

大数据处理综合实验

实验二 倒排索引 实验报告

组别：2020st39

组长：171860662 山越 1025331716@qq.com

目录

大数据处理综合实验.....	1
实验二 倒排索引 实验报告.....	1
组别：2020st39.....	1
组长：171860662 山越 1025331716@qq.com.....	1
一 . 实验设计.....	1
二 . 程序运行和实验结果说明.....	8
三 . 遇到的问题和不足之处.....	12
四 . 小组成员分工.....	12

一 . 实验设计

(一) 基础部分：带词频属性的文档倒排算法

1. 设计思路

由于需要实现带词频属性的文档倒排算法，所以我们按照课上所讲内容采取了基本的倒排索引结构，即一个单词对应多个包含文件名和有效负载的 posting。同时，考虑到直接采用这种结构所导致的规模瓶颈，我们对 Mapper 传递给 Reducer 的键值对做了 value-to-key conversion，以方便对传入 Reducer 的信息进行排序。

我们采用了默认的 InputFormat，即 map 函数的输入键值对为<行号, 该行内容>，所以输入 key 和 value 的类型分别是 Object 和 Text。map 中完成的动作为：对行内的每个词，发射当前文档名和出现一次的组合字符串。因为要实现值-键转换，所以 map 函数的输出键值对为<单词#文档名, 1>，即 key 为单词与文档名的组合字符串，类型为 Text；value 直接设为 1，类型为 IntWritable。

因为 map 的输出键值对中，key 不再是单纯的单词，所以为保证 shuffle 的过程中将同

一单词的记录发给同一个 Reducer，需要重新定义 Partitioner，使其分发记录时只根据 key 中的单词部分进行分发。

对于 reduce 函数，它会不断接收到上述键值对，且已根据其输入 key 进行了排序，即同一单词的记录总会相邻出现。此时，由于 Reducer 接收到的键值对是由 Mapper 输出的，且经过归并将相同 key 的 value 组成了列表，所以其输入 key 的类型仍为 Text，而输入 value 的类型则为 `Iterable<IntWritable>`。而 reduce 中完成的动作为：对某个单词，统计其在每个文档中出现的次数，统计其出现过的文档数，统计其出现过的总次数，最后算出词频、拼接得到输出 value、完成发射。所以最后输出的 key 为该单词，value 为其词频等信息，类型均为 Text。

2. 源代码说明

InvertedIndexMapper.java

```
//InvertedIndexMapper.java
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

public class InvertedIndexMapper extends Mapper<Object, Text, Text, IntWritable>
{
    @Override
    protected void map(Object key, Text value, Context context)
        // default RecordReader: LineRecordReader
        // key: line offset; value: line string
        throws IOException, InterruptedException
    {
        FileSplit fileSplit = (FileSplit)context.getInputSplit();
        String fileName = fileSplit.getPath().getName(); //获取文件名
        int pos = fileName.indexOf(".");
        if (pos > 0) {
            fileName = fileName.substring(0, pos); //去除.txt.segmented文件名后缀
        }

        Text word = new Text();
        StringTokenizer itr = new StringTokenizer(value.toString()); //分词
        IntWritable one = new IntWritable(1); //预备发射的value，表示出现一次
        while(itr.hasMoreTokens()) {
            word.set(itr.nextToken() + "#" + fileName); //预备发射的key，格式为"单词#文件名"
            context.write(word, one); //发射键值对<单词#文件名, 1>
        }
    }
}
```

InvertedIndexPartitioner.java

```
//InvertedIndexPartitioner.java
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.lib.partition.HashPartitioner;

public class InvertedIndexPartitioner extends HashPartitioner<Text, IntWritable> {

    @Override
    public int getPartition(Text key, IntWritable value, int numReduceTasks) {
        String term = key.toString().split("#")[0]; //把原来的key按照#拆分，取#前的单词部分
        return super.getPartition(new Text(term), value, numReduceTasks);
        //用单词部分进行Partition，确保同一单词发给同一Reducer
    }
}
```

InvertedIndexReducer.java

```
//InvertedIndexReducer.java
import java.io.IOException;
import java.util.Iterator;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class InvertedIndexReducer extends Reducer<Text, IntWritable, Text, Text> {
    // setup
    private String term = new String(); //存放当前key中的单词部分
    private String termPrev = " "; //存放之前传入的key中的单词部分
    private StringBuilder postingList = new StringBuilder(); //postings
    private int countWord = 0; //用以记录当前单词在所有文件中出现的总数
    private int countDoc = 0; //用以记录包含当前单词的文件数
    private float frequency = 0; //词频

    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
        this.term = key.toString().split("#")[0]; //从传入的key中拆分出单词
        if (!this.term.equals(this.termPrev)) {
            if (!this.termPrev.equals(" ")) { //当前单词与之前单词不同，且不是初始状态，所以之前单词的处理应该到此为止，应将其发射出去
                this.postingList.setLength(this.postingList.length() - 1); //消去结尾的";"
                this.frequency = (float)this.countWord / (float)this.countDoc; //计算词频
                context.write(new Text(this.termPrev), new Text(String.format("%.2f,%s", this.frequency, this.postingList.toString())));
                //发射键值对<单词, postingList>，其中postingList包含词频和该词在每个文档中出现的次数
                //下面需要统计当前新单词的数量等信息，所以重置变量的状态
                this.countWord = 0;
                this.countDoc = 0;
                this.postingList = new StringBuilder();
            }

            this.termPrev = this.term; //当前单词与之前单词不同，则更新termPrev
        }

        int sum = 0; //记录当前value列表中的数量，即key（单词#文件名）表示的该单词在该文件中出现的次数
        Iterator<IntWritable> it = values.iterator();
        while (it.hasNext()) { //遍历value列表，累加
            sum += it.next().get();
        }

        this.postingList.append(key.toString().split("#")[1] + ":" + sum + ";"); //在postingList中加入一个posting
        this.countWord += sum; //countWord自增当前单词在当前文件中出现的次数
        ++this.countDoc; //countDoc自增1
    }

    @Override
    public void cleanup(Context context) throws IOException, InterruptedException { //close
        //主要完成最后一个单词的键值对完善和发射，过程与之前类似
        this.postingList.setLength(this.postingList.length() - 1);
        this.frequency = (float)this.countWord / (float)this.countDoc;
        context.write(new Text(this.termPrev), new Text(String.format("%.2f,%s", this.frequency, this.postingList.toString())));
    }
}
```

InvertedIndexer.java

```
//InvertedIndexer.java
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class InvertedIndexer
{
    public static void main(String[] args)
    {
        try
        {
            Configuration conf = new Configuration();
            Job job = Job.getInstance(conf, "InvertedIndex");

            job.setJarByClass(InvertedIndexer.class);
            job.setMapperClass(InvertedIndexMapper.class);
            job.setPartitionerClass(InvertedIndexPartitioner.class);
            job.setReducerClass(InvertedIndexReducer.class);
            job.setOutputKeyClass(Text.class); //map的输出key类型为Text
            job.setOutputValueClass(IntWritable.class); //map的输出value类型为IntWritable

            FileInputFormat.addInputPath(job, new Path(args[0]));
            FileOutputFormat.setOutputPath(job, new Path(args[1]));
            System.exit(job.waitForCompletion(true) ? 0 : 1);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

(二) 选做部分一：针对词频进行全局排序

1. 设计思路

与基础部分类似，对每个词语的平均出现次数进行全局排序只能由 MapReduce 框架完成，内部排序会面临规模瓶颈。所以我们使用上述基础部分的输出文件作为全局排序的输入，此时我们选用在每行里分别读取 key 和 value 的输入方式作为 InputFormat，并对读取的键值对做 value-to-key conversion，将输入 value 中的词频部分作为输出的 key，而输出的 value 则用来保存信息。

我们采用了 KeyValueTextInputFormat，即 map 函数对输入文件的每一行分别读取 key 和 value，输入 key 和 value 的类型分别是 Object 和 Text。map 中完成的动作为：将输入 value 中的词频部分拆分出来作为输出的 key，而输出 value 则用来保存信息。因为要便于 MapReduce 框架进行排序，所以输出 key 类型为 FloatWritable，输出 value 类型为 Text。

对于 reduce 函数，它接收到的键值对已根据词频进行了排序。此时，其输入 key 的类型仍为 FloatWritable，而输入 value 的类型则为 Iterable<Text>。由于已完成排序，所以 reduce 中完成的动作仅仅是将 value 中保存的信息拆分开，那么最后输出的 key 为该单词，value

为其词频等信息，类型均为 Text。

2. 源代码说明

GlobalSortMapper.java

```
//GlobalSortMapper.java
import java.io.IOException;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class GlobalSortMapper extends Mapper<Object, Text, FloatWritable, Text> {
    @Override
    protected void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        //此时Mapper以基础部分的输出为输入，所以key是单词，value是词频以及该词在不同文件中出现的次数
        String num = value.toString().split(",")[0]; //从value中拆分出词频
        FloatWritable newKey = new FloatWritable(Float.parseFloat(num)); //由词频构成将要发射的key
        String tmpValue = key.toString() + "#" + value.toString(); //将要发射的value由单词和输入value组成，格式为"单词#词频,postings"
        context.write(newKey, new Text(tmpValue)); //发射键值对
    }
}
```

GlobalSortReducer.java

```
//GlobalSortReducer.java
import java.io.IOException;
import java.util.Iterator;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class GlobalSortReducer extends Reducer<FloatWritable, Text, Text, Text> {
    protected void reduce(FloatWritable key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
        Iterator<Text> itr = values.iterator();

        while(itr.hasNext()) {
            String tmp = ((Text)itr.next()).toString();
            String[] buf = tmp.split("#"); //将单词和词频等信息拆分
            context.write(new Text(buf[0]), new Text(buf[1])); //发射键值对
        }
    }
}
```

GlobalSort.java

```
//GlobalSort.java
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.KeyValueTextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class GlobalSort {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        if (args.length != 2) {
            System.err.println("Usage: InvertedIndex <in> <out>");
            System.exit(2);
        }

        Job job = Job.getInstance(conf, "InvertedIndex");
        job.setJarByClass(GlobalSort.class);
        job.setInputFormatClass(KeyValueTextInputFormat.class); //读取文件时，在每行中分别读取key和value
        job.setMapperClass(GlobalSortMapper.class);
        job.setReducerClass(GlobalSortReducer.class);
        job.setOutputKeyClass(FloatWritable.class); //map的输出key类型为FloatWritable
        job.setOutputValueClass(Text.class); //map的输出value类型为Text
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

(三) 选做部分二：计算 TF-IDF

1. 设计思路

鉴于基础部分的输出结果已经包含了计算 TF-IDF 的几乎全部信息，所以这一部分的输入定为基础部分的输出文件。不过，还缺少了语料库文档总数这个参数，所以要将语料库路径同样作为该部分 jar 包的运行参数，统计路径下的文件数量，将其通过可以全局传递的 Configuration 类型的变量 conf 传递给 Mapper。

这里我们同样采用了 KeyValueTextInputFormat，即 map 函数对输入文件的每一行分别读取 key 和 value，输入 key 和 value 的类型分别是 Object 和 Text。map 中完成的动作为：将输入 value 中的除词频之外的部分拆分出来进行分析，对于拆分的每一项，都表示输入 key 所代表的单词在某个文档中出现的次数；然后对拆分出的每一项，提取其中的作者名和出现次数计入一个 Map 表里，也就是说，该表记录当前单词在不同作者的文档中的 TF；最后遍历该表，计算 TF*IDF 并发射结果。这里的输出 key 为作者名，输出 value 为单词和 TF-IDF 的组合字符串，所以类型均为 Text。

由于 Mapper 的输出结果经过框架自动排序后已经可以作为整体输出，所以这一部分使用默认 Reducer。

2. 源代码说明

Tfidf.java

```
//Tfidf.java
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.KeyValueTextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class Tfidf {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        if (args.length != 3) {
            System.err.println("Usage: Tfidf <in> <out> <raw-in>");
            //输入参数个数为3，分别为InvertedIndex的输出文件、输出路径、InvertedIndex的原始输入文件
            System.exit(2);
        }

        FileSystem fs = FileSystem.get(conf);
        FileStatus []files = fs.listStatus(new Path(args[2])); //获取语料库文档总数
        conf.setInt("fileNumbers", files.length); //将上述语料库文档总数借助变量conf进行全局传递

        Job job = Job.getInstance(conf, "Tfidf");
        job.setJarByClass(Tfidf.class);
        job.setInputFormatClass(KeyValueTextInputFormat.class);
        //由于读取的是InvertedIndex的输出文件，所以对于每行分别读取key和value
        job.setMapperClass(TfidfMapper.class);
        //无须指定Reducer行为，所以使用默认Reducer
        job.setOutputKeyClass(Text.class); //map的输出key类型为Text
        job.setOutputValueClass(Text.class); //map的输出value类型为Text
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

TfidfMapper.java

```
//TfidfMapper.java
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class TfidfMapper extends Mapper<Object, Text, Text, Text> {
    protected void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        String []termFreq = value.toString().split(",")[1].split(";");
        //将InvertedIndex的输出value的除了词频以外的部分拆分，每一项格式为"文档名:单词出现次数"
        int fileNumbers = Integer.parseInt(context.getConfiguration().get("fileNumbers")); //获取语料库文档总数
        double idf = Math.Log((double)fileNumbers/(termFreq.length + 1)); //计算IDF
        Map<String, Integer> authorMap = new HashMap<>(); //存放键值对<作者, TF>
        for(String term:termFreq) { //对termFreq中每一项，从中拆分出作者和出现次数，填入authorMap
            String authorNumArticle = term.split(":")[0]; //对termFreq中每一项，用":"拆分
            String author = ""; //存储作者名
            for (int i = 0; i < authorNumArticle.length(); ++i) {
                if (authorNumArticle.charAt(i) < '0' || authorNumArticle.charAt(i) > '9')
                    //拆分出数字之前的作者名
                    author = author + authorNumArticle.substring(i, i + 1);
                else
                    break;
            }
            int freq = Integer.parseInt(term.split(":")[1]); //得到当前单词在当前文档中的出现次数
            if(authorMap.containsKey(author)) //若authorMap中存在key为author，那么将其value自增freq
                authorMap.put(author, authorMap.get(author)+freq);
            else //若不存在，则直接置为freq
                authorMap.put(author, freq);
        }
        for(Map.Entry<String, Integer> entry:authorMap.entrySet()) {
            Text name = new Text();
            name.set(entry.getKey());
            if(!entry.getKey().equals("")) //对authorMap中有效的每一项，发射键值对<作者, 单词#TF*IDF>
                context.write(name, new Text(key + "#" + String.format("%.02f", entry.getValue()*idf)));
        }
    }
}
```

二．程序运行和实验结果说明

(一) 输出结果

1. 基础部分的输出文件

输出文件在 HDFS 上的路径：/user/2020st39/Lab2_out/basic_out



2. “江湖”、“风雪”两个单词的输出结果

江湖：

江湖 116.06,卧龙生01:275;卧龙生02:329;卧龙生03:402;卧龙生04:105;卧龙生05:298;卧龙生06:244;卧龙生07:269;卧

风雪：

风雪 4.53,卧龙生01:3;卧龙生07:16;卧龙生08:1;卧龙生09:1;卧龙生12:36;卧龙生15:4;卧龙生18:9;卧龙生19:2;卧龙生22

3. 针对词频进行全局排序的输出文件

输出文件在 HDFS 上的路径：/user/2020st39/Lab2_out/sort_out

File - /user/2020st39/Lab2_out/sort_out/...
Page 1 of 13693

套套 1.00,卧龙生01:1;卧龙生02:1;卧龙生03:1;卧龙生05:1;卧龙生32:1;卧龙生33:1;卧龙生41:1;卧龙生46:1;卧龙生49:1;卧龙生54:1;李凉02:1;李凉04:1;李凉09:1;李凉11:1;李凉13:1;李凉17:1;李凉20:1;李凉31:1;李凉37:1;李凉39:1;梁羽生02:1;梁羽生03:1;梁羽生37:1;金庸07鹿鼎记:1
坚强有力 1.00,古龙34:1;古龙39:1;古龙59:1;李凉20:1;梁羽生10:1;金庸12倚天屠龙记:1
套口供 1.00,卧龙生22:1;李凉23:1;梁羽生07:1;梁羽生23:1
套下来 1.00,卧龙生05:1;卧龙生12:1;卧龙生41:1;古龙18:1;梁羽生10:1
套下去 1.00,卧龙生41:1;古龙09:1
套下 1.00,古龙26:1;李凉23:1;李凉29:1;梁羽生12:1;梁羽生16:1;梁羽生19:1;梁羽生22:1;梁羽生24:1;梁羽生25:1;梁羽生36:1;梁羽生37:1
套上去 1.00,卧龙生14:1;古龙39:1
坚忍不拔 1.00,李凉25:1;梁羽生17:1
坚执不从 1.00,金庸08笑傲江湖:1
奖许 1.00,金庸09书剑恩仇录:1;金庸13碧血剑:1
奖给 1.00,李凉10:1
奖杯 1.00,梁羽生32:1

Cancel
Download

4. TF-IDF 的输出文件

输出文件在 HDFS 上的路径： /user/2020st39/Lab2_out/TFIDF_out


File - /user/2020st39/Lab2_out/TFIDF_ou...
Page 1 of 2304

卧龙生 0#10.21
卧龙生 龟鹤遐龄#4.69
卧龙生 龟裂#5.35
卧龙生 龟背#21.43
卧龙生 龟肉#8.00
卧龙生 龟缩#12.76
卧龙生 龟甲#15.99
卧龙生 龟山#13.22
卧龙生 龟头#18.49
卧龙生 龟壳#38.44
卧龙生 龟兹#3.78
卧龙生 龟#265.81
卧龙生 鼋#7.65
卧龙生 龚#696.98
卧龙生 龙驹#10.44

Cancel
Download

(二) WebUI 执行报告

1. 基础部分

 **MapReduce Job job_1572597966684_3218** Logged in as: dr.who


Job Overview

Job Name: InvertedIndex
User Name: 2020st39
Queue: root.team39
State: SUCCEEDED
Uberized: false
Submitted: Wed Apr 22 12:46:23 CST 2020
Started: Wed Apr 22 12:46:28 CST 2020
Finished: Wed Apr 22 12:49:16 CST 2020
Elapsed: 2mins, 47sec
Diagnostics:
Average Map Time: 3sec
Average Shuffle Time: 1mins, 37sec
Average Merge Time: 8sec
Average Reduce Time: 38sec

ApplicationMaster		Start Time	Node	Logs
Attempt Number				
1		Wed Apr 22 12:46:24 CST 2020	slave007:8042	logs

Task Type	Total	Complete	
Map	218	218	
Reduce	1	1	
Attempt Type	Failed	Killed	Successful
Maps	0	0	218
Reduces	0	0	1

2. 针对词频进行全局排序

 **MapReduce Job job_1572597966684_3220** Logged in as: dr.who


Job Overview

Job Name: InvertedIndex
User Name: 2020st39
Queue: root.team39
State: SUCCEEDED
Uberized: false
Submitted: Wed Apr 22 12:56:28 CST 2020
Started: Wed Apr 22 12:56:31 CST 2020
Finished: Wed Apr 22 12:56:52 CST 2020
Elapsed: 20sec
Diagnostics:
Average Map Time: 4sec
Average Shuffle Time: 4sec
Average Merge Time: 0sec
Average Reduce Time: 0sec


ApplicationMaster		Start Time	Node	Logs
Attempt Number				
1		Wed Apr 22 12:56:28 CST 2020	slave017:8042	logs

Task Type	Total	Complete	
Map	1	1	
Reduce	1	1	
Attempt Type	Failed	Killed	Successful
Maps	3	0	1
Reduces	0	0	1

3. TF-IDF



MapReduce Job job_1572597966684_3222

Logged in as: dr.who

Job Overview

Job Name: TfIdf

User Name: 2020st39

Queue: root.team39

State: SUCCEEDED

Uberized: false

Submitted: Wed Apr 22 13:06:10 CST 2020

Started: Wed Apr 22 13:06:11 CST 2020

Finished: Wed Apr 22 13:06:29 CST 2020

Elapsed: 17sec

Diagnostics:

Average Map Time: 8sec

Average Shuffle Time: 2sec

Average Merge Time: 0sec

Average Reduce Time: 1sec

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Wed Apr 22 13:06:08 CST 2020	slave014:8042	logs

Task Type	Total	Complete
Map	1	1
Reduce	1	1

Attempt Type	Failed	Killed	Successful
Maps	0	0	1
Reduces	0	0	1

三 . 遇到的问题 and 不足之处

1. 计算 TF-IDF 时，我们将几乎所有动作都交由 Mapper 完成。反思时我们发现，当时的思路不够开阔，使得 Reducer 的功能没有发挥出来。其实这个 Job 同样可以由 Mapper 分发、在 Reducer 中完成统计，在效率上可能会有所提升。

四 . 小组成员分工

学号	姓名	分工
171860662	山越	提交集群和实验报告撰写
171860663	马少聪	基础和选做部分代码实现
171860664	谢鹏飞	基础和选做部分代码实现
171860681	冯旭晨	数据测试