MapReduce 大数据实验四—MyJoin_Hive

卢庆宁 MG20330040
邹德润 MG20330094
徐业婷 MF20330097
陈轶洲 MF20330010

December 7, 2020

1 实验任务描述

使用 MapReduce 完成两张表的 join 操作。实验数据在 hdfs://master001:9000/data/hive_myjoin 目录下。单机测试可以使用 FTP“实验要求”目录下的测试数据集。

1) 输入数据为 order.txt 和 product.txt;

2) 先将这两个文件通过使用 MapReduce 进行 join 操作，将结果输出到 HDFS 的个人目录上;

3) 进入 SQL On Hadoop 页面，使用 Hive 建表管理上一步输出的结果;

4) 最后在 Hive 上通过 show tables 能查看到名为 orders 的表，并且通过 select 语句能查出内容。

2 实验设计

2.1 实验环境

Linux 环境	Java 版本	Hadoop 版本	IDE	Maven	集群名称
Ubuntu18.04	jdk1.8	hadoop2.7.1	ItelliJ IDEA(mac os)	maven3.6	2021st20

2.2 实验任务流程

1) 小组开会进行任务的分配，伪代码的设计，并按照讨论得出的思路进行实现

2) 小组成员进行具体代码的编写，利用 maven 管理项目，生成 jar 包，进行本地测试

3) 进行集群的测试，查看结果，撰写实验报告

3 实验具体实现

本次实验的要求是将两张表中的数据进行连接，我们的实现借鉴了课堂 ppt 与书本代码的思路。

3.1 设计思路

Mapper 的输入为待合并的两张表中的每一行数据，输出的 Key 为标记了来源表的 Join 的字段内容，Value 为原表的记录。
Reducer 在接收到同一个 Key 的记录后遍历 Values，并根据 Key 中不同的来源表标签执行不同操作：“product”——将该条记录加入到一个 List 中；”order”——遍历上一种情况生成的 List，对 List 中的每一条来自产品的数据生成 join 结果并输出。(在区分复合键时进行二次排序，确保分类后 product 记录出现在 order 记录之前。)

3.2 覆写

进行了 Map 类和 Reduce 类的重写，并实现了 Partitioner、WritableComparable 和 WritableComparator 方法。

3.3 TaggedKey 复合键

3.3.1 实现思路

将两张表连接利用的是他们记录中相同的 product_id，除此以外我们还需要标记这条数据来自哪个文件，故需要设计一个 Tag (IntWritable 类型)。将这二者结合得到 <Text product_id, IntWritable Tag>，就是本实验中需要使用的复合键。
为了定义复合键我们还需要实现 WritableComparable 接口：利用二次排序，首先比较 product_id，若不同则返回比较结果，若相同则继续比较 Tag，并返回 Tag 的比较结果。

3.3.2 实际代码

```
//WritableComparable
public static class TaggedKey implements
    WritableComparable<TaggedKey>{
  private Text joinKey = new Text();//product_id
  private IntWritable tag = new IntWritable();//Tag
  //implement compareTo()
  public int compareTo(TaggedKey taggedKey){
    //compare product_id
    int compareValue = joinKey.compareTo(taggedKey.getJoinKey());
    //if product_id equals, then compare Tag
    if (compareValue == 0){
      compareValue = tag.compareTo(taggedKey.getTag());
    }
```

```
        return compareValue;
    }
    public void readFields(DataInput in) throws IOException{
        joinKey.readFields(in);
        tag.readFields(in);
    }
    //implement write() and set()
    public void write(DataOutput out) throws IOException{
        joinKey.write(out);
        tag.write(out);
    }
    public void set(String joinKey, int tag){
        this.joinKey.set(joinKey);
        this.tag.set(tag);
    }
    //implement get() to get product_id or Tag
    public Text getJoinKey(){
        return joinKey;
    }
    public IntWritable getTag(){
        return tag;
    }
}
```

3.4 Map 类

3.4.1 输入输出格式

输入	<LongWritable, Text>
输出	<TaggedKey, Text>=«pid,sourcefile>, record>

3.4.2 实现思路

首先获取文件名并去除文件名后缀（得到“product”或“order”），然后对内容进行读取，此时读取的是文件的每一行。根据来源文件设置 Tag(IntWritable 类型)：来自 product 的数据的 Tag 置为 1，来自 order 的数据的 Tag 置为 2。根据不同 Tag 就能从文本中提取不同来源文件的 product_id。将复合键 <Text product_id, IntWritable Tag> 作为 Key，数据 Text value 作为 value 输出。

3.4.3 实际代码

```
//Mapper
public static class DataMapper extends Mapper<LongWritable, Text,
    TaggedKey, Text>{
```

```
    @Override
    protected void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException{
      String[] columns = value.toString().split(" ");
      TaggedKey taggedKey = new TaggedKey();
      //setting <key, value> depends on filename
      FileSplit fileSplit = (FileSplit)context.getInputSplit();
      String fileName = fileSplit.getPath().getName();
      if(fileName.startsWith("product")){
        //columns[0] is pid in product.txt, tag is 1
        taggedKey.set(columns[0], 1);
        context.write(taggedKey, value);
      } else if(fileName.startsWith("order")){
        //columns[2] is pid in order.txt, tag is 2
        taggedKey.set(columns[2], 2);
        context.write(taggedKey, value);
      }
    }
  }
```

3.5 Partitioner 类

3.5.1 实现思路

默认情况下，Hadoop 会对 Key 进行哈希，以保证相同的 Key 会分配到同一个 Reducer 中。由于我们改变了 Key 的结构，因此需要重新编写分区函数：只利用 product_id 分区，而忽略 Tag 的不同。这样可以使不同文件中具有相同 product_id 的记录分到一个 Reducer 中。

3.5.2 实际代码

```
  //Partitioner
  public static class TaggedPartitioner extends Partitioner<TaggedKey,
      Text>{
    @Override
    public int getPartition(TaggedKey taggedKey, Text text, int
        numPartition){
      //partition TaggedKey by accessing product_id's hash
      return taggedKey.getJoinKey().hashCode()%numPartition;
    }
  }
```

3.6 WritableComparator 类

3.6.1 实现思路

同 Partitioner，调用 reduce 函数需要传入同一个 Key 的所有记录，这就需要重新定义分组函数.

3.6.2 实际代码

```
//WritableComparator
public static class TaggedJoinComparator extends WritableComparator{

   public TaggedJoinComparator(){
      super(TaggedKey.class, true);
   }

   @SuppressWarnings("rawtypes")
   @Override
   public int compare(WritableComparable a, WritableComparable b){
      TaggedKey key1 = (TaggedKey) a;
      TaggedKey key2 = (TaggedKey) b;
      return key1.getJoinKey().compareTo(key2.getJoinKey());
   }
}
```

3.7 Reduce 类

3.7.1 输入输出格式

输入	<TaggedKey, Iterable<Text»=«pid,sourcefile>,records[]>
输出	<NullWritable, Iterable>=<records[]>

3.7.2 实现思路

因为 partitioner 仅按照不同的 product_id 划分，所以对于一个 Reducer，包含了 text 和不同的 Tag，因此我们需要区分每条记录来自 product 还是 order。设置全局变量 List<String>products，遍历记录，当这条记录来自 produc 时，我们将该 Text 转换成 String 并存入 products 中；当这条记录来自 order 时，我们遍历 products，将其中每条 product 记录与该 order 记录进行 join 操作并输出。
因为最终只要求输出连接后的记录，所以将输出的 Key 置为 NullWritable。

3.7.3 实际代码

```
 //Reducer
public static class JoinReducer extends Reducer<TaggedKey, Text,
    NullWritable, Text>{
  List<String> products = new ArrayList<String>();//save records
      from product
  @Override
  protected void reduce(TaggedKey key, Iterable<Text> values,
      Context context)
  throws IOException, InterruptedException {
    for (Text value : values) {
      switch (key.getTag().get()) {
        case 1: // product
        products.add(value.toString());//add to products
        break;
        case 2: // order
        String[] order = value.toString().split(" ");
        for (String productString : products) {
          String[] product = productString.split(" ");
          //join data
          if(order[2].equals(product[0])) {
            List<String> output = new ArrayList<String>();
            output.add(order[0]);
            output.add(order[1]);
            output.add(order[2]);
            output.add(product[1]);
            output.add(product[2]);
            output.add(order[3]);
            //output
            context.write(NullWritable.get(), new
                Text(StringUtils.join(output, " ")));
          }
        }
        break;
        default:
        assert false;
      }
    }
  }
}
```

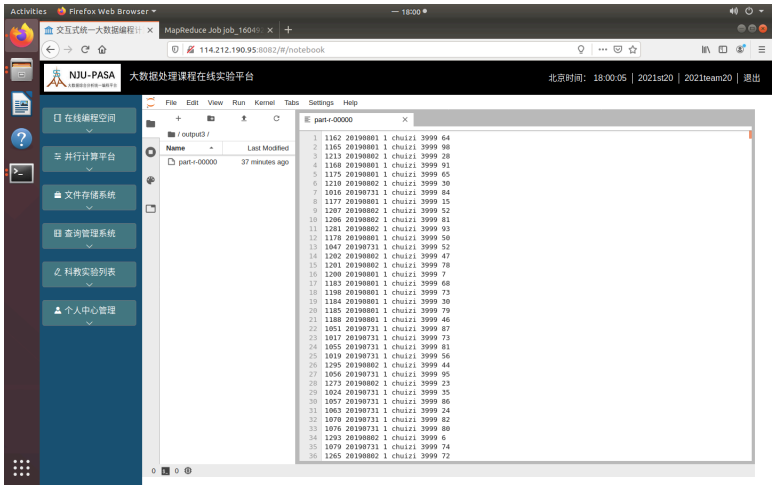
4 实验运行与结果分析

4.1 配置管理

利用 maven 进行包的依赖项的添加和项目的打包管理。生成 jar 包类型。具体见 pom.xml 文件。

4.2 生成 jar 包运行结果

输出结果文件在 HDFS 上的路径：*/user/*2021*st*20*/output*3 。两张表连接生成的结果文件为 part-r-00000，显示如下：

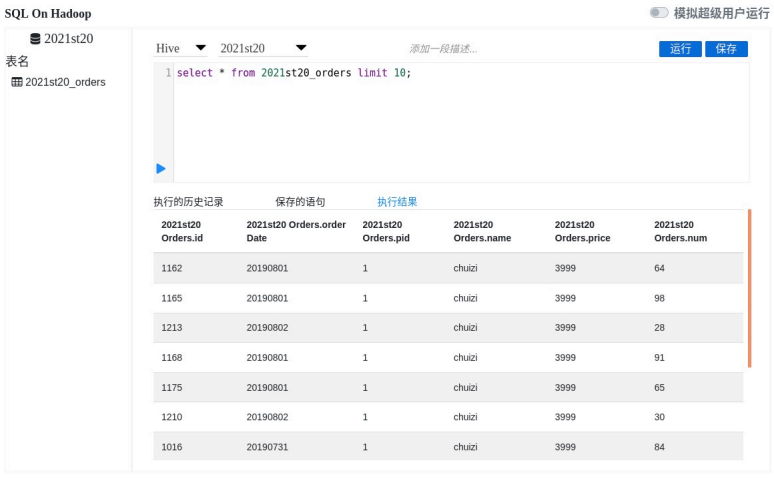


4.3 Hive 输出结果文件

利用上一小节生成的 part-r-00000 文件建表，在 SQL On Hadoop 中输入如下指令：

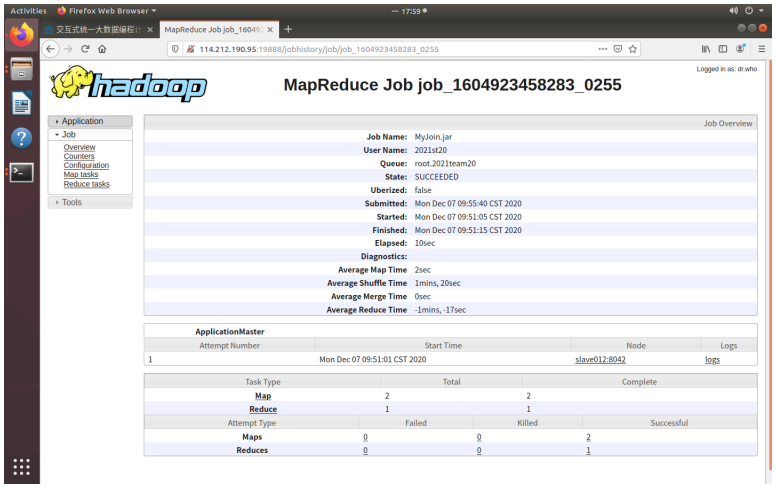
create table 2021st20_orders(id int,order_date string,pid string,name string,price int,num int) row format delimited fields terminated by ' ' location '/user/2021st20/output3/';

利用 select 语句查看此表：



4.4 Web-ui 结果

Job ID: job_1604923458283_0255



5 实验总结和参考

5.1 实验总结

1) 本次实现通过自己实现 hadoop 的 mapreduce 程序，了解了对于 map-reduce 程序编写，懂得了设计的重要性，并且通过 maven 管理了项目，使得项目的打包运行更方便；

2) 同时小组通过开会等，分工明确，互相帮助，共同完成这次实验，体现了团队协作的重要性，在一个项目中团队精神是不可缺少的；

3) 在实验中遇到了并行问题，通过询问助教后得到解决。因为 hadoop 不保证一个 reducer 只执行一次 reduce，所以设计时需要格外注意保护各个变量的信息；

4) 最后感谢助教和其他同学对本次实验提供的帮助。

5.2 实验参考

1) 深入理解 Reduce-side Join

2) Hadoop-MapReduce-Reduce 阶段调用 reduce() 方法过程源