 运行速度较快.

随着数据量的不断增长, 近年来大量基于分布式计算平台的频繁项集挖掘算法被提出. 文献 [23] 将 Apriori 算法直接在 Spark 计算框架上实现, 提出了 YAFIM 算法, 为后续 Apriori 算法的并行化研究奠定了基础, 该算法只是利用 Spark 加速了计算性能, 并未对 Apriori 算法做任何改进. 文献 [24] 在 Spark 平台上实现了 R-Apriori 算法, 该算法在生成候选 2 项集时使用 Bloom Filter 而非 HashTree 来存储, 提高了生成频繁项集的准确性, 但是其只是减少了第 2 次迭代过程的运行时间, 其他迭代过程的花费时间跟 YAFIM 相同. 文献 [25] 利用 Spark 的分区机制, 首先将原始数据集划分到不同的分区, 根据分区支持度阈值生成每个分区的频繁项集, 然后汇总各分区频繁项集, 扫描原始数据库获得全部频繁项集, 相较于 YAFIM 算法, 其所提出的 SARSO 算法有效减少了 shuffle 过程的数据量. 文献 [26] 在 R-Apriori 算法的基础上提出了 Adaptive-Miner 算法, 该算法在进行第 i 次迭代之前, 先根据第 $i-1$ 次迭代产生的频繁项集的数量 f 和每条事务包含的项的平均数量 g 计算第 i 次迭代可能花费的时间, 如果 f 比较大或者 g 比较小, 则使用 Bloom Filter 来生成频繁 k 项集, 否则继续按照传统 Apriori 算法产生频繁 k 项集, 相较于 R-Apriori 算法, Adaptive-Miner 算法可以减少每次迭代的时间, 但是每次迭代之前的选择策略计算会消耗额外的时间. 文献 [27] 和 Spark 的 mllib 包中都实现了分布式的 FP-Growth 算法[28], 进一步提高了 FP-Growth 算法的执行效率. 文献 [29] 结合 Apriori 和 Eclat 的思想提出了 HFIM 算法, 该算法也是由 k 项集迭代生成 $k+1$ 项集, 不同于采用 HashTree 的计数方式, 对于一个候选 k 项集, 首先获取其所有 $k-1$ 项集, 然后根据垂直格式的数据检索所有 $k-1$ 项集包含的所有共同事务 *ID* 的数量当做该 k 项集的计数. 文献 [30] 同样将水平格式数据转变为垂直格式的稀疏布尔矩阵, 提出了基于 Spark 的 FISM 算法, 当生成 k 项集时, 只要 $k-1$ 项集的布尔向量与 1 项集的布尔向量做一次“按位与”操作即可, 结果向量中 1 的个数即是 k 项集的支持度计数. 上述两种算法利用垂直数据格式避免重复扫描原始数据集, 较 Apriori 算法可以获得较大的性能提升. 目前最新的基于 Spark 的 Apriori 改进算法是由文献 [31] 提出的 EAFIM, 该算法在产生频繁 k 项集时, 不是由 $k-1$ 项集连接生成, 而是由每条事务先生成候选 k 项集, 然后使用 Spark 中的聚合算子统计得到计数值, 最终计算得到满足最小支持度计数的频繁 k 项集, 如果频繁 k 项集的数量小于频繁 $k-1$ 项集的数量, 则会更新原始数据库用于第 k+1 次迭代过程, 该算法已经证明优于 YAFIM 和 R-Apriori, 但是频繁地更新数据库会降低运行效率, 即使数据库变得越来越小. Spark 由于其基于内

存计算的特点, 有效避免了不必要的 I/O 开销, 其对于海量数据的处理显得非常快速高效, 所以本文将位运算与 Spark 进行结合, 设计了基于 Spark 的频繁项集挖掘算法 Fmafibs.

2 问题描述

挖掘频繁项集就是找出事务型数据库中所有出现频率不小于用户给定支持度阈值的项集. 相关定义如下.

给定 $I=\{i_1, i_2, \ldots, i_n\}$是由 n 个互不相同且相互独立的元素构成的集合, 集合 I 称为全集.

定义 1. 项. I 中的任一元素 i_j $(1\leqslant j\leqslant n)$ 称为一个数据项, 简称项, 项不可再分, 具有原子性.

定义 2. 项集. I 的任一非空子集 X 称为一个数据项集合, 简称项集. 包含 k 个不同项的项集称为 k 项集.

定义 3. 事务. 事务是形如 $T=(TID, X)$ 的二元关系集合, 其中 TID 是事务的唯一标识, X 是一个项集, 且满足 $X\subseteq I$ 且 $X\neq\varnothing$.

定义 4. 事务型数据库. $D=\{T_1, T_2, \ldots, T_m\}$是由 m 条事务组成的事务型数据库, 其中 $T_1.X\cup T_2.X\cup \ldots \cup T_m.X=I$.

定义 5. 项集支持度. 给定事务 T 和项集 Y, 如果 T 包含 Y, 当且仅当 $Y\subseteq T.X$.Y 的支持度为包含该项集的事务数量 $Y.count$ 占事务总量$|D|$的百分比, 记作 $support(Y)$, 则:

$$support(Y)=\frac{Y.count}{|D|} \tag{1}$$

定义 6. 频繁项集. 给定支持度阈值 $min\_sup\in(0, 1]$, X 为频繁项集, 当且仅当 $support(X)\geqslant min\_sup$, 否则 X 为非频繁项集. 包含 k 个不同项的频繁项集称为频繁 k 项集.

例如, 表 1 所示的事务型数据库由 10 条事务构成, 全集 $I=\{A, B, C, D, E, F, G, H\}$. 给定 3 个项集: AC, ABC, $ACDF$, 包含 AC 的事务为 (1, 2, 6, 8, 9, 10), 包含 ABC 的事务为 (1, 9, 10), 包含 $ACDF$ 的事务为 (2, 6), 则 $support(AC)=0.6$, $support(ABC)=0.3$, $support(ACDF)=0.2$. 若 $min\_sup=0.5$, 那么项集 AC 为频繁 2 项集, 项集 ABC 和 $ACDF$ 为非频繁项集.

表 1 样例数据库

TID	Item	TID	Item	TID	Item	TID	Item	TID	Item
1	*A B C E*	3	*D F G*	5	*B C D H*	7	*A D E F*	9	*A B C D*
2	*A C D E F*	4	*B D E*	6	*A C D F G*	8	*A C D E G*	10	*A B C F H*

频繁项集满足以下性质[3].

性质 1. 频繁项集的任一非空子集都是频繁项集.

性质 2. 非频繁项集的任一超集都是非频繁项集.

3 Fmafibs 算法

3.1 整体框架

Fmafibs 算法的整体示意图如图 1 所示. 算法主要由以下步骤构成.

(1) 生成频繁 1 项集. 抽取出数据库中所有不同的项, 根据最小支持度计数剪枝非频繁的 1 项集, 得到频繁 1 项集, 也即全集.

(2) 位串表达. 根据得到的频繁 1 项集将事务映射为位串.

(3) 模式增长. 利用位运算方法产生给定位串的所有子集.

(4) 位串分组. 借鉴分治的思想, 对超长位串进行分组, 每组同样用位运算方法生成候选项集.

(5) 位串连接. 对已经分组的位串, 不同组之间可能还存在频繁项集, 通过位串连接生成组间频繁项集.

(6) 挖掘所有频繁项集. 对所有位串进行聚合并解析, 得到全部频繁项集.

传统的产生频繁 k+1 项集的方法通常由两阶段构成.

第 1 阶段. 由两个前 $k-1$ 个项相同的频繁 k 项集相连接或者单个频繁 k 项集与其不包含的频繁 1 项集相连

接产生候选 $k+1$ 项集.

第 2 阶段. 扫描数据库统计候选 $k+1$ 项集的支持度计数, 筛选得到频繁 $k+1$ 项集.

上述两阶段过程通常会耗费大量时间. 许多改进的算法在候选 k+1 项集的生成过程中就获取其支持度计数, 但却需要增加额外的数据结构来存储信息, 导致空间复杂度提高. 综合上述问题, 本文所提算法采用位串表达一个特定的项集, 同时利用相关位运算生成候选项集, 无需重复扫描数据库对候选项集计数, 有效提高了挖掘效率.

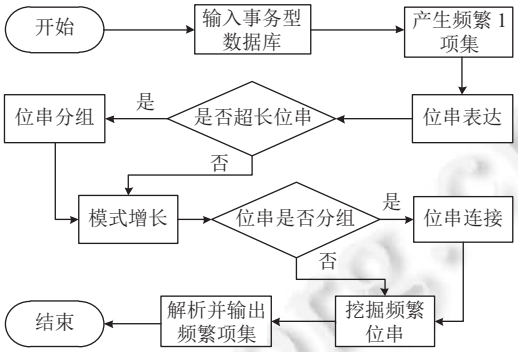

图 1 Fmafibs 算法示意图

3.2 位串表达

定义 7. 项集-位串关系. 对任一项集 $X\subseteq I$, 定义 $X\times I\rightarrow b_1b_2\ldots b_{n-1}b_n$ 为项集-位串关系. $b_1b_2\ldots b_{n-1}b_n$ 为项集 X 对应的位串, 记为 B_x, 其中 $b_j\in\{0, 1\}$, 当 $i_j\in X$ 时, b_j=1, 否则 b_j=0, 其中 j=1, 2, 3,…, n. 同理, 事务-位串关系记为 B_t.

例如, 事务 4 为{B, D, E}, 则 B_t=01011000; 事务 8 包含的某个项集 X 为{A, E}, 则 B_x=10001000.

定义 8. 位串-项集关系. 给定 n 位位串, 定义 $n\times I\rightarrow i_1i_2\ldots i_{n-1}i_n$ 为位串-项集关系. $i_1i_2\ldots i_{n-1}i_n$ 为位串对应的项集, 记为 X_b, 当 b_j=1 时, $i_j\in X_b$, 否则 $i_j\notin X_b$, 其中 j=1, 2, …, n.

例如, 给定位串为 10101100, 则 X_b={A, C, E, F}. 图 2 展示了项集与位串的相互映射过程.

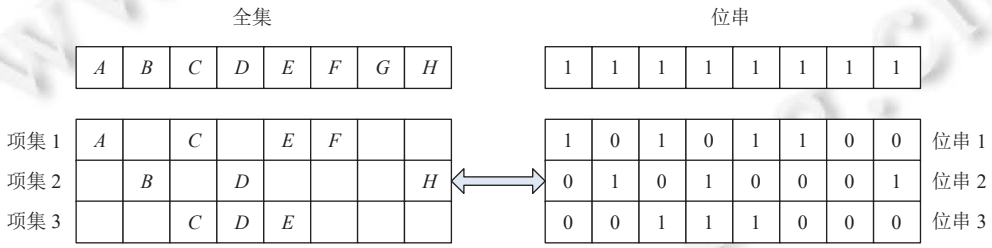

图 2 项集-位串关系映射

3.3 模式增长

如果一个项集不为空, 则其一定包含确定个数的非空子集. 根据定义 7 和定义 8 可知, 当全集确定时, 全集的任一非空子集都可以映射为位串, 并且非空子集对应的位串是其超集对应位串的子串.

定义 9. 位串非空子集. 给定一个位串 S, 则 S 的非空子集为所有元素 s 组成的, 其中元素 s 满足位不全为 0 且 s 中为 1 的位在 S 中对应的位必为 1.

例如, 位串 01100010, 则其子集为{00000010, 00100000, 00100010, 01000000, 01000010, 01100000, 01100010}, 对应的项集为{G, C, CG, B, BG, BC, BCG}.

根据上述定义, 我们使用位串的模式增长来代替由频繁 k 项集产生候选 k+1 项集的迭代过程, 为了快速得到位串的子集, 采用位运算的方法 BitwiseOperate[32], 其基本步骤为.

步骤 1. 将输入的位串与该位串对应整数的相反数的位串进行“按位与”运算;

步骤 2. 将步骤 1 中的结果对应的整数与最开始输入位串对应的整数相减;

步骤 3. 将最开始输入的位串与步骤 2 的结果进行“按位与”运算, 重复步骤 2 和 3, 直到所得位串各个位全为 0.

3.4 位串分组与连接

采用位串模式增长的策略可以加快频繁项集的生成, 但如果一个位串比较密集, 即对应项集中包含的项较多, 直接使用位串模式增长策略, 其整体计算效率会降低. 通过在本文所设实验条件下多次实验发现, 当位串中包含的 1 的个数超过 27 (不同实验条件可能会不同) 时会出现上述问题, 本文将包含 1 的个数超过 27 的位串称为超长位串. 在求解超长位串的子集时, 我们提出了一种基于事务垂直分组的策略. 首先对全集进行分割, 然后将事务按照全集的分割结果进行投影, 从而将一条事务划分为若干条子事务.

定义 10. 全集分割. 给定全集 $I=\{i_1, i_2,\ldots, i_n\}$, 将其划分为 n 个子集合, 每个子集合记为 S_{Ii}, 任意两个子集合满足 $S_{Ii}\cap S_{Ij}=\varnothing$, 其中 $i, j=1, 2,\ldots, n$.

例如, 将全集 $I=\{A, B, C, D, E, F, G, H\}$ 分为 3 组, 则 $S_{I1}=\{A, B, C\}$, $S_{I2}=\{D, E, F\}$, $S_{I3}=\{G, H\}$.

定义 11. 事务投影. 给定全集 I 的子集合 $S_I=\{S_{I1}, S_{I2},\ldots, S_{In}\}$, 对任一事务 T, T 中包含的项集 X 的某一部分投影记为 $T.X_i$, 则 $T.X_i=T.X\cap S_{Ii}$, $T.X$ 的全部投影为集合 $\{T.X_1, T.X_2,\ldots, T.X_n\}$, 其中 $i=1, 2,\ldots, n$.

例如, $T.X=\{B, C, D, H\}$, 则 $T.X_1=T.X\cap S_{I1}=\{B, C\}$, $T.X_2=T.X\cap S_{I2}=\{D\}$, $T.X_3=T.X\cap S_{I3}=\{H\}$.

得到全集 I 的每个子集合 S_{Ii} 后, 接下来我们将对属于同一事务的各子事务根据其对应的 S_{Ii} 分别转换为位串再进行位运算, 计算每条子事务对应的候选位串集合. 对属于同一 S_{Ii} 的候选位串进行计数, 过滤掉计数值不满足最小支持度计数的位串, 得到部分频繁位串. 以表 1 中的数据为例, 假设分为两组, 分组过程如图 3 所示.

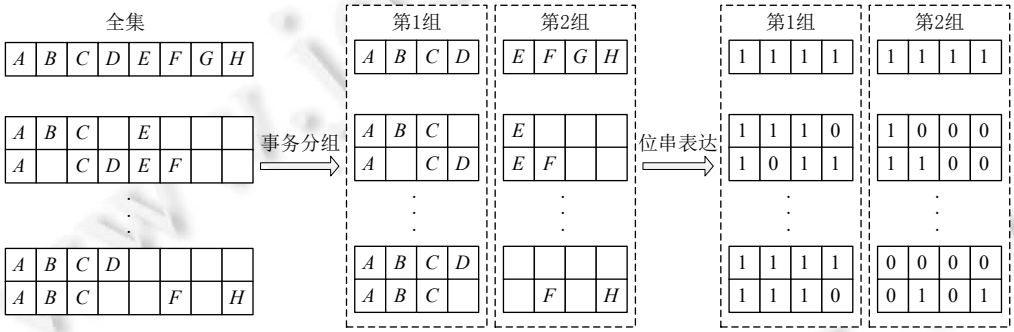

图 3 事务分组过程

得到组内频繁位串后, 需要将同一事务不同组产生的频繁位串进行连接, 得到组间的频繁位串. 以表 1 中的事务 7 为例, 组间位串连接及解析的过程如图 4 所示.

定理 1. 对同一位串, 使用模式增长策略产生的位串子集和先将位串进行分组再进行连接产生的位串子集是等价的.

证明: 不失一般性, 假设将某一位串分为 2 组, 多个组的情形可将 2 个组连接结果再和第 3 组连接依次推出.

给定位串 S, 其对应的位串子集记为 S_{sub}, 将 S 按全集的分割结果进行投影, 结果记为 S_1 和 S_2, 其对应的位串子集分别为 $S_{1\text{-}sub}$ 和 $S_{2\text{-}sub}$. 从以下两个方面进行证明:

(1) 证明 $S_{1\text{-}sub}\cup S_{2\text{-}sub}\subseteq S_{sub}$

对任意的元素 $e_1\in S_{1\text{-}sub}$ 和 $e_2\in S_{2\text{-}sub}$, 因 e_1 中置 1 的位在 S_1 中必为 1, e_2 中置 1 的位在 S_2 中也必为 1, 故 e_1e_2 中置 1 的位在 $S=S_1\cup S_2$ 中也必为 1, 故 $S_{1\text{-}sub}\cup S_{2\text{-}sub}\subseteq S_{sub}$.

(2) 证明 $S_{sub}\subseteq S_{1\text{-}sub}\cup S_{2\text{-}sub}$

对任意的元素 $e\in S_{sub}$, 将 e 按照全集的分割结果分成 e_1 和 e_2 两部分, 对应的 S 同样分成 S_1 和 S_2, 因 e_1 中置 1 的位必定在 S 的 S_1 部分中为 1, 故 $e_1\in S_{1\text{-}sub}$, 同理得 $e_2\in S_{2\text{-}sub}$, 故 $S_{sub}\subseteq S_{1\text{-}sub}\cup S_{2\text{-}sub}$.

故二者等价, 定理得证.

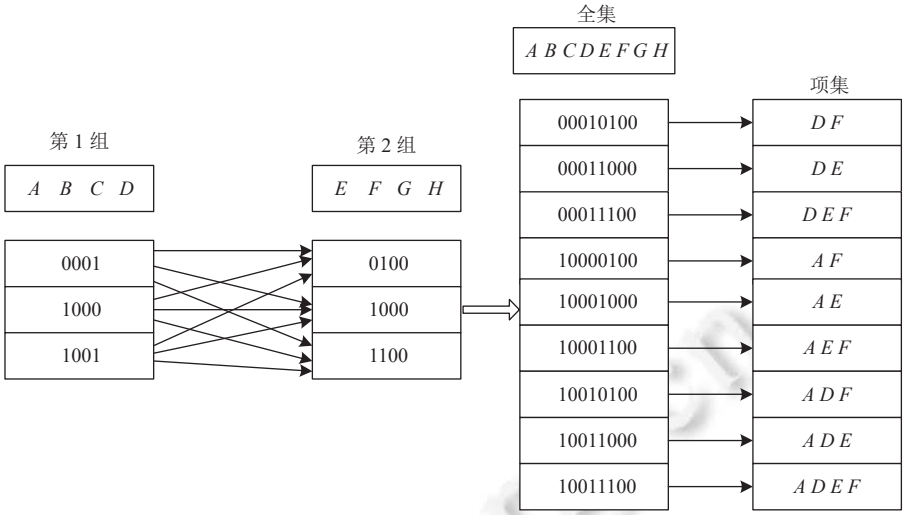

图 4 位串连接及解析过程

3.5 Fmafibs 算法的并行化实现

3.5.1 频繁 1 项集的并行计算

算法的第 1 阶段主要是将存储在 HDFS 上的数据集载入内存以 RDD 的形式存储, 统计数据库中包含的所有不同的项, 并对其进行计数, 过滤掉计数值不满足最小支持度计数的 1 项集, 筛选出频繁 1 项集, 并将频繁 1 项集进行分组. 其在 Spark 中的计算流程图如图 5 所示, 具体计算步骤如下.

步骤 1. 初始化 SparkContext 对象, 使用 textFile 算子读取存储在 HDFS 上的原始数据集, 设置 RDD 分区数, 得到初始 RDD:<*trans*>.

步骤 2. 计算求得总的事务数量, 与用户设置的支持度阈值 *min_sup* 相乘得到最小支持度计数 *min_count*.

步骤 3. 应用 flatMap 算子和 map 算子将 RDD:<*trans*> 转变为 RDD:<*item*, 1> 的形式.

步骤 4. 应用 aggregateByKey 算子对相同的 *item* 进行合并, 然后应用 filter 算子过滤出所有计数值满足最小支持度计数的候选 1 项集, 得到频繁 1 项集 *items*.

步骤 5. 应用数组的 sliding 方法对频繁 1 项集进行分组, 得到 *items_split*.

算法 1. 生成频繁 1 项集并分组.

输入: 事务数据库 *Database*, 用户支持度阈值 *min_sup*;
输出: 分组后的频繁 1 项集 *items_split*.

1. *rdd*0[*trans*]←*Database*
2. *min_count*=*rdd*0.*count* **min_sup*
3. map(*rdd*0[*trans*]) do begin
4. *rdd*1[*trans*]←*rdd*0.*trans*.*split*(“ ”)
5. end map
6. *rdd*2[*item*]←*rdd*1.*trans*.*flatMap*()
7. map(*rdd*2[*item*]) do begin
8. *rdd*3[*item*, 1]←*rdd*2.*item*
9. end map
10. *rdd*4[*item*, *count*]←aggregateByKey(*rdd*3[*item*, *count*])

11. *items*←filter(*rdd*4.*item.count*≥*min_count*)
12. *items_split*←broadcast(sliding(*items*))
13. return *items_split*

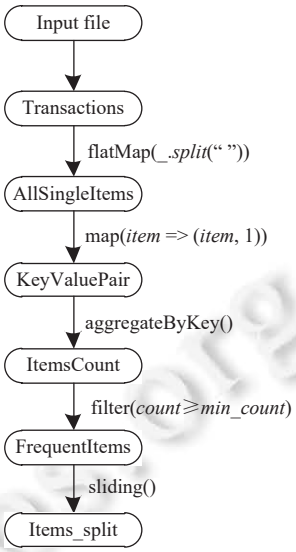

图 5 频繁 1 项集的计算过程

3.5.2 位串模式增长及位串分组并行计算

算法的第 2 阶段首先根据频繁 1 项集将每条事务转换为对应的位串, 然后判断位串是否是超长位串. 如果是超长位串, 则根据频繁 1 项集的分组结果将超长位串进行分组, 然后对不同组的位串应用模式增长的策略产生子位串集合; 如果某一位串不是超长位串, 则直接使用位串模式增长策略产生位串子集. 流程图如图 6 所示, 具体计算步骤如下.

步骤 1. 应用 zipWithIndex 算子对每条事务进行编号.

步骤 2. 应用 map 算子将每条事务根据频繁 1 项集转换为对应的位串.

步骤 3. 应用 map 算子对 RDD 中的超长位串根据分组后的频繁 1 项集进行分组.

步骤 4. 应用 map 算子对非超长位串使用 BitwiseOperate 方法, 产生由位串集合表示的候选项集的集合.

步骤 5. 应用 map 算子对超长位串的不同组中的位串使用 BitwiseOperate 方法, 产生各组内的候选位串.

步骤 6. 应用 aggregateByKey 算子对相同的位串进行聚合, 应用 filter 算子过滤掉所有支持度计数值小于最小支持度计数的位串, 得到各组内的频繁位串.

算法 2. 位串模式增长及分组计算.

输入: 事务 *rdd*0, 频繁 1 项集 *items*, 分组后的频繁 1 项集 *items_split*, 最小支持度计数 *min_count*;
输出: 非超长位串子集 *rdd*12 和超长位串各组的位串子集 *rdd*14.

1. zipWithIndex(*rdd*0[*trans*]) do begin
2. *rdd*5[*transId*, *trans*]←*rdd*0[*trans*]
3. end zipWithIndex
4. map(*rdd*5[*transId*, *trans*]) do begin
5. *array*←*rdd*5.*trans*.intersect(*items*)

6. *BitString*←*array*
7. *rdd*6[*transId*, *LongBitString*]←*BitString*
8. *rdd*7[*transId*, *NlongBitString*]←*BitString*
9. end map
10. map(*rdd*6[*transId*, *LongBitString*]) do begin
11. *rdd*8[*transId*, *groupId*, *BitString_Groupset*]←split *BitString* according to *items_split*
12. end map
13. map(*rdd*8[*transId*, *groupId*, *BitString_Groupset*]) do begin
14. *set*←*BitwiseOperate*(*BitString_Groupset*)
15. *rdd*9[*transId*, *groupId*, *set*]←*set*
16. end map
17. *rdd*10[*transId*, *groupId*, *BitString*]←*rdd*9[*transId*, *groupId*, *set*].*flatMapValues*()
18. map(*rdd*7[*transId*, *NLongBitString*]) do begin
19. *set*←*BitwiseOperate*(*NLongBitString*)
20. *rdd*11[*transId*, *set*]←*set*
21. end map
22. *rdd*12[*transId*, *BitString*]←*rdd*11[*transId*, *set*].*flatMapValues*()
23. *rdd*13[*transId*, *groupId*, *BitString*]←*aggregateByKey*(*rdd*10[*transId*, *groupId*, *BitString*])
24. *rdd*14[*transId*, *groupId*, *BitString*]←*filter*(*rdd*13.*BitString.count*≥*min_count*)
25. return *rdd*12, *rdd*14

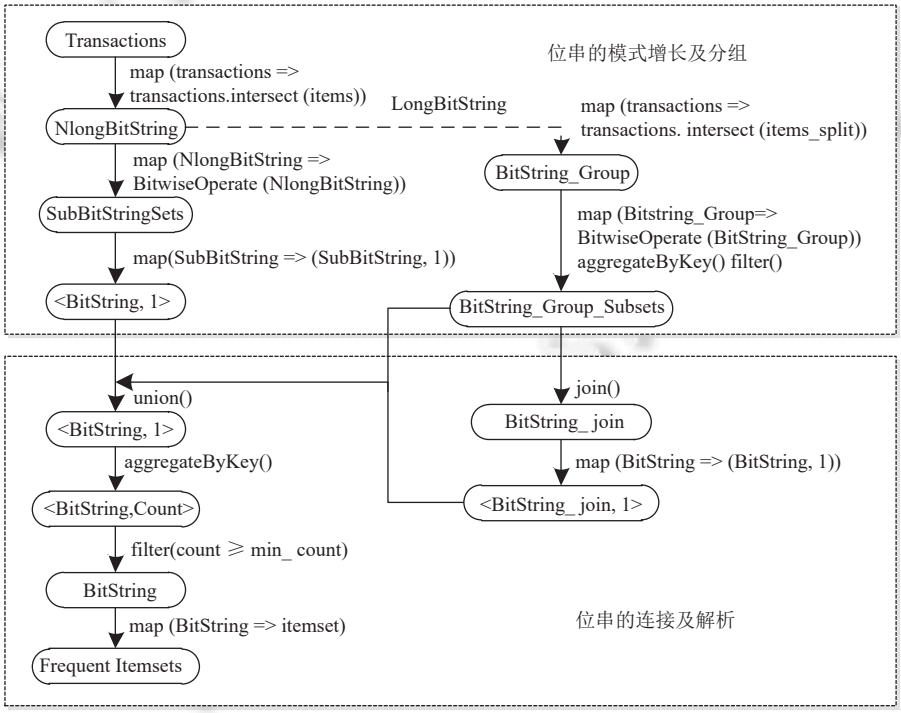

图 6 位串模式增长及分组连接解析过程

3.5.3 位串连接及解析并行计算

算法的第 2 阶段得到了非超长位串的所有子集以及超长位串分组后不同组内产生的位串子集, 本阶段通过对属于同一条事务不同组的位串做连接操作, 生成组间频繁位串, 最后通过聚合、筛选、解析获得全部频繁项集. 流程图如图 6 所示, 计算步骤如下.

步骤 1. 对同一条事务各组内产生的位串应用 join 算子做连接操作, 产生组间的候选位串.

步骤 2. 应用 union 和 aggregateByKey 算子对产生的所有候选位串进行聚合, 应用 filter 算子过滤掉所有支持度计数值小于最小支持度计数的位串, 得到全部的频繁位串.

步骤 3. 将位串根据全集解析为频繁项集.

算法 3. 位串连接及解析算法.

输入: 位串集合 *rdd*12 和 *rdd*14, 最小支持度计数 *min_count*;
输出: 全部频繁项集 *frequent_itemsets*.

1. map(*rdd*14[*transId*, *groupId*, *BitString*]) do begin
2. if *transId* is same and *groupId* is different do begin
3. then *rdd*15[*transId*, *BitString*]←join(*BitString*)
4. end if
5. end map
6. *rdd*16[*transId*, *BitString*]←map(*rdd*14[*transId*, *groupId*, *BitString*])
7. *rdd*17[*transId*, *BitString*]←union(*rdd*12[*transId*, *BitString*], *rdd*15[*transId*, *BitString*], *rdd*16[*transId*, *BitString*])
8. *rdd*18[*BitString*]←aggregateByKey(*rdd*17[*transId*, *BitString*])
9. *rdd*19[*BitString*]←filter(*rdd*18.*BitString.count*≥*min_count*)
10. *frequent_itemsets*←parse *rdd*19[*BitString*].collect()
11. return *frequent_itemsets*

3.6 时间复杂度分析

假设数据库中包含的事务数量为 T, 最长事务包含的项的个数为 S, 整个数据库有 K 个不同的项, Fmafibs 第 1 阶段的时间代价主要是生成频繁 1 项集, 其所花费的时间表示为 $O(T\times S+K)$; Fmafibs 第 2 阶段的时间代价主要为位串的模式增长, 假设 T_{nl} 表示非超长事务的数量, T_1 表示超长事务的数量, 非超长事务中长度最长的事务包含的候选项集的数量为 N_1, 超长事务中最长的子事务包含的候选项集的数量为 N_2, 共分为 M 组, 则第 2 阶段花费的时间表示为 $O(T_{nl}\times N_1+T_1\times N_2\times M)$; Fmafibs 第 3 阶段的时间代价主要是同一事务不同组之间的位串做连接操作, 不同组之间要做 $2^M-(M+1)$ 次连接, 第 3 阶段花费的时间表示为 $O((2^M-M-1)\times T_1\times N_2)$; 所以 Fmafibs 花费的总时间为 $O(T\times S+K+T_{nl}\times N_1+T_1\times N_2\times(2^M-1))$.

3.7 算 例

为了更好地理解本文所提出的算法, 我们将以表 1 所示的数据库为例来进一步阐述. 假设 *min_sup*=0.5, 则 *min_count*=5. 使用 flatMap 算子和 map 算子对每条事务进行格式转换, 例如事务 3 变为{(*D*, 1), (*F*, 1), (*G*, 1)}的形式, 使用 aggregateByKey 算子对单个项进行计数, 得到候选 1 项集: {(*A*, 7), (*B*, 5), (*C*, 7), (*D*, 7), (*E*, 5), (*F*, 5), (*G*, 3), (*H*, 2)}, 使用 filter 算子过滤掉支持度计数值小于 *min_count* 的 1 项集, 得到频繁 1 项集: {(*A*, 7), (*B*, 5), (*C*, 7), (*D*, 7), (*E*, 5), (*F*, 5)}. 使用 sliding 函数对频繁 1 项集进行分组, 这里假定分为 2 组, 第 1 组: {(*A*, 7), (*B*, 5), (*C*, 7)}, 第 2 组: {(*D*, 7), (*E*, 5), (*F*, 5)}. 这里我们假设将所有事务都进行分组, 因为分组的情况也包含不分组的情况, 根据频繁 1 项集的分组情况对每条事务进行分组, 分组结果如表 2 所示.

接下来对每条事务的各组应用 BitwiseOperate 方法, 根据子全集生成候选项集对应的位串, 例如第一组对应的子全集为{(*A*, 7), (*B*, 5), (*C*, 7)}, 则 (2, 1, *A*, *C*) 产生的位串集合为 (2, 1, 001, 100, 101), 结果如表 3 所示.

表 2 事务分组

Group1 (TID, GID, Item)	Group2 (TID, GID, Item)
(1, 1, *A B C*)	(1, 2, *E*)
(2, 1, *A C*)	(2, 2, *D E F*)
(3, 1, null)	(3, 2, *D F*)
(4, 1, *B*)	(4, 2, *D E*)
(5, 1, *B C*)	(5, 2, *D*)
(6, 1, *A C*)	(6, 2, *D F*)
(7, 1, *A*)	(7, 2, *D E F*)
(8, 1, *A C*)	(8, 2, *D E*)
(9, 1, *A B C*)	(9, 2, *D*)
(10, 1, *A B C*)	(10, 2, *F*)

表 3 项集的位串表达

Group1 (TID, GID, BitString)	Group2 (TID, GID, BitString)
(1, 1, 001, 010, 011, 100, 101, 110, 111)	(1, 2, 010)
(2, 1, 001, 100, 101)	(2, 2, 001, 010, 011, 100, 101, 110, 111)
(3, 1, null)	(3, 2, 001, 100, 101)
(4, 1, 010)	(4, 2, 010, 100, 110)
(5, 1, 001, 010, 011)	(5, 2, 100)
(6, 1, 001, 100, 101)	(6, 2, 001, 100, 101)
(7, 1, 100)	(7, 2, 001, 010, 011, 100, 101, 110, 111)
(8, 1, 001, 100, 101)	(8, 2, 010, 100, 110)
(9, 1, 001, 010, 011, 100, 101, 110, 111)	(9, 2, 100)
(10, 1, 001, 010, 011, 100, 101, 110, 111)	(10, 2, 001)

对属于同一组的位串使用 aggregateByKey 算子进行聚合, filter 算子过滤掉每条事务中计数值小于最小支持度计数的位串, 形成的结果如表 4 所示. 最终得到部分频繁位串: 第 1 组: (001, 010, 100, 101), 第 2 组: (001, 010, 100).

得到部分频繁位串后, 使用 join 算子对同一条事务不同组包含的频繁位串进行连接, 生成各组之间的候选位串, 结果如表 5 所示.

表 4 位串表达的频繁项集

Group1 (TID, GID, BitString)	Group2 (TID, GID, BitString)
(1, 1, 001, 010, 100, 101)	(1, 2, 010)
(2, 1, 001, 100, 101)	(2, 2, 001, 010, 100)
(3, 1, null)	(3, 2, 001, 100)
(4, 1, 010)	(4, 2, 010, 100)
(5, 1, 001, 010)	(5, 2, 100)
(6, 1, 001, 100, 101)	(6, 2, 001, 100)
(7, 1, 100)	(7, 2, 001, 010, 100)
(8, 1, 001, 100, 101)	(8, 2, 010, 100)
(9, 1, 001, 010, 100, 101)	(9, 2, 100)
(10, 1, 001, 010, 100, 101)	(10, 2, 001)

表 5 各组连接后的位串

(TID, Joined BitString)
(1, 001010, 010010, 100010, 101010)
(2, 001001, 001010, 001100, 100001, 100010, 100100, 101001, 101010, 101100)
(3, null)
(4, 010010, 010100)
(5, 001100, 010100)
(6, 001001, 001100, 100001, 100100, 101001, 101100)
(7, 100001, 100010, 100100)
(8, 001010, 001100, 100010, 100100, 101010, 101100)
(9, 001100, 010100, 100100, 101100)
(10, 001100, 010100, 100100, 101100)

使用 aggregateByKey 算子进行聚合, filter 算子过滤掉非频繁的位串, 得到的频繁位串为 (001100, 100100).

至此, Fmafibs 算法生成的全部的频繁位串为 (001, 010, 100, 101, 001, 010, 100, 001100, 100100), 转换为对应项集为 (*C*, *B*, *A*, *AC*, *F*, *E*, *D*, *CD*, *AD*).

4 实验结果与分析

4.1 实验条件

Spark 计算集群由 5 个节点构成, 包含 1 个主节点, 4 个计算节点. 每个节点 CPU 核数均为 4, 主节点运行内存为 6 GB, 硬盘容量为 2 TB, 计算节点运行内存为 4 GB, 硬盘容量为 2 TB. 每个节点均安装 CentOS 6.5, Spark 2.4.5, JDK 1.8.0. 实验程序采用 Scala 2.11.5 编写, 数据存储平台为 HDFS (Hadoop 2.6.0). 实验所用数据集均为频繁项集挖掘领域的基准数据集[33], 分别为 Mushroom, Chess, T10I4D100K, Kosarak, PAMP, Chainstore. 表 6 展示了 6 个数据集的基本特性. 对比算法为 Spark mllib 中实现的 FP-Growth 算法[28]和 EAFIM 算法[31].

表 6 不同数据集的基本特征

数据集	不同项的数量	事务的数量	密集程度
Mushroom	119	8124	密集
Chess	75	3196	密集
T10I4D100K	870	100 000	稀疏
Kosarak	41 270	990 002	稀疏
PAMP	141	1 000 000	密集
Chainstore	46 086	1 112 949	稀疏

4.2 有效性分析

为验证 Fmafibs 算法生成结果的正确性, 实验方案设计如下.

方案 1: 验证 Fmafibs 算法在 6 个数据集上产生的频繁 k 项集是否与 FP-Growth 产生的结果相同;

方案 2: 验证 Fmafibs 算法在 6 个数据集上产生的频繁项集的数量是否与 FP-Growth 产生的结果相同.

之所以这样设计, 是因为当支持度阈值设置较低时, 会挖掘出大量的频繁项集, 不易于直观比对. 所以我们先设置较高的支持度阈值来验证方案 1, 当方案 1 的结果得到正确验证后, 再减小支持度阈值, 验证方案 2.

方案 1 的实验结果如表 7–表 12 所示. 支持度阈值相同的情况下, Fmafibs 算法产生的频繁 k 项集与 FP-Growth 算法产生的频繁 k 项集及其支持度计数值相同.

表 7 Mushroom 的频繁项集 (*min_sup*=0.9)

频繁k项集	Fmafibs	FP-Growth
频繁1项集	[34]:7914 [85]:8124 [86]:7924 [90]:7488	[34]:7914 [85]:8124 [86]:7924 [90]:7488
频繁2项集	[34-85]:7914 [34-86]:7906 [85-86]:7924 [85-90]:7488	[34-85]:7914 [34-86]:7906 [85-86]:7924 [85-90]:7488
频繁3项集	[34-85-86]:7906	[34-85-86]:7906

表 8 Chess 的频繁项集 (*min_sup*=0.98)

频繁k项集	Fmafibs	FP-Growth
频繁2项集	[29-52]:3170 [29-58]:3180 [29-60]:3136 [52-60]:3138 [29-40]:3155 [58-60]:3148 [52-58]:3184 [40-52]:3159 [40-58]:3169	[29-52]:3170 [29-58]:3180 [29-60]:3136 [52-60]:3138 [29-40]:3155 [58-60]:3148 [52-58]:3184 [40-52]:3159 [40-58]:3169
频繁3项集	[29-52-58]:3169 [29-58-60]:3135 [52-58-60]:3137 [29-40-52]:3144 [29-40-58]:3154 [40-52-58]:3158	[29-52-58]:3169 [29-58-60]:3135 [52-58-60]:3137 [29-40-52]:3144 [29-40-58]:3154 [40-52-58]:3158
频繁4项集	[29-40-52-58]:3143	[29-40-52-58]:3143

表 9 T10I4D100K 的频繁项集 (*min_sup*=0.009)

频繁k项集	Fmafibs	FP-Growth
频繁2项集	[368-829]:1194 [529-598]:943 [217-346]:1336 [283-346]:910 [39-825]:1187 [368-692]:928 [227-390]:1049 [227-722]:995 [217-283]:926 [471-960]:935 [368-682]:1193 [390-722]:1042 [789-829]:1194 [39-704]:1107 [704-825]:1102	[368-829]:1194 [529-598]:943 [217-346]:1336 [283-346]:910 [39-825]:1187 [368-692]:928 [227-390]:1049 [227-722]:995 [217-283]:926 [471-960]:935 [368-682]:1193 [390-722]:1042 [789-829]:1194 [39-704]:1107 [704-825]:1102
频繁3项集	[227-390-722]:907 [39-704-825]:1035	[227-390-722]:907 [39-704-825]:1035

方案 2 的实验结果如图 7 所示, 其中 X 轴表示支持度阈值, Y 轴表示频繁项集的数量. 从图 7 中可以看出, 支持度阈值相同时, Fmafibs 算法产生的频繁项集的总数与 FP-Growth 算法产生的频繁项集的总数相同.

表 10 Kosarak 的频繁项集 (*min_sup*=0.09)

频繁*k*项集	Fmafibs	FP-Growth
频繁1项集	[3]:450 031 [6]:601 374 [1]:197 522 [11]:364 065	[3]:450 031 [6]:601 374 [1]:197 522 [11]:364 065
频繁2项集	[3-6]:265 180 [3-11]:161 286 [6-11]:324 013 [1-11]:91 882 [1>-6]:132 113	[3-6]:265 180 [3-11]:161 286 [6-11]:324 013 [1-11]:91 882 [1-6]:132 113
频繁3项集	[3-6-11]:143 682	[3-6-11]:143 682

表 11 PAMP 的频繁项集 (*min_sup*=0.88)

频繁*k*项集	Fmafibs	FP-Growth
频繁1项集	[44]:927 546 [53]:925 210 [63]:937 761 [106]:929 660 [131]:936 086 [134]:945 074 [137]:919 729 [140]:932 044	[44]:927 546 [53]:925 210 [63]:937 761 [106]:929 660 [131]:936 086 [134]:945 074 [137]:919 729 [140]:932 044
频繁2项集	[63-134]:892 992 [131-134]:886 788 [63-140]:883 746 [134-140]:892 368 [106-131]:895 453 [106-134]:880 651 [134-137]:880 156	[63-134]:892 992 [131-134]:886 788 [63-140]:883 746 [134-140]:892 368 [106-131]:895 453 [106-134]:880 651 [134-137]:880 156

表 12 Chainstore 的频繁项集 (*min_sup*=0.003)

频繁*k*项集	Fmafibs	FP-Growth
频繁2项集	[16 967-16 977]:7801 [16 967-39 684]:4155 [16 978-16 967]:4211 [16 975-16 967]:4488 [39 388-39 684]:3428 [13 743-16 967]:5528 [13 743-16 977]:3629 [5666-16 967]:3456 [16 881-16 977]:3474 [39 430-39 432]:4441 [39 430-13 743]:3778 [21 738-16 967]:4333 [3510-3482]:4252 [21 283-21 308]:3506 [11 780-11 783]:3788 [39 171-39 688]:3839	[16 967-16 977]:7801 [16 967-39 684]:4155 [16 978-16 967]:4211 [16 975-16 967]:4488 [39 388-39 684]:3428 [13 743-16 967]:5528 [13 743-16 977]:3629 [5666-16 967]:3456 [16 881-16 977]:3474 [39 430-39 432]:4441 [39 430-13 743]:3778 [21 738-16 967]:4333 [3510-3482]:4252 [21 283-21 308]:3506 [11 780-11 783]:3788 [39 171-39 688]:3839

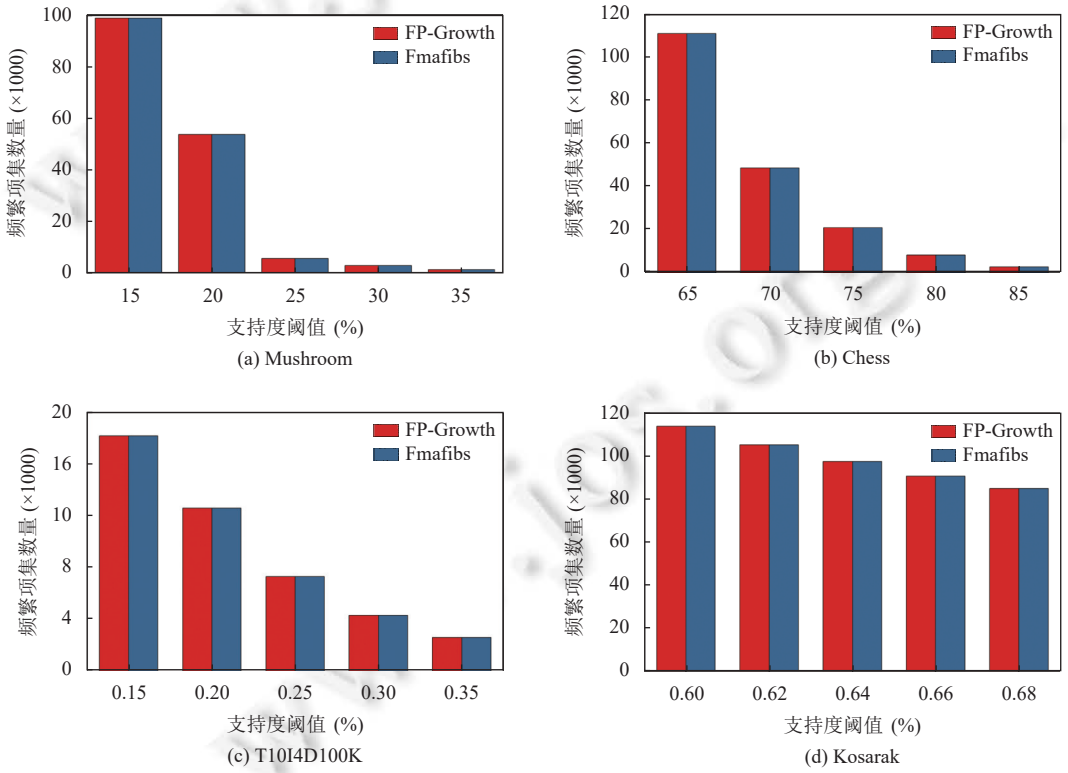

图 7 Fmafibs 与 FP-Growth 生成的频繁项集个数比较

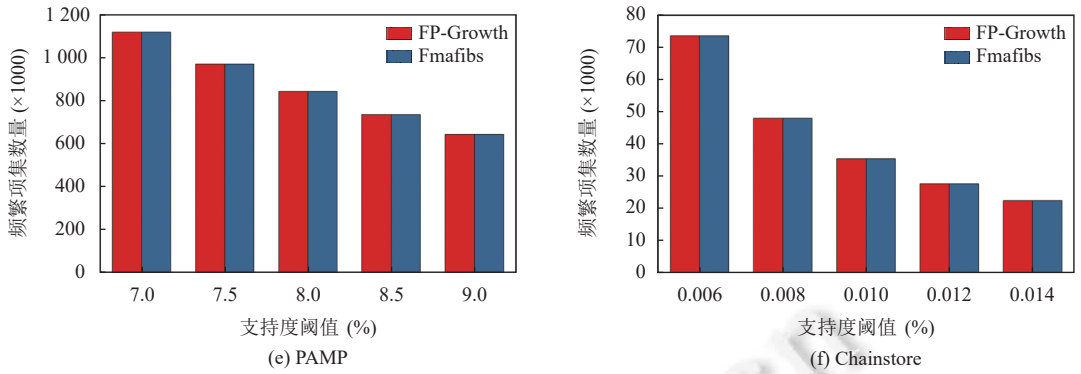

图 7　Fmafibs 与 FP-Growth 生成的频繁项集个数比较 (续)

方案 1 和方案 2 的实验结果证明了 Fmafibs 算法的正确性.

4.3　性能分析

为验证 Fmafibs 算法的执行效率, 采用运行时间作为性能评价指标, 比对 3 种算法在 6 个数据集上的运行时间. 实验结果如图 8 所示, 其中 X 轴表示支持度阈值, Y 轴表示算法的运行时间.

图 8(a) 和图 8(b) 展示的是 3 种算法在数据集 Mushroom 和 Chess 上的运行时间, 在不同的支持度阈值下, Fmafibs 算法的运行时间都少于 FP-Growh 算法和 EAFIM 算法的运行时间. 图 8(a) 中, 当支持度阈值由 10% 提高到 15% 时, Mushroom 数据集中的每条事务映射为位串后可以直接通过位操作函数得到候选项集, Fmafibs 算法所用时间大幅下降. 随着支持度阈值的提高, 每条事务中包含的项的个数逐渐减少, 与之对应的位串中 1 的个数也逐渐减少, 位操作函数的计算效率会进一步提高, 进而减少算法总的运行时间, 这也体现了位运算的速度优势. 当支持度阈值由 15% 提高到 35% 时, 总数据量变小, 每个节点数据处理的速度都很快, 此时 Spark 集群任务调度所花费时间占主要, 所以 Fmafibs 算法的运行时间比较平稳. Chess 数据集较 Mushroom 数据集更加密集, 不同项的个数少但事务冗余程度高, 所以支持度阈值改变时, Fmafibs 算法的整体运行时间无明显差别. 图 8(c) 展示了 3 种算法在数据集 T10I4D100K 上的运行时间, 在 0.1% 到 0.35% 阈值区间内, 频繁 1 项集的数量较多, 将超长位串划分为多组, 不同组之间需要频繁连接, 但 Fmafibs 算法的整体运行效率相对较高, 这也说明事务分组连接策略可以提升算法整体性能. Kosarak 数据集的大小远超前 3 个数据集, 但在同样的支持度阈值下, 从图 8(d) 中可以看出, Fmafibs 算法的性能远超另外两种算法. 在 PAMP 和 Chainstore 两个规模较大的数据集上, 我们设置了相对较小的支持度阈值区间来验证算法的时间性能, 如图 8(e) 和图 8(f) 所示, 在数据规模大而支持度阈值小的情况下, 3 种算法的运行时间都有所增加, 但相同的条件下, Fmafibs 算法仍然优于另外两种算法, 并且 3 种算法都能在可接受的时间范围内得出结果, 也从侧面上反映出 Spark 框架可以加速数据处理过程.

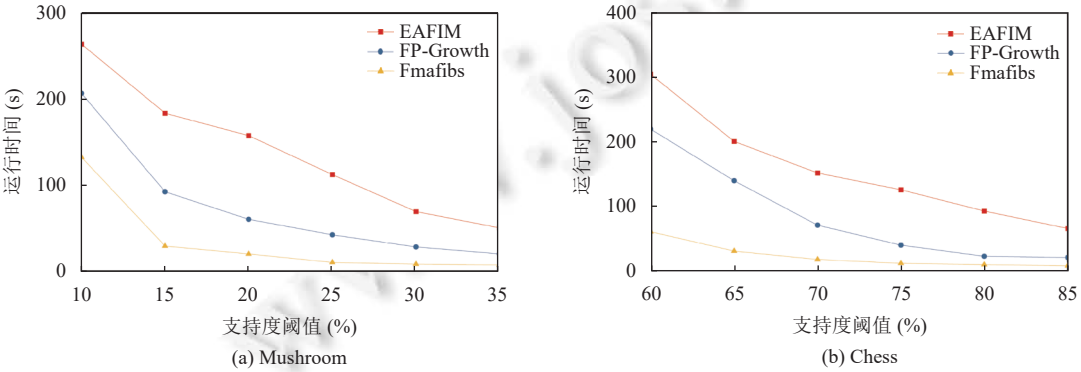

图 8　不同算法在 4 个数据集上的运行时间比较

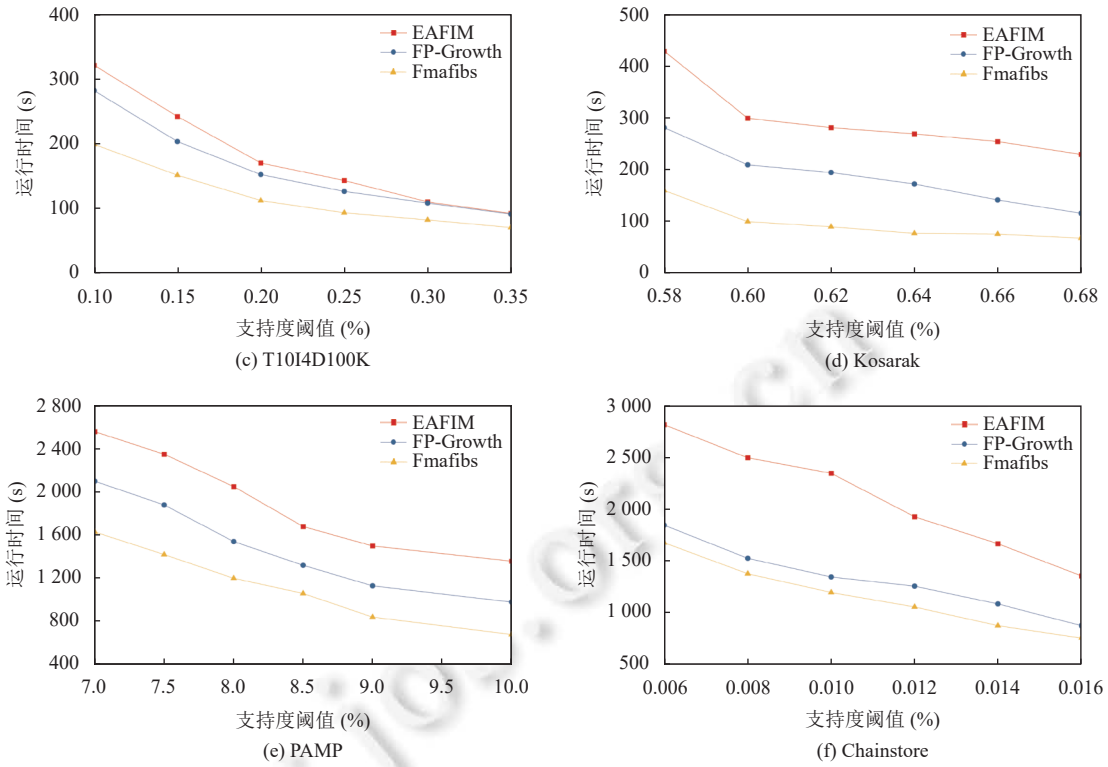

图 8　不同算法在 4 个数据集上的运行时间比较 (续)

4.4　加速比分析

定义 12. 加速比. 加速比[34]是指程序在单个节点上的运行时间与在多个节点上的运行时间的比值, 记作 S_p, 则:

$$S_p = \frac{T_1}{T_n} \tag{2}$$

其中, n 表示节点个数. 加速比是反映程序并行程度的最重要指标之一. 为验证 Fmafibs 算法的加速性能, 固定每个数据集的支持度阈值, 通过增加集群节点的数量来测试. 每个数据集的固定支持度阈值设置为: Mushroom: 35%, Chess: 80%, T10I4D100K: 0.3%, Kosarak: 0.6%, PAMP: 8%, Chainstore: 0.006%. 实验结果如图 9 所示, 其中 X 轴表示节点的个数, Y 轴表示 Fmafibs 算法的加速比.

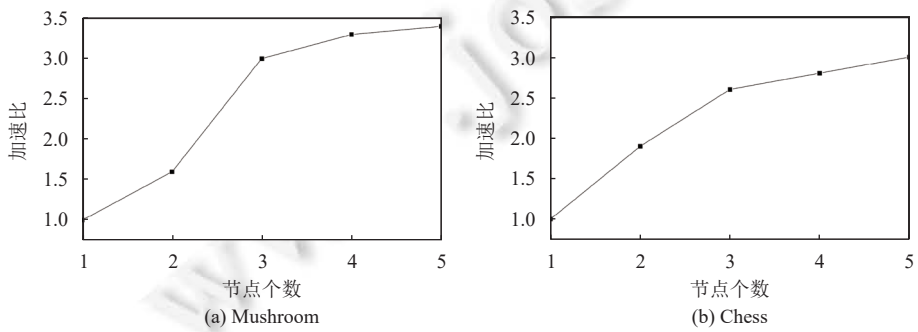

图 9　Fmafibs 算法在不同节点个数下的加速比

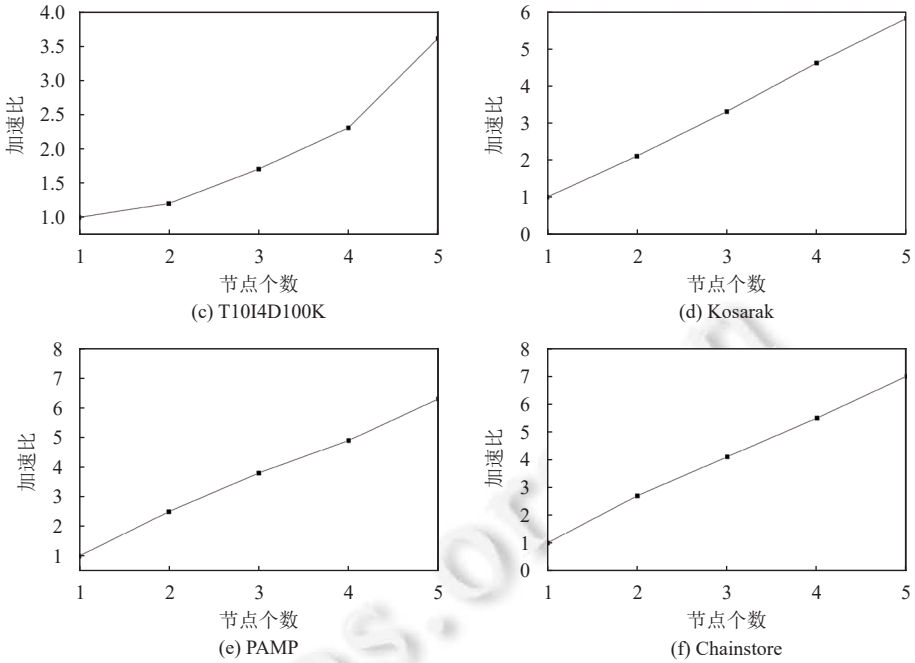

图 9 Fmafibs 算法在不同节点个数下的加速比 (续)

从图 9 中我们可以看出, Fmafibs 算法在 6 个基准数据集上的运行时间随着集群节点的增加而减少, 表明本文提出的算法具有良好的加速性能. 在 Mushroom 和 Chess 数据集上, 当节点由 3 个增加到 5 个时, 运行时间趋于平缓, 这是因为 Mushroom 和 Chess 数据集的总数据量相对较少, 在给定支持度阈值的情况下, 总数据量进一步降低, 集群的计算性能未能充分发挥, 继续增加集群节点数量对算法的整体运行时间影响不大. 图 9(c)–图 9(f) 表明在 T10I4D100K, Kosarak, PAMP, Chainstore 这 4 个数据集上 Fmafibs 算法的运行时间随集群节点个数的增加呈线性下降, 由于这 4 个数据集包含的事务数量较多, 集群节点的分布式处理能力得到充分利用, 通过增加集群节点, 减少每个节点的计算量, 可以有效缩短算法的运行时间, 尤其是在后 3 个数据集上加速性能尤为明显.

4.5 可扩展性分析

为验证算法的数据可扩展性, 通过记录 3 种算法在不同规模数据集上的运行时间来观察对比其性能. 各数据集支持度阈值设置如下: Mushroom: 35%, Chess: 80%, T10I4D100K: 0.3%, Kosarak: 0.6%, PAMP: 20%, Chainstore: 0.1%. 实验结果如图 10 所示, 其中 X 轴表示数据集的复制次数, Y 轴表示算法的运行时间.

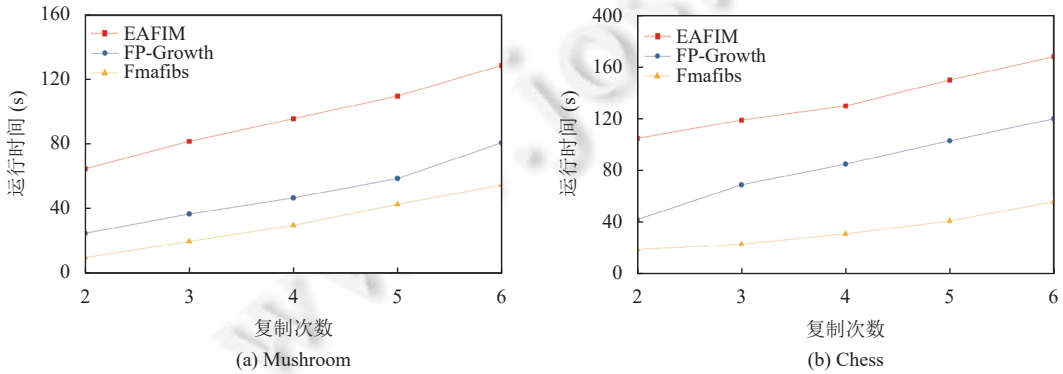

图 10 Fmafibs 算法的扩展性分析

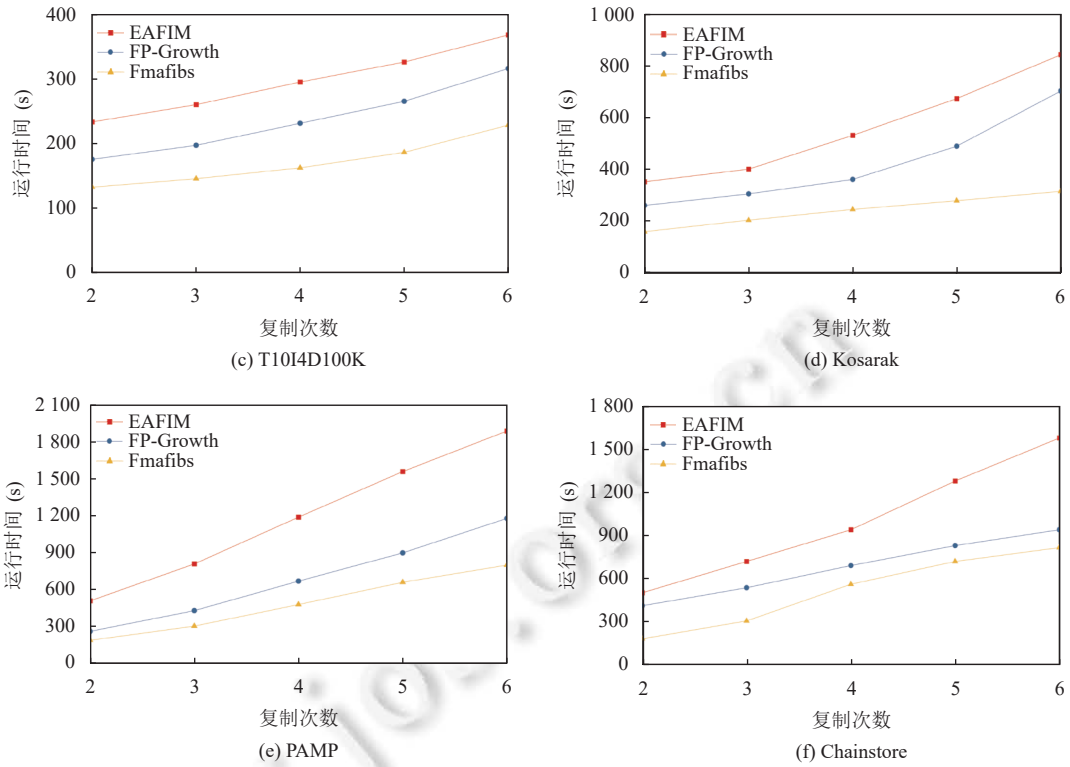

图 10 Fmafibs 算法的扩展性分析 (续)

从图 10 中可以看出, 随着数据规模的不断增加, 3 种算法的运行时间随之增加, 但 Fmafibs 算法仍然显示了良好的性能, 并且能够在可接受时间范围内对大规模的数据集进行快速处理, 说明 Fmafibs 算法具有良好的数据可扩展性.

5 总 结

为了提高海量数据中频繁项集的挖掘效率, 本文提出一种基于 Spark 的频繁项集快速挖掘算法 Fmafibs. Fmafibs 算法采用位串表达项集, 利用位运算快速生成候选频繁项集. 其次, 通过对事务进行垂直分组可以有效避免超长位串计算效率低的问题, 对分组后的位串采用连接策略生成同一事务不同组之间的候选频繁项集. 最后, 将所有项集进行聚合过滤得到最终频繁项集. 通过位串表达项集, 可以充分压缩事务集, 而位运算可以减少频繁项集的挖掘时间. 实验结果表明本文方法在保证挖掘结果准确的同时, 有效地提高了挖掘效率. 下一步我们将会重点研究事务分组之后是否会发生数据倾斜以及如何合理设置 Spark 分区数来加快执行速率, 进一步提高算法性能.

References:

[1] Nguyen TL, Vo B, Snasel V. Efficient algorithms for mining colossal patterns in high dimensional databases. Knowledge-based Systems, 2017, 122: 75–89. [doi: 10.1016/j.knosys.2017.01.034]

[2] Yu ZQ, Yu XH, Dong JW, Wang L. Distributed mining of frequent co-occurrence patterns across multiple data streams. Ruan Jian Xue Bao/Journal of Software, 2019, 30(4): 1078−1093 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5419.htm [doi: 10.13328/j.cnki.jos.005419]

[3] Agrawal R, Imielinski T, Swami A. Mining association rules between sets of items in large databases. In: Proc. of the 1993 ACM SIGMOD Int'l Conf. on Management of Data. Washington: ACM, 1993. 207–216. [doi: 10.1145/170035.170072]

[4] Chee CH, Jaafar J, Aziz IA, Hasan MH, Yeoh W. Algorithms for frequent itemset mining: A literature review. Artificial Intelligence Review, 2019, 52(4): 2603–2621. [doi: 10.1007/s10462-018-9629-z]

[5] Zhang C, Zhou J. Optimization algorithm of association rule mining for EMU operation and maintenance efficiency. Journal of Computer

Research and Development, 2017, 54(9): 1958–1965 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.2017.20160498]

[6] Li C, Liu H. Association analysis and *N*-Gram based detection of incorrect arguments. Ruan Jian Xue Bao/Journal of Software, 2018, 29(8): 2243−2257 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5531.htm [doi: 10.13328/j.cnki.jos.005531]

[7] Agrawal R, Srikant R. Fast algorithms for mining association rules. In: Proc. of the 20th Int'l Conf. on Very Large Data Bases. Santiago: VLDB, 1994. 487–499.

[8] Han JW, Pei J, Yin YW. Mining frequent patterns without candidate generation. In: Proc. of the 2000 ACM SIGMOD Int'l Conf. on Management of Data. Dallas: ACM, 2000. 1–12. [doi: 10.1145/342009.335372]

[9] Zaki MJ. Scalable algorithms for association mining. IEEE Trans. on Knowledge and Data Engineering, 2000, 12(3): 372–390. [doi: 10.1109/69.846291]

[10] Zhang R, Chen WG, Hsu TC, Yang HJ, Chung YC. ANG: A combination of Apriori and graph computing techniques for frequent itemsets mining. The Journal of Supercomputing, 2019, 75(2): 646–661. [doi: 10.1007/s11227-017-2049-z]

[11] Alghyaline S, Hsieh JW, Lai JZC. Efficiently mining frequent itemsets in transactional databases. Journal of Marine Science and Technology, 2016, 24(2): 184–191. [doi: 10.6119/JMST-015-0709-1]

[12] Zhu XL, Liu YG. An efficient frequent pattern mining algorithm using a highly compressed prefix tree. Intelligent Data Analysis, 2019, 23(S1): 153–173. [doi: 10.3233/IDA-192645]

[13] Zhang CK, Tian PB, Zhang XD, Liao Q, Jiang ZL, Wang X. HashEclat: An efficient frequent itemset algorithm. Int'l Journal of Machine Learning and Cybernetics, 2019, 10(11): 3003–3016. [doi: 10.1007/s13042-018-00918-x]

[14] Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proc. of the 9th USENIX Conf. on Networked Systems Design and Implementation. San Jose: USENIX Association, 2012. 2. [doi: 10.5555/2228298.2228301]

[15] Zhang P, Duan L, Qin P, Zuo J, Tang CJ, Yuan CA, Peng J. Mining Top-*k* distinguishing sequential patterns using Spark. Journal of Computer Research and Development, 2017, 54(7): 1452–1464 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.2017.20160553]

[16] Dong J, Han M. BitTableFI: An efficient mining frequent itemsets algorithm. Knowledge-based Systems, 2007, 20(4): 329–335. [doi: 10.1016/j.knosys.2006.08.005]

[17] Song W, Yang BR, Xu ZY. Index-BitTableFI: An improved algorithm for mining frequent itemsets. Knowledge-based Systems, 2008, 21(6): 507–513. [doi: 10.1016/j.knosys.2008.03.011]

[18] Xu ZY, Gu DY, Wei S. An efficient matrix algorithm for mining frequent itemsets. In: Proc. of the 2009 Int'l Conf. on Computational Intelligence and Software Engineering. Wuhan: IEEE, 2009. 1–4. [doi: 10.1109/CISE.2009.5364537]

[19] Fu XH, Chen DJ, Wang ZQ. Depth first frequent itemset mining based on bittable and inverted index. Journal of Chinese Computer Systems, 2012, 33(8): 1747–1751 (in Chinese with English abstract). [doi: 10.3969/j.issn.1000-1220.2012.08.023]

[20] Vo B, Hong TP, Le B. DBV-Miner: A dynamic bit-vector approach for fast mining frequent closed itemsets. Expert Systems with Applications, 2012, 39(8): 7196–7206. [doi: 10.1016/j.eswa.2012.01.062]

[21] Shahbazi N, Soltani R, Gryz J. Memory efficient frequent itemset mining. In: Perner P, ed. Machine Learning and Data Mining in Pattern Recognition. Cham: Springer, 2018. 16–27. [doi: 10.1007/978-3-319-96133-0_2]

[22] Liu JY, Jia XY. Multi-label classification algorithm based on association rule mining. Ruan Jian Xue Bao/Journal of Software, 2017, 28(11): 2865−2878 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5341.htm [doi: 10.13328/j.cnki.jos.005341]

[23] Qiu HJ, Gu R, Yuan CF, Huang YH. YAFIM: A parallel frequent itemset mining algorithm with spark. In: Proc. of the 2014 IEEE Int'l Parallel & Distributed Proc. Symp. Workshops. Phoenix: IEEE, 2014. 1664–1671. [doi: 10.1109/IPDPSW.2014.185]

[24] Rathee S, Kaul M, Kashyap A. R-apriori: An efficient apriori based algorithm on spark. In: Proc. of the 2015 Workshop on Ph.D. Workshop in Information and Knowledge Management. Melbourne: ACM, 2015. 27–34. [doi: 10.1145/2809890.2809893]

[25] Raj S, Ramesh D, Sethi KK. A Spark-based Apriori algorithm with reduced shuffle overhead. The Journal of Supercomputing, 2021, 77(1): 133–151. [doi: 10.1007/s11227-020-03253-7]

[26] Rathee S, Kashyap A. Adaptive-Miner: An efficient distributed association rule mining algorithm on Spark. Journal of Big Data, 2018, 5(1): 6. [doi: 10.1186/s40537-018-0112-0]

[27] Gassama ADD, Camara F, Ndiaye S. S-FPG: A parallel version of FP-growth algorithm under apache Spark™. In: Proc. of the 2nd IEEE Int'l Conf. on Cloud Computing and Big Data Analysis. Chengdu: IEEE, 2017. 98–101. [doi: 10.1109/ICCCBDA.2017.7951891]

[28] Li HY, Wang Y, Zhang D, Zhang M, Chang EY. PFP: Parallel FP-growth for query recommendation. In: Proc. of the 2008 ACM Conf. on Recommender Systems. Lausanne: ACM, 2008. 107–114. [doi: 10.1145/1454008.1454027]

[29] Sethi KK, Ramesh D. HFIM: A Spark-based hybrid frequent itemset mining algorithm for big data processing. The Journal of

Supercomputing, 2017, 73(8): 3652–3668. [doi: 10.1007/s11227-017-1963-4]

[30] Luo YH, Yang ZF, Shi HK, Zhang Y. A distributed frequent itemsets mining algorithm using sparse Boolean matrix on spark. In: Proc. of the 18th Asia-Pacific Web Conf. Suzhou: Springer, 2016. 419–423. [doi: 10.1007/978-3-319-45817-5_38]

[31] Raj S, Ramesh D, Sreenu M, Sethi KK. EAFIM: Efficient apriori-based frequent itemset mining algorithm on Spark for big transactional data. Knowledge and Information Systems, 2020, 62(9): 3565–3583. [doi: 10.1007/s10115-020-01464-1]

[32] Vance B, Maier D. Rapid bushy join-order optimization with Cartesian products. In: Proc. of the 1996 ACM SIGMOD Int'l Conf. on Management of Data. Montreal: ACM, 1996. 35–46. [doi: 10.1145/233269.233317]

[33] Fournier-Viger P, Lin JCW, Gomariz A, Gueniche T, Soltani A, Deng ZH, Lam HT. The SPMF open-source data mining library version 2. In: Proc. of the 2016 Joint European Conf. on Machine Learning and Knowledge Discovery in Databases. Riva del Garda: Springer, 2016. 36–40. [doi: 10.1007/978-3-319-46131-1_8]

[34] Amdahl GM. Validity of the single processor approach to achieving large scale computing capabilities. In: Proc. of the 1967 Spring Joint Computer Conf. Washington: AFIPS/ACM/Thomson Book Company, 1967. 483–485. [doi: 10.1145/1465482.1465560]

附中文参考文献:

[2] 于自强, 禹晓辉, 董吉文, 王琳. 分布式多数据流频繁伴随模式挖掘. 软件学报, 2019, 30(4): 1078–1093. http://www.jos.org.cn/1000-9825/5419.htm [doi: 10.13328/j.cnki.jos.005419]

[5] 张春, 周静. 动车组运维效率关联规则挖掘优化算法. 计算机研究与发展, 2017, 54(9): 1958–1965. [doi: 10.7544/issn1000-1239.2017.20160498]

[6] 李超, 刘辉. 一种基于关联分析与*N*-Gram的错误参数检测方法. 软件学报, 2018, 29(8): 2243−2257. http://www.jos.org.cn/1000-9825/5531.htm [doi: 10.13328/j.cnki.jos.005531]

[15] 张鹏, 段磊, 秦攀, 左劼, 唐常杰, 元昌安, 彭舰. 基于Spark的Top-*k*对比序列模式挖掘. 计算机研究与发展, 2017, 54(7): 1452–1464. [doi: 10.7544/issn1000-1239.2017.20160553]

[19] 傅向华, 陈冬剑, 王志强. 基于倒排索引位运算的深度优先频繁项集挖掘. 小型微型计算机系统, 2012, 33(8): 1747–1751. [doi: 10.3969/j.issn.1000-1220.2012.08.023]

[22] 刘军煜, 贾修一. 一种利用关联规则挖掘的多标记分类算法. 软件学报, 2017, 28(11): 2865−2878. http://www.jos.org.cn/1000-9825/5341.htm [doi: 10.13328/j.cnki.jos.005341]

丁家满 (1974－), 男, 教授, CCF 专业会员, 主要研究领域为数据挖掘, 云计算, 大数据.

贾连印 (1978－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为数据库. 数据挖掘.

李海滨 (1995－), 男, 硕士生, 主要研究领域为数据挖掘, 云计算, 大数据.

游进国 (1977－), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为大数据分析, 数据仓库, 图数据.

邓斌 (1997－), 男, 硕士生, 主要研究领域为数据挖掘, 软件工程.