文档倒排索引

1. 实验设计

参考教学ppt及书上示例代码，文档倒排索引主要设计以下函数:

**Mapper函数：**

由于此次倒排索引需要携带词频属性，因此在从文件中获取token之后，需要添加filename来共同作为键值对中的key。Token与filename之间以#相连，组成的key为text类型。在map阶段，每次生成的key其value值均为1，类型为IntWritable。

|  |
| --- |
| **private** **static** **final** IntWritable one = **new** IntWritable(1);  **private** Text word = **new** Text();  StringTokenizer tokenizer = **new** StringTokenizer(value.toString().toLowerCase());  **while** (tokenizer.hasMoreTokens()) {  word.set(tokenizer.nextToken() + "#" + fileName);  context.write(word, one);  } |

**Combiner函数**：

Mapper输出的中间结果中会包含大量相同主键的键值对，combiner将mapper输出的中间结果中的词频进行累加，来减少向reduce节点中传输的数据量。

|  |
| --- |
| int sum = 0;  **for** (IntWritable val : values)  sum += val.get();  result.set(sum);  context.write(key, result); |

**Partitioner函数：**

由于我们的key是token#filename 的形式，但我们是想对token进行词频统计。为了保证具有相同token的键值对可以分配到相同的reduce节点，因此需要暂时将key（token#filename）进行拆分。

|  |
| --- |
| **public** int getPartition(Text key, IntWritable value, int numReduceTasks) {  String term = key.toString().split("#")[0];  **return** **super**.getPartition(**new** Text(term), value, numReduceTasks);  } |

**Reducer函数：**

Reducer从patitioner拿到键值对后，需要对相同token在不同file中的计数进行叠加。因此此时我们需要把原本的key（word#filename）进行拆分，k-v由word#filename – sum 转化为word – filename：sum。在这部分我们使用lastword来记录上一个处理的token，curword记录当前处理的token，维护一个string类型的队列 postingList来保存相同token的文件:词频信息(即保存内容为 filename：sum 的字符串)，当某一token处理完成时，list的大小即为包含此token的文件数，然后将此list清空，然后用来继续保存下一个token的信息。

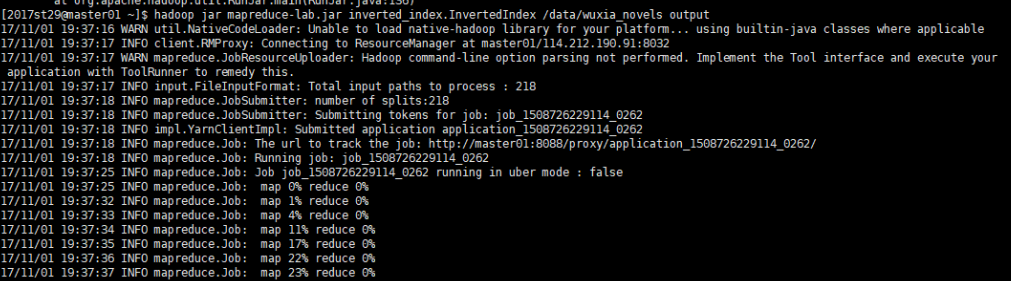
|  |
| --- |
| String[] keyPair = key.toString().split("#");  curWord.set(keyPair[0]);  String fileName = keyPair[1];  int sum = 0;  **for** (IntWritable val : values)  sum += val.get();  **if** (!lastWord.equals(curWord) && !postingList.isEmpty())*//当前token与上一个不同，即上一个token已处理完毕*  commitResult(context); *//自定义函数，详见下*  postingList.add(fileName + ": " + sum);  totalCount += sum;  lastWord.set(keyPair[0]); |
| **private** void commitResult(Context context) *//自定义函数*  **throws** IOException, InterruptedException {  StringBuilder builder = **new** StringBuilder();  builder.append(totalCount / (double) postingList.size()); *//平均词频*  builder.append(", ");  **for** (String str : postingList) {  builder.append(str); *//写入filename：sum*  builder.append("; ");  }  context.write(lastWord, **new** Text(builder.toString()));  totalCount = 0; *//清空，开始计算下一token*  postingList.clear();  } |

1. 运行截图

集群上执行InvertedIndex程序指令：

hadoop jar mapreduce-lab.jar inverted\_index.InvertedIndex /data/wuxia\_novels output

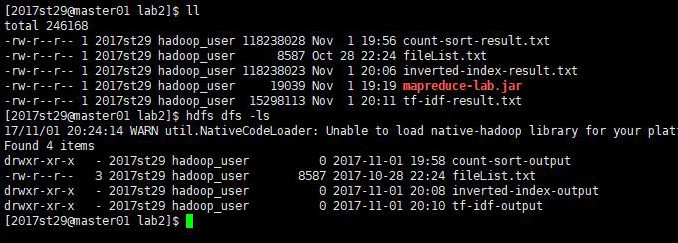
其输出内容前半部分：

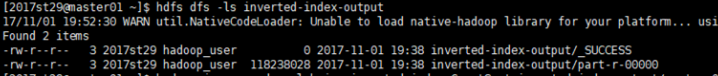




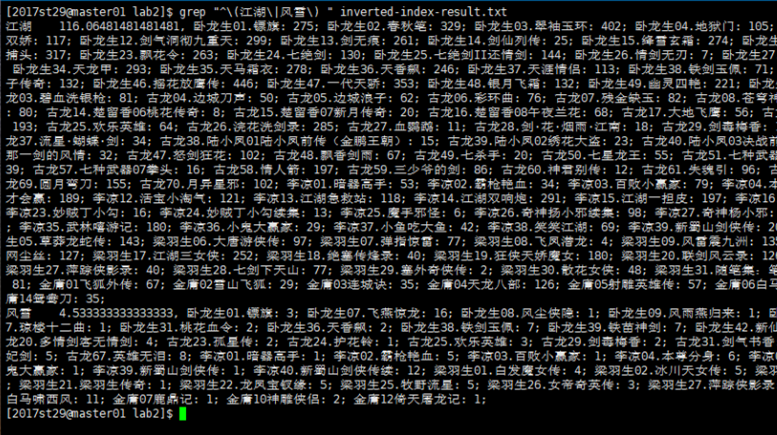
集群中输出文件的位置在$HOME/lab2中，

Hdfs下的路径即在inverted-index-output文件夹下





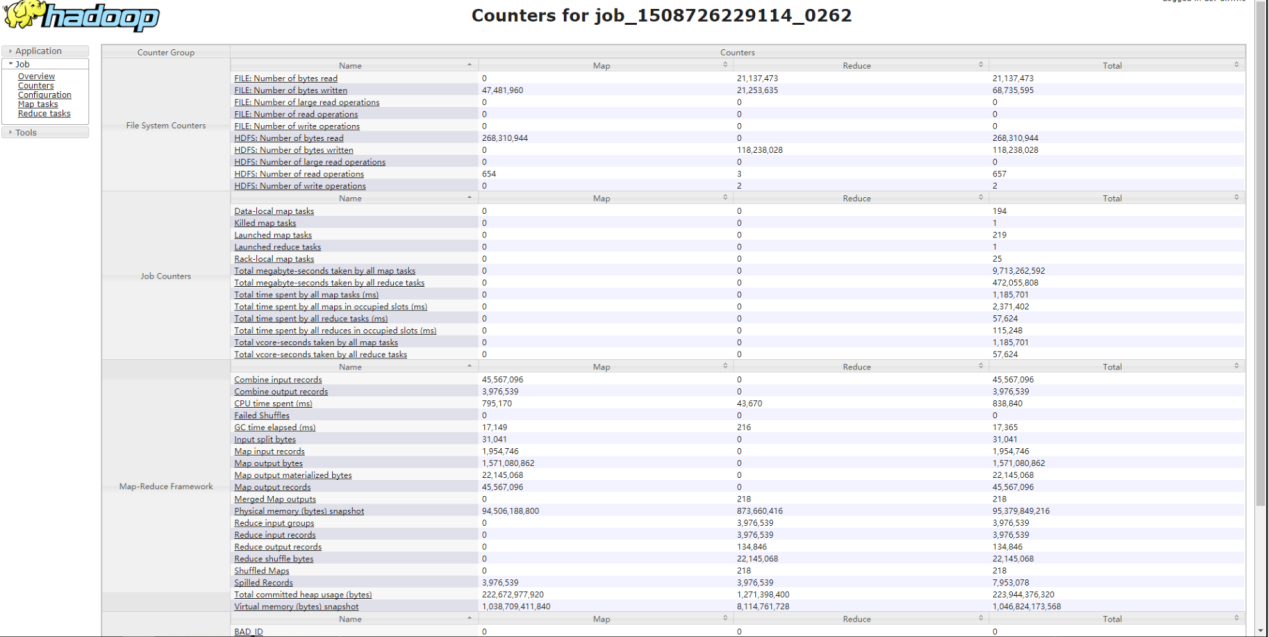
以“江湖”、“风雪”两个单词为例，它们的输出结果为（部分）：

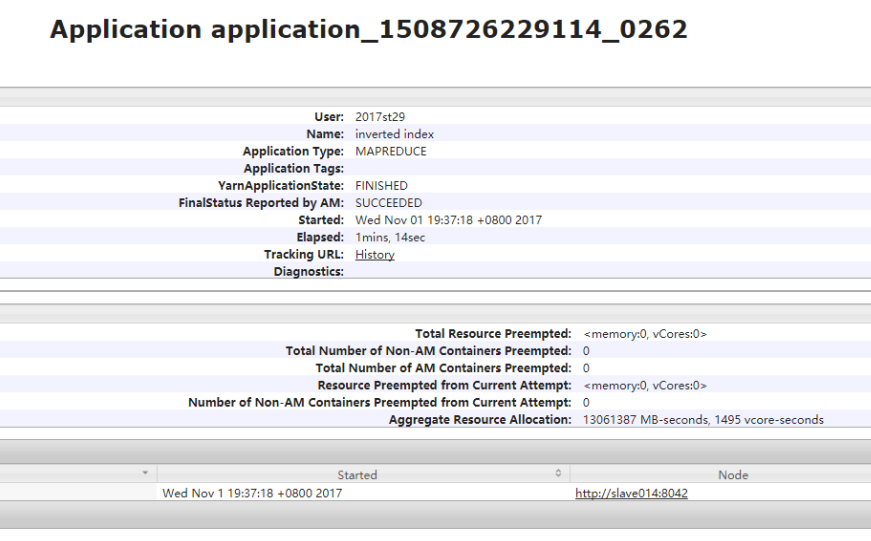






Mapreduce Job执行截图：





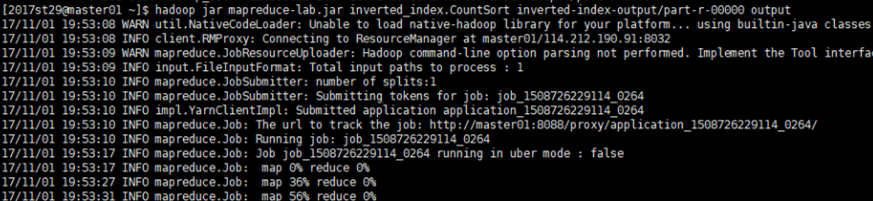
选做题：

两道选做题均以上述倒排索引算法的输出作为输入文件，代码以一并打包入mapreduce-lab.jar中。其集群本地输出文件与hdfs输出路径与倒排算法相同，在上面的路径截图的虚线框中体现。

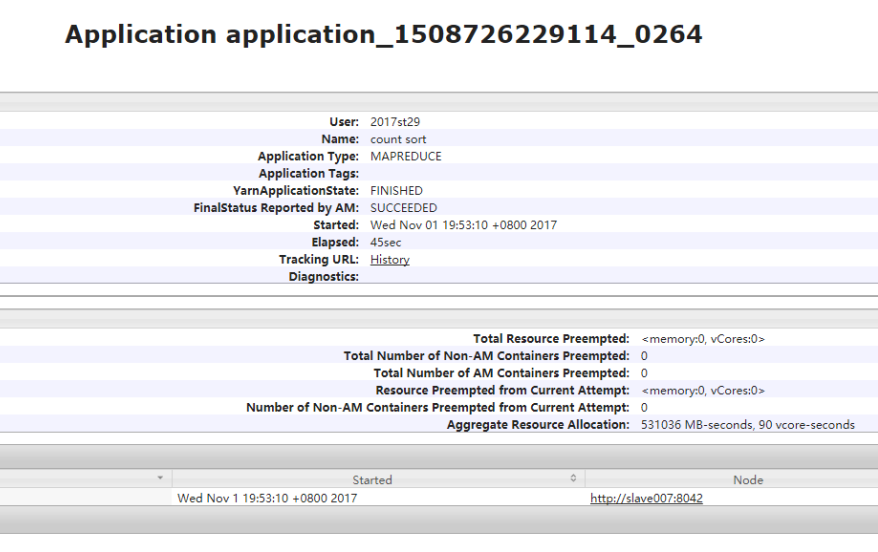
对每个词语的平均出现次数全局排序算法的执行命令为：

hadoop jar mapreduce-lab.jar inverted\_index.CountSort inverted-index-output/part-r-00000 output

其执行截图如下：







对每位作家，计算每个词语的TF-IDF算法的执行语句为：

hadoop jar mapreduce-lab.jar inverted\_index.TfIdf fileList.txt inverted-index-output/part-r-00000 output

其执行截图如下：

