MapReduce6 MapJoin和ReduceJoin

> 其实Join就是Mysql中表的连接操作，后面使用hive还是一条sql语句就可以解决了，这里反正比较麻烦，但是要了解一下

Reduce Join原理

Map端的主要工作

为来自不同表或文件的key/value对，打标签以区别不同来源的记录。然后用连接字段作为key，其余部分和新加的标志作为value，最后进行输出

Reduce端主要工作

在Reduce端以连接字段作为key的分组已经完成，我们只需要在每一个分组当中那些来源不同文件的记录（在Map阶段已打标记）分开，最后进行合并就ok了

缺点

它的缺点很明显就是会造成Map和Reduce端也就是Shffle阶段出现大量的数据传输，效率很低下

Reduce Join实战

需求

订单表

订单id	pid	数量
1001	01	1
1002	02	2
1003	03	3
1001	01	1
1002	02	2
1003	03	3

公司名称表

Pid	产品名称
01	小米
02	华为
03	格力

预期输出数据

```
订单id      产品名称      数量
1001          小米        1
1001          小米        1
1002          华为        2
1002          华为        2
1003          格力        3
1003          格力        3
```

过程分析

MapTask

1）Map中处理的事情

1. 获取输入文件类型
2. 获取输入数据
3. 不同文件分别处理
4. 封装Bean对象输出

```
01      1001    1   order
02      1002    2   order
03      1003    3   order
01      1001    1   order
02      1002    2   order
03      1003    3   order
01      小米      pd
02      华为      pd
03      格力      pd
```

2）默认对产品id进行排序

```
01      1001    1   order
01      1001    1   order
01      小米      pd
02      1002    2   order
02      1002    2   order
02      华为      pd
03      1003    3   order
03      1003    3   order
03      格力      pd
```

ReduceTask

1）Reduce方法缓存订单数据集合和产品表，然后合并

```
订单      产品名称        数量
1001       小米           1
1001       小米           1

1002       华为           2
1002       华为           2

1003       格力           3
1003       格力           3
```

代码

TableBean代码

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import org.apache.hadoop.io.Writable;

public class TableBean implenments Writable{

    private String order_id;  //订单id
    private String p_id;   //产品id
    private int amount; //产品数量
    private String pname; //产品名称
    private String flag;  //表的标记

    public TableBean() {
        super();
    }

    public TableBean(String order_id, String p_id, int amount, String pname,
String flag) {
        super();
        this.order_id = order_id;
        this.p_id = p_id;
        this.amount = amount;
        this.pname = pname;
        this.flag = flag;
    }

    public String getFlag() {
        return flag;
    }

    public void setFlag(String flag) {
        this.flag = flag;
    }

    public String getOrder_id() {
        return order_id;
    }

    public void setOrder_id(String order_id) {
        this.order_id = order_id;
    }

    public String getP_id() {
        return p_id;
    }

    public void setP_id(String p_id) {
        this.p_id = p_id;
    }

    public int getAmount() {
        return amount;
```

```
    }

    public void setAmount(int amount) {
        this.amount = amount;
    }

    public String getPname() {
        return pname;
    }

    public void setPname(String pname) {
        this.pname = pname;
    }

    @Override public void write(DataOutput out) throws IOException {
        out.writeUTF(order_id);
        out.writeUTF(p_id);
        out.writeInt(amount);
        out.writeUTF(pname);
        out.writeUTF(flag);
    }

    @Override public void readFields(DataInput in) throws IOException {
        this.order_id = in.readUTF();
        this.p_id = in.readUTF();
        this.amount = in.readInt();
        this.pname = in.readUTF();
        this.flag = in.readUTF();
    }

    @Override
    public String toString() {
        return order_id + "\t" + pname + "\t" + amount + "\t" ;
    }
}
```

TableMappe代码

```
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

public class TableMapper extends Mapper<LongWritable, Text, Text, TableBean>{
    String name;
    TableBean bean = new TableBean();
    Text k = new Text();

    @Override
    protected void setup(Context context) throws IOException,
InterruptedException {

        // 1 获取输入文件切片
        FileSplit split = (FileSplit) context.getInputSplit();

        // 2 获取输入文件名称
```

```
        name = split.getPath().getName();
    }

    @Override
    protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {

        // 1 获取输入数据
        String line = value.toString();

        // 2 不同文件分别处理
        if (name.startsWith("order")) {
            // 订单表处理
            // 2.1 切割
            String[] fields = line.split("\t");
            // 2.2 封装bean对象
            bean.setOrder_id(fields[0]);
            bean.setP_id(fields[1]);
            bean.setAmount(Integer.parseInt(fields[2]));
            bean.setPname("");

            // 打上标签
            bean.setFlag("other");
            k.set(fields[1]);
        }else {
            // 产品表处理
            // 2.3 切割
            String[] fields = line.split("\t");
            // 2.4 封装bean对象
            bean.setP_id(fields[0]);
            bean.setPname(fields[1]);
            bean.setFlag("pd");
            bean.setAmount(0);

            bean.setOrder_id("");
            k.set(fields[0]);
        }
        // 3 写出
        context.write(k, bean);
    }
}
```

TableReducer代码

```
import java.io.IOException;
import java.util.ArrayList;
import org.apache.commons.beanutils.BeanUtils;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class TableReducer extends Reducer<Text, TableBean, TableBean,
NullWritable> {
    @Override protected void reduce(Text key, Iterable<TableBean> values,
Context context) throws IOException, InterruptedException {
        // 1准备存储订单的集合
        ArrayList<TableBean> orderBeans = new ArrayList<>();
```

```
        // 2 准备bean对象
        TableBean pdBean = new TableBean();
        for (TableBean bean : values) {

            // 根据flag判定是哪个表来的数据
            if ("order".equals(bean.getFlag())) {
                // 订单表 // 拷贝传递过来的每条订单数据到集合中
                TableBean orderBean = new TableBean();
                try {
                    // 对象互拷，Web中常见操作
                    BeanUtils.copyProperties(orderBean, bean);
                } catch (Exception e) {
                    e.printStackTrace();
                }
                orderBeans.add(orderBean);
            } else {

                // 产品表 try
                {
                    // 拷贝传递过来的产品表到内存中
                    BeanUtils.copyProperties(pdBean, bean);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }


        // 3 表的拼接
        for(TableBean bean:orderBeans){
            // 遍历订单表，将名字设置进来即可
            bean.setPname (pdBean.getPname());
            // 4 数据写出去
            context.write(bean, NullWritable.get());
        }
    }
}
```

TableDriver代码

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class TableDriver {
    public static void main(String[] args) throws Exception {
        // 0 根据自己电脑路径重新配置
        args = new String[]{"e:/input/inputtable","e:/output1"};

        // 1 获取配置信息，或者job对象实例
        Configuration configuration = new Configuration();
        Job job = Job.getInstance(configuration);
```

```
        // 2 指定本程序的jar包所在的本地路径
        job.setJarByClass(TableDriver.class);

        // 3 指定本业务job要使用的Mapper/Reducer业务类
        job.setMapperClass(TableMapper.class);
        job.setReducerClass(TableReducer.class);

        // 4 指定Mapper输出数据的kv类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(TableBean.class);

        // 5 指定最终输出的数据的kv类型
        job.setOutputKeyClass(TableBean.class);
        job.setOutputValueClass(NullWritable.class);

        // 6 指定job的输入原始文件所在目录
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        // 7 将job中配置的相关参数，以及job所用的java类所在的jar包， 提交给yarn去运行
        boolean result = job.waitForCompletion(true); System.exit(result ? 0 :
1);
    }
}
```

总解

缺点：这种方式中，合并的操作是Reduce阶段完成的，Reduce端的处理压力太大，Map节点的运算负载则很低，资源利用率不高，而且Reduce阶段极易产生数据倾斜。

解决方案：Map端实现数据合并

MapJoin

应用场景

MapJoin适用于一张表十分小，一张表十分大的场景

对于Reducer数据倾斜的处理办法是：在Map端缓存多张表，提前处理业务逻辑，这样增加Map端业务，减少Reduce端数据的压力，尽可能减少数据倾斜

通常都是用hive进行mapjoin操作

采用DistributedCache

（1）在Mapper的setup阶段，奖文件读入到缓存集合中

（2）在驱动函数加载缓存

```
//缓存普通文件到Tas运行节点
job.addCacheFile(new URI("file://e:/cache/pd.txt"));
```

代码

```
import java.io.BufferedReader;
import java.io.FileReader;
```

```
import java.io.IOException;
import java.util.HashMap;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

/**
 * map-join中小表的数据如下：
 *
 *    1    Beijing
 *    2    Guangzhou
 *    3    Shenzhen
 *    4    Xian
 *
 * 大表的数据如下：
 *
 *    Beijing Red Star              1
 *    Shenzhen Thunder              3
 *    Guangzhou Honda               2
 *    Beijing Rising                1
 *    Guangzhou Development Bank    2
 *    Tencent                       3
 *    Back of Beijing               1
 */
public class MapJoin {

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);
        job.setJarByClass(MapJoin2.class);
        //此方法已过时,被job.addCacheFile()所取代
        //DistributedCache.addCacheFile(new
URI("hdfs://10.16.17.182:9000/test/in/address.txt"), conf);
        //加载小表到 分布式缓存DistributedCache
        job.addCacheFile(new Path(args[0]).toUri());
        job.setMapperClass(MJMapper.class);
        job.setNumReduceTasks(0);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        FileInputFormat.addInputPath(job, new Path(args[1]));
        FileOutputFormat.setOutputPath(job, new Path(args[2]));
        System.exit(job.waitForCompletion(true)? 0:1);
    }

    public static class MJMapper extends Mapper<LongWritable, Text, Text, Text>{

        /**
         * 此map是存放小表数据用的
         * 注意小表的key是不能重复的，类似与数据库的外键表
         * 在这里的小表，就相当于一个外键表
```

```
         * **/
        private HashMap<String, String> map=new HashMap<String, String>();

        @Override
        protected void setup(Context context) throws IOException,
InterruptedException {

            BufferedReader br=null;         // 读取文件流
            String line;

            // 获取DistributedCached里面 的共享文件
             Path[] paths = context.getLocalCacheFiles();

            for(Path path : paths){
                //如果是 address文件
                if(path.getName().indexOf("address") >= 0){
                    br=new BufferedReader(new FileReader(path.toString()));

                    //读取文件中的每一行
                    while((line=br.readLine()) != null){
                        String[] splited = line.split("\t");
                        //将小表解析成 key/value 存放进map
                        map.put(splited[0], splited[1]);
                    }
                }
            }
        }

        /**
         * map阶段读取并处理大表中的数据
         * 小表中的数据是加载到HashMap中的，无需从hdfs读取
         */
        @Override
        protected void map(LongWritable key, Text value, Context context)
                throws IOException, InterruptedException {

            if(value==null || ("").equals(value.toString())){    //跳过空值
                return;
            }

            String[] splited = value.toString().split("\t");
            if(map.get(splited[1]) != null){    //map中大表的 key 对应的 value 不为
空
                Text keyOut = new Text(splited[0]);         //key=大表的第一列

                Text valueOut = new Text(map.get(splited[1]));    //value=小表的
第二列
                context.write(keyOut, valueOut);
            }
        }
    }
}
```