

大数据综合处理实验

实验四

组长：韩畅，组员：李展烁、王一之、闫旭芃

2020 年 5 月 14 日

1 实验规划与设计

1.1 任务分配

171860551, 韩畅：组长,算法设计与实验规划, Pair_Int类, partitionr类, combiner 类编写, 程序调试与优化
171860550, 王一之：算法设计与实验规划, 程序优化, 添加程序代码, 主要报告编写, github集中控制
171860549, 闫旭芃：算法设计与实验规划,driver类以及代码整合,修改,hadoop集群调试提交, 部分报告补充
171840565, 李展烁：算法设计与实验规划,mapper类,reducer类,combiner类编写,参与程序优化和代码修改

1.2 任务要求

使用MapReduce 完成对数据的二次排序。

1.3 设计思路

map过程将每行的两个值处理为自定义数据结构Pair_Int作为key, 为减少传输量, value为空。重写partitioner, 保证分区时第一个值相同的值分配到同一个reducer上。在Pair_Int中设置比较函数, 每个分区会调用此比较函数, 此时便完成了二次排序。重载Comparator, 保证分组时第一个值相同的结果在同一批次任务中处理,reduce阶段将传入的值转化为text直接输出即可。

1.3.1 自定义数据类型

自定义一个Pair_Int存储数值对

```
public class Pair_Int implements WritableComparable<Pair_Int> {  
  
    private int lf = 0;  
    private int rt = 0;  
}
```

图 1. 自定义Pair_Int

重载读写方法,, 重载toString方法方便最终输出, 以二次排序的规则重载compareTo方法, 同时重载WritableComparator提供字节流比较

```
@Override
public int compareTo(Pair_Int pI) {
    // 第一列的值升序排列，而第二列的值降序排列

    if (lf != pI.lf) {
        return lf < pI.lf ? -1 : 1;
    } else if (rt != pI.rt) {
        return rt > pI.rt ? -1 : 1;
    } else {
        return 0;
    }
}
```

(a) 重载比较

```
public static class SSComparator extends WritableComparator {
    public SSComparator() {
        super(Pair_Int.class);
    }

    // 使用二进制字典序排列
    @Override
    public int compare(byte[] arg0, int arg1, int arg2, byte[] arg3, int arg4, int arg5) {
        return compareBytes(arg0, arg1, arg2, arg3, arg4, arg5);
    }
}

// 在WritableCparator里激活新定义的comparator
static {
    WritableComparator.define(Pair_Int.class, new SSComparator());
}
```

(b) 重载比较2

图 2. Pair_Int部分代码实现1

```
// 必须重载的读入和输出方法，为了利用编码范围，需要加上偏移量
@Override
public void readFields(DataInput dI) throws IOException {

    lf = dI.readInt() + Integer.MIN_VALUE;
    rt = Integer.MAX_VALUE - dI.readInt();
}

@Override
public void write(DataOutput dO) throws IOException {

    dO.writeInt(lf - Integer.MIN_VALUE);
    dO.writeInt(Integer.MAX_VALUE - rt);
}
```

(a) 重载读写

```
@Override
public String toString() {
    return this.lf + "\t" + this.rt;
}

@Override
public int hashCode() {
    // 由于第二列的值的范围不超过100，因此使用此函数即可确保任意不同的一二列值对的哈希码不同
    return lf * 100 + rt;
}

@Override
public boolean equals(Object obj) {

    // 当且仅当两列完全相同时相等，不同类型，存在不相等均返回false
    if (obj instanceof Pair_Int) {
        Pair_Int p = (Pair_Int) obj;
        return p.lf == lf && p.rt == rt;
    } else {
        return false;
    }
}
```

(b) 部分其他函数

图 3. Pair_Int部分代码实现2

1.3.2 主功能Map设计思路

读入数据，每行的两个值以自定义数据结构Pair_Int的方式作为key，value都为空

```
public class SecondSortMapper
    extends Mapper<Text, Text, Pair_Int, NullWritable> {
    private final Pair_Int key = new Pair_Int();

    @Override
    public void map(Text key, Text value, Context context) throws IOException, InterruptedException {
        this.key.set(Integer.parseInt(key.toString()), Integer.parseInt(value.toString()));
        context.write(this.key, NullWritable.get()); // key中为intpair, value为空
    }
}
```

图 4. Mapper实现

1.3.3 重写Partitioner

重写Partitioner自定义分区，根据第一个值进行分区，保证第一个值相同在同一个reducer上

```
public class L1Partitioner extends Partitioner<Pair_Int, IntWritable> {
    @Override
    // 保证相同的第一列值必然分配到同一个reducer上
    public int getPartition(Pair_Int key, IntWritable val, int num) {
        return Math.abs(key.getLf() * 100) % num;
    }
}
```

图 5. Partitioner部分实现

1.3.4 排序

mapreduce此阶段对每个分区内进行排序，key为自定义类型，所以在Pair_Int中自定义排序方法，见2

compareTo函数首先第一个值，若相同则比较第二个值，完成第一列升序，第二列降序的比较。

1.3.5 自定义分组

为保证结果正确，第一个值相同的结果要在同一个reduce任务中处理因此重载RawComparator进行自定义分组

```
public class SecondSortGroupingComparator implements RawComparator<SecondarySort.Pair_Int> {
    @Override
    public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
        return WritableComparator.compareBytes(b1, s1, Integer.BYTES, b2, s2, Integer.BYTES);
    }

    //根据第一个字段分组
    @Override
    public int compare(SecondarySort.Pair_Int p1, SecondarySort.Pair_Int p2) {
        return Integer.compare(p1.getLf(), p2.getLf());
    }
}
```

图 6. 重写Comparator

1.3.6 主功能Reduce设计思路

输入的key中已经是排好序的结果，自定义类型已经实现了toString，直接输出即可

```
public class SecondSortReducer extends Reducer<Pair_Int, NullWritable, Pair_Int, NullWritable> {

    @Override
    public void reduce(Pair_Int key, Iterable<NullWritable> values, Context context)
        throws IOException, InterruptedException {
        for (NullWritable ignored : values) { //int对在key中, value为空, 直接忽略
            context.write(key, NullWritable.get());
        }
    }
}
```

图 7. 重写Reducer

1.3.7 Key-Value类型协调

map输出类型为intPair, IntWritable, value实际为空(NullWritable)

reduce输出类型为intPair, IntWritable, value同样为空(NullWritable)，但key中会调用Pair_Int的toString，输出实际为text

1.4 代码演示

1.4.1 Map阶段代码演示

已经包含在设计思路图片中，见图4

1.4.2 Reduce阶段代码演示

同上,见图7

2 实验结果展示

2.1 结果文件在HDFS上的路径

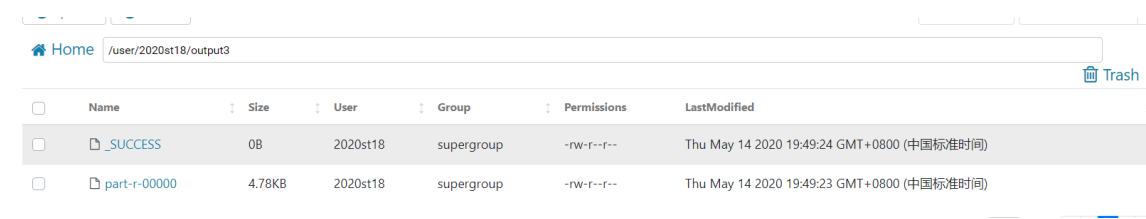


图 8. HDFS路径截图

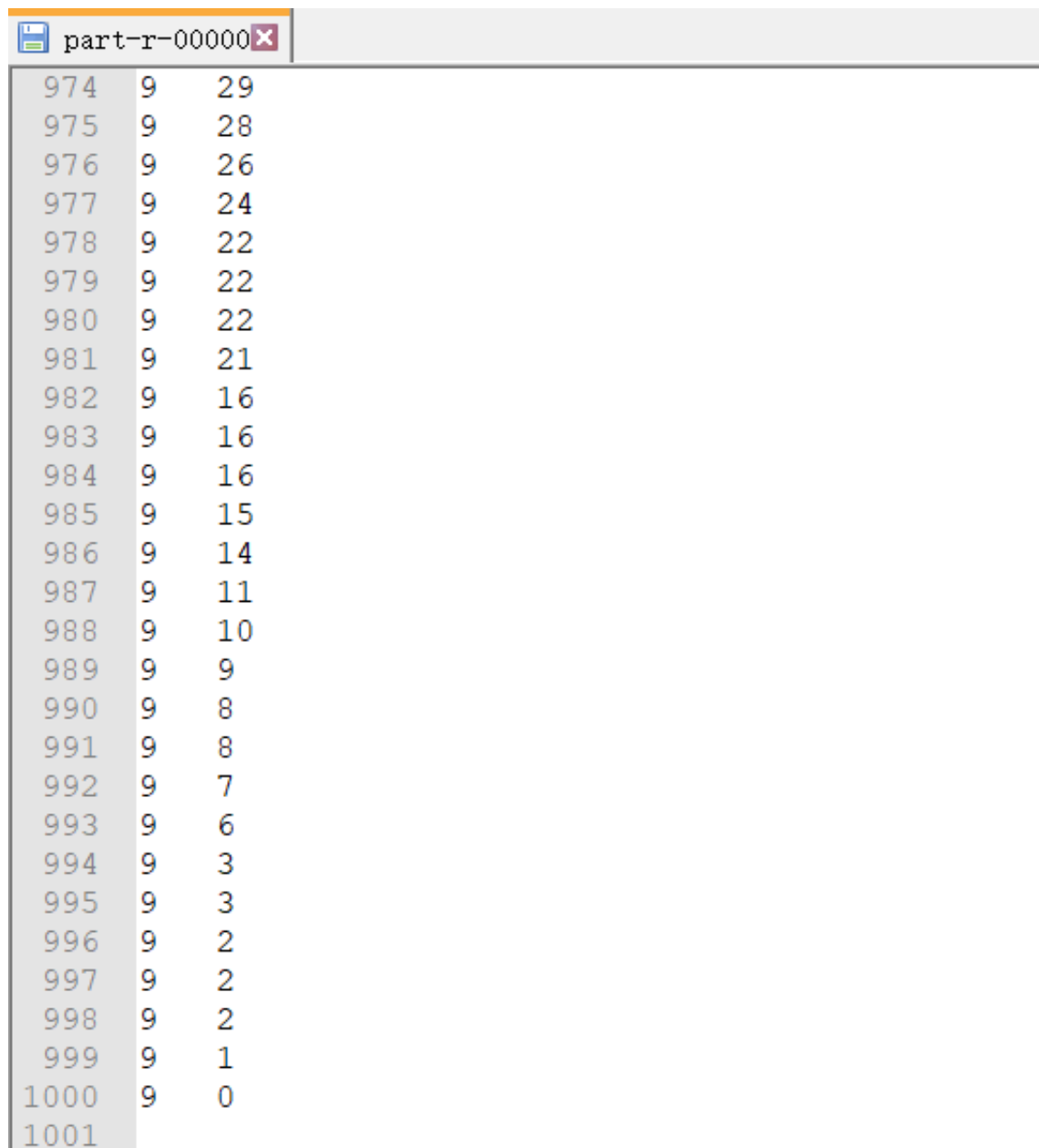
2.2 输出结果文件的部分截图



图 9. web实验结果查看

part-r-00000		
1	0	98
2	0	98
3	0	97
4	0	97
5	0	96
6	0	95
7	0	93
8	0	91
9	0	91
10	0	90
11	0	90
12	0	87
13	0	87
14	0	86
15	0	86
16	0	85
17	0	85
18	0	84
19	0	84
20	0	83
21	0	83
22	0	77
23	0	77
24	0	76
25	0	75
26	0	71
27	0	71
28	0	69
29	0	68
Normal text file		
length : 4,897 lines : 1,001		
Ln : 1 Col : 1 Sel : 0 0		

图 10. 结果开头



974	9	29
975	9	28
976	9	26
977	9	24
978	9	22
979	9	22
980	9	22
981	9	21
982	9	16
983	9	16
984	9	16
985	9	15
986	9	14
987	9	11
988	9	10
989	9	9
990	9	8
991	9	8
992	9	7
993	9	6
994	9	3
995	9	3
996	9	2
997	9	2
998	9	2
999	9	1
1000	9	0
1001		

图 11. 结果结尾

2.3 Web UI 报告内容展示

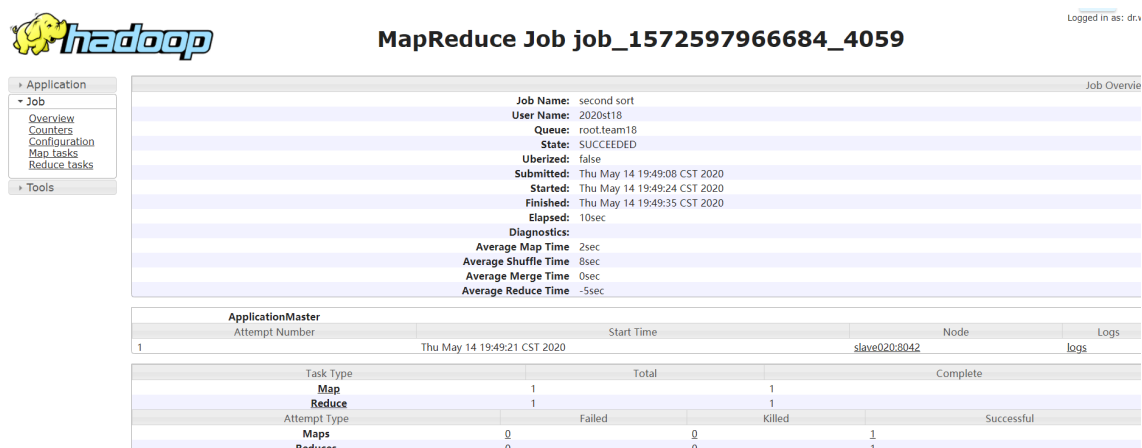


图 12. WebUI执行报告

3 实验经验总结与改进方向

- 1) 同上次实验相比，这次同样自定义了数据类型，但是比较是直接在字节流中比较，避免创建对象
- 2) 这次算法的巧妙之处在于：升序降序的实现是序列化时使用了一个技巧，使得Pair_Int对象按IF升序、按rt降序排序等价于序列化后byte数组的字典序，进而可以直接使用Writable.compareBytes对序列化的byte数组进行比较，减小了序列化的开销。在write方法中：先写入Integer.MIN_VALUE，将 -2^{31} 到 $2^{31}-1$ 映射在0x00000000到0xffffffff，此时byte[]字典序等价于int升序；再写入Integer.MAX_VALUE-rt，将 $2^{31}-1$ 到 2^{31} 映射在0x00000000到0xffffffff，此时byte[]字典序等价于int降序。在readFields方法中，再相应地还原。