一种基于MapReduce模型的高效频繁项集挖掘算法

朱 坤 黄瑞章 张娜娜

(贵州大学计算机科学与技术学院 贵阳 550025) (贵州省公共大数据重点实验室 贵阳 550025)

摘 要 由于互联网技术急速发展及其用户迅速地增加,很多网络服务公司每天不得不处理TB级甚至更大规模的数据量。在如今的大数据时代,如何挖掘有用的信息正变成一个重要的问题。关于数据挖掘(Data Mining)的算法在很多领域中已经被广泛运用,挖掘频繁项集是数据挖掘中最常见且最主要的应用之一,Apriori则是从一个大的数据集中挖掘出频繁项集的最为典型的算法。然而,当数据集比较大或使用单一主机时,内存将会被快速消耗,计算时间也将急剧增加,使得算法性能较低,基于MapReduce的分布式和并行计算则被提出。文中提出了一种改进的MMRA(Matrix MapReduce Algorithm)算法,它通过将分块数据转换成矩阵来挖掘所有的频繁k项集;然后将提出的算法和目前已经存在的两种算法(one-phase算法、k-phase算法)进行比较。采用Hadoop-MapReduce作为实验平台,并行和分布式计算为处理大数据集提供了一个潜在的解决方案。实验结果表明,改进算法的性能优于其他两种算法。

关键词 Hadoop,MapReduce,分布式计算,数据挖掘,频繁项集挖掘,Apriori算法

中图法分类号 TP399 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.07.006

Efficient Frequent Patterns Mining Algorithm Based on MapReduce Model

ZHU Kun HUANG Rui-zhang ZHANG Na-na

(College of Computer Science and Technology,Guizhou University,Guiyang 550025,China)

(Guizhou Provincial Key Laboratory of Public Big Data,Guiyang 550025,China)

Abstract Along with the rapid development of Internet and the rapid growing group of users,many Internet services companies have to manage TB size or higher amount of data every day. How to find useful information in this big data era is becoming an important problem. The data mining algorithm has been widely used in many fields,and finding frequent itemsets is one of the most common and primary applications of data mining,and Apriori algorithm is the most typical algorithm for finding frequent itemsets from a big transaction database. However,when the dataset size is rather huge or a single host is used,the memory would be quickly exhausted and the computation time would also increase dramatically,which make the algorithm performance inefficient. Parallel and distributed computing based on the MapReduce framework has been proposed. An improved reformative MMRA (Matrix MapReduce Algorithm) algorithm which should convert the blocked data into matrixs to find all frequent k-itemsets was proposed in this paper,and the proposed algorithm was compared with current two existed algorithms(one-phase algorithm and k-phase algorithm). Adapting Hadoop-MapReduce as the experiment platform,parallel and distributed computing offer a potential solution for processing vast amount of data. Experimental results show that the proposed algorithm outperforms the other two algorithms.

Keywords Hadoop,MapReduce,Distributed computing,Data mining,Frequent itemset mining,Apriori algorithm

1 引言

随着计算机的普及和互联网技术的进步,网络上每天都会产生TB级甚至PB级的海量数据。如何能够更快速、低成本、高效地在这些数据中找出有价值的知识变得非常重要。另外,从海量数据中挖掘出隐含的、先前未知的、有价值的信息的数据挖掘方法也已经发展成为一门非常重要的学科[1]。

关联规则挖掘方法在数据挖掘中已被普遍应用,其目的是找出各事务数据项之间的相关关系,其本质是获取频繁项集[2]。关联规则的挖掘通常情况下分为以下两个步骤:1)依

到稿日期:2016-08-20 返修日期:2016-10-25 本文受国家自然科学基金(61462011,61202089),高等学校博士学科专项科研基金(20125201120006),贵州大学引进人才科研项目(2011015)资助。

朱 坤(1990—),男,硕士生,主要研究方向为数据挖掘、大数据分析和应用,E-mail:kun_zhu90@163.com;黄瑞章(1979—),女,博士,副教授,主要研究方向为数据挖掘、机器学习等;张娜娜(1992—),女,硕士生,主要研究方向为数据挖掘、数据并行处理算法。

次扫描事务数据库并生成候选项集，然后与最小支持度阈值比较，找出频繁项集。2)利用频繁项集产生其关联规则，如果某规则的置信度计数符合最小置信度阈值，那么此关联规则称作强关联规则。而在挖掘频繁项集这一过程时，最为经典的一个算法是 Apriori[3]。Apriori 算法采用循环渐进的方法求出全部频繁项集，即根据前一个频繁项集推导出下一个频繁项集[4]。随着数据集的不断增大，Apriori 算法的缺点也凸显出来，当数据量较大、候选项集较多时，该算法的效率则会明显降低；另外，反复扫描事务数据库需要很长时间，也使算法的实现时间延长。目前单个主机已经不能承担大规模的运算，传统的 Apriori 算法也已无法解决现有的海量数据处理问题，因而把 Apriori 并行改进后放置到 Hadoop 上进行实现的方法被提出。

Hadoop 平台是在 Google 提出 MapReduce 模型之后产生的一种开源平台[5]，其原理是将海量数据（PB 级以上）进行分布式并行计算，具有易扩展、高效率、低成本等特性。Hadoop 是基于 Java 设计语言研发的，在搭建部署方面不限于某个固定的操作系统，因此能够容易地部署在低廉的计算机节点设备中。中小型公司或科研机构可以轻松地在 Hadoop 平台上处理海量数据，因而在众多产业和学术机构中 Hadoop 被高度重视。

在 MapReduce 的基础上挖掘频繁项集已有一些相关工作。LIU 等人[6]通过减少 Map 端低频候选项集的产生来提高运算效率。YANG 等人[7]在 MapReduce 框架上通过大量节点来操纵大数据集，最后获取频繁项集。Ye 等人[8]通过在一个并行的对称式多重处理计算机上运行程序来缩短运算时间。KOVACS F 等人[9]利用 GPUS 来并行处理频繁项集以加快运算速度。HUANG Y S 等人[10]通过在 Map 阶段归并全部候选项集，在 Reduce 阶段验证候选项集的计数值是否满足最小支持度来提高性能。

LI L 和 ZHANG M[11]在 2011 年提出了一个在 Hadoop 平台上操作的 One-phase 算法。其思路是通过 One-phase（一次 MapReduce 计算）改进传统的 Apriori 算法，使得此算法能够适用于基于 MapReduce 框架的分布式并行计算。One-phase 算法通过一次运算生成全部的候选项集。在 Map 阶段输出全部候选项集，并输出各个候选项集和其对应的 value 等于 1 的键值对。在 Reduce 阶段将 key 相同的 value 值进行合并，再将其值与最小支持度进行比较。虽然该算法的计算流程很简单，但是由于它是一次产生所有的候选项集，因此会产生很多非高频项集，内存占用严重，当内存不足时速度变得非常缓慢，甚至存在运算失败的情况。

LI N 等[12]在 2012 年提出了一个能够在 Hadoop 上并行处理的 k-phase 算法。其采用 k-phase（基于 MapReduce 的多阶层的计算），先求一阶，再求二阶，通过一阶一阶地计算找出高频项集。因为该算法不需要一次储存所有的候选项集，需要分成很多阶层去处理运算，所以内存的用量较 One-phase 方法的更少。但是若需要计算的阶数越来越多，所花费的基本时间就会相对增加，进而影响算法的整体完成时间。

Spark 中的 Mllib 已经实现了 Apriori 算法的频繁项集挖掘，这里把该频繁项集挖掘算法简称为 SMApriori 算法。SMApriori 算法的实现过程如下：首先，将数据加载至内存，运用 Map 和 Reduce 执行频繁 1-项集的挖掘；然后，执行频繁 $(k+1)$ 项集的挖掘，对于频繁 k-项集，利用 Spark Map 任务中的连接方法进行自连接运算，生成候选 $k+1$ 项集；扫描数据，对 $k+1$ 项集执行 Map 运算；最后对 Map 运算后的 $k+1$ 项集执行 Reduce 运算，从而挖掘出所有的频繁 $(k+1)$ 项集[13]。Spark 中的 Mllib 虽然已经实现了 Apriori 算法的频繁项集挖掘，但使用的还是原始的 Apriori 算法，效率仍然不高。

综上所述，虽然基于 MapReduce 框架的并行挖掘频繁项集的方法比较简单，但是大量 I/O 资源被浪费、网络通讯负担过重、数据占用空间过大、内存占用严重、效率低等缺陷仍无法得到有效解决。

本文设计了一个基于 MapReduce 编程模型的高效率频繁项集挖掘算法——MMRA 算法。该算法先将数据分块，然后在各节点对分块数据进行矩阵映射，建立各阶权值矩阵，并统计各矩阵每行的权值，再分别计算各个节点上的频繁 k-项集，最后将各个节点计算出的所有频繁 k-项集组合起来。

本文第 2 节介绍了 MapReduce 编程模型；第 3 节介绍了 Apriori 算法及相关原理；第 4 节介绍了 MMRA 算法的具体实现；第 5 节介绍了实验结果与实验分析；最后总结全文。

2 MapReduce 编程模型

MapReduce 是一个能够高效处理海量数据的分布式并行运算编程模型，把此流程用两个函数来描述：Map 和 Reduce，即映射和归约[14]，某些情况下还需要 Combiner 和 Partition，即合并和分区。

在 Map 阶段，MapReduce 架构把数据分割成 M 个片段，同时也会有 M 个 Map。Map 的输入是数据片中的键值对[key1，value1]，其中，[key1，value1]通过配置 Job 中设置的 InputFormat 来进行解析处理（InputFormate 指定输入文件的内容格式），并通过 Context 对象的 write()函数来获取 map()的输出。Map 任务调用程序员编写的 Map 函数，执行后产生并且输出中间状态〈key2，value2〉集合。然后将 key2 对应的数据进行排列，产生并输出新的〈key2，list(value2)〉集合，从而同一个键会对应一系列值形成的集合。

在 Combiner 阶段，可根据 Combine 函数将中间状态的输出结果进行合并——将 key 相同的 value 值进行合并。它首先处理本地 Map 的输出，然后将输出结果作为 Reduce 的输入。Combine 函数的主要作用是通过减少中间结果[key，value]对的数目来减少系统的消耗。Combiner 函数是一个优化 MapReduce 的方法。由于 MapReduce 的运行受到网络宽带大小的限制，因此可以利用 Combiner 函数将 Map 和 Reduce 之间的数据传输量最大程度地减少。

在 Partition 阶段，将 Reduce 任务的数量定义为 r，然后可以按照 key 的取值把中间输出状态分成 r 份。通常使用哈希(hash)函数进行划分(如：hash (key) mod r)，以确保某一

范围内的 key 一定都是由某一 Reduce 任务来进行处理，从而大大简化 Reduce 的处理过程[15]。

在 Reduce 阶段，输入的数据为〈key2，value2〉片段，经用户定义的 Reduce 函数执行完后，输出用户需要的〈key3，value3〉键值对。然后，每个 Reduce 任务向运行 Map 任务的若干计算节点发出请求，进而获取存在于 key 值区间内的数据。在 Reduce 操作中，key 都是相同的，用户只需要设计函数来对 value 的集合进行循环迭代处理即可。实现 Reduce 函数后，任何一个 Reduce 均会使用生成的输出文件（HDFS）来存放最后的结果。

其中，JobTracker 是主控服务，且只有一个。它主要进行调度和管理 TaskTracke，将 Map 任务和 Reduce 任务分发给一些空闲的 TaskTracke，并行化处理这些任务，并且控制这些任务的执行完成情况。TaskTracke 是从服务，可以有多个，负责任务的执行。如果有 TaskTracke 出现异常，JobTracker 则把其负责的任务重新分配到其他空闲的 TaskTracke 上执行[16]。

对于 HDFS 而言，默认的 HDFS 块大小为 64 MB，HDFS 所在集群分为两种节点：NNode 和 DNode。NNode 节点负责管理多个数据节点，它的运作方式是以管理者和工作者的方式进行的；此外，NNode 节点负责文件系统的命名，且控制着整个系统的运行。DNode 节点为文件系统（HDFS）的执行者，它负责存储数据块并具有定位数据块的功能，还可以按时向名称节点传递存储后的数据块列表。

另外，MapReduce 框架采用主从结构，由一台服务器作为主节点负责所有的协调和调度，其余服务器作为从节点执行任务，其中主节点只有一个，从节点可以有多个。

当用户执行 MapReduce 过程时，MapReduce 编程模型的执行流程如图 1 所示。

（1）首先把 HDFS 上的数据分片，划分后的数据块不大于 HDFS 默认的数据块大小，从而保证了较小的数据集存储到一个节点上，以便于本地计算。

（2）启动各个节点上的程序，执行 Map 任务，Map 的数目是由划分后的小数据集的数目决定的，有多少个小数据集就会有多少个 Map 任务。但是 Map 的数量和节点的个数不一样，每个节点可以处理多个 Map 任务。Reduce 的数量可以由用户指定。

（3）调用自定义的 Map 任务的程序，将输入数据按照键值对的方式输入，接着将它们分发给 Map 函数，执行完成后把生成的中间值暂时存放到 Cache 当中。

（4）分区函数处理 Cache 中的所有键值对，然后周期性地存储在本地的磁盘上。将本地磁盘上键值对的存储位置等内容发送到主节点上，然后再通过它把存放的位置信息传递到 Reduce 主节点程序。

（5）当 Reduce 主节点程序收到从节点发来的处理结果信息后，需要以 RPC（远程过程调用协议）的方式读取 Map 输出的信息。当得到信息后，会对每个关键字对应的结果进行排序，这样可以将相同关键字的值汇聚到一起。

（6）主节点处理经过排序的信息，把所有 key 对应的 value值集合进行处理，然后调用用户改写的 Reduce 函数。

（7）Map 和 Reduce 执行实现后，主节点通知其用户定义的程序，然后再输出存放结果的文件。

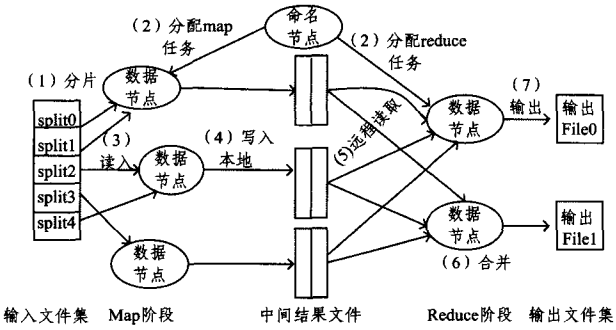

图 1　MapReduce 编程模型的执行流程图

3　Apriori 算法及相关原理

3.1　Apriori 算法

Apriori 算法步骤如下。

输入：事物数据集、最小支持度阈值 min_Sup、最小置信度阈值 min_Conf

输出：关联规则

1）将事物数据集逐条进行扫描，再结合 min_Sup 产生频繁 1-项集。

2）把 L_K 进行裁剪和连接，生成候选 C_{K+1}，再结合 min_Sup，生成 L_{k+1}（频繁）项集。

3）若 L_{K+1} 非空，则 k 值增 1，重新执行步骤 2）；否则，执行步骤 4）。

4）由步骤 3）获取全部频繁 k-项集，再结合 min_Conf，生成强关联规则，算法完毕。

3.2　相关性质、推论及定理

性质 1　频繁项集的所有非空子集一定是频繁项集；任何非频繁项集都不可能是频繁项集的子集。

推论 1　L_k 是事务数据库 D 的频繁 k-项集，那么 L_{k-1} 中含有 L_k 的 $k-1$ 项子集的数目必须为 k。

推论 2　若频繁 k-项集可以生成频繁 $k+1$ 项集，那么频繁 k-项集中的数目一定大于 k。

性质 2[17]　存在一频繁 k-项集 L_k，若 $|L_k|<k+1$，那么事务集最大频繁项集的数目为 k，其中 $|L_k|$ 为频繁 k-项集的数目。

性质 3　非频繁项集的任一超集也一定是非频繁项集。

定理 1[18]　当 $k-1$ 项集产生 k 项集时，L_{k-1} 进行自连接运算，如果两个项集的前 $k-2$ 项不相同，那么不再对这两个项集进行连接，这是由于产生的项集可能是非频繁项集或者是相同的项集。

推论 3[19]　对于两个 $k-1$ 频繁项目集 I_m 和 I_n，若 I_m 和 I_n 不可以进行连接运算，那么对 I_m 和 I_n 以后的任意项目集均不需要进行连接判断。

4　MMRA 算法的实现

MMRA 算法流程的示意图如图 2 所示。

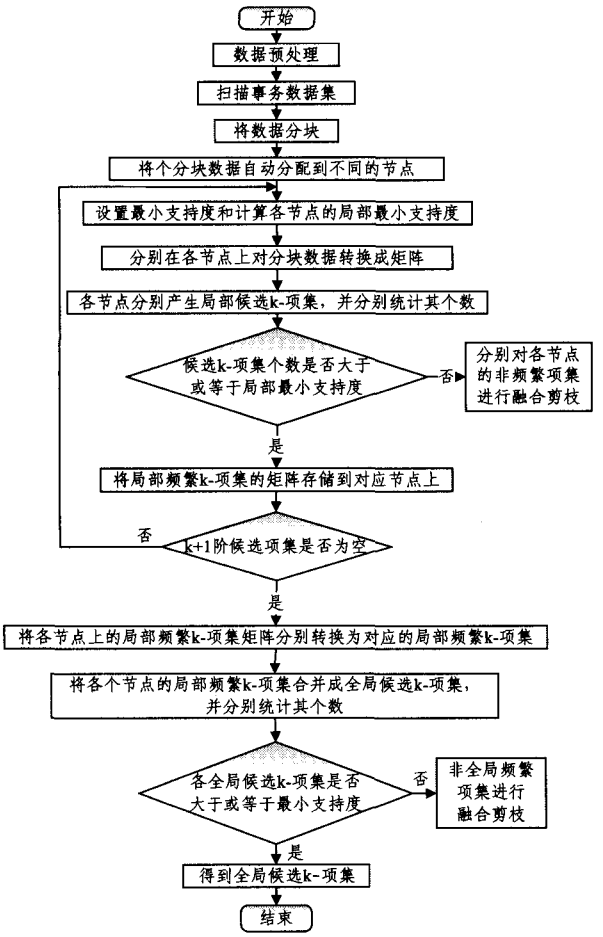

图 2 算法流程示意图

MMRA 算法具有通用性，由于文章篇幅限制以及为了更好地描述与理解，从第(6)步开始，用一个具体的例子来详细描述 MMRA 算法的过程，其余过程均与该例子的过程类似。

(1)安装并配置 Hadoop 集群，Hadoop 平台分别是由 1—6 个节点组成的 6 个集群；然后启动服务器。

(2)将数据进行预处理运算，首先对数据进行噪音处理等操作，然后进行集成、变换、归约等一系列操作后，形成相对规范的数据集。

(3)扫描事物数据集。

(4)将数据分块。由于形成的块的大小对算法有一定的影响，若各块数据集过小，其就会造成块过多，网络传输的数据量则明显增加，通讯压力变大，I/O 资源浪费严重，内存占用量过大；如果各分块数据集过大，则每个节点的计算压力就会过大，内存浪费严重，影响运算效率。因此需要设置合适的分块数据集大小。

(5)将各分块数据自动分配到不同的节点。

(6)用一个具体的例子详细描述 MMRA 算法的过程，其余过程均与该例子的过程类似，这也体现了 MMRA 算法的通用性。

设全局的最小支持度计数是 s，总数是 n，数据块大小是 m，那么局部的最小支持度计数是 $(m/n) * s$，本例中，全局最小支持度计数是 12，总数为 16，一个数据分块的大小为 4，则局部最小支持度计数为 3。

(7) 分别在各节点上对分块数据进行矩阵映射。

以一个节点上的数据处理过程为例，事务集如表 1 所列，矩阵及权值矩阵存储运算过程如图 3 所示，将图 3 的第一个表中的数据作为事务数据集，事务数据集包括 TID 和事务项目。过程(一)将事务数据集转换成矩阵的存储形式，对事务数据集进行扫描，当扫描事务编号 D_1 的事务项目 F_1，F_2，F_3 时，矩阵的第一行从低位到高位依次赋值，分别为 11100；扫描事务编号 D_2 的事务项目 F_1，F_2，F_4 时，矩阵的第二行依次为 11010；同理，扫描完成事务编号 D_3，D_4，D_5，D_6，得到事务矩阵。只要把事务数据集映射成矩阵，以后就只需要对矩阵进行操作，因此只需要对事务数据集扫描一次即可，从而极大地压缩了数据量，节省了大量内存，还避免了多次扫描整个数据集造成的大量 I/O 资源的浪费。

表 1 事务表

事务编号(事务 ID)	事务项目
D_1	F_1 F_2 F_3
D_2	F_1 F_2 F_4
D_3	F_2 F_3 F_4
D_4	F_2 F_3 F_4 F_5
D_5	F_1 F_2 F_4
D_6	F_1 F_2 F_3 F_4

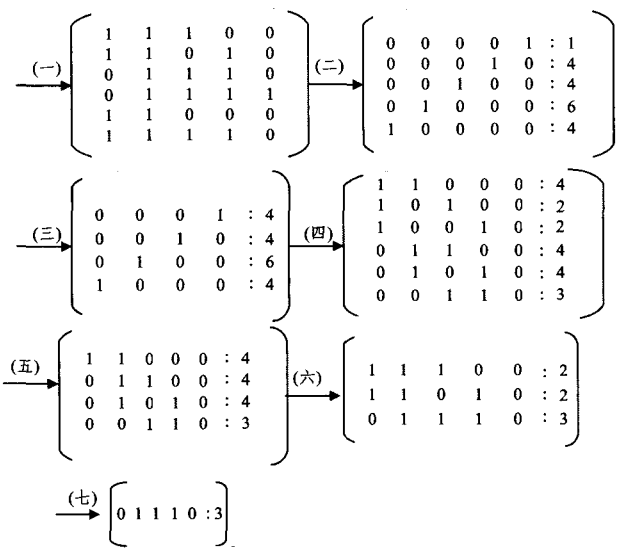

图 3 矩阵及权值矩阵存储运算的过程

(8)分别在各节点上找出各数据分块的局部频繁 k-项集。

1)在过程(二)中，将事务数据集转化为原始矩阵后，扫描该矩阵，建立权值矩阵，产生了一阶局部候选项集，同时统计了矩阵列和生成权值(每个一阶局部候选项集的计数)。

2)在过程(三)中，将各行权值与本例中设置的局部的最小支持度计数 3 进行比较，若其大于或等于局部的最小支持度计数 3，那么本行局部的候选 1-项集成为局部频繁 1-项集，而且保留这一行；若其小于局部的最小支持度计数 3，那么该行局部的候选 1-项集是非局部频繁 1-项集，则对该项集进行融合剪枝并去除该项集。最后将局部的频繁 1-项集的矩阵存储到此节点中。

3)验证二阶频繁项集是否为空，如果不为空，则进行过程

(四)，1-阶频繁项集对应矩阵内任意两项自连接作或运算(例如矩阵的第一行 00010 和第二行 00100 作或运算后得到 00110)生成 2-候选频繁项集相关矩阵，并统计各项集的权值(例如对于 00110，则计算 00110 在原始矩阵的全部行中出现的次数)。

4)在过程(五)中，与过程(三)类似，将各行的权值与局部最小支持度计数 3 作比较，如果其大于或等于局部最小支持度计数 3，那么此行局部候选 2-项集为局部频繁 2-项集，而且保留此行数据；如果其小于局部最小支持度计数 3，那么该行局部候选 2-项集为非局部频繁 2-项集，则对该项集进行融合剪枝并去除该项集。最后将局部频繁 2-项集的矩阵存储到此节点中。

5)检验三阶频繁项集是否为空，如果不为空，则进行过程(六)，二阶频繁项集矩阵内的任意两项自连接作或运算(例如矩阵的第一行 11000 和第二行 01100 作或运算后得到 11100)连接产生三阶候选频繁项集矩阵，并统计各项集权值(例如对于 11100，则计算 11100 在原始矩阵的全部行中出现的次数)。

6)在过程(七)中，与过程(三)、过程(五)类似，将各行的权值与局部最小支持度计数作比较，如果其大于或等于局部最小支持度计数 3，那么此行局部候选 3-项集为局部频繁 3-项集，而且保留此行数据；如果其小于局部最小支持度计数 3，那么该行局部候选 3-项集为非局部频繁 3-项集，则对该项集进行融合剪枝并去除该项集。最后将局部的频繁 3-项集的矩阵存储到此节点中。

7)检验 4-频繁项集是否为空，由于其为空，则生成局部的频繁项集的过程完成。将全部节点上存在的局部的频繁 k-项集矩阵分别转换为对应的局部的频繁 k-项集。

(9)将各节点上的局部频繁 k-项集矩阵分别转换为对应的局部频繁 k-项集(例如矩阵的一行 00110:3 可以转换为 F_3F_4:3；同理，其他行也是如此)。

(10)转换成局部的频繁 k-项集后，将所有节点的局部的频繁 k-项集合并成全局的候选 k-项集，并分别统计各个全局的候选 k-项集的个数(分别将全部节点上的相同的各阶局部的频繁 k-项集合并，例如对于一个二阶局部频繁项集 F_2F_4，需要将 F_2F_4 在所有节点上出现的次数相加)。

(11)检验各全局候选 k-项集的数目，若大于或等于全局最小支持度计数，则获得全局频繁 k-项集；否则，把非频繁 k-项集全局剪枝。最终得到全局频繁 k-项集。

①第一个 Mapper 类的伪代码：

```
class FirstMapper{
    //事务数据集
    List tSet;
    //该函数实现的是一个节点上的数据分块转化为矩阵的过程，且遍历事务集时需反复执行下面过程
    public void transMatrix(tSet){
    //定义一个 BigInteger 类型的变量 matrixSet，用于存储矩阵元素；把一个矩阵元素存储在一个 String 型的矩阵 matrixArray 中
    BigInteger matrixSet;
    String matrixArray[];
    For each matrixSet in matrixArray[]
      //记录新元素的位置，并加入到 matrixArray[]中，支持度计数设置为 1
      local(matrixArray[i]);
      matrixArray[]. add(matrixSet );
      count=1;
      //将项目所在位置设置为 1
      1=local(matrixArray[i]);
    End;
    }
    map(key1,value1){
      //存储局部候选项集
      Map localCI;
      //遍历 matrixArray
      For each value in matrixArray
        //对 matrixArray 中的一行 value 1 和另一行 value 2 作或运算，并将结果存储在 temp 中，temp=genMatrix(value1 | value2);
        //由 k 项集生成 k+1 项集的过程中，如果 temp 中 1 的个数不等于 matrixArray 中 1 的个数加 1，则说明不是 k+1 项集，将其剪枝；反之，将得到的 k+1 项集存储到 localCI 中
        if(temp. count(1)! =matrixArray. count(1)+1){
            delete(temp);
        }else{
            localCI. add(temp);}
      End;
      For each item I in localCI
        Output (I,1);
      End;
    }
}
```

②第一个 Reducer 类的伪代码：

```
class FirstReducer{
    //存储局部频繁项集
    reduce(key2,value2){
      Int sum1=0;
      Map localFI;
      //对 key 值相同的局部候选项集计数
      For each value2 in localCI
      sum1+=value2;
      End;
      //得出局部的最小支持度计数
    minLocalSupport=(num/n) * Apriori. supportRate;
      //得出局部频繁项集
      If (sum1 >=minLocalSupport)
      output(key2,sum1);
    }
    localFI. add(key2,sum1);
}
```

③第二个 Mapper 类的伪代码：

```
class SecondMapper{
    map(key3,value3){
      Int sum2=0;
      //存储全局候选项集
      Map globalCI;
      //将各节点产生的局部的频繁项集组合产生全局的候选项集
      sum2 +=count;
    }
    globalCI. add(key3,sum2);
}
```

④第二个 Reducer 类的伪代码：

```
class SecondReducer{
  reduce(key4,value4){
    //得出全局的最小支持度计数
    minGlobalSupport=num * Apriori. supportRate;
    If (sum2>=minGlobalSupport)
    output(key4,sum2);
    }
  globalFI. add(key4,sum2);
  }
```

5 实验结果与实验分析

5.1 实验环境

整个实验是在 Hadoop 下进行的，所用软件为Hadoop2.4.0。整个平台是由 6 个节点组成的集群，其中有 1 个主节点和 5 个从节点。硬件设备全部为 x86 架构的 PC，主设备节点使用 Intel 至强处理器，CPU 主频为 3.4GHz，内存为 4GB；从设备节点使用 AMD 四核处理器，CPU 主频为2.7GHz，内存大小为 2GB。

5.2 实验数据集

One-phase，k-phase 和 MMRA 3 种算法均在 Frequent Itemset Mining Implementations Repository 的 WebDocs 事务数据集上进行实验，数据真实可靠，并且相比其他数据集，该数据集经数据预处理后具有较高的频繁项集挖掘效率，可以很好地应用于数据挖掘算法中。此事务数据集大小为 1.37GB，含有 1692082 条事务数据，共有 5267656 个不同的项目，一个事物最多含有 71472 个不同的项目，平均每条事务含有 177 个项目。

5.3 实验分析

图 4 示出了在全局最小支持度阈值为 0.45%时，One-phase，k-phase 和 MMRA 3 种算法在块数不同的情况下的运行时间。通过实验可得，在块数相同的情况下，MMRA 算法的执行完成时间少于另外两种算法。另外，当块的个数变多时，One-phase 算法的执行时间一直下降。对于 k-phase 算法，随着块数的增加，当块数小于或等于 8 时，运行时间逐渐减少；当块的个数大于 8 时，运行时间逐渐增多；当块的个数等于 8 时，k-phase 算法的性能最好。对于 MMRA 算法，当块数小于或等于 6 时，运行时间逐渐减少；当块数大于 6 时，运行时间逐渐增多；当块数等于 6 时，MMRA 算法所用时间最少，说明此时 MMRA 算法的性能是最好的。这是因为数据块过多时，网络传输的数据量则会增加，通讯压力变大，效率明显降低。

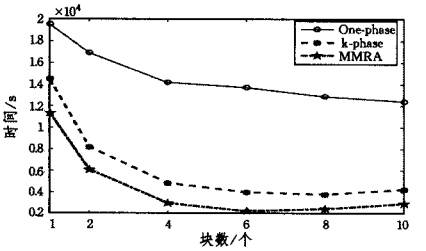

图 4 块数不同时 3 种算法的执行完成时间

图 5 示出了 MMRA 算法在块数不同时的加速比。随着块数的增加，当块数小于或等于 6 时，MMRA 算法能够得到接近线性的加速比；当块数大于 6 时，加速比则快速减小。这也说明分块过多时，网络传输的数据量太多，效率会降低。

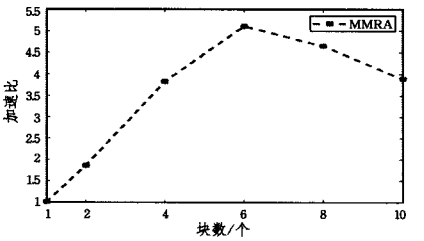

图 5 MMRA 算法在块数不同时的加速比

图 6 示出了在块数为 6 时 One-phase，k-phase 和 MMRA 3 种算法在不同支持度下的运行时间。通过实验可知，在全局最小支持度阈值相等的情况下，MMRA 算法的运行时间要少于其他两种算法，由此说明 MMRA 算法完成的效率更高。另外，对于 One-phase，k-phase，MMRA 3 种算法，当全局最小支持度阈值增大时，可以看出 3 种算法的完成时间都在减少，这主要是由于当全局支持度阈值增大时，全局频繁项集的数目在减少。

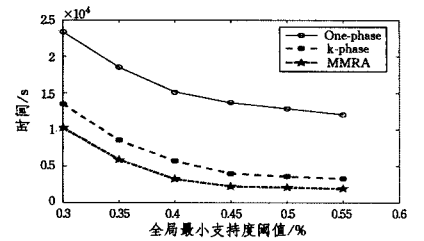

图 6 全局最小支持度阈值不同时 3 种算法的执行完成时间

图 7 示出了 3 种算法在节点数目不同时的执行时间。本实验中，将 One-phase，k-phase 和 MMRA 3 种算法的最小支持度均设置为 0.35%，测试了 3 种算法分别在1—6 个节点的集群环境上运行所需要的时间。随着集群节点数目不断地增加，3 种算法的运行时间均逐渐减少，且在节点数目相同时，MMRA 算法的运行时间少于其他两种算法，另外，由于 One-phase 算法在 1 个节点的运行时间过长(这里没在图中标出)，因此在利用 Hadoop 集群处理大数据集时，在单个节点处理 Map 和 Reduce 的数目不变的情况下，在一定节点范围内，增加集群节点的数目，可以提高并行算法的运行速度。

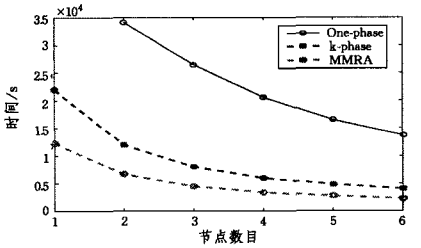

图 7　3 种算法在节点数目不同时的执行时间

结束语　本文针对多次扫描整个数据集造成大量 I/O 资源浪费、数据占用空间过大、效率低等缺点，提出了一种基于 MapReduce 模型的高效频繁项集挖掘算法——MMRA 算法。该算法先将数据分块，然后在各节点上对分块数据进行矩阵映射，建立各阶权值矩阵，再分别计算局部频繁 k-项集，最后合并计算出全局频繁 k-项集。实验结果表明，在块数、支持度、节点数目不同时，MMRA 算法与 One-phase 算法、k-phase 算法相比在性能方面有一定的提升。

在本文的基础上，下一步工作将改善算法执行效率，同时提高应用的准确性，从而解决实际问题，为实际应用服务。

参 考 文 献

[1] HE Y, WANG W Q, XUE F. Study of Massive Data Mining Based on Cloud Computing[J]. Computer Technology and Development, 2013, 23(2): 69-72. (in Chinese)
贺瑶，王文庆，薛飞. 基于云计算的海量数据挖掘研究[J]. 计算机技术与发展，2013，23(2)：69-72.

[2] TIAN S, WANG S, LIU Y, et al. An algorithm for mining frequent itemsets on uncertain dataset[J]. Computer Modelling and New Technologies, 2014: 264-272.

[3] CHANG R, LIU Z. An improved apriori algorithm[C]// 2011 International Conference on Electronics and Optoelectronics. 2011: 1476-1478.

[4] AGRAWAL R, IMIELINSKI T, SWAMI A. Mining Association Rule between Sets of Items in Large Databases[C]// ACM SIGMOD Conf. Management of Data, 1993: 207-216.

[5] LI W W, ZHAO H, ZHANG Y, et al. Research on massive data mining based on MapReduce[J]. Computer Engineering and Applications, 2013, 49(20): 112-117. (in Chinese)
李伟卫，赵航，张阳，等. 基于 MapReduce 的海量数据挖掘技术研究[J]. 计算机工程与应用，2013，49(20)：112-117.

[6] LIU S H, LIU S J, CHEN X S, et al. IOMRA– A High Efficiency Frequent Itemset Mining Algorithm Based on the MapReduce [C] // IEEE 17th International Conference on Computation Model. 2014: 1290-1295.

[7] YANG X Y, LIU Z, FU Z. MapReduce as a Programming Model for Association Rules Algorithm on Hadoop[C]// The 3rd International Conference on Information Sciences and Interaction Sciences. Chengdu, China, IEEE, 2010: 99-102.

[8] YE Y B, CHANG C C. A Parallel Apriori Algorithm for Frequent Itemsets Mining[C] // Fourth International Conference Software Engineering Research, Managementand Applications. 2006: 87-94.

[9] KOVACS F, ILLES J. Frequent itemset mining on hadoop. Computational Cybernetics (ICCC)[C]// The IEEE 9th International Conference. 2013: 241-245.

[10] YU K M, LEE M G, HUANG Y S, et al. An Efficient Frequent Patterns Mining Algorithm Based on MapReduce Framework [C]// International Conference on Software Intelligence Technologies and Applications & International Conference on Frontiers of Internet of Things. 2014: 1-5.

[11] LI L, ZHANG M. The Strategy of Mining Association Rule Based on Cloud Computing[C]// The International Conference on Business Computing and Global Informatization. Washington, DC, USA, 2011: 475-478.

[12] LI N, ZENG L, HE Q, et al. Parallel Implementation of Apriori Algorithm Based on MapReduce[C]// The 13th ACIS International Conference on Software Engineering, Networking and Parallel& Distributed Computing. Kyoto, 2012: 236-241.

[13] NING Y H. Study of Some Techniques in Data Mining Based on Spark[D]. Hangzhou: China Jiliang University, 2015. (in Chinese)
宁永恒. 基于 Spark 的若干数据挖掘技术研究[D]. 杭州：中国计量学院，2015.

[14] LIU D D, CHEN J, LIANG F, et al. A Performance Analysis for Hadoop under Heterogeneous Cloud Computing Environments [J]. Journal of Interation Technology, 2012, 1(4): 46-51. (in Chinese)
刘丹丹，陈俊，梁锋，等. 云计算异构环境下 Hadoop 性能分析[J]. 集成技术，2012，1(4)：46-51.

[15] DONG X H, LI R X, ZHOU W W, et al. Performance Optimization and Feature Enhancements of Hadoop System[J]. Journal of Computer Research and Development, 2013, 50(S2): 1-15. (in Chinese)
董新华，李瑞轩，周湾湾，等. Hadoop 系统性能优化与功能增强综述[J]. 计算机研究与发展，2013，50(S2)：1-15.

[16] YING Y, LIU Y J. Survey of Developments of MapReduce Parallel Computing Technology[J]. Computer System Application, 2014, 23(4): 1-6. (in Chinese)
应毅，刘亚军. MapReduce 并行计算技术发展综述[J]. 计算机系统应用，2014，23(4)：1-6.

[17] KHARE N, ADLAKHA N, PARDASANI K R. An Algorithm Ior Mining Multidimensional Association Rules using Boolean Matrix[C]// 2010 International Conference in Recent Trends in Information, Telecommun ication and Computing. Kochi, Kerala, 2010: 95-99.

[18] LIU H Z, DAI S P, JIANG H. Quantitative association rules mining algorithm based on matrix[C]// 2009 International Conference on Computational Intelligence and Software Engineering. Wuhan, 2009: 1-4.

[19] TIAN S, WANG S, LIU Y, et al. An algorithm for mining frequent itemsets on uncertain dataset[J]. Computer Modelling and New Technologies, 2014, 18(11): 264-272.