FiDoop-DP: Hadoop 集群环境下的频繁项集并行挖掘

荀亚玲, 张继福, 秦啸

摘要

传统的挖掘频繁项集的并行算法的目的是通过在一组计算节点之间平等地划分数据来平衡负载。我们从发现现有并行频繁项集挖掘算法的一个严重性能问题开始研究。在大数据集的情况下，现有解决方案中的数据划分策略由于计算节点之间传输的冗余事务而产生较高的通信和挖掘开销。我们通过使用 MapReduce 编程模型开发一种名为 FiDoop-DP 的数据分区方法来解决这个问题。FiDoop-DP 的首要目标是提高 Hadoop 集群上并行频繁项集挖掘的性能。FiDoop-DP 的核心是基于 Voronoi 图的数据分区技术，它利用了事务之间的相关性。FiDoop-DP 结合了相似性度量和局部敏感哈希技术，将高度相似的事务放到数据分区中，以在不创建过多冗余事务的情况下改进局部事务。实验结果表明，FiDoop-DP 通过消除 Hadoop 节点上的冗余事务，有利于减少网络负荷和计算负载。FiDoop-DP 将现有并行频繁模式方案的性能显著提高了高达 31%，平均提高了 18%。

关键词：频繁项目集挖掘；并行数据挖掘；数据分区；mapreduce 编程模型；hadoop 集群

1 引言

传统的并行频繁项集挖掘技术（即 FIM）侧重于负载平衡；数据在集群的计算节点之间进行平均分区和分布。通常，数据之间缺乏相关性分析导致数据局部性较差。数据配置的缺失增加了数据洗牌成本和网络开销，降低了数据划分的有效性。在本研究中，我们发现冗余的事务传输和项集挖掘任务很可能是由不适当的数据分区决策造成的。因此，FIM 中的数据分区不仅影响网络流量，而且还影响计算负载。我们的证据表明，数据分区算法除了应注意负载平衡的问题外，还应注意网络和计算负载。我们提出了一种使用 MapReduce 编程模型的并行 FIM 方法，称为 FiDoop-DP。FiDoop-DP 的关键思想是将高度相关的事务分组到一个数据分区中；因此，冗余事务的数量被大幅减少。重要的是，我们展示了如何在 Hadoop 集群的数据节点上划分和分配一个大型数据集，以减少通过在远程节点上进行冗余事务所引起的网络和计算负载。FiDoop-DP 有利于提高集群上并行 FIM 的性能。

1.1 FiDoop-DP 要解决的数据划分问题

在 Hadoop 集群中，在 shuffling 阶段需要传输的数据量很大程度上取决于中间结果的本地性和均衡性。不幸的是，当一个数据分区方案对中间结果进行分区时，数据的局部性和平衡性将被完全忽略。在现有的基于 hadoop 的 FIM 应用程序中，传统的数据划分方案存在重大的性能问题，原因如下：数据分区中的传统方法的目的是使用哈希函数或一组等间距的范围键来产生平衡的分区[1][2]。有趣的是，我们发现过多的计算和网络负载很可能是由并行 FIM 中不适当的数据分区造成的。图 1 提供了一个示例，显示了各种项目分组和数据分区决策及其对通信和计算负载的影响。

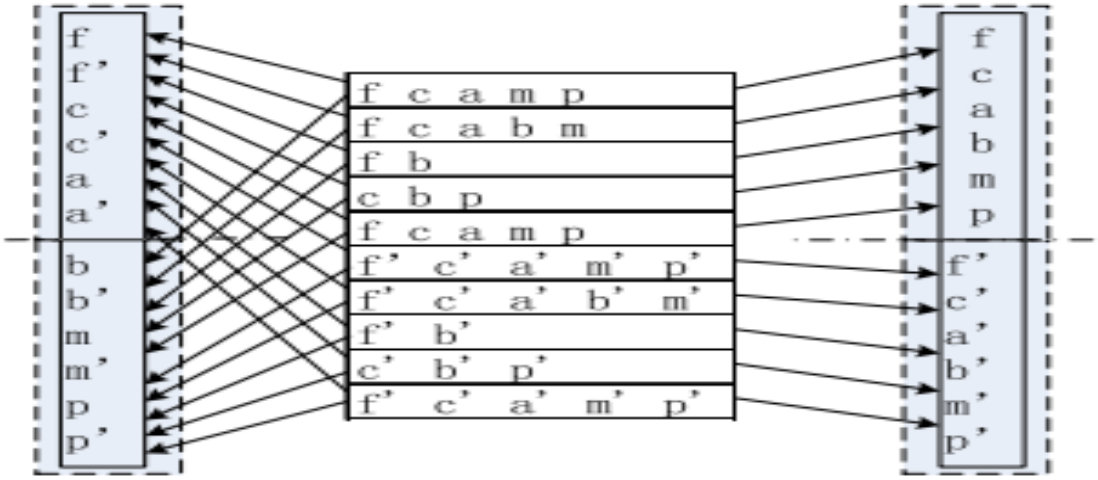

图 1: 项目分组和数据划分实例

在图 1 中，中间表中的每一行代表一个事务 (即，一共有 10 个事务。12 个项目（如 f、c、a 等）在事务数据库中进行管理（参见图 1 中的左列和右列）。请注意，这两列表示用一条中线分开的两种分组策略。传统的分组策略按下降频率法将项目均匀分为两组（见图 1 左侧一栏）。不幸的是，这个分组决策迫使用户在处理之前将所有事务传输到两个分区。我们认为，通过在跨节点网络流量和负载平衡之间进行良好的权衡，可以降低如此高的事务转移开销。在挖掘频繁项集的多阶段并行过程中，冗余挖掘任务往往出现在后期阶段。在启动并行挖掘程序之前，预测这些冗余的任务往往不难。因此，在并行挖掘过程之前执行的现有数据划分算法不足以解决冗余任务的问题。

FiDoop-DP 的首要目标是提高运行在 Hadoop 集群上的并行 FIM 应用程序的性能。这一目标是在 FiDoop-DP 中通过消除多个节点上的冗余事务来减少网络和计算负载来实现的。为了缓解图 1 中所示的过度网络负载问题，我们发现项目和事务之间的相关性创造了充足的机会来显著降低事务转移开销（见图 1 右边的一栏）。这种新的分组决策使得构建小的 FP 树成为可能，从而降低了通信和计算成本。我们将数据划分方案纳入到基于 hadoop 的频繁模式树（FP-tree）算法中。图 2 概述了我们的 FiDoop-DP 处理过程，它包括四个步骤。在这个流程图中，我们优化了第二个 MapReduce 作业的数据分区策略，因为它是 FiDoop-DP 中最复杂、最耗时的作业。在第二个 MapReduce 作业中，各 mapper 将频繁的 1 项集（图 2 中的 FList）划分为 Q 组，同时根据分组信息将事务分配给计算节点。然后，各 reducer 对其划分到的数据建树并完成挖掘任务。

1.2 并行 FP-growth 算法

在这项研究中，我们重点关注一种流行的 FP-Growth 算法，称为并行 FP-Growth 或 Pfp。FP-Growth 通过构造和挖掘压缩的数据结构（即 FP-tree）而不是整个数据库来有效地发现频繁项集。Pfp 的设计目的是通过将事务数据库划分为独立分区, 因为它保证每个分区包含与该组的特性（或项）相关的所有数据。

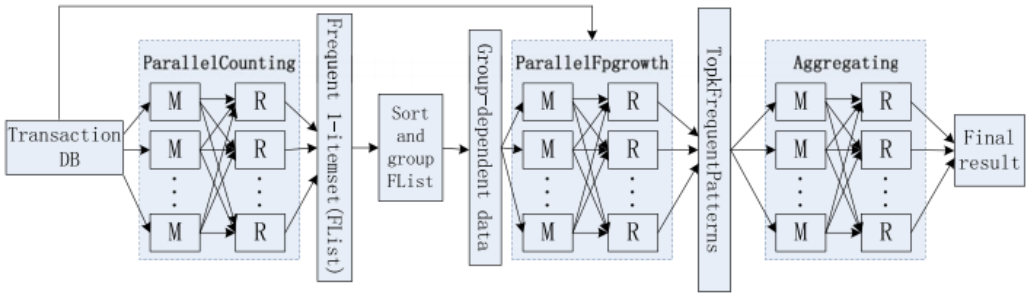

图 2: pfp 实现流程图

给定一个事务数据库数据库，图 2 描述了并行 FP-Growth 的流程。并行算法包括四个步骤，其中三个步骤是 MapReduce 作业。

步骤 1。并行计数：第一个 MapReduce 作业对驻留在数据库中的所有项的支持值进行计数，以并行地发现所有频繁的项或频繁的 1 项集。值得注意的是，此步骤只是扫描数据库一次。

步骤 2。将频繁 1 项集排序为 FList：第二步按频率递减排序；排序的频繁 1 项集缓存在一个名为 FList 的列表中。步骤 2 由于其简单性和集中式控制，是一个非 MapReduce 过程。

步骤 3。并行 fp 增长：这是 Pfp 的核心步骤，其中 mapper 阶段和 reducer 阶段执行以下两个重要功能。mapper 程序-分组项目并生成与组相关的事务。首先，mapper 程序将 FList 中的所有项划分为 Q 组。组的列表被称为组列表或 GList，其中每个组都被分配了一个唯一的组 ID（即 Gid）。然后，根据业务列表将事务划分为多个组。也就是说，每个 mapper 输出一个或多个键-值对，其中一个键是一个组 ID，其对应的值是一个生成的依赖于组的事务。在组依赖的分区上的减少 fp 增长。通过执行本地 FP-Growth 来生成本地频繁的项目集。每个 reducer 通过逐个处理一个或多个依赖于组的分区来执行本地 FP-Growth，并将发现的模式在最终输出。

步骤 4。聚合：最后一个 MapReduce 作业通过聚合在步骤 3 中生成的输出来生成最终结果。

第二个 MapReduce 作业（即步骤 3）是整个数据挖掘过程的性能瓶颈。map 任务应用第二轮扫描，根据 FList 对每个事务进行排序和修剪，然后将 FList 中排序的频繁 1 项集分组，形成组列表 GList。接下来，将每个事务放入一个组依赖的数据分区中；因此，构造了多个数据分区。每个数据分区对应于由 Gid 标识的一个组。上述分区方法确保了对一组 GList 的数据完整性。

另一个缺点是，这种数据的完整性是以数据冗余为代价的，因为一个事务可能在多个数据分区中有重复的副本。毫不奇怪，数据分区中的数据冗余是不可避免的，因为必须保持分区之间的独立性，以最小化同步开销。冗余事务会导致过多的数据传输成本和本地 fp-growth 的计算负载。

1.3 基本方法概述与问题阐述

文献 [4] 表明[3]，大多数现有的并行 FP-Growth 算法基本遵循图 2 所示的处理流程，其中第二个 MapReduce 作业耗时最多。其实验结果也指出:(1) 局部 fp-growth 开销占整体挖掘成本的 50% 以上，(2) 分组策略是影响后续的数据分区和本地 fp-growth 性能的最重要的因素。利用分组策略将重新排序后的事物数据集被划分并分配给相应的 reducers，并将这些事物插入相应的 Fp-tree。也就是说，分组策略不仅直接影响着 partition 阶段的数据传送量，也影响局部 Fp-Growth 阶段的计算量。因此，我们建议在运行分组和分区策略之前，先将输入数据进行聚集操作。该数据划分策略考虑到事物之间的相关性以优化分组处理。

传统的基于 MapReduce 的 FIM 算法采用了 Hadoop 中缺省实现的的数据分区策略, 且采用了一个简单的分组策略。该分组策略将 FList 中频繁 1-项集个数除以组的个数，从而确定每个组的大小，即该分组策略是一种平均分配方法。

BPFP 算法 (the balanced parallel Fp-Growth algorithm) 是对分组和分区策略的改进，其将负载均衡思想结合到 Pfp 算法中，对 Pfp 做了进一-步改进。BPFP 以平衡负载的方式将 FList 中所有项目分割成 Q 个分组。BPFP 使用在 Fp-Growth 执行过程中的迭代次数来估计挖掘负载，其输入是每个项目的条件模式基。在 FList 中每个项目的位置被认为是该项目在条件模式基的最长路径的长度。同时，迭代次数是正比于条件模式基的最长路径的。因此，项目 i 的负载被估计为 T = logL, 其中 T 表示所估计的负载，L 表示项目 i 在 FList 中的位置。由以上描述可以看出，BPFP 只关心每个节 i 点 CPU 资源的计算均衡性，它只是按计算负载均衡性将 FList 中的项目分成 Q 组[4]。然而，图 2 显示如果在分组时不考虑事物之间的相关性，大量的事物将被重复拷贝给集群中的多个节点来保证相对于每个组的数据完整性。换句话说，参与传输和计算的事务量不可避免地增加，数据传输开销 (即 shufling 成本) 和局部 Fp-Growth 负载将显著增加。

2 本文方法

2.1 本文方法概述

在第二部分，我们了解到 pfp 算法的第二个阶段是整个性能的瓶颈，如果能够改进这一阶段，那么就可以提升性能。在这个阶段执行之前，我们先考虑事务和项目之间的相关性，从而划分事务。将非常相似的事务划分到一个分区中，以防止事务重复传输到远程节点。本文采用了基于 Voronoi 图的数据划分技术，Voronoi 图是将某个空间分割成若干区域的技术，在构造 Voronoi 图时，一组种子点被预先指定，对于一个区域，若其中的每一个点到它所在区域的种子点的距离比到其他区域种子点的距离近，则该区域称为该种子点的 Voronoi 区域。

2.2 距离度量

为了衡量两个对象之间的相关性，本文采用 Jaccard 相似度作为距离度量，数据库中的每条事务被看作是一个集合，集合 A 和集合 B 之间的 Jaccard 相似度定义为：

$$J(A,B)=\frac{|A\cap B|}{|A\cup B|}.$$

J(A,B) 是一个介于 0 和 1 范围的值；当两个集合互不相交时，J(A,B) 取 0，当两个集合完全相同时为 1，否则，取 0 和 1 之间的任意值，两个集合的 Jaccard 指数越接近 1 时，两个集合之间的距离越近，反之，越远。

2.3 K-means 枢轴的选择

选择枢轴直接影响 voronoi 基于图划分的剩余对象的均匀性系数。我们使用基于 k-means 的选择策略来选择枢轴。k -means 聚类的目的是将 n 个对象划分为 k 组。其中每个对象属于具有最近均值的分组。当 k 个聚类被产生后，每个聚类的中心点将被作为基于 voronoi 图数据划分的种子点。当种子点选择好后，需计算每个对象到这些种子点的距离，以确定各对象所属的分区，我们开发了基于 LSH

的分组和分区策略。在此之前，我们先了解一下 minhash 技术。

2.4 minhash 和 LSH

minhash 是 LSH 的一种，它可以用来快速估算两个集合的相似度[5]。它能在保持原数据相似性的情况下将原始大集合压缩成更小的集合，进而缩短计算时间，即 minhash 采用更小的由“特征矩阵”（原始矩阵）“minhash”后形成的“签名矩阵”来计算两个数据集集合间的相似度。但当数据量非常大时，minhash 就不适用了。此时采用局部敏感哈希—-LSH，LSH 不像 minhash 那样需要反复比较数量过多的元素对，而是仅扫描数据库，就能确定所有可能相似的对[6][7]。LSH 将特征空间中的事物映射到若干桶中，且相似的事物将会被映射到相同的桶中。其形式化定义如下：对于 hash 族 H，如果任意两点 p 和 q 满足如下条件，那么 H 被称为（R,c,P1,P2）-敏感的

$$\text{If} \parallel p-q \parallel \leq R, \text{then } Pr_H(h(p)=h(q)) \geq P_1.$$
$$\text{If} \parallel p-q \parallel \leq cR, \text{then } Pr_H(h(p)=h(q)) \leq P_2.$$

当 P1 ＞ P2 时，该 hash 族才有意义。上述条件 1) 确保原本相似的两个点可以以较大概率映射到相同的桶内[8]；条件 2) 确保原本不相似的点以极小的概率被映射到同一个桶内。

3 复现细节

本次复现主要实现了 pfp 算法，也就是前面提到的四个过程。代码结构如下图所示：

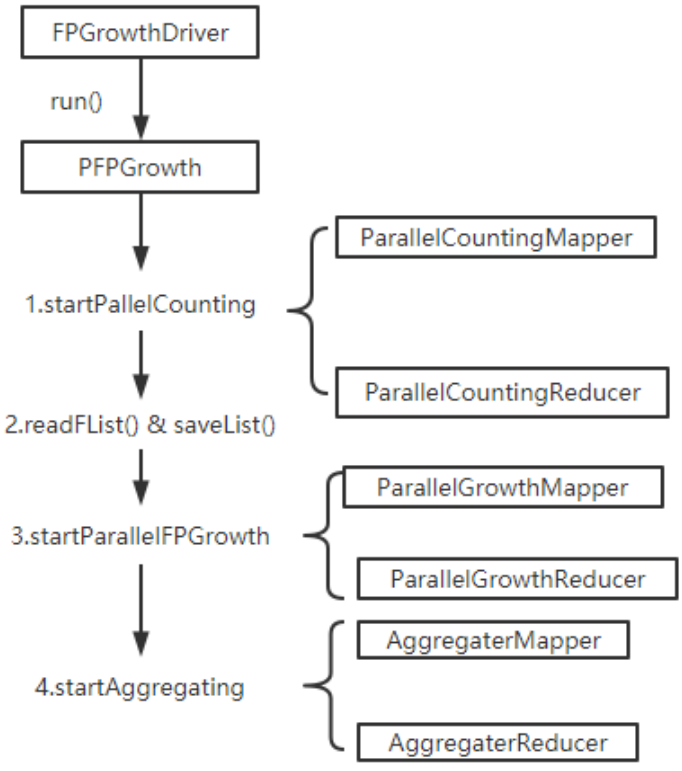

图 3: 代码结构图

从驱动类 Driver 开始，调用 run 方法。run 方法中主要是解析命令行传入的参数，参数有-s，-k,-g,-method 等，-s 表示支持度阈值，-k 是要求出的前 s 个频繁项。-g 表示分组数,-method 表示执行的方式是串行还是并行。PFPGrowth 是并行化 FP-Growth 算法的驱动类。runPFPGrowth(params) 方法内初始化了一个 Configuration 对象，之后调用 runPFPGrowth(params, conf) 方法。runPFPGrowth(params, conf) 方法包括了并行化 FP-Growth 算法的五个关键步骤。其中，startParallelCounting(params, conf) 对

应 Step1 和 Step2，通过类似 WordCount 的方法统计每一项的支持度，其输出结果将被 readFList() 和 saveList() 用于生成 FList 之后，将按照用户输入的命令行参数 NUMGROUPS 来计算每一个 group 所含项的个数，并将其存储到 params。startParallelCounting 方法初始化了一个 Job 对象，该 Job 对象调用 ParallelCountingMapper 和 ParallelCountingReducer 来统计支持度。第三步是整个算法阶段的关键步骤，parallelgrowthmapper 和 parallelgrowthreducer 代码分别如下：

```
@Override
protected void map(LongWritable offset, Text input, Context context)
  throws IOException, InterruptedException {

  String[] items = splitter.split(input.toString());

  OpenIntHashSet itemSet = new OpenIntHashSet();

  for (String item : items) {
    if (fMap.containsKey(item) && !item.trim().isEmpty()) {
      itemSet.add(fMap.get(item));
    }
  }

  IntArrayList itemArr = new IntArrayList(itemSet.size());
  itemSet.keys(itemArr);
  itemArr.sort();

  OpenIntHashSet groups = new OpenIntHashSet();
  for (int j = itemArr.size() - 1; j >= 0; j--) {
    // generate group dependent shards
    int item = itemArr.get(j);
    int groupID = PFPGrowth.getGroup(item, maxPerGroup);

    if (!groups.contains(groupID)) {
      IntArrayList tempItems = new IntArrayList( initialCapacity: j + 1);
      tempItems.addAllOfFromTo(itemArr, from: 0, j);
      context.setStatus("Parallel FPGrowth: Generating Group Dependent
      wGroupID.set(groupID);
      context.write(wGroupID, new TransactionTree(tempItems, support:
    }
    groups.add(groupID);
  }

}
@Override
```

```
@Override
protected void reduce(IntWritable key, Iterable<TransactionTree> values, Context context) throws IOExcep
  TransactionTree cTree = new TransactionTree();
  for (TransactionTree tr : values) {
    for (Pair<IntArrayList,Long> p : tr) {
      cTree.addPattern(p.getFirst(), p.getSecond());
    }
  }

  List<Pair<Integer,Long>> localFList = Lists.newArrayList();
  for (Entry<Integer,MutableLong> fItem : cTree.generateFList().entrySet()) {
    localFList.add(new Pair<Integer,Long>(fItem.getKey(), fItem.getValue().toLong()));
  }

  Collections.sort(localFList, new CountDescendingPairComparator<Integer,Long>());

  if (useFP2) {
    org.apache.mahout.fpm.pfpgrowth.fpgrowth2.FPGrowthIds fpGrowth =
      new org.apache.mahout.fpm.pfpgrowth.fpgrowth2.FPGrowthIds();
    fpGrowth.generateTopKFrequentPatterns(
        cTree.iterator(),
        freqList,
        minSupport,
        maxHeapSize,
        PFPGrowth.getGroupMembers(key.get(), maxPerGroup, numFeatures),
        new IntegerStringOutputConverter(
            new ContextWriteOutputCollector<IntWritable,TransactionTree,Text,TopKStringPatterns>(context
            featureReverseMap),
        new ContextStatusUpdater<IntWritable,TransactionTree,Text,TopKStringPatterns>(context));
  } else {
    FPGrowth<Integer> fpGrowth = new FPGrowth<~>();
    fpGrowth.generateTopKFrequentPatterns(
        new IteratorAdapter(cTree.iterator()),
        localFList,
        minSupport,
        maxHeapSize,
        new HashSet<Integer>(PFPGrowth.getGroupMembers(key.get(),
                                                       maxPerGroup,
                                                       numFeatures).toList()),
        new IntegerStringOutputConverter(
            new ContextWriteOutputCollector<IntWritable,TransactionTree,Text,TopKStringPatterns>(context
            featureReverseMap),
        new ContextStatusUpdater<IntWritable,TransactionTree,Text,TopKStringPatterns>(context));
  }
}

@Override
```

图 4: 代码图

第三步中 mapper 会读取 Glist,Glist 是一张 hashmap，键是项，值是项所对应的 group-id，所有 group-id 相同的数据会被推送到同一个 reducer.reducer 在本地递归构建条件 fp-tree, 并挖掘频繁模式，第四步归约阶段依次将频繁模式每一个 item 作为 key，然后输出包含该 key 的这一条频繁模式。所以，每一个 mapper 的输出是 <key=item, value= 该节点上的包含该 item 的频繁模式 >，reducer 汇总所有 mapper 的输出结果，并输出最终的结果。本代码参考了 mahout 里面的代码。

4 实验结果分析

由于原文中使用的数据集是 IBM Quest Market-Basket Synthetic Data Generator 生成的，年代久远，故我采用的数据集是实验室中的数据集。总共做了两个实验，第一个实验是测试不同的支持度大小对该算法的影响，数据集大小为 20M 时，只关注第二个 mapreduce 作业的性能，当支持度大小分别为 0.0005%，0.001%，0.0015%，0.002%，0.0025% 时，其运行时间分别如下：

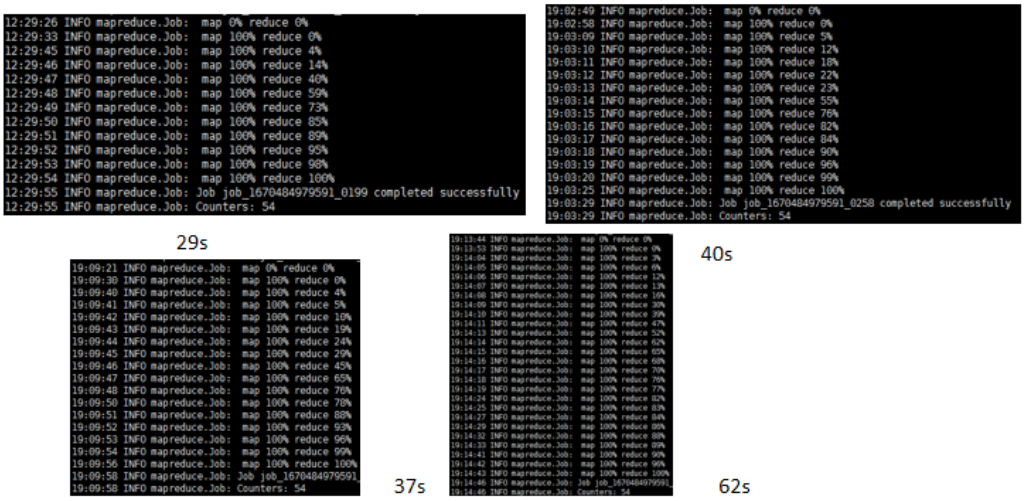

图 5: 运行时间实验结果图

原文中当支持度不断增大时，pfp 的算法运行时间逐渐减小，这里的实验数据有所波动，是因为论文中的数据集比较大，产生的时间差异比较明显，而这里的数据集较小，看不出来区别。但在不同支持度下产生的 FList 大小趋势与原文一致，实验结果如下图所示:

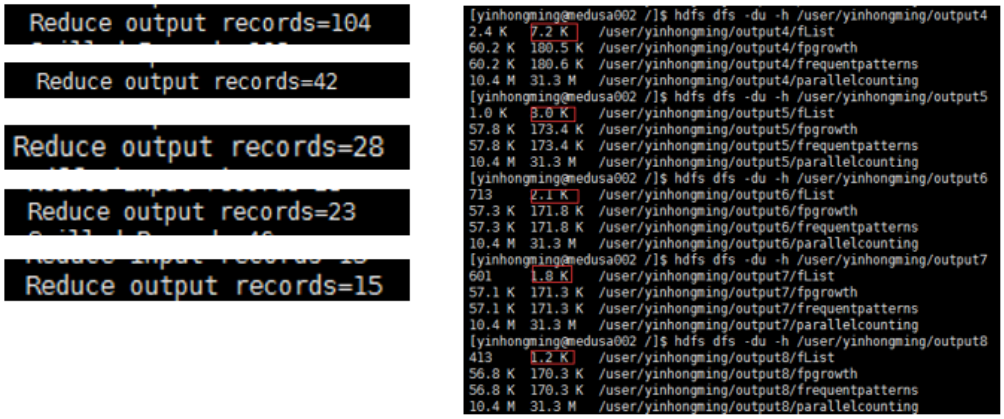

图 6: FList 大小实验结果图

整理成列表并与文中实验结果对比如下：

minsupport	0.0005%	0.001%	0.0015%	0.002%	0.0025%
FList	7.2K	3.0K	2.1K	1.8K	1.2K
OutRecords	104	42	28	23	15

Table 4.2 FList size and output record number for different minimum support

minsupport	0.0005%	0.001%	0.0015%	0.002%	0.0025%
FList	14.69k	11.6k	9.71k	6.89k	5.51k
OutRecords	745	588	465	348	278

图 7: 对比图

和原文的数据趋势相同，随着最小支持度的增加，pfp 算法会过滤掉更多的频繁一项集，由此降低事物划分代价，减少了事物在分区间的重复事物传输量。因此 FList 和 outrecords 不断变小。第二个实验是测试数据集大小对 pfp 算法的影响，还是只关注第二个 mapreduce 阶段，当数据集分别为 20M,30M,40M,50M 时，其运行时间如下：

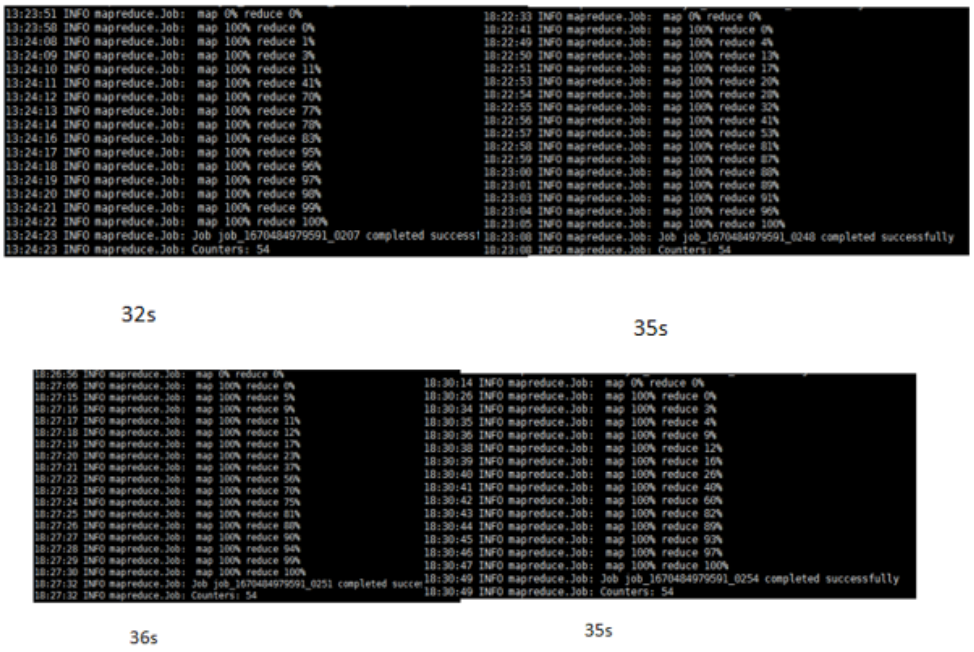

图 8: 不同数据集对算法的影响

原文中随着数据集的不断扩大，算法的运行时间会不断增加，此次测试的数据趋势大致一致。这意味着数据集增大使算法需要多次扫描多余的数据集，从而导致并行挖掘过程减慢，即增加的数据集会增加扫描时间。

5 总结与展望

本部分对整个文档的内容进行归纳并分析目前实现过程中的不足以及未来可进一步进行研究的方向。为了更好的挖掘频繁项集，我们分析了 pfp 算法并找出了其中耗时较长和开销较大的阶段，即第二个 mapreduce 阶段。通过对第二个 mapreduce 阶段进行改进，即本文采用的是基于 Voronoi 图的数据分区，将高度相似的事务划分到一个分区，进而减少冗余事务的传输，从而减小运行时间和开销。在这项研究中，我们引入了一个相似度度量来促进数据感知的分区。作为未来的研究方向，我们可以考虑用该指标来研究异构 Hadoop 集群上的高级负载平衡策略。其次，在本文中，我们主要进行了两个实验，分别研究了支持度和数据集大小对算法的影响。第一个实验的数据有点瑕疵，在未来可以采用更大的数据集进行测试，并分析实验结果。此外，我们还可以做一组与事务相关的实验，将事务相关性大小考虑进去，进一步分析算法的运行时间等因素。

参考文献

[1] CURINO C, JONES E P C, ZHANG Y, et al. Schism: a workload-driven approach to database replication and partitioning[J]., 2010.

[2] BENJAMAS N, UTHAYOPAS P. Impact of I/O and execution scheduling strategies on large scale parallel data mining[C]//2012 6th International Conference on New Trends in Information Science, Service Science and Data Mining (ISSDM2012). 2012: 654-660.

[3] ZHOU L, ZHONG Z, CHANG J, et al. Balanced parallel fp-growth with mapreduce[C]//2010 IEEE youth conference on information, computing and telecommunications. 2010: 243-246.

[4] ZHOU L, ZHONG Z, CHANG J, et al. Balanced parallel fp-growth with mapreduce[C]//2010 IEEE youth conference on information, computing and telecommunications. 2010: 243-246.

[5] LESKOVEC J, RAJARAMAN A, ULLMAN J D. Mining of massive data sets[M]. Cambridge university press, 2020.

[6] STUPAR A, MICHEL S, SCHENKEL R. Rankreduce–processing k-nearest neighbor queries on top of mapreduce[J]. Large-Scale Distributed Systems for Information Retrieval, 2010, 15: 3.

[7] BAHMANI B, GOEL A, SHINDE R. Efficient distributed locality sensitive hashing[C]//Proceedings of the 21st ACM international conference on Information and knowledge management. 2012: 2174-2178.

[8] PANIGRAHY R. Entropy based nearest neighbor search in high dimensions[J]. arXiv preprint cs/0510019, 2005.