华中科技大学

# 大数据课程实验报告

**标题：大数据下的交易排序统计与结果聚类分析**


组　　员：　　徐永鑫 U201610002

组　　员：　　陈佳杰 U201614678

组　　员：　　叶翔宇 U201611143

学　　院：　　计算机科学与技术学院

班　　级：　　计算机卓越 1601 班

指导教师：　　莫 益 军


2018 年 12 月 29 日

# 摘要

本文基于选题二的任务要求，生成了数据规模为 1MB~10GB 的多组数据集合，将它们存放在 Hadoop 分布式文件系统中。并分别对组数据集进行了排序和聚类的操作，并将获得的结果写回 Hadoop 分布式文件系统中。

在课程实验的过程中，小组成员首先调查研究了 Hadoop 的框架、逻辑以及 MapReduce 的工作原理，确定了需要用到的技术以及工具。在完成调研之后着手调研环境的搭建，首先在本机上进行了伪分布环境的搭建和测试，后利用三台云主机进行了集群环境搭建和测试。

在环境搭建完成后，小组成员完成了排序和聚类的 Mapper 和 Reducer 的算法设计。首先利用管道模拟测试检验程序的正确性，后在伪分布和集群环境上进行了综合测试，在测试过程中遇到了内存分配不足的问题。小组成员使用了配置文件修改、程序优化以及迭代次数减少等手段，最终得以运行 10GB 数据级别的排序程序以及 2GB 数据级别的 Kmeans 聚类程序。

在运行数据集成功之后，小组成员继续针对于 1GB 的排序程序进行参数调整优化，在修改文件分块大小、并行处理线程数等参数后，将原本 180 秒左右的运行时间缩短到 70s 左右。

最后，小组成员总结了整个实验过程，记录心得，写成此报告。

小组 GitHub(Issue 内有调研过程): https://github.com/csee1601/big_data_project

# 组员分工

| | |
|---|---|
| 徐永鑫 | 全局调研、数据集生成、集群环境搭建、程序编写、管道伪分布集群测试、参数调优，性能测试。<br>报告中负责技术指导，摘要与系统优化部分，统筹排版审核。 |
| 陈佳杰 | Hbase 与 Spark 调研、辅助配置环境、报告逻辑设计。报告中负责问题定义、配置总结、测试总结、排序算法介绍、展望等。 |
| 叶翔宇 | MR 流程与细节研究、数据集取样、报告逻辑设计。<br>报告中负责 Kmeans 算法介绍、MR 原理介绍、作图、总结等。 |

# 目　录

# 1 任务描述

## 1.1 问题定义

**统计与排序：**

将 10GB 左右的数据存入分布式存储系统中，通过 MapReduce 任务，统计某一确定年份（2017 年）销售总额排序，得到 20 个省份 2017 年单年的销售总额的数据后进行排序，排序结果存入分布式存储系统中。

**聚类：**

将 10GB 左右的数据存入分布式存储系统中，通过 MapReduce 任务根据销售单价和销售额两个维度进行聚类处理，得到聚类的中心样本，并可视化聚类结果，对异常数据进行分析。

## 1.2 数据集描述

随机生成若干条交易信息记录，记录包含省份、年份、单价和销售量，如下表 1-1 所示：

表 1-1 数据集举例

| 省份 | 年份 | 销售量 | 销售单价 |
|---|---|---|---|
| beijing | 2016 | 1612 | 8.03 |
| tianjin | 2017 | 1134 | 9.2 |
| shandong | 2018 | 2765 | 8.41 |
| hubei | 2017 | 6500 | 2.32 |
| hebei | 2017 | 123 | 21.23 |

为了便于聚类，随机生成了异常销售数据，在生成销售量和销售单价时作特殊处理。随机生成过程中，每 10 条销售信息中，有 2 条异常数据。

其中，有 1 条信息的销售量在(100,200)之间和销售单价在(20,25)之间，有 2 条信息的销售量在(6000,9000)之间和销售单价在(1,3)之间，这些信息作为异常数据，方便之后的聚类处理。其他的正常数据的销售量在(2000,3000)和销售单价在(5,10)之间。

数据生成了 8 组，从 1M 到 10G 之间都有，典型的数据集有 100M、500M、1000M 和 100000M。

## 1.3 实验环境

表 1-2 实验环境信息表

| 属性 | 信息 |
| --- | --- |
| 主机版本 | CentOS 7.4.1708 |
| 内存 | 8GB |
| 主机数量 | 3 台 |
| VCPU 数量 | 4 VCPU |
| 磁盘 | 100GB |
| Hadoop 版本 | Hadoop 2.7.6 |
| Java 版本 | 1.8.1_45 |

# 2 总体设计

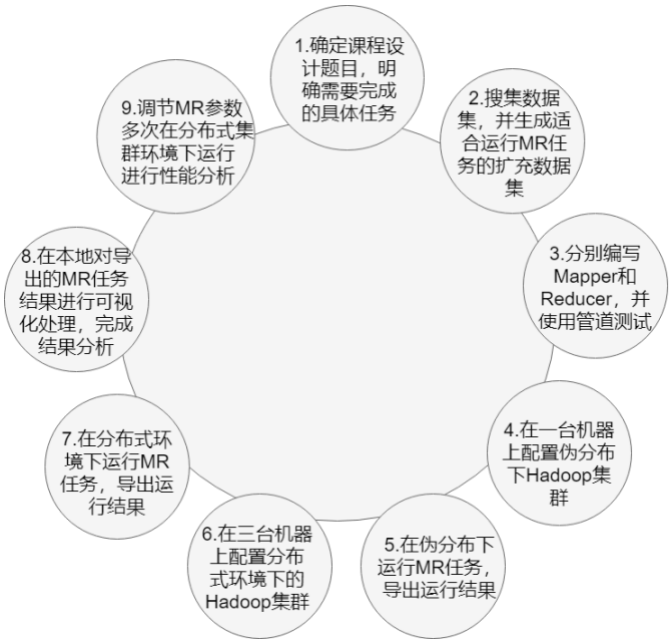


图 2-1 工作流程设计

工作流程概述如上图 2-1，为了完成本次课程设计，课设小组首先对课程设计提供的选题进行调研，最终确定本小组的课程设计题目为“**大数据下的交易排序统计与交易结果聚类分析**”。

完成选题后，针对选择课题内容与课题中涉及的数据体征，课程设计小组明确了该课题所包含的两项具体任务：对交易数据进行按总成交金额的排序；对交易数据进行按成交量和单笔成交额的聚类。

确定选题并明确任务要求后，开始进行数据的搜集。这里课程小组为了保证课程设计的有效推进，创建了多数量级的数据作为实验数据。

准备工作完成后，着手进行算法设计与程序编写。由于本次课程设计涉及两个 MR 任务：排序与聚类，故针对不同任务编写了两套 Mapper 程序与 Reducer 程序。完成程序编写后利用管道在小数据集下进行模拟测试，检验程序的正确性。

为了保证程序在分布式集群环境下的正常运行，课程小组首先在一台机器上进行伪分布下的 Hadoop 集群配置（具体配置方法见附录），并在伪分布下使用小数据集分别运行通过管道模拟正确性检测的两套 MR 程序，比较结果是否符合预期。

完成伪分布下的 Hadoop 配置与 MR 任务运行后，参考伪分布配置过程在三台机器上配置分布式 Hadoop 集群环境（具体配置方法见附录）。完成配置后使用大数据集分别运行两套 MR 程序，并将运行的结果导出至本地。

在本地对导出的 MR 任务结果进行数据可视化操作，并进行结果分析。通过调节 MR 任务参数，多次在分布式集群环境下运行 MR 任务，对比不同参数下 MR 任务完成时间，对任务进行性能分析。

# 3 详细设计

## 3.1 环境搭建

### 3.1.1 Hadoop 配置背景

Hadoop 集群中每一个计算节点都有自己的配置文件，集群特定的配置文件如下表 3-1 所示：

表 3-1 Hadoop 配置文件

| 配置文件 | 作用 |
| --- | --- |
| core-site.xml | 核心 Hadoop 配置文件 |
| hdfs-site.xml | HDFS 属性配置文件 |
| Mapred-site.xml | MapReduce 属性配置文件 |
| yarn.xml | YARN 属性配置文件 |

在本次集群搭建过程中，我们选择主机 xu 作为 master 节点，主节点有 Namenode、ResourceManager、SecondaryNameNode 三个主要守护进程，子节点 chen 和 ye 有 DataNode 和 NodeManager 两个守护进程。

### 3.1.2 hadoop 搭建流程


图 3-1 Hadoop 配置流程

设置 hosts 让 xu，chen，ye 成为各主机 IP 地址的别名，方便后续配置文件和控制台指令的操作。

将三台主机设置成 SSH 免密登录，让节点之间数据和信息交换时可以通过秘钥会话直连。

安装并配置 Java 和 Hadoop 的本地环境变量。修改配置文件，

让 Hadoop 以我们指定的集群模式配置并启动。启动 HDFS 和 YARN，查看守护进程，观察 matser 节点和 slaves 节点是否正常运行。运行例子程序调试，观察程序是否执行成功。

安装完成后，需要通过修改前文介绍的四个基本配置文件来搭建集群环境，设置 Master 节点和 Slaves 节点的属性，让主节点的 NameNode 和子节点的 DataNode 能够正常进行 MapReduce 任务。主要配置文件简介如下。（注：后期会有参数性能调优）core-site.xml 是 Hadoop 的核心配置，其目的主要是对 master 节点的设置，配置 master 节点的 IP 地址和端口号，让各个节点确定主节点 NameNode 的存在。

### 3.1.3 Hadoop 配置文件介绍

**core-site.xml**

core-site.xml 是 Hadoop 的核心配置，对 master 节点的设置，配置 master 节点的 IP 地址和端口号，让各个节点确定主节点 NameNode 的存在。

**hdfs-site.xml**

hdfs-site.xml 主要负责记录 hdfs 文件系统在 NameNode 和 DataNode 中具体的存放位置，确定元数据存放的本地位置。

**Mapred-site.xml**

Mapred-site.xml 主要是设置 MapReduce 的运行框架和运行方式，在本集群中指定 MR 框架为 YARN 方式，基于资源管理系统 YARN。还可以设置运行 MR 任务时的属性，比如内存大小，MapReduce 的 task 分配等。

**yarn-site.xml**

yarn-site.xml 主要是制定节点资源管理器 NodeManager 以及总资源管理器 ResourceManager 的配置，另外还配置了 NodeManager 和 ResourceManager 的通讯地址。

## 3.2 Map-Reduce 原理

### 3.2.1 MapReduce 的产生

MapReduce 概念来自于“谷歌三大论文”中的最后一篇《MapReduce: Simplified Data Processing on Large Clusters》，并作为算法模型建立在另外两篇论文中分别提出的文件系统 Google File System(GFS)和数据模型 Bigtable 之上应用于大规模集群环境下的分布式计算问题，三者之间的关系如下图所示。


图 3-2 大数据 MR 模型关系

Apache Hadoop 是一个基于 MapReduce 原理对大数据的一种解决方案，目前为止，经历了两个大阶段，第一个是 MR1.0，是对 MapReduce 的基本实现，在 MR2.0(又称 YARN)中，对 MR 核心进行了很大的改动，支持了更多语言和框架。

### 3.2.2 YARN 的守护进程

YARN 主要的守护进程总共包含 4 个，分别是 **Resource Manager、Scheduler、Application Master** 和 **Node Manager。**

**Resource Manager**

RM 是一个全局的资源管理器，集群只有一个，负责整个系统的资源管理和分配，包括处理客户端请求、启动/监控 APP master、监控 Node Manager、资源的分配与调度。它主要由两个组件构成:：Scheduler 和 Applications Manager。

**Scheduler**

调度器 Scheduler 负责根据容量、队列等限制条件将 Container 包含的系统资源分配给各个正在运行的应用程序。。

**Application Master**

AM 负责管理 YARN 内运行的应用程序的每个实例。为应用程序申请资源并进一步分配给内部任务。负责协调来自 Resource Manager 的资源，并通过 Node Manager 监视容易的执行和资源使用情况。

**Node Manager**

Node Manager 整个集群有多个，负责每个节点上的资源和使用。处理来自于 Resource Manager 和 Application Master 的命令。管理由 Resource Manager 分配的 Container 并向 Resource Manager 定时汇报所在节点上的资源使用情况和各个 Container 的运行状态。

### 3.2.3 MapReduce 的过程

MapReduce 过程的核心操作时 Map 与 Reduce。集群环境下的并行计算中，大部分 Map Task 与 Reduce Task 的执行是在不同节点上的。Reduce 执行时需要跨节点地去拉取其它节点上的 Map Task 结果。

Map，Reduce 和 Shuflle 三个过程的官方示意图如下图所示。

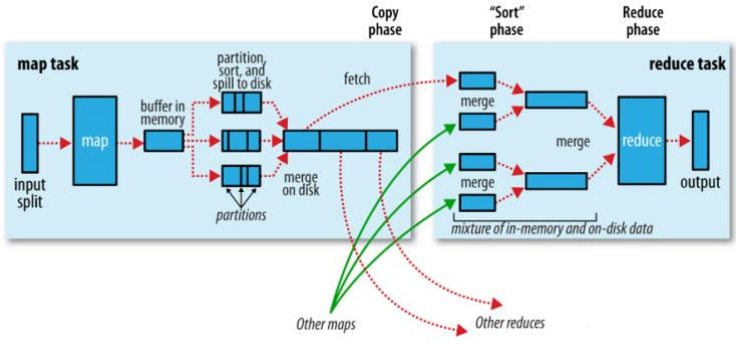


图 3-3 MapReduce 过程示意图

**Map**

在 Map Task 执行时，对其输入来源 HDFS 的 Block，Map Task 只读取 Split。Split 与 Block 的对应关系默认为一对一，这一过程称之为 Input 过程。Map Worker 的输出是 Key/Value 对，然后将数据写入内存缓冲区中，缓冲区的作用是批量收集 Map 结果，减少磁盘 IO 的影响。写入缓冲区之前，Key 与 Value 值都会被序列化成字节数组，这一过程称之为 Partition 过程。

**Shuflle**

从内存往磁盘写数据的过程被称为 Spill。Spill 是由单独线程来完成，不影响往缓冲区写 Map 结果的线程。在 Reduce Worker 从 Map Worker 取数据之前 Map Worker 会对具有相同 Key 的 Key/Value 对进行 Combine 操作，即将其相同 Key 的 Key/Value 相加。当 Map 很大时，每次 Spill 会产生一个 Spill File，最终的输出只有一个文件，在最终输出之前会对多个中间过程多次产生的 Spill 文件 Spill file 进行合并，此过程被称为 Merge。

上述过程都是在 Map Worker 中完成的，故称之为 Map 端的 Shuflle 操作。

**Reduce**

Reduce 过程就是在 Shuflle 过程完成 Copy 过程和 Merge 过程后，从 Reduce Worker 的内存或磁盘中读取价值对做归并操作，Reduce 的具体功能由 User Program 确定，用以完成最终的集群环境下的并行计算任务。

### 3.2.4 MapReduce 的实例——Word Count

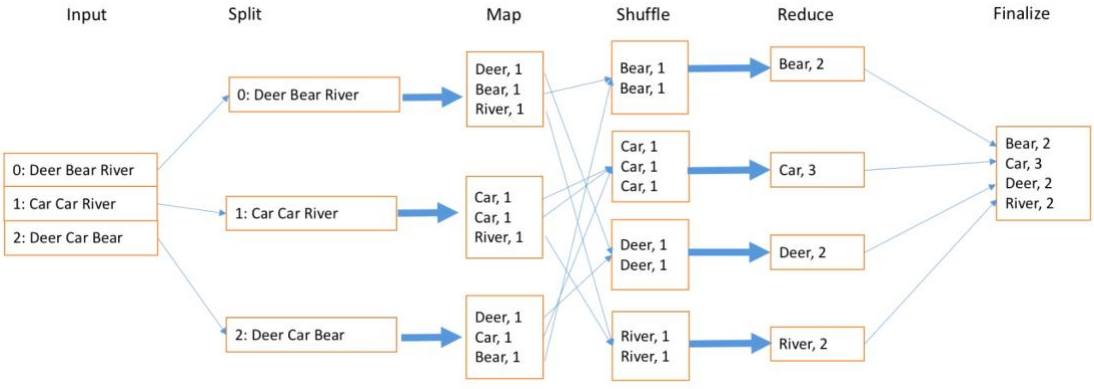


图 3-4 wordcount 实例示意图

下面通过 MR 的原理，分析 MapReduce 的“HelloWorld”程序“Word Count”。

Input：输入为一个有多行的文本文件，内容为由英文单词组成的句子；

Split：将文本数据的每一行切分成一个 Block，作为 Map 过程的输入；

Map：没个 Map Worker 统计取得的 Block 中每个单词出现的频数，生成键值对；

Shuflle：在数据被 Reduce Worker 取得之前，对 Map 过程得输出数据进行简单得归并以减少单一节点上内存和磁盘交换的次数，节约网络传输成本；

Reduce：把所有值相同的键值对相加，实现的数据的归并，得到每个单词在全文档中出现的频数；

Finalize：把 Reduce 过程的输出取得作为最终 MapReduce 任务的输出结果，完成 Word Count 任务。

数据集为若干条<省份，年份，销售量，销售单价>的销售信息。

数据生成：对于 20 个不同省份随机生成 2016 年，2017 年，2018 年的销售信息。

## 3.3 大数据排序

### 3.3.1 任务要求

数据集为若干条<省份，年份，销售量，销售单价>的销售信息。

数据生成：对于 20 个不同省份随机生成 2016 年，2017 年，2018 年的销售信息。

### 3.3.2 算法实现

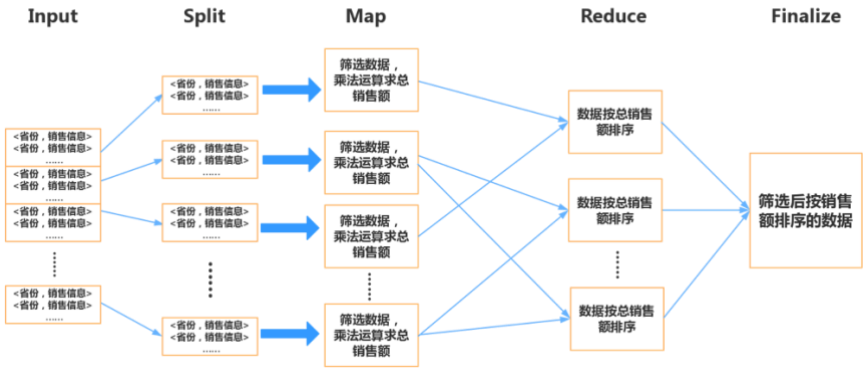


图 3-5 排序算法实现明细

**Mapper:**

将 10.6GB 大小的销售信息数据集通过 MapReduce 任务进行处理，对每个省的总销售额进行统计，并对结果排序。将 10.6GB 大小的销售信息数据集分为若干个数据块，这些数据块作为 Map task 的输入，计算出销售额，同时进行年份信息的数据筛选和同省份信息的归并。若干个 Map task 经过 Shuflle 过程后将若干个销售量信息键值对表<省份，销售额>输出到 Reducer。

**Reducer:**

将 Map task 输出的若干个销售量信息键值对表<省份，销售额>作为 Reduce task 的输入，Reduce task 对这些键值对表按照 key 值省份进行归并，将相同的省

份的键值对合并，得到该省份总的销售额。

使用排序函数对合并后的销售量信息键值对表<省份，销售额>按照 value 值销售额大小进行排序，得到最终的 20 个省份 2017 年单年的销售总额的排序表。

## 3.4 Kmeans 聚类

K-means 集群算法是一种基于形心的无监督学习算法，是针对没有标签的一组数据聚类问题的一种有效且简单的算法。K-means 算法的最终结果是根据算法中给定的参数 K，把对一组数据聚类为 K 个簇。在本次 课程设计中，生成的数据集是不含标签的交易额和交易量数据，没有规定聚类结果中聚类簇的数目且数据在二维欧式空间中具有几何意义，故采用 K-means 算法进行大数据下交易数据聚类的 MR 任务是合适的。

### 3.4.1 任务要求

基于 K-means 算法在 Hadoop 分布式集群下运行 MR 任务。设置聚类数目 K，根据数据集中”销售量“ 与”销售单价“ 信息，对某一年的销售数据进行聚类，并可视化聚类结果。将含有 m 个数据特征的数据集映射到一个 m 维的欧氏空间中；

### 3.4.2 k-means 算法原理

1、将含有 m 个数据特征的数据集映射到一个 m 维的欧氏空间中，

2、初始时，随机选择 K 个数据项作为这 K 个簇的形心$c_i(i \in \{1,2,\dots k\})$ 。每个簇心代表一个簇，也就是一组数据项构成的集合。然后对数据集中的所有的 n 个数据项，计算这些数据项与$c_i$的欧式距离。如对于数据项$(x_m \in \{1,\dots n\})$它与其中的一个簇心$c_i$最近，则将$x_m$归类为簇$c_i$ 。初步将数据集划分为 K 个簇，点$x_m$划分到簇$c_i$基于以下公式。

$$arg \min_{c_i \in C} dist(c_i, x_m)^2$$

3、计算类中所有数据项的各个维度的均值，得到这 K 个簇的形心，并更新为所在簇的新形心。每个簇计算一次，更新所有簇形心，记簇$c_i$中所有数据元素的集合为$S_i$，计算新簇形心依据以下公式；

$$c_i = \frac{1}{|S_i|} \sum_{x_i \in S_i} x_i$$

4、对上一步计算得到的新的形心，重复进行第 2，3 步的工作，直到各个类的形心不再变化或迭代次数达到程序设计阈值为止。

### 3.4.3 算法实现

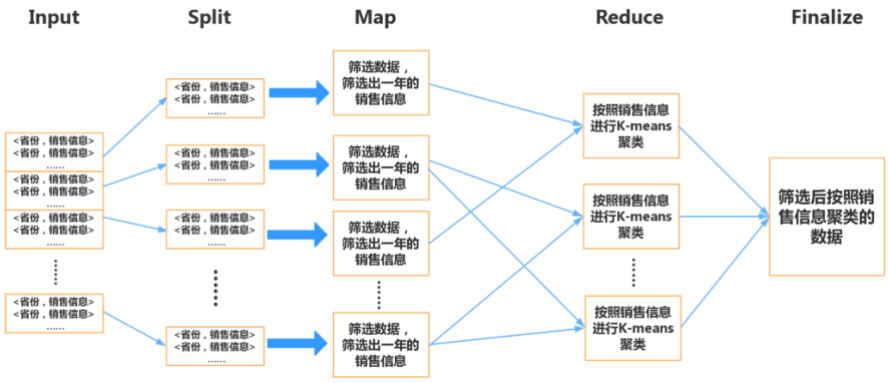


图 3-5 Kmeans 聚类算法实现明细

**Mapper:**

在 Mapper 中将待处理的 block 数据块进行筛选操作，根据关键字”年份“筛选出某一年的交易数据，将数据生成<省份:销售信息>键值对作为 Mapper 的输出经过 Shuflle 过程后传递给 Reducer；

**Reducer:**

分别以”销售量“和”销售单间“为横纵坐标把数据块中交易信息数据映射到二维欧氏空间上；在二维欧氏空间中随机选取 n 个数据点作为初始 K 个簇的形心；计算每个销售信息簇中所有数据项的二维欧氏距离，得到这 K 个簇的形心，并更新为所在簇的新形心。每个簇都这样计算一次，更新所有簇形心；

对上一步计算得到的新的形心，重复进行第 2，3 步的工作，直到各个簇的形心不再变化或迭代次数达到程序设计阈值为止

最终导出聚类结果并在本地进行二维欧氏空间下的可视化处理。通过可视化数据并结合数据特征，分析销售数据的分布情况，得出结论

# 4 测试与结果分析

## 4.1 运行步骤

为了确保程序的正确性以及风险的最低性，测试步骤分为首先在本机进行管道模拟测试，再在伪分布上运行程序测试，最终在集群上完成测试。

管道模拟的核心思想是用前者的输出作为后者的输入，以达到数据输入传递给 Map 再传递给 Shuflle_Sort，最后传递给 Reduce 输出的效果，这样可以模拟

Mapper 数量为 1，Reducer 数量为 1 的效果。

**管道模拟：**

```
cat data.csv | python map.py | sort -t "," -k 1 | python reduce.py
```

用管道来模拟排序程序的正确性。data.csv 中存储的数据作为 Map 任务的输入，Map 任务的输出定向至 sort 函数中，这一步骤是用来模拟 shffule-sort 的过程，结果作为 Reduce 热任务的输入，得到统计排序得到结果。

**伪分布与集群测试：**

```
hadoop fs -mkdir -p /sort_py/output
```

首先，在 HDFS 下新建排序任务的目录，规定输出文件的位置。

```
hadoop fs -put ./data.csv /sort_py/
```

再将数据存入 HDFS 下的排序任务主目录下。

```
hadoop jar hadoop-streaming-2.7.6.jar \
-input "/sort_py/data.csv" \
-output "/sort_py/output" \
-mapper "python map.py" \
-reducer "python reduce.py" \
-file ./map.py \
-file ./reduce.py
```

使用 Streaming 程序，设置输入和输出的位置，以及 MapReduce 程序在本地的位置，开始运行 MapReduce 并等待结果。

## 4.2 测试与结果分析

### 4.2.1 管道模拟测试

**统计排序结果：**

```
(venv) ┌─vinstarry@ubuntu ~/PycharmProjects/mapreduce/sort
└─$ cat data.csv | python map.py | sort -t "," -k 1 | python reduce.py
chongqing,356374.86
shanghai,332413.0
heilongjiang,320487.34
shanxi,300194.2
liaoning,281923.4
jilin,281032.83
hunan,261978.63
shandong,257203.43
guangxi,255927.05
zhejiang,252607.53
beijing,234250.54
jiangsu,229470.7
tianjin,221445.35
hubei,212885.31
guangdong,208576.18
hainan,193558.12
henan,192882.05
sichuan,188926.88
hebei,134993.88
fujian,128991.86
```

图 4-1 统计排序管道模拟测试结果

通过观察结果，不难发现统计已经完成且数据按照销售总额排序呈现，这证

明了程序的正确性，于是可以用 MapReduce 对统计排序进行测试。

**聚类测试结果：**

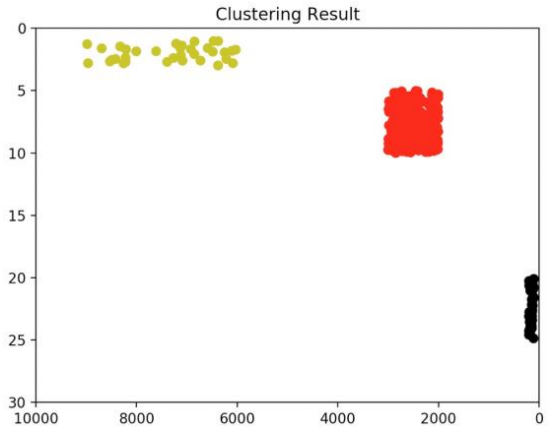


图 4-2 聚类管道模拟测试结果

对结果进行分析，异常数据主要分布在黄色区域(左上角)黑色区域(右下角)。黄色区域数据的销售量主要分布于(6000,9000)，销售单价主要分布于(1,3)跟数据生成时设定的异常数据的值相同。对黑色数据分析后的结果也相同。

因此，图片 4-2 证明了程序的正确性，接下来可以用 MapReduce 对 Kmeans 聚类程序进行测试。

### 4.2.2 伪分布测试

测试集群之前，为了降低风险，首先选择在伪分布上测试数据量为 30K 级别与 200M 级别的数据。

由于 30K 级别数据与 200M 级别的数据在结果数据量上差别较大，但是形式上差别不大，此处仅仅展示了 30K 级别的数据在伪分布 Hadoop 下的运行结果，200M 的结果不再展示。

**统计排序结果：**

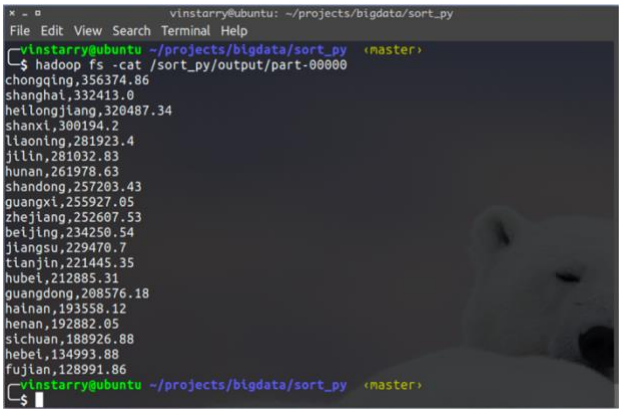


图 4-3 统计排序伪分布测试结果图

在 Hadoop 伪分布下运行程序，结果与管道模拟相同，即用 MapReduce 任

务实现统计排序成功。

**聚类结果：**

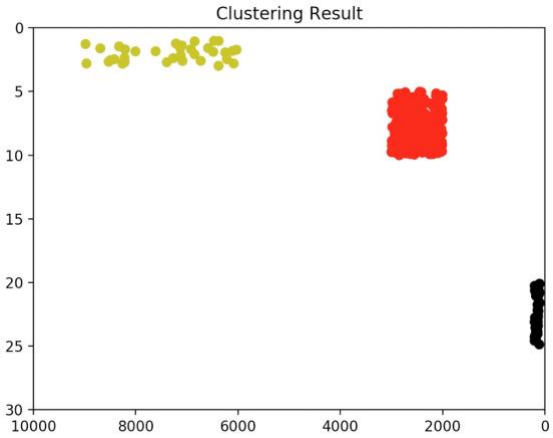


图 4-3 统计排序伪分布测试结果图

在 Hadoop 伪分布下运行程序，并将结果输入可视化程序展现，可以发现聚类分布情况与管道测试相同，即用 MapReduce 任务实现对数据的聚类成功。

### 4.2.3 集群测试

伪分布的运行步骤几乎与集群运行步骤一致。

**排序结果：**


图 4-4 统计排序集群测试结果图

在集群中的 10GB 数据排序结果正确。为了进行性能测试，又对多组数据进行了多次测试：以下是测试结果表，我们进行了 3×8 组测试。

表 4-1 统计排序伪分布模拟测试结果图

| 数据规模(MB) | 1 | 10 | 100 | 200 | 300 | 400 | 1000 | 10000 |
|---|---|---|---|---|---|---|---|---|
| 测试次数 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 3 |
| MR 时间(s) | 2.87 | 3.2 | 17.35 | 38.62 | 57.12 | 72.57 | 180.22 | 1825.98 |
| 单线程时间(s) | 0.13 | 1.13 | 13.2 | 26.2 | 38.2 | 失败 | 失败 | N/A |
| 多线程时间(s) | 0.25 | 2.2 | 24.1 | 49.1 | 84.3 | 失败 | 失败 | N/A |

其中，单线程和多线程的 python 程序由于内存不够的原因，在运行到 400M

左右的吞吐量时已经无法继续运行。

根据测试结果画出运行时间与数据规模曲线图如下：

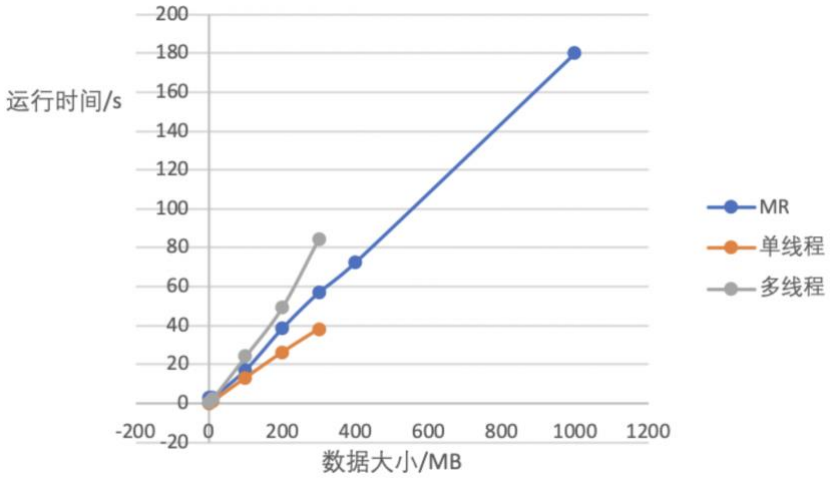


图 4-5 性能测试曲线图

由于统计排序程序的重点在于汇总，仅仅对于 20 个数据进行排序，曲线近似于线性曲线。

MR 任务在同一数据大小情况下比单线程慢，主要原因是因为 MR 过程中 Map 任务分配的数量较大，并且 MR 过程中做的 IO 操作较多，相比于单线程来说没有优势。后续再优化 MR 参数会使得运行结果更优。

多线程在同一数据大小情况下是最慢的，主要是因为在写多线程中加了读写锁，对性能有较大影响。

总的来说，因为这个排序任务重点在于统计，时间复杂度较小，且数据规模不大，导致 MR 的优势没有体现出来。

**聚类结果：**

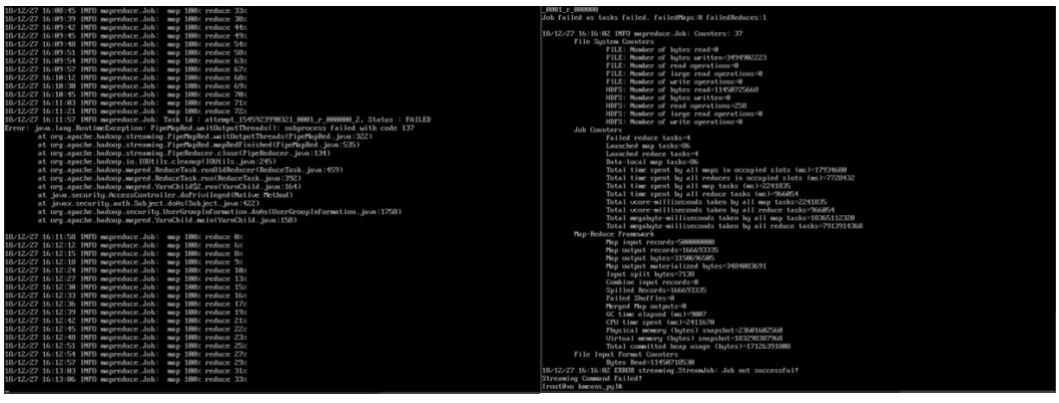

图 4-6 聚类测试内存不足错误提示截图

在运行数据量为 10GB 的聚类任务时失败，Reduce 任务进行到 72%时重新开始 Reduce，最后 Reduce 任务均失败。失败原因是进行 Reduce 任务时内存不足。

随后减少算法的输出与复杂度，将 K-means 算法的迭代次数从 10 次改为 5 次，输出减少了省份一列，重新运行数据量为 2GB 的测试数据，运行成功。

# 5 系统优化

## 5.1 优化过程

由于本次实验编程语言使用的是 python，所以必须在运行前修改系统配置文件以达到参数调优的效果，实验者采用控制变量的方法，每次只修改一个参数，进行 5 次模拟测试，测试取平均值得到平均运行时间，注：测试结果都是在数据规模为 1G 的数据进行排序得到的。

Hadoop 具有多大 190 多条参数可以进行配置，有许多与本次任务类型不匹配，经过多次测试之后，实验者选择了 4 个典型的配置参数进行说明。

**1.dfs.block.size:**

dfs.block.size 说明了 HDFS 存储数据的分块大小，如果设置过大，则同一时间只有较少 map 进行计算，若果值设置的过小，浪费可用 map 个数资源，而且若文件分块过小会造成 NameNode 内存浪费。实验者进行多次参数调优，每次调整参数后重新上传文件，在 1G 的数据集上进行排序操作，得到以下结果。

表 5-1 dfs.block.size 参数调整表

| dfs.block.size 值 | 134217728(默认) | 67108864 | 33554432 | 16777216 |
|---|---|---|---|---|
| 块规模(MB) | 128M | 64M | 32M | 16M |
| Mapper 数量 | 9 | 17 | 35 | 69 |
| 运行次数 | 5 | 5 | 5 | 5 |
| 平均时间(s) | 185.23 | 173.34 | 73.73 | 99.52 |

可以看出，由于数据规模较小，将块大小设置为 32M，效果最优，用折线图进行模拟，得到结果如下：

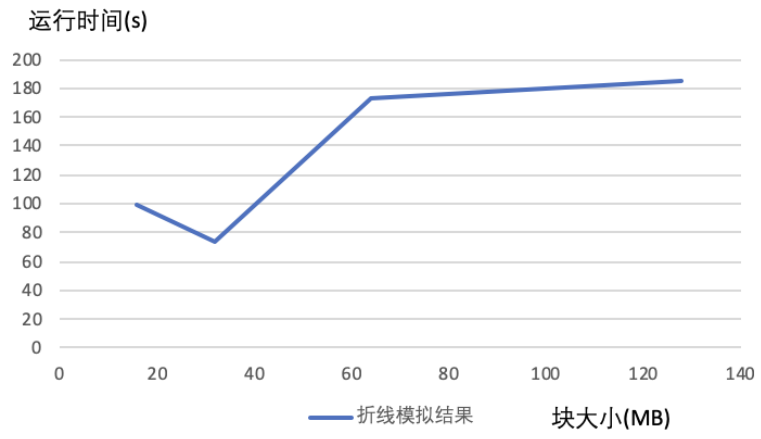


图 5-1 折线模拟运行结果(1)

**2.dfs.namenode.handler.count:**

NameNode 有一个工作线程池用来处理客户端的远程过程调用及集群守护进程的调用。处理程序数量越多意味着要更大的池来处理来自不同 DataNode 的并发心跳以及客户端并发的元数据操作。对于大集群或者有大量客户端的集群来说，通常需要增大参数 dfs.namenode.handler.count 的默认值 10。设置该值的一般原则是将其设置为集群大小的自然对数乘以 20，即$20\log N$，$N$为集群大小，从定义上看这个参数对小的集群影响不大，测试结果如下：

表 5-2 dfs.namenode.handler.count 参数调整表

| dfs.namenode.handler.count | 10 (默认) | 20 | 30 | 40 |
|---|---|---|---|---|
| 运行次数 | 5 | 5 | 5 | 5 |
| 平均时间(s) | 73.73 | 73.40 | 72.55 | 73.21 |

可以看出，该参数对效率的影响微乎其微，主要是由于集群数量较小的原因，将值设置为 30，效果最优，用折线图进行模拟，得到结果如下：

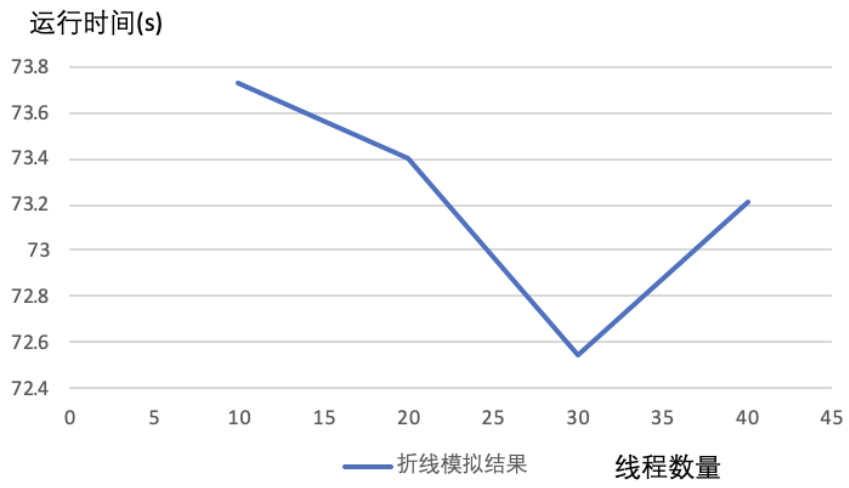


图 5-2 折线模拟运行结果(2)

**3.io.sort.factor:**

io.sort.factor，排序文件时要同时合并的流的数目，当 map 的中间结果非常大，调大 io.sort.factor，有利于减少 merge 次数，进而减少 map 对磁盘的读写频率，达到优化作业的目的。实际测试结果如下表：

表 5-3 io.sort.factor 参数调整表

| io.sort.factor | 5 | 10 (默认) | 15 | 20 |
|---|---|---|---|---|
| 运行次数 | 5 | 5 | 5 | 5 |
| 平均时间(s) | 73.78 | 72.55 | 72.98 | 73.21 |

从上表 5-3 可以看出，io.sort.factor 参数对运行表现影响不大，主要原因是数据规模只有 1G，且对 Map 筛选统计后的文件只有原数据的 1/3 左右的数据量，减少 merge 的次数没有显著提升性能。

**4.mapred.reduce.parallel.copies：**

mapred.reduce.parallel.copies 决定作为 Client 端的 Reduce 同时从 Map 端获取数据的并行度，换言之，mapred.reduce.parallel.copies 制定了单次 Reduce 主动获取数据的 Map 数量。这与网络性能、CPU、RAM 和存储都密切相关，调整结果如下：

表 5-5 mapred.reduce.parallel.copies 参数调整表

| mapred.reduce.parallel.copies | 5 (默认) | 10 | 15 | 20 |
|---|---|---|---|---|
| 运行次数 | 5 | 5 | 5 | 5 |
| 平均时间(s) | 72.55 | 72.18 | 71.72 | 73.02 |

根据表 5-5 可以看出，适应数量的 mapred.reduce.parallel.copies 可以在很小程度上优化运行时间。

## 5.2 优化结果

通过以上的优化步骤，可以看出，dfs.block.size 对运行表现影响较大，dfs.namenode.handler.count、io.sort.factor 和 mapred.reduce.parallel.copies 对运行表现影响较小，但对这三个参数进行合理调整，可以得到有效稳定得提升。

随后，重新运行 8 组数据集合，得到以下的运行表现结果图。

表 5-6 优化前后运行时间记录表

| 数据规模(MB) | 1 | 10 | 100 | 200 | 300 | 400 | 1000 | 10000 |
|---|---|---|---|---|---|---|---|---|
| 测试次数 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 3 |
| 优化前时间(s) | 2.87 | 3.2 | 17.35 | 38.62 | 57.12 | 72.57 | 180.22 | 1825.98 |
| 优化后时间(s) | 2.99 | 3.32 | 9.03 | 21.58 | 23.93 | 30.51 | 71.72 | 717.84 |

对于极小的数据，优化前后的差别不大。之后，优化后的表现明显由于优化前的表现。由于数据最大只有 10GB，严格意义上仍然不能算做一个大数据，在这样的数据级别下，32MB 分块大小的运行表现明显优于 128MB 的分块。

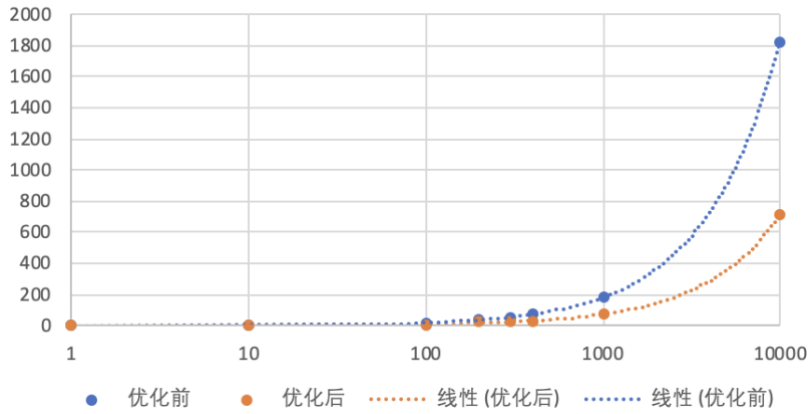


图 5-3 优化前后运行时间对比图

## 5.3 优化策略分析

关于 Hadoop 切片大小(Mapper)个数的参数调整策略，本文结合实验结果与网络资料，得出以下的结论：

如果每个任务只运行很短的时间(比如说几十秒)，那么应该减少任务的数量，因为 Mapper 和 Reducer 的任务首先需要启动 JVM(将 JVM 加载进入内存中，然后需要初始化 JVM，这些是非常耗时的)。如果数据量很大，规模>100G，可以选择更大的切片。

关于读入文件的类型，根据资料与实验者小部分测试，二进制的文件速度会明显由于可读的文件(如 txt,csv 等)，因为将数字信息转化成 UTF-8 字符串需要很大的开销。

对于比较小的文件，尽量合并它们，使它们接近一个分块的大小，否则会造成很大程度的分块浪费，也会增加内存消耗。

关于内存参数的调整，最基本的调整策略就是使用尽可能多的，不会造成 swap 的内存。

因为 IO 操作很多时候都是大数据任务的瓶颈，在本文的聚类实验中，也可以体会到，一开始多输出了一列省份信息时，数据规模为 2G 的数据进行聚类 MR 任务时会由于内存不足退出任务，而减少一列省份信息后运行成功。所以提高 Hadoop 运行表现一个很重要的策略是压缩 IO，查阅资料后得知，在 Mapper 中应用 70%左右的堆栈内存去处理 Spill buffer 是比较理想的。

Mapper 任务的参数调整，是关乎整个 MR 任务性能的至关重要的一环，因为 Mapper 任务是隐含的，不像 Reduce 任务需要显示的规定每一步操作，数量大小等，当应对非常多的文件时，Hadoop 应该将文件分块为更小更多的文件块，方便 Mapper 进行并行处理。

修改 Mapper 参数应该从以下几点出发：

1. 尽可能的复用 JVM，如前文所述，启动、初始化和结束 JVM 是非常大的时间开销。

2．尽量让每个 Mapper 任务的运行时间在 1~3 分钟，如果任务的时间小于一分钟，需要增加参数 mapred.min.split.size，分配更少的 Mapper 槽，从而减少 Mapper 的数量。

修改 Reducer 的参数应该考虑以下几点：

1. Reducer 的线程数量应当合适，Mapper 的数量越多，Reducer 应该使用更多线程处理

2. 本文的实验均只使用了一个 Reducer，但实际上很多任务有 Reducer 参与，这时应该控制 Reducer 的数量，并且让它们负载均衡。

# 6 总结与展望

## 6.1 课程实验总结

本次课程设计完成了伪分布环境下和分布式环境下 Hadoop 集群的配置，实现了 GB 级别数据量上的 MapReduce 任务的运行，得到了 MR 任务的排序结果以及基于 K-means 算法的聚类结果。

在完成上述任务的基础上，课设小组在本次课程设计中还完成了对聚类结果的本地可视化处理与结果分析；通过比较 MR 处理排序任务与本地单线程和多线程处理排序任务的运行时间来对 MR 进行了性能分析；通过调节 MR 运行参数、聚类算法迭代次数，以及 Mapper 和 Reducer 的数目及单个 Mapper 和 Reducer 上切片大小来对分布式集群环境下的 MR 任务进行性能调优。

## 6.2 课程实验展望

通过本次 Hadoop 平台的实践任务，可以将实践学习到的 MapReduce 操作运用到实际的大数据环境中，比如，任务一中的大数据排序可以运用在网购平台商品仓库的调度上，将用户的网购信息和用户的所在地关联起来，通过 MapReduce 统计并排序，得到不同地区用户对某一商品需求量的排序，合理去分配这一商品在不同地区仓库的存放量，这样能让快递速度大幅度加快。

在本次 MapReduce 性能优化过程中，仅对单个 Mapper 和 Reducer 的切片大小进行研究，效果比较明显，可以进一步拓展，对 Hadoop NameNode 和 DataNode 用于 RPC 的线程进行调参，或者对 YARN 容器进行调参，减少资源的浪费。这样可以对 MapReduce 任务有更加深刻地理解。

还可以对 Hadoop 平台有更多的拓展，对于需要实时查询的数据集，用 HBase 数据库来装载；对于类 SQL 的数据集，可以用 Hive 来排序和查询；对于数据挖掘与机器学习等需要迭代的 MapReduce 的算法，可以使用 Spark 计算引擎。以后进一步在 Hadoop 平台上开发和实践，针对不同类型的数据，不同的处理需求，运用不同的工具进行大数据处理，是必不可少的。

# 附录

## 统计排序程序

**sort_Map.py**

```
import sys

data_list = []
result_dict = {}

'''
从输入流中以‘,’为分隔解析出省份，年份，销售量，销售单价信息
存入结构化的销售信息表 data_list 中
data_list 作为 Map 的数据主体
'''
for line in sys.stdin:
    data = line.strip().split(',')
    data_list.append(data)

'''
从销售信息表 data_list 中剔除非 2017 年的信息，作初步筛选
将每一条销售信息的销售量和销售单价求积，得到销售量
再将同省份的销售量键值对<省份，销售额>作归并处理
得到归并后的销售量信息键值对表 result_dict<省份，销售额>
'''
for item in data_list:
    if item[1] == '2017':
        prov = item[0]
        sales = float(item[2]) * float(item[3]) #销售额 = 销售量 * 销售单
价
        if prov in result_dict.keys():
            result_dict[prov] += sales  #如果存在同名 key 省份，则将 value 销
售量相加归并
        else:
            result_dict[prov] = sales   #如果不存在同名 key 省份，则生成新的
键值对<prov,sales>

'''
将销售信息键值对表 result_dict<省份，销售额>作为输出流
'''
for item in result_dict:
```

```
    print(item + "," + str(round(result_dict[item],2)))
```

**sort_Reduce.py**

```
import sys

data_list = []
result_dict = {}

'''
输入是归并后的销售量信息键值对表
将其以‘,’为分隔解析并存入 data_list
data_list 作为 Reduce 的数据主体
'''
for line in sys.stdin:
    data = line.strip().split(',')
    data_list.append(data)

'''
将 Map 后的若干个 data_list 按照 key 值省份合并
得到最终的未排序的 result_dict<省份，销售额>
'''
for item in data_list:
    prov = item[0]
    sales = float(item[1])
    if prov in result_dict.keys():
        result_dict[prov] += float(sales)    #如果 key 值省份相同，则将 value
相加
    else:
        result_dict[prov] = sales            #如果不存在同名省份，则生成新的
result_dict 键值对<prov,sales>

'''
将 reslut_dict 按照销售额排序
得到最终的结果 final_list
'''
final_list = sorted(result_dict.items(), key=lambda item:item[1],
reverse=True)

for item in final_list:
    print(item[0] + ',' + str(item[1]))
```

# Kmeans 聚类程序

**Map.py**

```
import sys

data_list = []

'''
将数据存放至列表 data_list
列表的每一项是包含有省份、年份、销售额、销售量等元素的子列表
同时子列表中还包含一个标志元素，当标志元素为“1”时表示数据的年份为“2016”，反之
标志元素为“0”
'''
def phrase_input():
    for line in sys.stdin:
        data = line.strip().split(',')  #读取 CSV 文件中每一列数据至列表
        if data[1] != '2016':
            #当年份不为“2016”，标志元素置 0
            data.append(0)
        else:
            #否则标志元素置 1
            data.append(1)
        data_list.append(data)  #子列表加入列表中

'''
输入列表中标志元素为 1 的元素项
Reducer 只接收标志元素为 1 的元素项
'''
def output_result():
    for item in data_list:
        if item[-1] == 1:
            #标志元素为 1 则输出
            print(item[0] + "," + item[2] + "," + item[3])


if __name__ == '__main__':
    phrase_input()
    output_result()
```

**kmeans_Reduce.py**

```
import sys
import random

data_list = []  #数据列表
iter_time = 1   #总递归次数
centroids = []  #


'''
读取 Mapper 筛选后输出的数据至 data_list
列表的每一项是包含有省份、年份、销售额、销售量等元素的子列表
同时子列表中还包含一个标志元素，表示所属簇的中心点编号
'''
def phrase_input():
    for line in sys.stdin:
        data = line.strip().split(',')
        data.append(0)
        data[1] = int(data[1])
        data[2] = float(data[2])
        data_list.append(data)


'''
K-means 算法
输入为 data_list

'''
def K_means(cluster_num = 3, iteration_time = 10):
    global iter_time
    global data_list
    global centroids
    random_centroids_number = [0] * cluster_num
    if (len(data_list) < 10) or (len(data_list) < cluster_num):
        #数据量太小或数据量小于期望簇数，中止本次迭代
        print("Bad data!")
        return
    if cluster_num == 1:
        #期望簇数为 1，中止本次迭代
        return
    if iter_time == 1:
        #第一次迭代需要随机选取数据点作为簇的形心
        for i in range(0, cluster_num):
            temp = random.randint(0, len(data_list) - 1)
```

```
            while temp in centroids:
                #随机选取的形心必须互不相同
                temp = random.randint(0, len(data_list - 1))
            random_centroids_number[i] = temp   #形心编号
            centroids.append([data_list[temp][1], data_list[temp][2]])  #
形心在二维平面上的坐标
        tag_items(cluster_num)  #按照选取的形心进行聚类
        iter_time = iter_time + 1
        return K_means(cluster_num, iteration_time)

    elif iter_time == iteration_time:
        #达到最大迭代次数
        return
    else:
        calculate_centroids(cluster_num)    #计算新形心
        tag_items(cluster_num)  #从新把数据划分到新形心代表的各个簇
        iter_time = iter_time + 1   #迭代次数加 1
        return K_means(cluster_num, iteration_time) #递归进行迭代

'''
计算所有数据距离最近的簇的中心点
记录每个数据距离最近的簇的中心点的编号和坐标
'''
def tag_items(cluster_num = 3):
    global data_list
    global centroids
    distance_square = [0] * cluster_num
    for item in data_list:
        for i in range(cluster_num):
            #计算到每个形心点之间的欧氏距离
            distance_square[i] = (abs(item[1] - centroids[i][0]) ** 2) +
(abs(item[2] - centroids[i][1]) ** 2)
        min = distance_square[0]
        min_pos = 0
        for i in range(cluster_num):
            #选取距离最小的形心点，记录编号与坐标
            if distance_square[i] < min:
                min_pos = i
                min = distance_square[i]
        #数据子列表的最后一项标志所述簇的中心点编号
        item[-1] = min_pos



'''
```

```
计算新的簇的中心点
更新中心点标号列表与坐标列表
'''
def calculate_centroids(cluster_num = 3):
    global data_list
    global centroids
    sum_x = [0] * cluster_num
    sum_y = [0] * cluster_num
    points_num = [0] * cluster_num
    for item in data_list:
        #计算一个簇中所有元素的 x，y 的坐标之和
        sum_x[item[-1]] += item[1]
        sum_y[item[-1]] += item[2]
        points_num[item[-1]] += 1
    for i in range(cluster_num):
        #计算一个簇中所有元素的 x，y 的坐标平均值，即为新的形心坐标
        centroids[i][0] = sum_x[i] / points_num[i]
        centroids[i][1] = sum_y[i] / points_num[i]


'''
输出 K-means 得出的聚类结果
'''
def print_result():
    for item in data_list:
        print(str(item[0]) + "," + str(item[1]) + "," + str(item[2]) +
"," + str(item[-1]))
    return


if __name__ == '__main__':
    phrase_input()
    K_means(3,10)
    print_result()
```

# 配置文件

core-site.xml

```
<configuration>
    <property>
        <name>fs.default.name</name>
        <value>hdfs://xu:9000</value>
        <description>
            fs.default.name 用于设置默认文件系统的 URL，URL 用于确定文件系统的主机， 端口等。
            本集群系统中 hdfs://xu:9000 为集群中主节点 namenode 的 URL。
        </description>
    </property>
</configuration>
```

hdfs-site.xml

```
<configuration>
    <property>
        <name>dfs.namenode.name.dir</name>
        <value>/opt/hadoop/data/nameNode</value>
        <description>
            dis.namenode.dir 用于确定在本地文件系统上的 DFS 名称节点应存储名称表（fsimage）。
            fsimage 的内容会被存储到以逗号分隔的列表的目录中，然后在所有的目录中复制名称表目录，用于冗余。
            fsimage:元数据镜像文件。存储某一时段 NameNode 内存元数据信息。
        </description>
    </property>

    <property>
        <name>dfs.datanode.data.dir</name>
        <value>/opt/hadoop/data/dataNode</value>
        <description>
            dis.datanode.dir 用于确定 datanode 在本地文件系统上的 DFS 数据节点应存储名称表。
        </description>
    </property>

    <property>
        <name>dfs.replication</name>
        <value>2</value>
        <description>
```

```
            dfs.replication 用于设置 hdfs 副本系数。
        </description>
    </property>

    <property>
        <name>dfs.datanode.handler.count</name>
        <value>10</value>
        <description>
            datanode 上用于处理 RPC 的线程数。
        </description>
    </property>

    <property>
        <name>dfs.block.size</name>
        <value>33554432</value>
        <description>
            HDFS 中数据 block 大小。
        </description>
    </property>

    <property>
        <name>dfs.namenode.handler.count</name>
        <value>30</value>
        <description>
            namenode 中用于处理 RPC 的线程数。
        </description>
    </property>

</configuration>
```

yarn-site.xml

```
<configuration>
    <property>
        <name>yarn.acl.enable</name>
        <value>0</value>
        <description>
            类似客户端和管理员等的请求这样面向用户的 API，查看访问的 ACL 决定谁可以通过 RPC 接口查看一些所有应用程序的相关细节，WEB UI 服务及 WEB 服务。
        </description>
    </property>

    <property>
        <name>yarn.resourcemanager.hostname</name>
```

```
        <value>xu</value>
        <description>
            设置 resourcemanager 的主机名称。
        </description>
    </property>

    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
        <description>
            NodeManager 上运行的附属服务。需配置成 mapreduce_shuffle，才可运行 MapReduce。
        </description>
    </property>

    <property>
        <name>yarn.nodemanager.resource.memory-mb</name>
        <value>8196</value>
        <description>
            NodeManager 总的可用物理内存。
            注意，该参数是不可修改的，一旦设置，整个运行过程中不可动态修改。
        </description>
    </property>

    <property>
        <name>yarn.scheduler.maximum-allocation-mb</name>
        <value>8196</value>
        <description>
            单个可申请的最大内存资源量。比如设置为 1536，则运行 MapRedce 作业时，每个 Task 最多
可申请 1536MB 内存。
        </description>
    </property>

    <property>
        <name>yarn.scheduler.minimum-allocation-mb</name>
        <value>64</value>
        <description>
            单个可申请的最小内存资源量。比如设置为 128，则运行 MapRedce 作业时，每个 Task 最少可
申请 128MB 内存。
        </description>
    </property>

    <property>
        <name>yarn.nodemanager.vmem-check-enabled</name>
        <value>false</value>
```

```
        <description>
            是否启动一个线程检查每个任务正使用的虚拟内存量，如果任务超出分配值，则直接将其杀掉，默认是 true。
        </description>
    </property>
</configuration>
```

mapred-site.xml

```
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
        <description>
            使用 yarn 运行 mapreduce，作为资源管理框架。
        </description>
    </property>

    <property>
        <name>io.sort.factor</name>
        <value>5</value>
        <description>
            执行 merge sort 时，每次同时打开 spill 文件的数量。
        </description>
    </property>

    <property>
        <name>mapred.reduce.parallel.copies</name>
        <value>20</value>
        <description>
            Reduce shuffle 阶段 copier 线程数。
        </description>
    </property>

    <property>
        <name>yarn.app.mapreduce.am.resource.mb</name>
        <value>8192</value>
        <description>
            MR ApplicationMaster 占用的内存量
        </description>
    </property>

    <property>
        <name>mapreduce.map.memory.mb</name>
        <value>8192</value>
        <description>
```

```
            每个 Map Task 需要的内存量
        </description>
    </property>

    <property>
        <name>mapreduce.reduce.memory.mb</name>
        <value>8192</value>
        <description>
            每个 Reduce Task 需要的内存量
        </description>
    </property>
</configuration>
```