实验三 MyJoin_Hive

171180554 夏宇 171860617 肖维城

1. 实验设计

实验任务是将product和order以pid作为关键字进行join,于是通过map按照pid对数据进行划分,将相同pid的数据送往同一个reducer,通过reducer进行合并输出。

数据结构:

使用自定义数据类型OrderBean对数据进行包装,其构成为:

```
Integer tag;//order is 1, product is -1
private Integer oid;
private String odata;
private Integer pid;
private String oamount;
private String pname;
private String price;
```

其中, tag用于区分是product还是order的记录, product取值为-1, order取值为1。

整体流程思路:

Map阶段:

输入输出:

输入<Key, Value>为: LongWritable, Text;

输出<Key, Value>为: OrderBean, NullWritable (用作占位符)。

具体过程:

- 1. 使用job.**setInputFormatClass**(TextInputFormat.class)做为输入格式
- 2. 进入Mapper的map()方法,生成一个List,将product和order中的记录包装为OrderBean,输出的**Key**和**Value**类型分别为: OrderBean,NullWritable。
- 3. 在map阶段的最后,会先调用job.**setPartitionerClass**(JoinPartitioner.class)对Map的结果进行分区,按照pid值进行划分,目的是使将相同的pid的数据分到同一个reducer上。
- 4. 每个分区内中又调用job.**setSortComparatorClass**()设置的key比较函数类排序,实验中没有设置,所以使用OrderBean的实现的**compareTo**方法对输出的结果做二次排序,将相同分区中不同的pid分开,并将相同pid的区域内,tag指示为-1(product)的数据排在tag指示为1(order)的之前,最后按照oid排序。

Reduce阶段:

输入输出:

输入<Key, Value>为: OrderBean, NullWritable; 输出<Key, Value>为: OrderBean, NullWritable。

具体过程:

1. shuffle阶段: reducer开始获取所有映射到这个reducer的map输出值。

- 2. 构造一个key对应的value迭代器,这里需要使用job.**setGroupingComparatorClass**()设置的分组函数类。分组中,只需要OrderBean里面的pid相同,便将这些key分为同一组,将它们放在一个迭代器中。
- 3. 最后进入Reducer的**reduce()**方法,在迭代器中,product记录放置在第一个位置,Reducer将 pname和price取出,用来填充后面的一系列order数据的pname和price部分,最后写入到输出中。

2. 实验代码

OrderBean

OrderBean实现WritableComparator接口: 定义compareTo函数:

```
public int compareTo(OrderBean orderBean) {
    //实现product数据在order前面:
    //先按照pid排,再在相同pid中按照tag排列,最后再按照oid进行排序
    if(this.pid.equals(orderBean.pid)) {
        if(this.tag.equals(orderBean.tag)) {
            return this.oid.compareTo(orderBean.oid);
        }
        else
            return this.tag.compareTo(orderBean.tag);
    }
    else
        return this.pid.compareTo(orderBean.pid);
}
```

Mapper

```
public class JoinMapper extends Mapper<LongWritable, Text,OrderBean,</pre>
NullWritable> {
    private String filename;
    OrderBean orderBean;
    @override
    protected void setup(Context context) throws IOException,
InterruptedException {
        FileSplit fs = (FileSplit) context.getInputSplit();
        //得到文件名称
        filename = fs.getPath().getName();
        orderBean = new OrderBean();
    }
    @override
    protected void map(LongWritable key, Text value, Context context)
                        throws IOException, InterruptedException {
        //按照空格划分数据
        String[] fields = value.toString().split(" ");
        //根据不同名称对oederBean进行赋值, setTag以划分
        if (filename.equals("product.txt")) {
            orderBean.setTag(-1);
            orderBean.setPid(Integer.parseInt(fields[0]));
            orderBean.setPname(fields[1]);
            orderBean.setPrice(fields[2]);
```

```
}else if(filename.equals("order.txt")){
    orderBean.setTag(1);
    orderBean.setOid(Integer.parseInt(fields[0]));
    orderBean.setOdata(fields[1]);
    orderBean.setPid(Integer.parseInt(fields[2]));
    orderBean.setOamount(fields[3]);
}
//以orderBean为key, NullWritable为value(占位符), context对(k,v)进行收集
context.write(orderBean,NullWritable.get());
}
}
```

partitioner

```
public class JoinPartitioner extends Partitioner<OrderBean, NullWritable> {
    @Override
    public int getPartition(OrderBean key, NullWritable value, int
numReduceTasks) {
        //按照pid值对数据进行划分,均匀分配到不同的reducer,且保证相同的pid分配到一个
reducer上
        return key.getPid() % numReduceTasks;
    }
}
```

Grouping

```
public int compare(WritableComparable a, WritableComparable b) {
    //只需要OrderBean里面的pid相同,便将这些key分为同一组
    OrderBean oa = (OrderBean) a;
    OrderBean ob = (OrderBean) b;
    return oa.getPid().compareTo(ob.getPid());
}
```

Reducer

```
protected void reduce(OrderBean key, Iterable<NullWritable> values, Context
                              throws IOException, InterruptedException {
context)
   Iterator<NullWritable> iterator = values.iterator();
   iterator.next();
   //第一条数据为product,取出pname和price的值
   String tempPname = key.getPname();
   String tempPname = key.getPrice();
   //迭代到下一条order数据,使用pname和price填充缺失值
   while (iterator.hasNext()) {
       iterator.next();
       key.setPname(pname);//迭代的时候key会指向下一条数据
       key.setPrice(price);
       context.write(key, NullWritable.get());
   }
}
```

reference

Partitioner说明:

```
https://blog.csdn.net/will_cruise/article/details/88751112
```

https://www.cnblogs.com/cangos/p/6429609.html

Join参考:

```
https://www.cnblogs.com/ivanny/p/5702461.html
```

SortComparator和GroupingComparator说明:

```
http://www.wangt.cc/2017/07/hadoop-mapreduce%E5%B7%A5%E4%BD%9C%E6%B5%81%E7%A8%8B%EF%BC%88partitionersortcomparatorgroupingcomparator%EF%BC%89/
```

https://www.linuxidc.com/Linux/2013-08/88603.htm

http://blog.sina.com.cn/s/blog_7581a4c30102veem.html

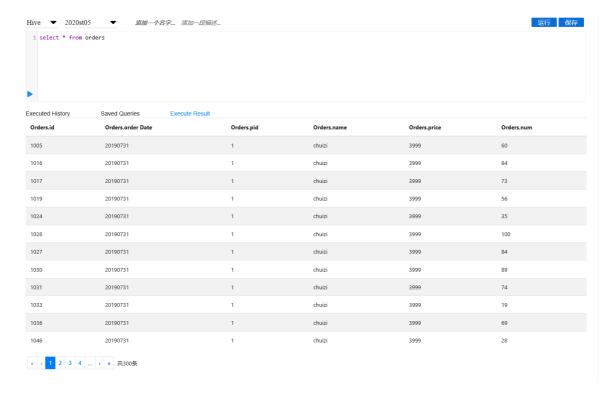
3. 实验结果

在编译完成jar包 myjoin.jar 后,使用命令 hadoop jar myjoin.jar JoinMain /data/exercise_3 exp3_out 进行join任务。

join结果存放在 /user/2020st05/exp3_out 中。

而后执行建表操作,并将表命名为 orders ,使用查看表命令 select * from orders 可以显示表内容。

部分内容如下所示:



4. WebUI 执行报告

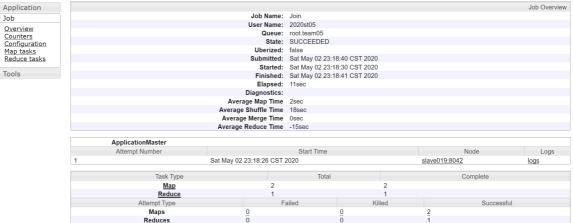
在大数据实验平台的历史记录中,查找join任务运行的记录

application 1572597966684 3797 2020st05 Join MAPREDUCE

详细结果为:

→ Tools





5. 实验分工

夏宇同学负责代码编写,编译jar包并在实验平台上测试,测试、完成建表,完成实验报告。 肖维城同学负责项目构建、代码编写,编译jar包并在实验平台上测试,测试建表,完成实验报告。