基于分布式协调系统的并行频繁模式增长算法的优化

王 洁[1] 戴清灏[1,2] 李 环[1]
(首都师范大学管理学院 北京 100089)[1] (中国科学院计算技术研究所 北京 100190)[2]

摘 要 频繁模式挖掘可以发现数据中频繁出现的模式,是关联规则挖掘的重要步骤。并行频繁模式算法将其应用到并行环境中,以对海量数据进行挖掘。在Apache软件基金会的Mahout项目实现的基础上,对计数和排序阶段以及算法的执行顺序提出了新的优化策略。优化后的设计将计数信息存储在分布式协调系统上,充分地利用了分布式协调系统的高可用性、适宜存储元数据信息的特点。该设计减小了小文件在分布式文件系统(HDFS)上的开销,同时保留了其优点,还能使计数过程和排序过程并行执行,减小了计算节点的内存开销。对比了文件系统I/O的开销,并分析了实现设计中的难点,为未来的工作打下了基础。

关键词 频繁模式增长算法,并行数据挖掘,分布式协调系统,性能优化
中图法分类号 TP312 **文献标识码** A

Tuning of Parallel Frequent Pattern Growth Algorithm Based on Distributed Coordination System

WANG Jie[1] DAI Qing-hao[1,2] LI Huan[1]
(School of Management, Capital Normal University, Beijing 100089, China)[1]
(Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China)[2]

Abstract Frequent pattern mining can find frequent pattern in data, and it's an important step in the association rules mining. Parallel frequent pattern(PFP) algorithms apply it into parallel environment, which is suitable for massive data. Based on the implementation of Apache Mahout, this paper proposed a design for optimizing the counting and sorting parts of PFP using distributed coordination system. This design takes advantage of distributed coordination system and reduces the consumption on HDFS and memory of data node. Another benefit is that the counting procedure and sorting procedure start parallelly. At last this paper analyzed the experimental result and the difficulties for implementation for further study.

Keywords Frequent pattern growth algorithm, Parallel data mining, Distributed coordination system, Performance tuning

1 引言

频繁模式挖掘(Frequent Pattern)是数据挖掘研究中的一个重要课题,也是关联规则、相关性分析、序列模式等数据挖掘任务的基础。频繁模式增长(Frequent Pattern Growth),即FP-Growth算法[11],是一种不产生候选集并且时空效率都很高的频繁模式挖掘算法。FP-Growth的核心是建立一个FP-Tree数据结构,然后从FP-Tree中进行挖掘,它对于挖掘长和短的频繁模式,都是有效的和可规模化的,并且比Apriori算法快一个数量级。

海量数据挖掘是一个应用数据挖掘算法的、具有挑战性的场景,将频繁模式算法应用到海量数据的挖掘中具有重要的意义。但FP-Growth算法比较复杂,难以应用到并行环境中。这是由于与Apriori算法相比,FP-Growth在构建FP-Tree之前,需要先对数据集进行计数,统计出每个项(Item)出现的次数,然后才能对事务(Transaction)进行排序。而在并行与分布式环境下进行计数和排序,会遇到以下问题:第一是计数粒度的划分,第二是通信和同步的开销,第三是内存大小的限制[8]。

目前在并行环境下已经有了一些关联规则数据挖掘的实现方法。例如Mohammad [1]等人提出了并行搜索频繁模式的方法。Lamine M.[2]等人讨论了异构平台下的工作流管理。S. Parthasarathy[3]等人使用了共享内存的方式来保证一致性,并讨论了如何降低False Sharing。A. Javed[4]等人用传统的方式实现并行频繁模式,通过将计数本地化再归并来保证计算性能。Iko Pramudiono[5]等人使用的也是使数据本地化的设计,以保证计数的性能。Gregory Buehrer[6]等人的设计减少了网络开销。Gregory Buehrer的文章中也提到了将计数信息存储到单个节点上不可行,每一次信息更新时在每个节点上的同步开销也太大,取而代之的依然是分布式计数(Count Distribution),其与上文的A. Javed的实现类似,即每个节点先计数,再归并。

但是,以上的设计都存在一些缺点或不足。Mohammad等人重点在于搜索频繁模式,而不在计数上,且提及了分布式

到稿日期:2011-04-15 返修日期:2011-07-26 本文受国家信息安全测评中心项目(CNITSEC-KY-2007-22)资助。
王 洁(1977—),女,博士,讲师,主要研究方向为并行计算、数据挖掘、机器学习,E-mail:wangjie@ncic.ac.cn;戴清灏(1989—),男,主要研究方向为数据挖掘、云计算;李 环(1963—),女,高工,主要研究方向为信息管理、计算机网络。

计数算法(Count Distribution Algorithm)上的3个难题,其中之一就是受限于内存大小,即计数信息如果太大则不宜存放在各个节点的内存中。Lamine M等人的设计则采用了Apriori算法,它无疑会产生大量的候选集,不适宜海量数据的场景。S. Parthasarathy等人的设计虽然可以在共享的内存中保持一致性,但处理内存分配和降低错误共享时会产生不小的复杂性,且也是基于Apriori算法。A. Javed的设计中全局计数信息仍然是存储到单个节点上,不适宜海量数据的场景。Iko Pramudiono等人的设计类似于A. Javed的设计,只是将一致的计数信息分别存储在各个节点上,这样的设计也会受到内存大小的限制。Gregory Buehrer的实现中,分布式计数阶段会有一个问题,就是当最小支持度值降低时,候选集的数量会增加。因为每个节点都会产生完整的计数信息,所以会消耗很多的空间资源,同样,当项数量很多时也会有这个问题。而且,以上所提到的所有设计,都是在计数完成以后才开始排序。

针对现有并行挖掘算法的不足,提出了一种新的基于分布式协调系统的计数-排序的设计来对并行频繁模式增长PFP-Growth(Parallel FP-Growth)算法进行优化。

2 Mahout中PFP-Growth算法的实现与不足

目前实现了并行频繁模式挖掘算法的比较著名的开源项目是Mahout,因此本文的优化策略是基于Mahout项目中的实现提出的。Mahout是由Apache软件基金会开发的开源项目,其主要目标是创建一些可伸缩的数据挖掘和机器学习算法。Mahout中的并行频繁模式挖掘算法是在Hadoop计算平台上,采用MapReduce编程模型和分布式文件系统HDFS,基于FP-Growth算法来实现的。

2.1 Mahout中PFP-Growth算法的实现

Mahout中并行频繁模式增长算法实现的设计思路来自Haoyuan Li, Yi Wang等人的论文[7],该方法通过执行多个MapReduce任务的方式实现频繁模式挖掘。方法主要分为ParallelCounting, GroupingItem, TransactionSorting, PFP-Growth, Aggregating 5个阶段。本文优化的是Parallel-Counting,GroupingItem和TransactionSorting阶段,这3个阶段在Mahout的实现中依先后顺序逐一执行。

在PalallelCounting阶段,算法会做简单的并行计数,并将得到的计数信息以键值对的形式序列化后存储在HDFS中。这一步会得到一个F-list,记录着所有项的出现次数。

在GroupingItem阶段,将把从前一步中得到的所有无重复的项分组,按照指定好的组数目,平均分配在各自的组中,每组有一个Id。这一步是在单个节点上串行地执行。这一阶段的结果是产生一个G-List。

在TransactionSorting阶段,是对每个Transaction中的Item进行排序。这个排序过程基于之前的F-List。在这个过程中没有用到G-List。

在这3个过程之后进行的是递归建立FP-tree的过程。因为F-list和G-list文件较小,所以会将执行这一过程的节点下载到内存中。

2.2 Mahout中PFP-Growth算法的不足

目前在Mahout中并行FP-Growth算法的实现存在一些不足,具有一定的优化空间。首先,Mahout中对于Parallel-Counting,GroupingItem和TransactionSorting 3个阶段是顺序执行,但实际上这3个过程存在并行化的可能性。例如在TransactionSorting中并不需要G-List。第二,Mahout中F-list和G-list的信息会被存储进HDFS,经过Hadoop管线[10]存入HDFS,这一过程会有复制产生冗余的开销,而HDFS也不适合存储F-list,G-list这种较小的文件。第三,Mahout的实现中,也不能提前得出排序的结果。但实际上项的排序结果可以提前得出。例如在计数过程中,已经统计得到项A出现了a次,项B出现了b次,如果还剩余r条事务记录未计数,且$(b+r)<a$,则最后项B出现的次数一定小于A。在频繁模式算法中,只需要得到两个项的计数之间的大小关系就足够了。而且在数据挖掘之前,可以根据总数据集的大小和每条记录的大小得到总记录数和估计值。根据总记录数和已经记录的数目,就可以得到剩余记录数或其估计值。最重要的是,排序一个事务Transaction,只需要该事务所包含的项的计数之间的大小关系即可。

3 基于分布式协调系统的PFP-Growth算法优化策略

结合以上的分析,针对Mahout中并行频繁模式增长PFP-Growth算法实现上的不足,提出了基于分布式协调系统的优化策略。

3.1 分布式协调系统

分布式协调系统是在分布式环境下,用来维护配置、名称、状态等信息的系统,可提供分布式同步与组服务等。目前使用比较广泛的是Apache软件基金会的Zookeeper项目[9]。

分布式协调系统本身也是分布式的,即由多个节点组成。集群节点共同维护一个类似文件系统的树状数据结构。和文件系统不一样的是,每个名称节点既可以存储数据,也可以作为父节点,即每个名称节点既可以存储数据,也可以有子节点。这样的层次的名称空间(name space)是存储元数据(meta-data)的理想组织方式,通常存储的信息都是Kb级别的。每个集群的节点都有这个数据结构的完全拷贝。

Zookeeper实现了paxos算法,能够保证最终一致性和可用性。由于它是分布式的架构,因此可以将通信的开销分担到多个节点上。客户端可以通过Zookeeper的API连接上Zookeeper集群的任一节点,并进行创建、删除、修改、同步和获取子节点的操作,还可以在指定节点上添加监视,一旦节点被修改,就可以触发相应的事件,即事件驱动。

3.2 优化策略

针对Mahout中PFP-Growth算法实现的不足,设计了以下3个优化策略,即计数信息存储的优化、算法执行顺序的优化以及搜集计数信息的优化。

3.2.1 计数信息存储的优化

设计的优化策略之一是对并行频繁模式增长算法中计数信息部分的存储进行优化,将计数信息存储在Zookeeper上。例如对于项A,当把计数信息搜集好了以后,就写入/Item/A节点,计数信息累加在/A节点的数据中(如/Item/A节点不存在,则创建之)。

同样,分组信息也存储在Zookeeper中,例如对于项B,

要将其分入分组 2 中，则在 Zookeeper 的/Group/2 节点的数据中追加一个 B。由于此过程是串行的，也可简单地实现一次更新一个 Group。

在创建和修改节点的过程中，都会对节点加上事件监听，如果节点数据发生变化，则 Zookeeper 将事件发送到客户端更新数据。同样，在 3.2.2 节提到的 TransactionSorting 中，也是对相应的节点加上事件监听，一旦节点数据发生变化，就更新数据，直到得到足够的信息进行排序。

Zookeeper 机群的数据结构如图 1 所示。

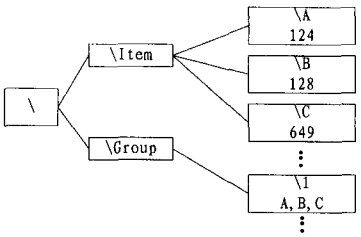

图 1　Zookeeper 机群数据结构

3.2.2　算法执行顺序的优化

在 Mahout 的实现中，并行频繁模式增长算法是顺序执行的，但实际上并非一定如此。例如 TransactionSorting 过程并不依赖 G-list。而且依本文第 2 节所述，TransactionSorting 与 ParallelCounting 过程可以同步进行，因为并不用等到所有记录都处理完就可以得到两个项之间计数的大小关系。

频繁模式挖掘中，通常项的种数很多，但是每个事务中包含的只是所有项中非常小的一部分。排序部分在 Zookeeper 上只对要排序的 Transaction 中含有的项所对应的节点加上监视，当数据发生更新时，就将数据更新到本地。同时更新的还有已经处理的记录数。利用在用户开始时所输入的估计的总的记录数和 Zookeeper 上记录的已经处理的记录数，执行计数任务的节点就能够得到未处理的记录数。一旦确定 Transaction 中项之间的两两大小关系，就可以进行排序。这样的设计实现了排序的提前开始(可以利用 Hadoop 的 MultithreadedMapper 来实现多线程)。并且这样的设计类似于生产者/消费者的模型，通常每个 Map 任务以独立线程的形式进行，在没有得到 Item 计数的大小关系之前处于阻塞状态，由 Zookeeper 的事件驱动。

当 GroupingItem 和 ParallelCounting 都完成之后，则可以开始 ParallelFPGrowth 过程。

优化后的算法执行顺序过程如图 2 所示。

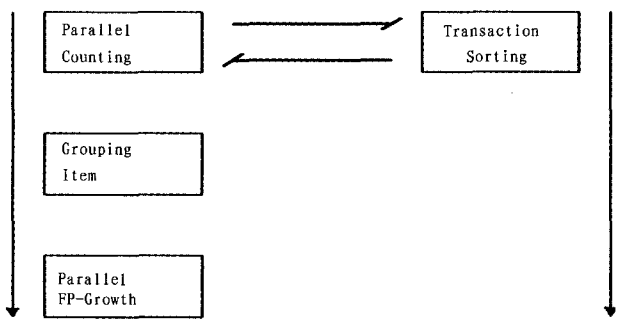

图 2　优化后的算法执行顺序

3.2.3　搜集计数信息的优化

Mahout 中的 PFP-Growth 算法是在 MapReduce 编程模型下实现的。在 Mahout 的实现中，其在 Map 阶段搜集单个项的信息，然后以〈Item，1〉的键值对的形式输出，经过 Combiner 归并后到 Reducer 进行进一步的归并。

在海量数据挖掘的场景中，如果在 Map 阶段就往 Zookeeper 上更新计数信息，必然会产生大量的写操作。而在 Reduce 阶段再往 Zookeeper 上更新计数信息，则又不够及时。所以本文的设计是在 Combiner 阶段往 Zookeeper 上更新计数信息(Hadoop 支持 Combiner)，而 Reduce 阶段则不进行任何操作。在客户端与 Zookeeper 之间的会话设计上，可以使一个 Mappper 使用一个会话，因为一个 Mapper 上通常会执行多个 Combiner，而且都是串行的方式执行，这样既减少了会话的数量，也减少了建立结束会话的开销。实验证明，Zookeeper 能够承受实验数据集中的会话开销。

3.3　本文设计策略的优点

与已有的并行频繁模式增长算法相比较，本文所设计的优化策略具有以下几方面的优点。

首先，相比传统的关系型数据库用单节点来存储的形式，分布式协调系统的不加锁机制能更加高效地读写信息。而且其分布式的架构也能更好地分担多个节点的通信，且没有硬盘 I/O。

其次，相比存储在 HDFS 中的高可用性，分布式协调系统也有高可用性的保证，而且 Zookeeper 有一套 fail-over 机制，它既可以保证可用性，同样又没有硬盘 I/O，从而提高了性能。

第三，Zookeeper 这样的分布式协调系统使用的是最终一致性模型(paxos 算法)，适合在计数一致性要求不是特别高的场景下使用，如项 A 已经计数出现了 200 次，项 B 已经计数出现了 30 次，那么即使某个 Combiner 的 20 次 B 的计数没有写成功，也不影响 A 与 B 之间的关系，即 $A>B$。

第四，这样的设计减小了内存的开销，即每个 Map 任务的线程只会从 Zookeeper 下载它所处理的 Transaction 中所含的 Item 信息。通常这是总计数信息的很小一部分。同样，在 ParallelFPGrowth 阶段，每个 Reduce 都要用到一个 Group 的所有信息，而 Mahout 中的实现是下载所有 Group 的信息。优化之后存储在 Zookeeper 上，就可以单独下载指定 Group 的信息，这通常只是总的 Group 信息的几十分之一。

第五，计数和排序同时开始。即 3.2.2 节描述的 ParallelCounting 和 TransactionSorting 并行执行，有利于加快排序的完成。

第六，Hadoop 自身的优化。从 Mahout 的 0.5 版开始，针对 F-list 和 G-list 的存储，利用 Hadoop 中的 Distributed Cache 来进行优化。这样的机制实际是在 TaskTacker 执行任务之前，将 F-list 和 G-list 都下载到本地。但是这样的设计只能加快读取的效率，从网络开销上看，仍然要把所有的数据都下载到本地，并不能减少网络的开销。

3.4　实现的难点

基于本文设计实现优化策略存在如下难点。1)多线程编程。本文的实现涉及多线程编程，线程还会在网络通讯时发生阻塞等；2)与分布式协调系统的交互。比如创建、删除节点，更新同步数据，异步回调等。3)在后续的 ParallelFP-

Growth 过程中与 Zookeeper 的交互。因为在 ParallelFP-Growth 过程中也需要用到 F-list 的信息，所以也要考虑后续的过程中与 Zookeeper 的交互。

4 实验结果

本文设计的优化算法与 Mahout 中的 PFP-Growth 算法保持了相同的时间复杂度，主要设计实验对比了算法优化前后的空间开销。实验平台为一个 16 个节点的机群，每个节点有一颗 Intel 2.5GHz 的双核处理器，内存为 2G，使用思科路由连接在同一网段。实验机群 16 个节点中 1 个节点为 master，11 个节点为 slave，3 个节点为 Zookeeper。所有节点上使用的都是 Ubuntu 10.04 Server 操作系统和 openjdk6，实验使用的 Hadoop 版本是 0.20.203.0，Mahout 的版本是 0.4，Zookeeper 的版本是 3.3.3。在分布为较小规模和较大规模的数据集上测试了本文优化策略的性能，所采用的测试数据集分别是 T40I10D100K 和 Webdocs，大小分别是 14.8Mb 和 1.38G。其中 T40I10D100K 是 IBM 数据生成器产生的模拟数据，共有 100000 条记录，而 Webdocs 是一个真实的数据集，共有 1692082 条记录。实验中最小 support 值都设为 1%。

实验结果如表 1 和表 2 所列。

表 1 优化前后的空间开销对比(在数据集 T40I10D100K 上)

Contents	Before Tuning	After Tuning
HDFS_BYTES_WRITTEN	85207	86
FILE_BYTES_WRITTEN	19001	292
REDUCE SHUFFLE BYTES	0	0
REDUCE OUTPUT RECORDS	942	0

表 2 优化前后的空间开销对比(在数据集 Webdocs 上)

Contents	Before Tuning	After Tuning
HDFS_BYTES_WRITTEN	12656116	86
FILE_BYTES_WRITTEN	284014127	17190
REDUCE SHUFFLE BYTES	37579111	308
REDUCE OUTPUT RECORDS	5267656	0

从表 1 和表 2 中可以看到，采用优化策略后的并行算法极大地减少了 HDFS 的空间开销以及 Reduce 中 Shuffle 的空间开销。这是因为通常在机群环境下，数据会被复制产生冗余，则实际带来的空间开销和网络资源开销会更大，同时该设计也减少了计算节点的内存开销，即计算节点不用下载整个 F-list 和 G-list，为计算节点提高了性能。

结束语 综上所述，本文分析了并行频繁模式增长算法，总结了之前相关算法在设计上的优缺点，详细分析了 Mahout 项目的并行频繁模式增长实现上的不足之处。并在此基础上提出了一种新的利用分布式协调系统的优化设计。本文的设计针对计数和排序部分进行改进，使得计数和排序部分可以同时进行，排序任务能够提前开始，并减少了网络开销。本文的方法使得并行频繁模式的性能得到了提升，但是这样的设计本身也增加了一定的复杂度与实现难度，这对实现本文的设计提出了挑战。后续我们将继续本文工作，降低优化算法实现的时空复杂度，并在海量数据集上进一步验证优化后的算法性能。

参 考 文 献

[1] El-Hajj M, Zaiane O R. Parallel Leap: large-scale maximal pattern mining in a distributed environment[C]// Proceedings of the 12th International Conference on Parallel and Distributed Systems(ICPADS'06). 2006:135-142

[2] Aouad L M, Le-Khac N-A, kechadi T M. Distributed Frequent Itemsets Mining in Heterogeneous Platforms[J]. Journal of Engineering, Computing and Architecture, 2007, 1(2)

[3] Parthasarathy S, Zaki M J, Ogihara M, et al. Parallel Data Mining for Association Rules on Shared-Memory Systems[J]. Knowledge and Information Systems, 2001, 3(1)

[4] Asif J, Ashfaq K. Frequent Pattern Mining on Message Passing Multiprocessor Systems[J]. Distributed and Parallel Databases, 2004, 16(3):321-334

[5] Iko P, Masaru K. Parallel FP-Growth on PC Cluster[C]// Proceedings of the 7th Pacific-Asia conference on Advances in knowledge discovery and data mining(PAKDD'03). 2003

[6] Buehrer G, Parthasarathy S, Tatikonda S, et al. Toward terabyte pattern mining: an architecture-conscious solution[C]// Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming(PPoPP '07). ACM, New York, NY, USA, 2007:2-12

[7] Li Hao-yuan, Wang Yi, Zhang Dong, et al. Pfp: parallel fp-growth for query recommendation[C]// Proceedings of the 2008 ACM conference on Recommender systems (RecSys '08). ACM, New York, NY, USA, 2008:107-114

[8] Zhang Yu-zhou, Wang Jian-yong, Zhou Li-zhu. Parallel Frequent Pattern Discovery[J]. Challenges and Methodology, Tsinghua Science & Technology, 2007, 12(6):719-728

[9] Hunt P, Konar M, Junqueira F P, et al. ZooKeeper: wait-free coordination for internet-scale systems[C]// Proceedings of the 2010 USENIX conference on USENIX annual technical conference(USENIXATC'10). USENIX Association, Berkeley, CA, USA, 2010

[10] HDFS Architecture Guide [EB/OL]. http://hadoop.apache.org/common/docs/current/hdfs_design.html#N10159, 2011

[11] Han J, Pei J, Yin Y. Mining Frequent Patterns without Candidate Generation[C]// Proceedings of the 2000 ACM SIGMOD International Conference of Management of Data. ACM press, May 2000:1-12

(上接第 159 页)

[10] 张健沛，杨悦，杨静，等. 基于最优划分的 K-means 初始聚类中心选取算法[J]. 系统仿真学报，2009，21(9)：2586-2590

[11] Keogh E, Chu S, Hart D, et al. An On-line Algorithm for Segmenting Time Series[C]// Proc. of the IEEE International Conference on Data Mining. San Jose, USA, 2001:289-296

[12] Keogh E, Kasetty S. On Need for Time Series Data Mining Benchmarks: A Survey and Empirical Demonstration[C]// Proc of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Edmonton, Canada, 2002:102-111

[13] 汪小帆，李翔，陈关荣. 复杂网络理论及其应用[M]. 北京：清华大学出版社，2006：165-171