FBDP-作业6

邵一淼 191098180

需求分析

在作业5的数据集基础上完成莎士比亚文集单词的倒排索引，输出按字典序对单词进行排序，单词的索引按照单词在该文档中出现的次数从大到小排序。单词忽略大小写，忽略标点符号（punctuation.txt），忽略停词（stop-word-list.txt），忽略数字，单词长度>=3。输出格式如下：

单词1: 文档i#频次a, 文档j#频次b...

单词2: 文档m#频次x, 文档n#频次y...

需求总体与老师提供的InvertedIndex源码相差不多，只需要在完成倒排索引之后，对每个文档中的频次排序，可以使用treemap来实现排序

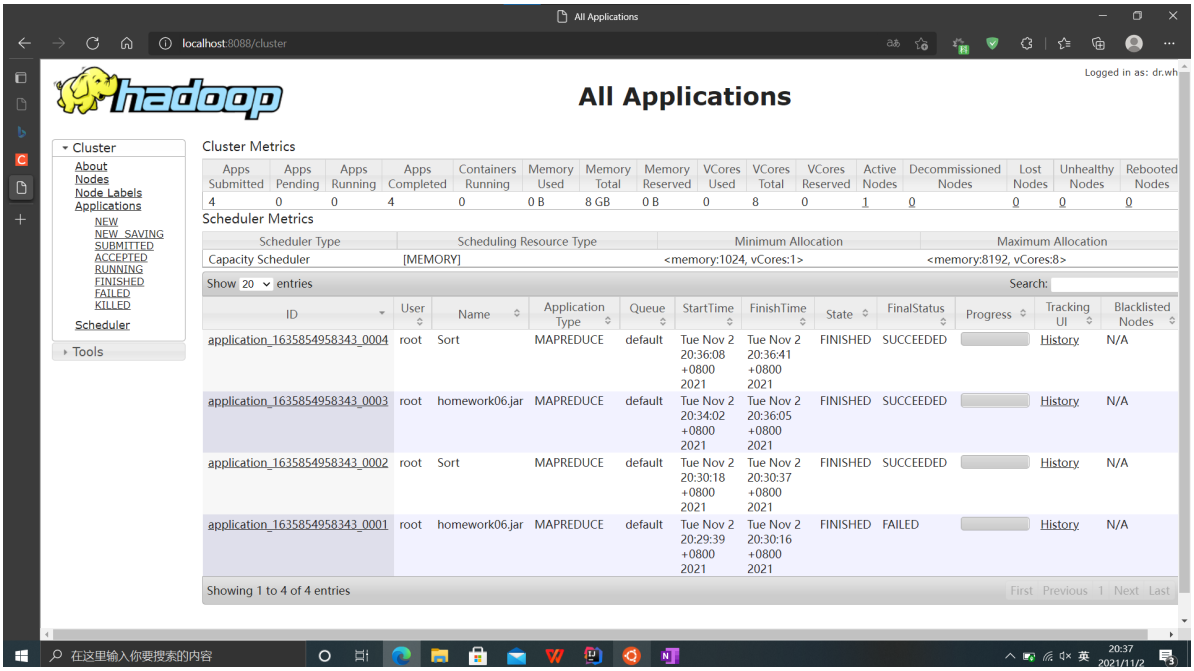
设计思路

类	功能
InvertedIndexRunner	程序的主类
InvertedIndexMapper	程序的mapper
InvertedIndexReducer	程序的reducer
InvertedIndexCombiner	程序的combiner
InvertedIndexSort	完成排序任务

实验结果

实验结果保存在results文件夹下

以下是集群运行截图



实验改进

1. 减少Job

   本实验的设计思路为设计两个job，一个job完成倒排索引，一个job完成对单词在文档中出现的次数排序。第二个job可省略，在第一个job写入的时候就可以完成排序。

2. 算法的更迭

   思路一、最开始

```
class MAPPER
    procedure MAP(docid n, doc d)
        for all term t in doc d
            EMIT(term t, <n, 1>)

class REDUCER
    method INITIALIZE
        t(pre) <-- null
    procedure REDUCER(term t, postings[<docid n1, tf1>,<docid n2, tf2>....])
        P <-- new ASSOCIATIVE_SORTED_MAP
        if t(pre) != t AND t(pre) != null
            EMIT(t, P)
            P.RESET
        for all posting<n, tf> in postings[....]
            P{n, tf} = P{n, tf++};
    method CLOSE
        EMIT(term t, P)
```

   以上实现中，Mapper每次都提交一个三元组<term t, docid n, 1>，然后在Reducer端，在对相同的term进行每个doc 的tf统计，最后提交。方法确实很简单，基本上所有的工作都在Reducer端完成，Mapper端只是分解doc提交而已，如果一篇文章很长，每个term出现的次数又比较多，那么这样导致 mapper提交次数过多，生成的中间结果过多，网络传输就会很频繁，从来在shuffle和sort阶段很费时费力。思路二就简单解决了这个问题

   思路二、

```
class MAPPER
    procedure MAP(docid n, doc d)
        H <-- new ASOOCIATIVEARRAR
        for all term t in doc do
            H{t} <-- H{t} + 1
        for all term t in H do
            EMIT(term t, posting<n, H{t}>)

class REDUCER
    procedure REDUCER(term t, postings[<docid n1, tf1>,<docid n2, tf2>....])
        P <-- new LIST
        for all posting <a,f> in postings[<n1,f1>,<n2,f2>..]
            APPEND(P,<a,f>)
        SORT(P)
        EMIT(term t, posting P)
```

该实现在mapper端词频做了统计，然后再提交各Reducer，这样就大大减少了mapper的提交次数。这样还是存在问题的。
基本的MapReduce执行框架不能保证Mapper端提交的value被发送到reducer的排序问题，就是说，不能保证在Reducer端相同的key所对应的value列表的值是有序的。reducer端对于相同的key必须缓存其值列表。然后再在内存进行排序，最后在写到磁盘，这样很有可能会导致reducer端out-of-memory 。

一个比较好的解决办法是让MapReduce框架帮我们做value的排序工作，以此来保证传递给reducer的相同的key所对应的values列表值是有序的。

思路三、

```
class MAPPER
    method MAP(docid n, doc d)
        H <-- new ASOOCIATIVEARRAR
        for all term t in doc d do
            H{t} <-- H{t} + 1
        for all term t in H do
            EMIT(tuple <t,n>, tf H{t})

class REDUCER
    method INITIALIZE
        pre(t) <-- null
        P <-- POSTINGSLIST
    procedure REDUCER(tiple <t,n>,tf[f])
        if t != pre(t) and pre(t) != 0 then
            EMIT(term t ,postings P)
            P.RESET()
        P.ADD(<n,f)
        pre(t) <- t
    method CLOSE
        EMIT(term t, postings P)
```

这是一种比较典型的MapReduce value-to-key转换的设计模型，这样的话，在Reducer端缓存的postings列表的内存使用率将大大降低，从而减少out-of-memory发生