華中科技大學

研究生（分布式系统）课程报告

题目：MapReduce 实现倒排索引

学　　号：M201977184

姓　　名：程　　诚

专　　业：计算机技术

指 导 教 师：金海 石宣化

院（系、所）：计算机科学与技术学院

2020 年 1 月 16 日

目 录

目 录 I
1 实验目的 1
1.1 倒排索引简介 1
1.2 MapReduce 简介 2
1.3 MapReduce 原理 2
2 实验内容 4
2.1 实验具体内容 4
2.2 实验要求 4
3 实验方法 5
3.1 代码逻辑分析 5
3.2 源代码清单 6
4 实验结果 9
5 总结与收获 10

1 实验目的

本次实验主要考察如何使用 MapReduce 编程模型建立倒排索引，并通过实验了解倒排索引和 MapReduce 编程模型基本原理。

1.1 倒排索引简介

倒排索引（Inverted Index）是文本检索系统中最常用的一种数据结构，被应用于各种大型搜索引擎。倒排索引又叫反向索引（Inverted Index）、倒排文件（Inverted File）等，是一种用于存储从文本内容（如单词、数字等）到其所在表或者所在文件等映射的索引数据结构（如表 1.1 所示），而建立倒排索引的过程实际上就是用一定的文件处理时间换取高效检索能力的过程。

目前倒排索引主要有两个变种，分别是记录级别的倒排索引（Record-Level Inverted Index）和单词级别的倒排索引（Word-Level Inverted Index）。记录级别的倒排索引的记录包含有每个单词到其所在文档列表的映射，同时也可以记录单词词频等信息。而单词级别的倒排索引不仅囊括了前者，还同时记录有单词在其所在文档中的位置，显然，建立单词级别的倒排索引需要花费跟多的时间。

与倒排索引相对应的则是正排索引（Forward Index）。正排索引记录的是文档到单词的映射，是一种十分简单的索引方式，如表 1.2 所示。但显然这种索引方法相比之下十分不利于文本检索，当需要检索文档中的单词时，需要顺序遍历每一个文档索引，检索效率低下。

表 1.1 倒排索引

Words	Documents
“and”	{Document2, Document3}
“is”	{Document1, Document2}

“mapreduce”	{Document1, Document2, Document3}
“powerful”	{Document1}
“simple”	{Document1,Document2}

表 1.2 正排索引

Document	Words
Document1	{“mapreduce”,”is”,”simple” }
Document2	{“mapreduce”,”is”,”powerful”,”and”,”simple” }
Document3	{“mapreduce”,”and”,”mapreduce”}

1.2 MapReduce 简介

MapReduce（简称 MR）是一种分布式计算框架，同时也是一种编程模型，主要是用于大规模数据集的并行运算，最早由 Google 提出。其设计初衷是为了解决 Google 搜索引擎中大规模的网页数据的并行化处理。而后 Yahoo!基于 MapReduce 开发设计了开源分布式批处理系统 Hadoop，使得 MR 模型得到迅速推广和普遍应用。现如今已经出现了许多支持 MapReduce 编程模型的分布式计算系统框架，如：Storm、Spark、Flink 等等。

1.3 MapReduce 原理

MR 编程模型主要由两个阶段组成：Map 和 Reduce，通常情况下用户只需要实现 map()和 reduce()两个函数，并提交至对应的分布式系统，即可实现分布式计算，因此用户只需专注于领域知识，解决领域问题，无需考虑分布式计算底层细节，如：网络通信、任务分发、任务调度等。其中 Map 阶段主要任务是解析输入的文件内容，将文件的文本内容映射成某类型键值对输入，经过自定义处理之后，再转换成某类型键值对进行输出；而 Reduce 阶段的主要任务是接受分组排

序后的 Map 输出键值对，在对这些键值对进行一系列处理后，又以键值对的形式输出到最终文件中。

MR 模型除了最主要的 Map 和 Reduce 阶段，中间还有 Shuffle 阶段，而 Shuffle 阶段又可以细分为 Spill、Combine、Partition 等等。Shuffle 阶段的主要任务是将 Map 的输出进行分组合并，然后通过网络传输给 Reducer 进行后续处理。由于 Map 阶段的输出会通过网络传给 Reducer 所在主机，因此为了减小网络 I/O 消耗，需要将数据事先合并，这样可以减小网络传输所需带宽，这也是 Combine 阶段的作用，即将 Map 输出的键值对在 Mapper 本地进行事先合并。MapReduce 大致过程如图 1-1 所示。

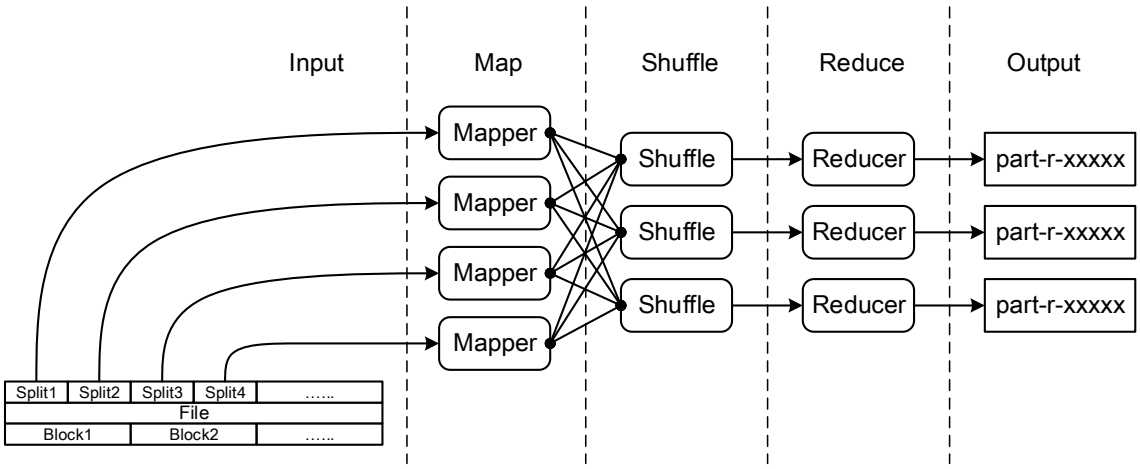

图 1-1 MapReduce 过程

2 实验内容

2.1 实验具体内容

基于平台指定的三个文本文件，为其建立倒排索引，三个文本文件的文件名及其具体内容如下所示：

- file1.txt：“mapreduce is simple”
- file2.txt：“mapreduce is powerful and simple”
- file3.txt：“mapreduce and mapreduce”

预期输出如下所示（第一项代表输出文件第一行内容，后续以此类推）：

- and file3.txt:1;file2.txt:1;
- is file2.txt:1;file1.txt:1;
- mapreduce file1.txt:1;file2.txt:1;file3.txt:2;
- powerful file2.txt:1;
- simple file2.txt:1;file1.txt:1;

其中第一行内容表示“and”单词在 file3.txt 文件中出现 1 次，在 file2.txt 文件中出现一次，后续以此类推。

2.2 实验要求

根据平台编辑器中的代码提示，将代码区中/**********Begin**********/与/**********End**********/之间的代码补全，即分别实现 InvertIndex_origin 类中的 Map 类中的 map 方法、Combine 中的 reduce 方法、Reduce 中的 reduce 方法。在编辑界面中只需要补齐 InvertIndex_origin 文件中内容即可，其余文件均保持不变。

3 实验方法

3.1 代码逻辑分析

3.1.1 Mapper

- 输入：<文本偏移量 , 整行文本内容>
- 处理过程：通过 Split 获取文件名，然后对本行文本内容进行分词，将“单词:文件名”作为 keyOut，valueOut 可以直接设置为空的 Text 类型变量，也可以设置为“1”
- 输出：<单词:文件名 ,“1”>

3.1.2 Combiner

- 输入：<单词:文件名 , {1,1,1,1,1,1,1……}>
- 处理过程：从 valueIn 中获取文件名信息，对 valueIn 使用英文冒号“:”进行分词，最后一个分词结果即是文件名，将前面的分词结果组合在一起即是单词，将单词作为 keyOut，遍历 valueIn 统计词频，将文件名和词频组合在一起形成 valueOut
- 输出：<单词 , 文件名:sum>

3.1.3 Reducer

- 输入：<单词 , {文件名 1:sum1 , 文件名 2:sum2……}>
- 处理过程：将 valueIn 中相同文件名对应的词频相加，然后将每个元素拼接在一起作为 valueOut，中间使用英文分号“;”隔开，keyIn 直接作为 keyOut，即

单词

- 输出：<单词 ，文件名 1:sum1;文件名 2:sum2……>

3.2 源代码清单

3.2.1 Mapper

```
public static class Map extends Mapper<Object, Text, Text, Text>
{
        private Text keyInfo = new Text(); // 存储单词和 URL 组合
        private Text valueInfo = new Text(); // 存储词频
        private FileSplit split; // 存储 Split 对象

        // 实现 map 函数
        @Override
        public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {
            // 获得<key,value>对所属的 FileSplit 对象
            split = (FileSplit) context.getInputSplit();
            StringTokenizer itr = new
StringTokenizer(value.toString());
            String fileName = split.getPath().getName();// 通过分片
Split 获取文件名
            while (itr.hasMoreTokens()) {
                // key 值由单词和文件名组成，value 值初始化为 1. 组成 key-
value 对:
                // 如: (MapReduce:file1.txt, 1)
                /**********Begin**********/
                // 创建 Key
                keyInfo.set(itr.nextToken() + ":" + fileName);
                /*// 创建 Value
                valueInfo.set("1");*/
                // 将 Key-Value 压入到 context 中
                context.write(keyInfo, valueInfo);
```

```
            /**********End**********/
        }
    }
}
```

3.2.2 Combiner

```
public static class Combine extends Reducer<Text, Text, Text,
Text> {
    private Text info = new Text();

    // 实现 reduce 函数， 将相同 key 值的 value 加起来
    // 并将(单词:文件名, value) 转换为 （单词， 文件名:value）
    @Override
    public void reduce(Text key, Iterable<Text> values,
Context context) throws IOException, InterruptedException {
        /**********Begin**********/
        // 存储 ValueOut
        Text valueOut = new Text();
        // 统计词频
        int count = 0;
        for (Text text : values) {
            count++;
        }
        // 重新设置 value 值由 URL 和词频组成
        String[] words = key.toString().split(":");
        String fileName = words[words.length - 1];
        valueOut.set(fileName + ":" + count);
        // 重新设置 key 值为单词
        // 为了避免前面单词中出现":"，在这里对 String 数组前面的 String
进行组合
        StringBuilder stringBuilder = new StringBuilder();
        for (int i = 0; i < words.length - 1; i++) {
            stringBuilder.append(words[i]);
        }
        info.set(stringBuilder.toString());
```

```
            context.write(info, valueOut);
            /**********End**********/

        }
    }
```

3.2.3 Reducer

```
public static class Reduce extends Reducer<Text, Text, Text,
Text> {
        private Text result = new Text();
        // 实现 reduce 函数，将相同单词的 value 聚合成一个总的 value，每个
value 之间用`;`隔开，最后以`;`结尾
        @Override
        public void reduce(Text key, Iterable<Text> values,
Context context) throws IOException, InterruptedException {
            /**********Begin**********/
            // 因为题中设置了使用 Combiner，所以
            // 为了避免当文件过大时使用的是不同的 Mapper 处理的同一个文件，从
而导致后来的 Combiner 处理之后在 ValuesIn
            // 中可能存在文件名重复的情况，这个时候就需要辨别，所以在这里使用
LinkedHashMap 将同文件名
            HashMap<String, Integer> hashMap = new
LinkedHashMap<>();
            StringBuilder stringBuilder = new StringBuilder();
            Text valueOut = new Text();
            for (Text valueIn : values) {
                String[] words = valueIn.toString().split(":");
                String fileName = words[0];
                if (hashMap.containsKey(fileName)) {
                    Integer sum = Integer.valueOf(words[1]) +
hashMap.get(fileName);
                    hashMap.put(fileName, sum);
                } else hashMap.put(fileName,
Integer.valueOf(words[1]));
            }
```

```
            for (String fileName : hashMap.keySet()) {

stringBuilder.append(fileName).append(":").append(hashMap.get(fil
eName)).append(";");
            }
            valueOut.set(stringBuilder.toString());
            /**********End**********/
            context.write(key, valueOut);}}}
```

4 实验结果

本次实验的测试结果如图 4-1 所示。代码执行时长为：87.83 秒，消耗内存为 1296.29MB，本次测评耗时（编译、运行总时间）55.569 秒。根据本次实验结果可以分析得出此 MapReduce 程序具体的执行过程，如图 4-2 所示。

图 4-1 实验结果

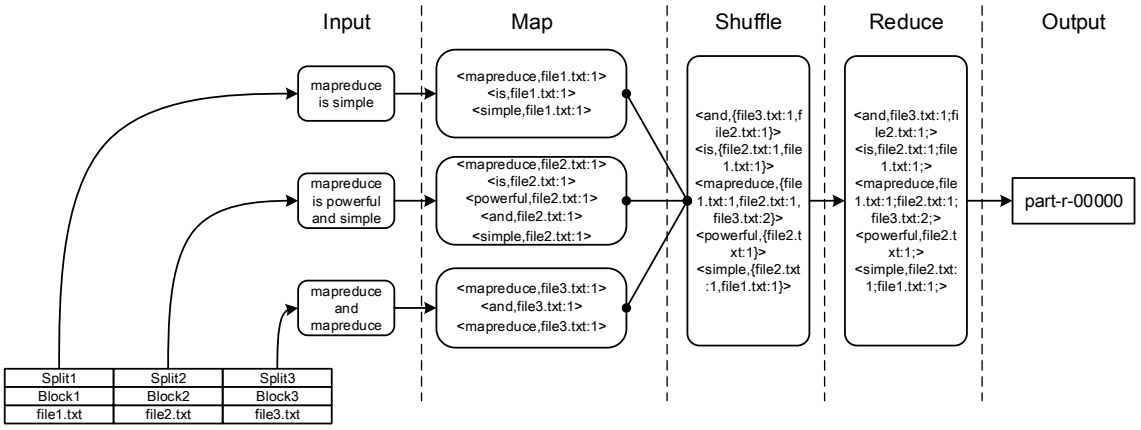

图 4-2 程序执行过程

5 总结与收获

在本次实验中，当输入的文本文件中单个文件较大时（需要多个 Split 表示），就需要多个 Mapper 对同一个文本文件进行处理，这就会导致在 Reduce 阶段的输入的 value 中，可能存在类似于“file1.txt:2,file2.txt:3,file1.txt:1……”这种情况，即同一个文件的词频是作为不同的 value 中的元素传入的，所以在 Reduce 阶段采用的类似于 LinkedHashMap 这样的数据结构来将相同文件名的词频进行求和（不使用 HashMap 的原因是其中的元素存储不一定按照插入顺序，与预期结果冲突），再将其进行以英文分号“;”作为分隔符进行拼接，最后作为 Reduce 阶段输出的 value 进行输出。这样一来就能确保即便当输入文件很大时，程序的输出结果也是正确的。

通过本次 Educoder 平台的实验，本人掌握了倒排索引的基本概念和数据结构，学习了 MapReduce 编程模型的逻辑结构和基本原理，并能够通过 MapReduce 编程实现一些简单的分布式运算。