

# 大数据处理程序设计实验报告

## 任务1 数据预处理

### 1.1. 实验设计说明

#### 1.1.1 主要设计思路

预处理部分需要做三件事情：

1. 做一次 MapReduce 输出人名
2. 使用 hanlp 开源工具进行分词
3. 整理别名表 nickname 供任务2使用

Map 阶段：

setup() 导入 person\_name\_list 以供分词，需要注意的是 hanlp 分词字典 CustomDictionary 可以包含带空格的词汇，在 map 中，用 hanlp 开源工具提供的接口 HanLP.\*segment\*(value.toString()) 来进行分词，比较分词后的词性是否是 "userDefine" 来判断词语是不是人名

KEYIN:Object:文档偏移量，map 中没有使用到

VALUEIN:Text:文档中的一段，用来做分词的参数

KEYOUT:Text:输出键为该段的全部人名，格式

为 "personName1\tABpersonName2\tAB...personNamen\tAB"，因为一个人名可能包含空格，所以名字之间用制表符 \TAB 分开

VALUEOUT:NullWritable:因为输出键就已经足够了，所以值为空

Reduce 阶段：

遍历所有的 NullWritable value，每一个 value 都要单独写入一次 key 以防止出现两段文章里输出是相同的名字和顺序，例如：

第一段输出:Harry Potter Hermione

第二段输出:Harry Potter Hermione

这两段输出一样，如果不按照每一个 value 单独写一次 key 就有可能漏掉

KEYIN:Text:与 map 的输出键格式相同，表示一段的所有出现的名字

VALUEIN:NullWritable:与 map 的输出键格式相同，表示空，占位

KEYOUT:Text:输出键为该段的全部人名，格式

为 "personName1\tABpersonName2\tAB...personNamen\tAB"，因为一个人名可能包含空格，所以名字之间用制表符 \TAB 分开

VALUEOUT:NullWritable:因为输出键就已经足够了，所以值为空

### 1.1.2. 主要算法及性能分析

以下是任务1代码的主要内容 NameProcessDriver 类:

```
public class NameProcessDriver {
    public static class NameProcessMapper extends Mapper<Object, Text, Text,
NullWritable> {
        @Override
        protected void setup(Context context) throws IOException {
            //获取person_name_list.txt文件,并加入字典中
            FileSystem fs = FileSystem.get(context.getConfiguration());
            Path in = new Path(FilePath.personName);
            FSDataInputStream fsIn = fs.open(in);
            LineReader lineIn = new LineReader(fsIn,
context.getConfiguration());
            Text line = new Text();
            while (lineIn.readLine(line) > 0) {
                CustomDictionary.insert(line.toString().trim(), "userDefine");
            }
            fsIn.close();
        }

        public void map(Object key, Text value, Context context) throws
IOException, InterruptedException {
            List<Term> terms = HanLP.segment(value.toString());
            //开源HanLP进行分词,它会将字典中自定义的"userDefine"的词性识别出来,即使这个词
            中间有空格

            StringBuilder sb = new StringBuilder();
            int cnt = 0;
            for (Term i : terms) {
                if (i.nature.toString().equals("userDefine")) {
                    cnt++;
                    sb.append(i.word + "\t");
                    //使用制表符可以避免名字中本身就有空格的干扰
                }
            }
            if (0 != cnt) {
                context.write(new Text(sb.toString().trim()),
NullWritable.get());
            }
        }
    }

    public static class NameProcessReducer extends Reducer<Text, NullWritable,
Text, NullWritable> {
        @Override
        protected void reduce(Text key, Iterable<NullWritable> values, Context
context) throws IOException, InterruptedException {
            for (NullWritable it : values) {
                context.write(key, it.get());
                //对每一个value的遍历保证了不会遗漏掉哪一段
            }
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
```

```

//设置环境参数
Job job = Job.getInstance(conf, "NameProcess");
//设置程序类名
job.setJarByClass(NameProcessDriver.class);
//为作业设置map类
job.setMapperClass(NameProcessDriver.NameProcessMapper.class);
//为作业设置Reduce类
job.setReducerClass(NameProcessDriver.NameProcessReducer.class);
//设置Map输出
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(NullWritable.class);
//设置Reduce输出
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(NullWritable.class);

FileSystem fs = FileSystem.get(conf);
FileStatus[] listStatus = fs.listStatus(new Path(args[0]));
//遍历目录下的所有文件，跳过person_name_list.txt
for (FileStatus file : listStatus) {
    if (file.getPath().getName().startsWith("person")) {
        continue; //跳过person_name_list.txt
    }
    FileInputFormat.addInputPath(job, file.getPath());
}
//设置输出文件路径
FileOutputFormat.setOutputPath(job, new Path(args[1]));

job.waitForCompletion(true); /* 启动作业 */
}
}

```

任务1要求输出的是每段的人名，所以 `map-reduce` 过程需要处理大量的小说原文，故过程较慢，且难以改进。

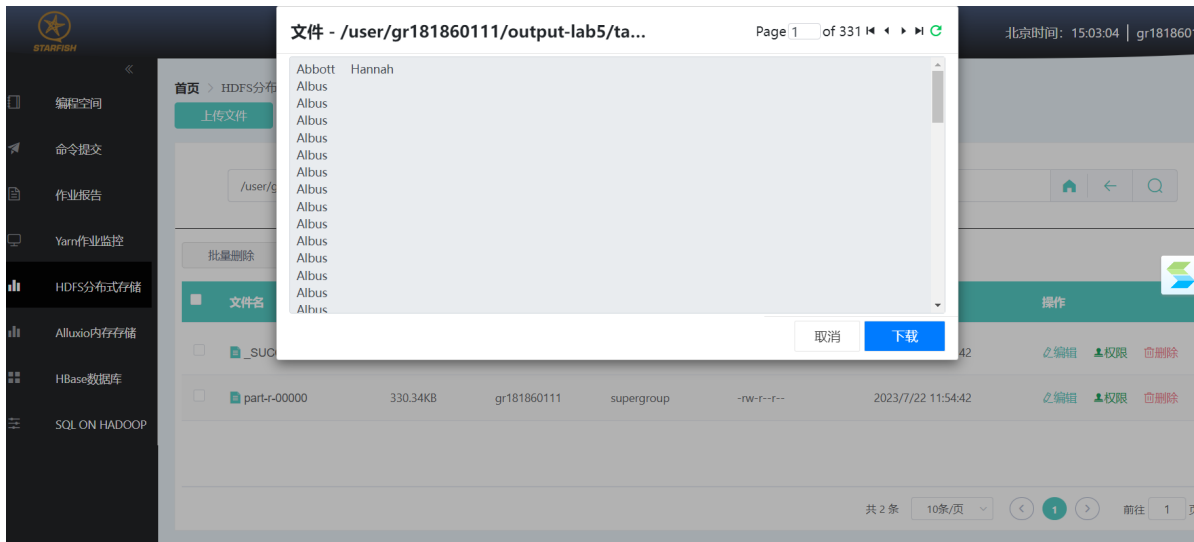
## 1.2. 实验结果

### 1.2.1. 输出文件路径

[程序输出路径] `/user/gr181860111/output-lab5/task1/`

[别名表路径] `/user/gr181860111/nickName.txt`

### 1.2.2. 部分输出结果



## 任务2 特征抽取:人物同现统计

### 2.1. 实验设计说明

#### 2.1.1. 主要设计思路

首先需要说明的是，任务2的输入是任务1的输出，同时还有别名表 `nickName.txt`

别名表 `nickName.txt` 由笔者本人基于以下手段，手动完成:

[别名表路径] `/user/gr181860111/nickName.txt`

规则1: 每一行唯一表示一个人物的所有名字，不同行必然不同人

规则2: 较长的名字作为统一名字，较短的名字作为别名

规则3: 名字与别名之间用制表符"`\t`"来分隔，以避免英文名中间空格对分隔的影响

规则4: 删除 `person_name_list.txt` 在任务1输出中所有没有出现过的名字

删除了一些没有用的名字,共32个，中间数据中记录了删除它们的原因，以下展示部分

	person_name_list.txt中没有用的名称	没有用的原因
1	Mad-eye Moody	拼写错误 Mad-Eye Moody
2	Godric	全部被Godric Gryffindor取代
3	Moaning Murtle	压根没这个人Moaning Murtle, 应该是Moaning Myrtle
4	Hooch	全部被Madam Hooch取代
5	Ernie Mcmillan	拼写错误 Ernie McMillan
6	Gibleroy Lockhart	拼写错误 Gilderoy Lockhart
7	Murtle	压根没这个人, 它应该是Moaning Myrtle的别名, 拼写错误
8	Rowena	是这个人Rowena Ravenclaw的别名且未单独出现
9	Aurthor	没有这个人, 同样没有下面的Aurthor Weasley
10	Helga	是Helga Hufflepuff的别名, 且没单独出现过
11	Aurthor Weasley	没有这个人, 同样没有上面的的Aurthor
12	Mad-eye	是Mad-eye Moody的别名, 但是拼写错了, 应该是Mad-Eye
13	Pince	是Madam Pince的别名, 但是从未单独出现过
14	Madam Maxime	没有这个人
15	Jorkins	Bertha Jorkins的别名, 且从未单独出现
16	Bill Weasley	没有这个人
17	Porfessor	没有这个人
18	Professor Minerva Mcgonagall	叫Mcgonagall人有, 但是没这个叫Professor Minerva Mcgonagall的名字

#### Map 阶段:

统计一行内容中所有同现的人物对, 发射, 需要注意的是, 对于以下的一行内容

```
Harry Harry Harry Ron Ron
```

应该发射两组

```
Key:<Harry,Ron> Value:3
```

```
Key:<Ron,Harry> Value:2*3
```

#### Map 的输入输出参数:

**KEYIN:**Object ,这里没用到

**VALUEIN:**Text ,表示输入一行的信息,格式为 Name1\tName2\tName3...\tNamen\t

**KEYOUT:**Text 表示人物对, 格式为 <Name1,Name2>

**VALUEOUT:**IntWritable 表示人物对同现的次数

#### Combine 阶段:

将相同人物对的同现次数求和, 然后发射, 以减少网络传输的次数, 提高效率

**KEYIN:**Text 表示人物对, 格式为 <Name1,Name2>

**VALUEIN:**IntWritable 表示人物对同现的次数

**KEYOUT:**Text 表示人物对, 格式为 <Name1,Name2>

**VALUEOUT:**IntWritable 表示人物对同现的次数

Reduce 阶段:

将相同人物对的同现次数求和

KEYIN:Text 表示人物对, 格式为 <Name1,Name2>

VALUEIN:IntWritable 表示人物对同现的次数

KEYOUT:Text 表示人物对, 格式为 <Name1,Name2>

VALUEOUT:IntWritable 表示人物对最终的同现的次数

## 2.1.2. 主要算法及性能分析

以下是实现任务2主要功能的 ConcurrencyRelationDriver 类中的内容

HarryMapper 实现 Map 功能

```
public static class HarryMapper extends Mapper<Object, Text, Text, IntWritable>
{
    public static Map<String, String> nickname_table = new HashMap<String,
String>();

    protected void setup(Context context) throws IOException,
InterruptedException {
        super.setup(context);
        FileSystem hdfs = FileSystem.get(context.getConfiguration());
        //创建hdfs文件系统
        Path rPath = new Path(FilePath.nickName);
        //设置路径为预设的别名表路径
        BufferedReader in = new BufferedReader(new
InputStreamReader(hdfs.open(rPath)));
        //创建读取对象
        String str = in.readLine();
        //接下来按行读取, 一行就是一个人的所有名称, 一行中的第一个是统一名称
        while (str != null) {
            String[] current = str.split("\t");
            for (int i = 1; i < current.length; ++i) {
                nickname_table.put(current[i], current[0]);
                //后面的词是别名, 最前面的词是统一后的名称
            }
            str = in.readLine();
        }
        in.close();
    }

    @Override
    protected void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
        HashMap<String, Integer> names = new HashMap<>();
        //names存储一段中所有名字归一化后的统一名字和响应数量
        //如果是Ron Ron Ron Harry Harry 那么<Ron,Harry>和<Harry,Ron>同现为3*2次
        String[] allNames = value.toString().split("\t");//各个名字之间用制
表符连接,防止名字本身有空格
        String temp = "";
        for (String name : allNames) {
            temp = nickname_table.get(name);
```

```

        if (temp == null) {
            temp = name; //如果在别名哈希表中没有找到,说明是统一后的名字
        }
        //下面统计各个名字以及其出现的次数
        if (names.keySet().contains(temp)) {
            int newInt = names.get(temp).intValue() + 1;
            names.put(temp, new Integer(newInt));
        } else {
            names.put(temp, new Integer(1));
        }
    }
    for (String name1 : names.keySet()) {
        for (String name2 : names.keySet()) {
            if (!name1.equals(name2)) {
                int nameFreq1 = names.get(name1).intValue();
                int nameFreq2 = names.get(name2).intValue();
                context.write(new Text("<" + name1 + "," + name2 + ">"),
                    new IntWritable(nameFreq1 * nameFreq2)); //<1,2>和<2,1>都是需要发射的
            }
        }
    }
    names.clear();
}
}

```

HarryCombiner 实现 Combine 功能

```

public static class HarryCombiner extends Reducer<Text, IntWritable, Text,
IntWritable> {
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context
context)
        throws IOException, InterruptedException {
        //单个节点上求和key的value,减少传输次数
        if (values == null) return;
        int sum = 0;
        for (IntWritable val : values) {
            sum = sum + val.get();
        }
        context.write(key, new IntWritable(sum));
    }
}

```

HarryReducer 实现 Reduce 功能

```

public static class HarryReduce extends Reducer<Text, IntWritable, Text,
IntWritable> {
    protected void reduce(Text key, Iterable<IntWritable> values, Context
context)
        throws IOException, InterruptedException {
        if (values == null) return;
        int sum = 0;
        for (IntWritable val : values) {
            sum = sum + val.get();
        }
        context.write(key, new IntWritable(sum));
    }
}

```

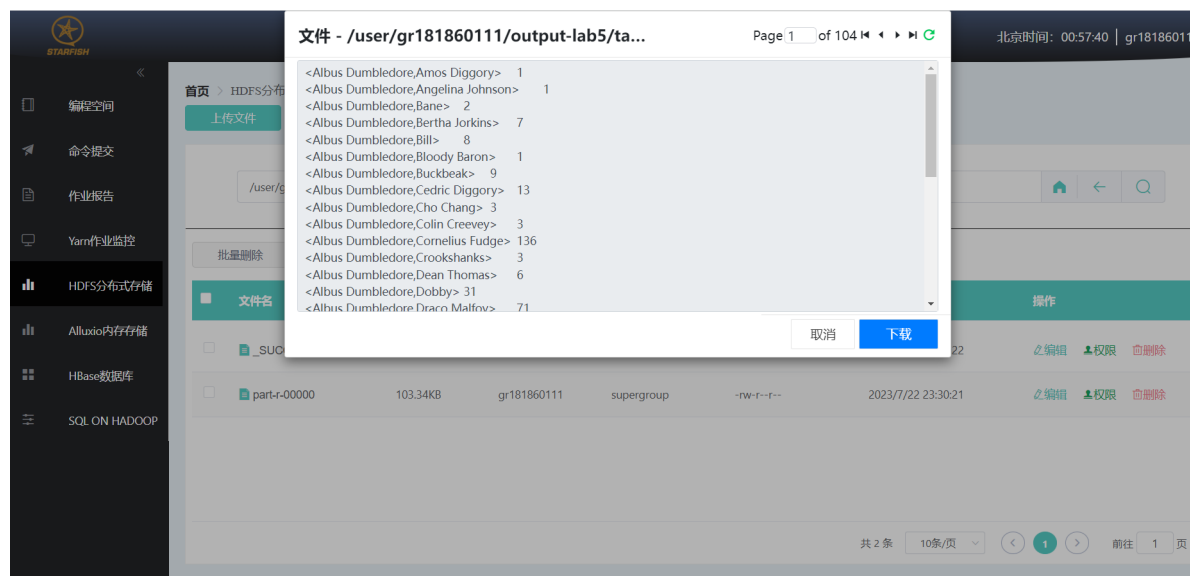
这里用 `HarryCombiner` 减少网络传输信息量，提高性能

## 2.2. 实验结果

### 2.2.1. 输出文件路径

[程序输出路径] `/user/gr181860111/output-lab5/task2/`

### 2.2.2. 部分输出结果



## 任务3 特征处理 人物关系图构建与特征归一化

### 3.1. 实验设计说明

#### 3.1.1. 主要设计思路

Task3的输入是Task2的输出

Map 阶段

把 `<Name1,Name2> \t times` 处理为 `Key:Name1 Value:Name2,times` 发射

Combine 阶段



把同一个节点上的相同 `Key:Name1` 对应的 `Value:Name2,times` 合并为

`Key:Name1 Value:Name2,times\tName3,times\t...\tNamen,times\t` 并发射

Reduce 阶段

把同一个 `Key:Name1` 下的所有 `Value:Name2,times\tName3,times\t...\tNamen,times\t` 分段统计

统计所有其他人物出现的次数以及出现的总次数，进行归一化

输出格式 `Key:Name0 Value:[Name1,x.xx|Name2,x.xx|...|Namen,x.xx]`

### 3.1.2. 主要算法及性能分析

以下是实现任务3主要功能的 `RelationGraphDriver` 类中的内容

`HarryMapper` 实现 map 阶段的功能

```
public static class HarryMapper extends Mapper<Object, Text, Text, Text> {
    @Override
    protected void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
        String[] name_times = value.toString().split("\\t");
        //分词为["<name1,name2>", "times"]
        String name_temp = name_times[0].substring(1, name_times[0].length()
- 1);

        //name_temp为"name1,name2"
        String[] names = name_temp.split(",");
        //names为["name1","name2"]
        String name1 = names[0];
        String name2 = names[1];
        String times = name_times[1];
        context.write(new Text(name1), new Text(name2 + "," + times));
        //发射Key:"name1",Value:"name2,times"
    }
}
```

`HarryCombiner` 实现 combine 功能

```
public static class HarryCombiner extends Reducer<Text, Text, Text, Text> {
    protected void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {
        //将一个节点上的相同的Key:name1的Value进行合并,制表符分隔
        if (values == null) return;
        String temp = "";
        for (Text val : values) {
            temp = temp + val.toString() + "\\t";
        }
        context.write(key, new Text(temp));
    }
}
```

`HarryReducer` 实现 reduce 功能

```
public static class HarryReduce extends Reducer<Text, Text, Text, Text> {
    protected void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {
```

```

        if (values == null) return;
        List<String> myValues = new ArrayList();
        //myValues 存储"name,times"数据,下面累加所有的times
        double sum = 0.0;
        for (Text val : values) {
            String[] people = val.toString().split("\t");
            for (String p : people) {
                int times = Integer.parseInt(p.split(",")[1]);
                sum = sum + times;
                myValues.add(p);
            }
        }
        StringBuffer res = new StringBuffer();
        //创建一个StringBuffer对象来存储需要输出的字符串,最终格式为
        [name1,p1|name2,p2|...|namen,pn]
        res.append("[");
        for (String val : myValues) {
            String person = val.split(",")[0];
            res.append(person);
            res.append(",");
            double times = (double) Integer.parseInt(val.split(",")[1]);
            double pTimes = times / sum;
            res.append(pTimes);
            res.append("|");
        }
        int l = res.length();
        res.deleteCharAt(l - 1);
        //最后一个"|"需要删除
        res.append("]");
        context.write(key, new Text(res.toString()));
    }
}

```

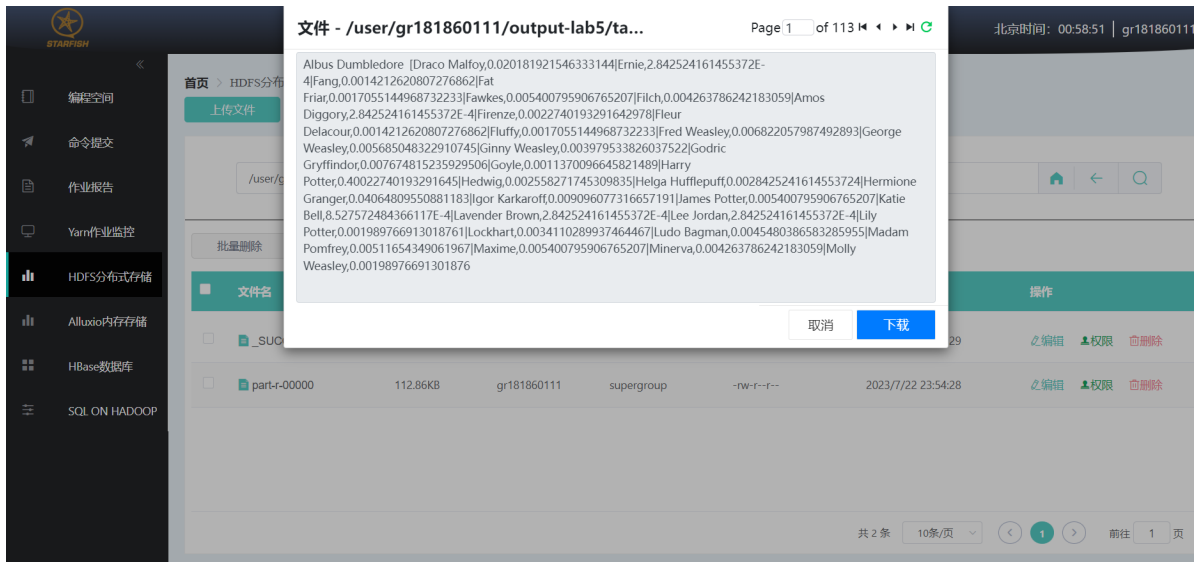
这里用 `HarryCombiner` 减少网络传输信息量, 提高性能

## 3.2. 实验结果

### 3.2.1. 输出文件路径

[程序输出路径] `/user/gr181860111/output-lab5/task3/`

### 3.2.2. 部分输出结果



## 任务4：数据分析：基于人物关系图的PageRank计算

### 4.1. 实验设计说明

#### 4.1.1. 主要设计思路

在人物关系图中，与许多"主角"有较深关系的人物大概率也是"主角"，可以利用PR值的大小衡量一个人物是否是"主角"，一个人物的PR值取决于：

1. 与其有关系的人物的个数(即关系图上以该人物为点的度数)
2. 与其有关系的人物的的重要程度(即相邻节点的PR值)
3. 与其有关的人物与他的关系的深浅(即有向关系图上人物之间边的权重)

计算一个人物  $Role_i$  的PR值的公式如下：

$$PR(Role_i) = \sum_{Role_j \in B_i} (PR(Role_j) * Weight(Edge_{ji}))$$

$B_i$  为与角色  $i$  有关系的所有角色节点集合

$Weight(Edge_{ji})$  为有向图中边  $Edge_{ji}$  的权

PageRank迭代过程可以这么理解:

对于任务三处理后的105个人物

我们建立了一个有向图邻接矩阵  $105 \times 105$ ，邻接矩阵A存储的是有向图的归一化后的边权

需要注意的是， $Edge_{ij}$ 和 $Edge_{ji}$ 是不一样的，因为这是一个有向图

设置所有的105个节点的PR值初始值为1.0，列向量PR的元素表示顺序105个人物角色节点的PR值

一直令邻接矩阵A左乘PR,直到收敛或达到预定迭代次数

$$PR_{new} \leftarrow A * PR_{last}$$

### 4.1.2. 主要算法和类设计

分为三个阶段

第一阶段：通过 `GraphBuilder` 类进行图的建立

第二阶段：通过 `PageRankIter` 类进行迭代更新PR值，直到收敛或者到达预定迭代次数

第三阶段：通过 `PageRankViewer` 类对最终迭代完毕的结果进行排序

这三个阶段的 MapReduce 设计如下：

`GraphBuilder`：

Map：将 Task3 的行作为输入的值，格式为 `Role0\t[Role1,x.xx|Role2,x.xx|...|RoleN,x.xx]`

Map 将之转化为 `Key:Role0 Value:1.0|Role1,x.xx|Role2,x.xx|...|RoleN,x.xx` 进行发射

```
public static class GraphBuilderMapper extends Mapper<Object, Text, Text, Text>
{
    @Override
    protected void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
        //输入的value的格式为:"Role0\t[Role1,x.xx|Role2,x.xx|...|RoleN,x.xx]"
        String cur_line = value.toString();
        String[] Items = cur_line.split("\t");
        String Init_PR = "1.0";
        String values = Init_PR + "|" + Items[1].substring(1,
Items[1].length() - 1);
        context.write(new Text(Items[0]), new Text(values));
        //发射Key:Role0 Value:1.0|Role1,x.xx|Role2,x.xx|...|RoleN,x.xx
    }
}
```

Reduce：该阶段直接将节点接收到的数据直接发射

```
public static class GraphBuilderReducer extends Reducer<Text, Text, Text, Text>
{
    protected void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {
        for (Text v : values) {
            context.write(key, v);
        }
    }
}
```

`PageRankIter`：

Map：输入格式为建图之后的输出，输入 value 为

`Role0\t[Role1,x.xx|Role2,x.xx|...|RoleN,x.xx]`

将 `<Key:thisRole,value:Role1,x.xx|Role2,x.xx|...|RoleN,x.xx>` 发射出去

因为  $PR\_new\_i = PR\_last\_0 * R\_0\_i + PR\_last\_1 * R\_1\_i + \dots + PR\_last\_(n-1) * R\_(n-1)\_i$

故对 `Role0` 的关系列表中的所有角色：

将 `<key:Role_i,value:last_PR*A_thisKey_i>` 发射出去

Map 阶段一个 `map` 函数发射了 `K+1` 条信息，`K` 为 `Role0` 的关系列表中角色个数

```
public static class PageRankIterMapper extends Mapper<Object, Text, Text, Text>
{
    @Override
    protected void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
        //Value格式 "Role0\tPR|Role1,x.xx|Role2,x.xx|...|RoleN,x.xx"
        String cur_line = value.toString();
        String []Items = cur_line.split("\t");
        //Items: ["Role0", "PR|Role1,x.xx|Role2,x.xx|...|RoleN,x.xx"]
        int index = Items[1].indexOf("|");
        String list = Items[1].substring(index + 1);
        //list: "Role1,x.xx|Role2,x.xx|...|RoleN,x.xx"
        context.write(new Text(Items[0]), new Text(list));
        //Key:Role0 Value:Role1,x.xx|Role2,x.xx|...|RoleN,x.xx
        String []values = Items[1].split("\\|");
        //values: ["PR","Role1,x.xx",...,"RoleN,x.xx"]
        double last_PR = Double.parseDouble(values[0]);
        for(int i = 1; i < values.length; ++i) {
            String[] cur_value = values[i].split(",");
            double val = last_PR * Double.parseDouble(cur_value[1]);
            context.write(new Text(cur_value[0]), new
Text(String.valueOf(val)));
            //根据PR_new_i = PR_last_0 * R_0_i + PR_last_1 * R_1_i + ... +
PR_last_(n-1) * R_(n-1)_i
        }
    }
}
```

`Combine`：如果收到的是 `<key:thisRole,value:Role1,x.xx|Role2,x.xx|...|RoleN,x.xx>` 列表信息，重新发射出去

如果收到的是其他信息，就把 `value: Double` 累加起来发射出去

```
public static class PageRankIterCombiner extends Reducer<Text, Text, Text, Text>
{
    protected void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {
        double count = 0.0;
        for(Text value : values){
            if(value.toString().contains(",")){
                context.write(key, value); //包含list重新发出去
            }
            else{
                count += Double.parseDouble(value.toString()); //否则累加
            }
        }
        context.write(key, new Text(String.valueOf(count)));
    }
}
```

Reduce:如果收到的是 <Key:thisRole,Value:Role1,x.xx|Role2,x.xx|...|RoleN,x.xx> 列表信息, 就记录下来

如果收到的是其他信息, 就把所有的PR值分量加起来

<Key:Role0,Value:PR\_new|Role1,x.xx|Role2,x.xx|...|RoleN,x.xx> 作为结果发射, 正好和 Map 输入相呼应

```
public static class PageRankIterReducer extends Reducer<Text, Text, Text, Text>
{
    protected void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {
        String value_list = "";
        double PR = 0.0;
        for(Text value : values){
            if(value.toString().contains(",")){
                value_list = value.toString();
            }
            else{
                PR += Double.parseDouble(value.toString());
            }
        }
        context.write(key, new Text(String.valueOf(PR) + "|" + value_list));
    }
}
```

RankViewer:

Map:输入 <Key:Role0,Value:PR|Role1,x.xx|Role2,x.xx|...|RoleN,x.xx>

输出 <Key:PageRank(DoubleWritable),Value:Role(Text)>

```
public static class RankViewerMapper extends Mapper<Object, Text,
DoubleWritable, Text> {
    @Override
    protected void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
        String[] split0 = value.toString().split("\t");
        String[] split1 = split0[1].split("\\|");
        context.write(new DoubleWritable(Double.parseDouble(split1[0])), new
Text(split0[0]));
        //发射Key:PageRank值,Value:Name
    }
}
```

Reduce:输入 <Key:PageRank(DoubleWritable),Value:Role(Text)>

输出 <Key:Role(Text),Value:PageRank(DoubleWritable)>

```

public static class RankViewerReducer extends Reducer<DoubleWritable, Text,
Text, DoubleWritable> {
    @Override
    public void reduce(DoubleWritable key, Iterable<Text> values, Context
context)
        throws IOException, InterruptedException {
        for (Text text : values) {
            context.write(text, key);
        }
    }
}

```

实现这个 MapReduce 排序任务的比较器：

```

public static class DecDoubleCompare extends WritableComparator {
    public DecDoubleCompare() {
        super(DoubleWritable.class, true);
    }
    @Override
    public int compare(WritableComparable a, WritableComparable b) {
        return -a.compareTo(b);
        //compareTo小于返回-1,大于返回1,完全相等返回0
        //所以compare方法大于返回-1,小于返回1,完全相等返回0,从大到小排序
    }
}

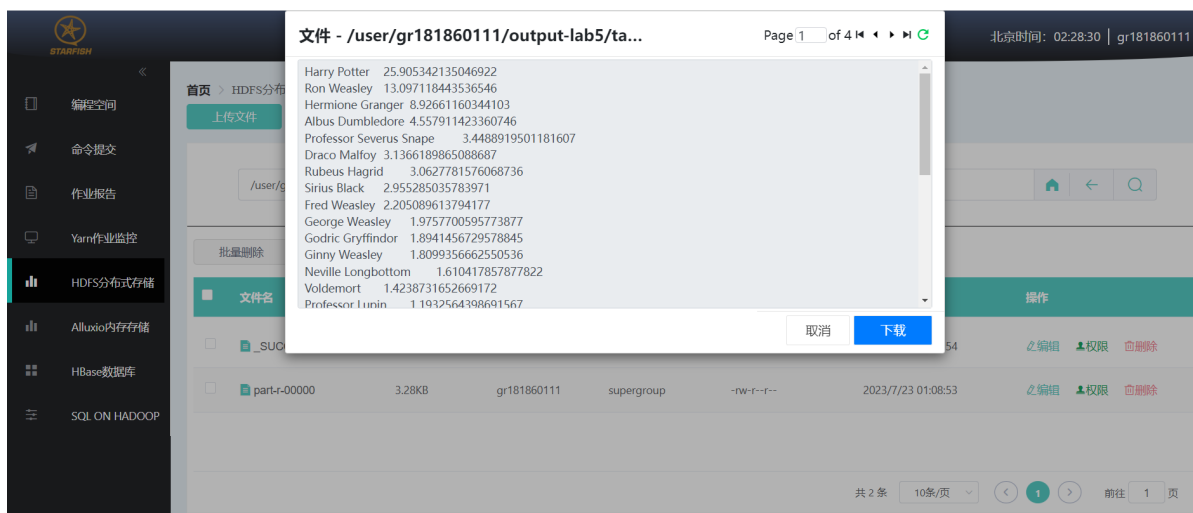
```

## 4.2. 实验结果

### 4.2.1. 输出文件路径

[程序输出路径] /user/gr181860111/output-lab5/task3/FinalRank

### 4.2.2. 部分输出结果



注:根据多次测试观察, 这个结果大概是迭代了14次左右输出的

## 任务5: 数据分析: 在人物关系图上的标签传播

## 5.1. 实验设计说明

### 5.1.1. 主要设计思路

任务5的需求：

采用标签传播算法根据之前的任务关系图将关联度比较大的人物分到同一标签。

在 Label Propagation 算法过程中，首先将每个节点的标签初始化为自己的名字，之后每一轮更新所有节点的标签，对于每一个节点，考察邻居节点的标签，将权重最大的标签作为该节点的新标签赋值给当前节点，当权重最多的标签不唯一时，随机选择一个标签赋值给当前节点。迭代直到各个人物标签基本稳定。

标签的更新采用了一个哈希表来存储每个任务最新的标签来获取更新的标签。

在 `map` 中,对上一轮的迭代结果逐行读入，将人物名、记录的标签、邻居及附带的比例信息逐一提取，发送<人物名,标签><人物名,邻居列表>两种键值对，并打上对应标签便于识别。

在 `reduce` 中，根据发送来的<邻居,标签#人物名>和类成员记录已更新的人名建立人名与标签的关系，遍历邻居列表，将标签与对应的权重放入一个哈希表中，最后选择权重最高的标签作为新标签。

`main` 函数中同时使用两个条件来控制迭代，分别是总次数和 `Is_Continue` 函数。`Is_Continue` 函数根据两次迭代的结果的标签变化比例来决定是否返回真。

为了结果的直观，最后将结果使用一个 `mapreduce` 过程将同一个标签的人物放在同一行，输出为最后的文件。

### 5.1.2. 主要算法和类设计

`LabelPropagationDriver` 类：

`Map` 类：

```
protected static class HarryMapper extends Mapper<LongWritable, Text, Text,
Text> {
    @Override
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        //输入value的格式: "Role0\tLabel[R1,x.xx|R2,x.xx|...|Rn-1,x.xx]"
        String line = value.toString();
        int index1 = line.indexOf("\t");
        int index2 = line.indexOf("[");
        int index3 = line.indexOf("]");
        String name = line.substring(0, index1); //name为Role0
        String label = line.substring(index1 + 1, index2); //label是标签
        String name_list = line.substring(index2 + 1, index3);
        //name_list是R1,x.xx|R2,x.xx|...|Rn-1,x.xx
        StringTokenizer tokenizer = new StringTokenizer(name_list, "|");
        while (tokenizer.hasMoreTokens()) {
            String[] neighbor = tokenizer.nextToken().split(",");
            context.write(new Text(neighbor[0]), new Text("1#" + label + "#"
+ name));
        }
        context.write(new Text(name), new Text("2#" + name_list));
        /*发射
        * 第一组:Key:Role1 value:"1#label#name"
        * 第二组:Key:name value:"2#name_list"
        */
    }
}
```



```
}
```

Reduce 类:

```
protected static class HarryReducer extends Reducer<Text, Text, Text, Text> {
    HashMap<String, String> name2label_hash = new HashMap<>();
    //name2label_hash存储已经更新好label的<Role,label>

    @Override
    public void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {
        //String label = "";
        //当前Key:Role 的label
        String name_list = "";
        //name_list记录R1,x.xx|R2,x.xx|...|Rn-1,x.xx
        HashMap<String, String> relation_name_label = new HashMap<>();
        //relation_name_label记录所有与当前的Key:Role1邻居的名字和标签 entry:
        <name,label>
        for (Text text : values) {
            String str = text.toString();
            if (str.charAt(0) == '1') {
                String[] neighbor = str.split("#");
                relation_name_label.put(neighbor[2], neighbor[1]);
            } else if (str.charAt(0) == '2') {
                name_list = str.split("#")[1];
            }
        }

        HashMap<String, Double> label_rank_map = new HashMap<>();
        //label_rank_map存储label和对应的label总边权和
        StringTokenizer name_list_token = new StringTokenizer(name_list,
        "|");

        //name_list_token:["R1,x.xx","R2,x.xx",..."Rn-1,x.xx"]
        while (name_list_token.hasMoreTokens()) {
            String[] name_rank = name_list_token.nextToken().split(",");
            //name_rank:["Ri","x.xx"]
            Double current_rank = Double.parseDouble(name_rank[1]);
            //current_rank x.xx, 是矩阵中A_Ri_thisKey
            String current_label = "";
            //current_label表示["Ri","x.xx"]中Ri这个角色的label
            //current_label更新好了就拿更新的,没更新好,就拿当前的Key的Label
            if (name2label_hash.containsKey(name_rank[0])) {
                current_label = name2label_hash.get(name_rank[0]);
            } else {
                current_label = relation_name_label.get(name_rank[0]);
            }
            Double label_rank;
            //将所有相同label的边权加起来,没有则加进去,有则更新其值
            if ((label_rank = label_rank_map.get(current_label)) != null) {
                label_rank_map.put(current_label, label_rank +
current_rank);
            } else {
                label_rank_map.put(current_label, current_rank);
            }
        }

        name_list_token = new StringTokenizer(name_list, "|");
    }
}
```

```

//重新分词,因为前面已经迭代遍历过了
double max_rank = Double.MIN_VALUE;
List<String> max_list = new ArrayList<>();
while (name_list_token.hasMoreTokens()) {
    String[] neighbor = name_list_token.nextTokens().split(",");
    //neighbor: ["Ri", "x.xx"]
    String current_label = "";
    //current_label Ri的label
    if (name2label_hash.containsKey(neighbor[0])) {
        current_label = name2label_hash.get(neighbor[0]);
    } else {
        current_label = relation_name_label.get(neighbor[0]);
    }
    double current_rank = label_rank_map.get(current_label);
    //current_rank表示当前的label的得分, 以下加入得分高的key
    if (max_rank < current_rank) {
        max_list.clear();
        max_rank = current_rank;
        max_list.add(neighbor[0]);
    } else if (max_rank == current_rank) {
        max_list.add(neighbor[0]);
    }
}

Random random = new Random();
int index = random.nextInt(max_list.size());
String target_name = max_list.get(index);
String target_label = "";
String my_name = key.toString();
if (name2label_hash.containsKey(target_name)) {
    target_label = name2label_hash.get(target_name);
} else {
    target_label = relation_name_label.get(target_name);
    name2label_hash.put(my_name, target_label);
}
//防止出现单个孤立节点
if (target_label == null) {
    target_label = my_name;
}
context.write(key, new Text(target_label + "[" + name_list + "]"));
}
}

```

LabelPropagationViewer 类:

Map 类:

```

public static class HarryMapper extends Mapper<Object,Text,Text,Text> {
    public void map(Object key,Text value,Context context)
        throws IOException, InterruptedException{
        String cur_line = value.toString();
        int index1= cur_line.indexOf("\t");
        int index2= cur_line.indexOf("[");
        String name=cur_line.substring(0,index1);
        String label=cur_line.substring(index1+1,index2);
        context.write(new Text(label),new Text(name));
    }
}

```

Reduce 类:

```

public static class HarryReducer extends Reducer<Text,Text,Text,Text> {
    public void reduce(Text key,Iterable<Text> values,Context context)
        throws IOException, InterruptedException{
        String output="";
        for(Text value:values){
            output=output+value.toString()+" ";
        }
        context.write(key,new Text(output));
    }
}

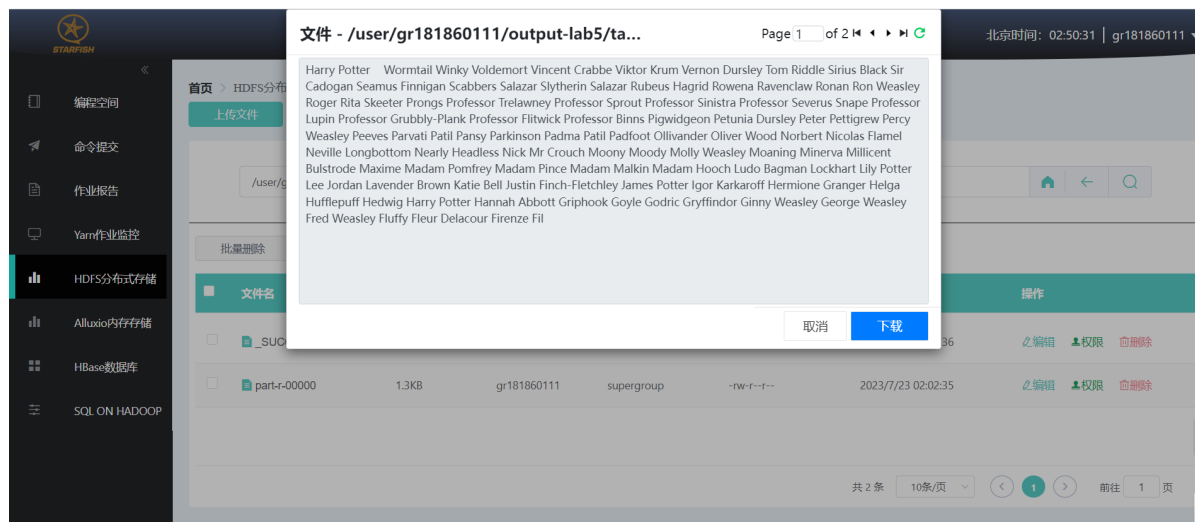
```

## 5.2. 实验结果

### 5.2.1. 输出文件路径

[输出文件路径] /user/gr181860111/output-lab5/task5/Final\_label1

### 5.2.2. 部分输出结果:



文件 - /user/gr181860111/output-lab5/ta... Page 1 of 2

Harry Potter Wormtail Winky Voldemort Vincent Crabbe Viktor Krum Vernon Dursley Tom Riddle Sirius Black Sir Cadogan Seamus Finnigan Scabbers Salazar Slytherin Salazar Rubeus Hagrid Rowena Ravenclaw Ronan Ron Weasley Roger Rita Skeeter Prongs Professor Trelawney Professor Sprout Professor Sinistra Professor Severus Snape Professor Lupin Professor Grubbly-Plank Professor Flitwick Professor Binns Pigwidgeon Petunia Dursley Peter Pettigrew Percy Weasley Peeves Parvati Patil Pansy Parkinson Padma Patil Padfoot Ollivander Oliver Wood Norbert Nicolas Flamel Neville Longbottom Nearly Headless Nick Mr Crouch Moony Moody Molly Weasley Moaning Minerva Millicent Bulstrode Maxime Madam Pomfrey Madam Pince Madam Malkin Madam Hooch Ludo Bagman Lockhart Lily Potter Lee Jordan Lavender Brown Katie Bell Justin Finch-Fletchley James Potter Igor Karkaroff Hermione Granger Helga Hufflepuff Hedwig Harry Potter Hannah Abbott Griphook Goyle Godric Gryffindor Ginny Weasley George Weasley Fred Weasley Fluffy Fleur Delacour Firenze Fil

操作

取消 下载

共 2 条 10条/页 1 页

