## 3.6 MapReduce Join关联

### 3.6.1 Reduce join(合并)

1．原理

Map端的主要工作：为来自不同表（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端的主要工作：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将那些来源于不同文件的记录（在map阶段已经打标志）分开，最后进行合并就ok了。

### 3.6.2 Reduce join案例实操

1．需求

表 订单数据表t\_order

|  |  |  |
| --- | --- | --- |
| id | pid | amount |
| 1001 | 01 | 1 |
| 1002 | 02 | 2 |
| 1003 | 03 | 3 |



表 商品信息表t\_product

|  |  |
| --- | --- |
| pid | pname |
| 01 | 小米 |
| 02 | 华为 |
| 03 | 格力 |

sql:select \* from t\_order o join t\_product p on o.pid=p.pid; 笛卡尔积，join条件：t\_order.pid=t\_product.pid;

将商品信息表中数据根据商品pid合并到订单数据表中。

表 最终数据形式

|  |  |  |
| --- | --- | --- |
| id | pname | amount |
| 1001 | 小米 | 1 |
| 1004 | 小米 | 4 |
| 1002 | 华为 | 2 |
| 1005 | 华为 | 5 |
| 1003 | 格力 | 3 |
| 1006 | 格力 | 6 |

通过将关联条件作为map输出的key，将两表满足join条件的数据（包含数据来源于哪一个文件的标识），发往同一个reduce task，在reduce中进行数据的串联，如图所示



图 reduce端表合并

1）创建商品和订单合并后的bean类

|  |
| --- |
| package com.bigdata.mapreduce.table;  import java.io.DataInput;  import java.io.DataOutput;  import java.io.IOException;  import org.apache.hadoop.io.Writable;  public class TableBean implements 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" ;  }  } |

2）编写TableMapper程序

|  |
| --- |
| package com.bigdata.mapreduce.table;  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>{  TableBean bean = new TableBean();  Text k = new Text();    @Override  protected void map(LongWritable key, Text value, Context context)  throws IOException, InterruptedException {    // 1 获取输入文件类型  FileSplit split = (FileSplit) context.getInputSplit();  String name = split.getPath().getName();    // 2 获取输入数据  String line = value.toString();    // 3 不同文件分别处理  if (name.startsWith("order")) {// 订单表处理  // 3.1 切割  String[] fields = line.split("\t");    // 3.2 封装bean对象  bean.setOrder\_id(fields[0]);  bean.setP\_id(fields[1]);  bean.setAmount(Integer.parseInt(fields[2]));  bean.setPname("");  bean.setFlag("0");    k.set(fields[1]);  }else {// 产品表处理  // 3.3 切割  String[] fields = line.split("\t");    // 3.4 封装bean对象  bean.setP\_id(fields[0]);  bean.setPname(fields[1]);  bean.setFlag("1");  bean.setAmount(0);  bean.setOrder\_id("");    k.set(fields[0]);  }  // 4 写出  context.write(k, bean);  }  } |

3）编写TableReducer程序

|  |
| --- |
| package com.bigdata.mapreduce.table;  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) {  if ("0".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 {  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){  bean.setPname (pdBean.getPname());  // 4 数据写出去  context.write(bean, NullWritable.get());  }  }  } |

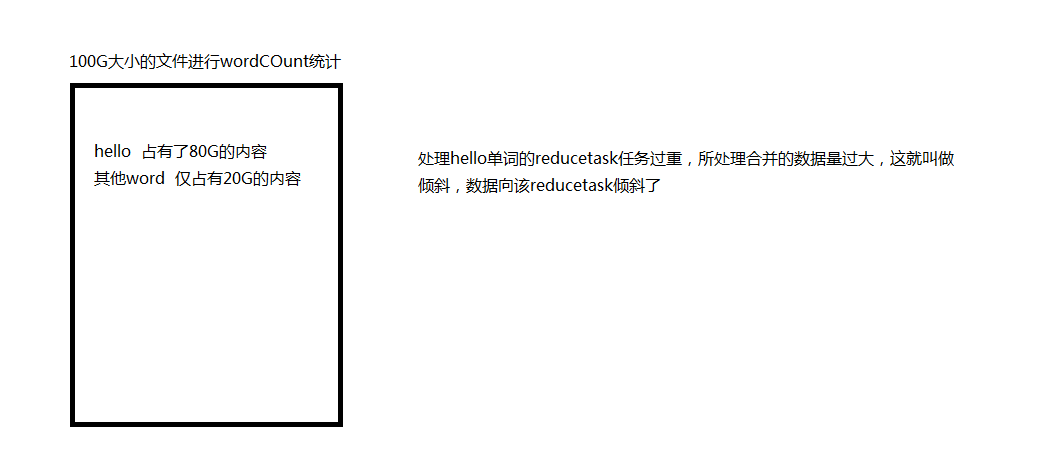
4）编写TableDriver程序

|  |
| --- |
| package com.bigdata.mapreduce.table;  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 {  // 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);  }  } |

3）运行程序查看结果

|  |
| --- |
| 1001 小米 1  1001 小米 1  1002 华为 2  1002 华为 2  1003 格力 3  1003 格力 3 |

缺点：这种方式中，合并的操作是在reduce阶段完成，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，且在reduce阶段极易产生数据倾斜（同一个reduce接收到的数据量很大）



解决方案： map端实现数据合并

### 3.6.3 Map join（合并）

1．使用场景

一张表十分小、一张表很大。

2．解决方案

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，尽可能的减少数据倾斜。

3．具体办法：采用distributedcache

（1）在mapper的setup阶段，将文件读取到缓存集合中。

（2）在驱动函数中加载缓存。

job.addCacheFile(new URI("file:/e:/mapjoincache/pd.txt"));// 缓存普通文件到task运行节点，如图21所示



图21 map端表合并

### 3.6.4 Map join案例实操

1．分析

适用于关联表中有小表的情形；

可以将小表分发到所有的map节点，这样，map节点就可以在本地对自己所读到的大表数据进行合并并输出最终结果，可以大大提高合并操作的并发度，加快处理速度。

2．实现代码

（1）先在驱动模块中添加缓存文件

|  |
| --- |
| package test;  import java.net.URI;  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 DistributedCacheDriver {  public static void main(String[] args) throws Exception {  // 1 获取job信息  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);  // 2 设置加载jar包路径  job.setJarByClass(DistributedCacheDriver.class);  // 3 关联map  job.setMapperClass(DistributedCacheMapper.class);  // 4 设置最终输出数据类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(NullWritable.class);  // 5 设置输入输出路径  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  // 6 加载缓存数据  job.addCacheFile(new URI("file:///e:/inputcache/pd.txt"));    // 7 map端join的逻辑不需要reduce阶段，设置reducetask数量为0  job.setNumReduceTasks(0);  // 8 提交  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

（2）读取缓存的文件数据

|  |
| --- |
| package test;  import java.io.BufferedReader;  import java.io.FileInputStream;  import java.io.IOException;  import java.io.InputStreamReader;  import java.util.HashMap;  import java.util.Map;  import org.apache.commons.lang.StringUtils;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class DistributedCacheMapper extends Mapper<LongWritable, Text, Text, NullWritable>{  Map<String, String> pdMap = new HashMap<>();    @Override  protected void setup(Mapper<LongWritable, Text, Text, NullWritable>.Context context)  throws IOException, InterruptedException {  // 1 获取缓存的文件  BufferedReader reader = new BufferedReader(new InputStreamReader(new FileInputStream("pd.txt"),"UTF-8"));    String line;  while(StringUtils.isNotEmpty(line = reader.readLine())){  // 2 切割  String[] fields = line.split("\t");    // 3 缓存数据到集合  pdMap.put(fields[0], fields[1]);  }    // 4 关流  reader.close();  }    Text k = new Text();    @Override  protected void map(LongWritable key, Text value, Context context)  throws IOException, InterruptedException {  // 1 获取一行  String line = value.toString();    // 2 截取  String[] fields = line.split("\t");    // 3 获取产品id  String pId = fields[1];    // 4 获取商品名称  String pdName = pdMap.get(pId);    // 5 拼接  k.set(line + "\t"+ pdName);    // 6 写出  context.write(k, NullWritable.get());  }  } |

。

# 第4章 Yarn

## 4.1 Yarn概述

Yarn是一个资源调度平台，负责为运算程序提供服务器运算资源，相当于一个分布式的操作系统平台，而MapReduce**等**运算程序则相当于运行于操作系统之上的应用程序。

## 4.2 Yarn基本架构

  YARN主要由ResourceManager、NodeManager、ApplicationMaster（AM）和Container等组件构成，如图23所示。



图23 Yarn基本架构

## 4.3 Yarn工作机制

1．Yarn运行机制，如图24所示



图24 Yarn工作机制

2．工作机制详解

（1）Mr程序提交到客户端所在的节点。

（2）Yarnrunner向Resourcemanager申请一个Application。

（3）rm将该应用程序的资源路径返回给yarnrunner。

（4）该程序将运行所需资源提交到HDFS上。

（5）程序资源提交完毕后，申请运行mrAppMaster。

（6）RM将用户的请求初始化成一个task。

（7）其中一个NodeManager领取到task任务。

（8）该NodeManager创建容器Container，并产生MRAppmaster。

（9）Container从HDFS上拷贝资源到本地。

（10）MRAppmaster向RM 申请运行maptask资源。

（11）RM将运行maptask任务分配给另外两个NodeManager，另两个NodeManager分别领取任务并创建容器。

（12）MR向两个接收到任务的NodeManager发送程序启动脚本，这两个NodeManager分别启动maptask，maptask对数据分区排序。

（13）MrAppMaster等待所有maptask运行完毕后，向RM申请容器，运行reduce task。

（14）reduce task向maptask获取相应分区的数据。

（15）程序运行完毕后，MR会向RM申请注销自己。

## 4.5 资源调度器

目前，Hadoop作业调度器主要有三种：FIFO、Capacity Scheduler和Fair Scheduler。Hadoop2.7.2默认的资源调度器是Capacity Scheduler。

具体设置详见：yarn-default.xml文件

<property>

<description>The class to use as the resource scheduler.</description>

<name>yarn.resourcemanager.scheduler.class</name>

<value>org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler</value>

</property>

　　1．先进先出调度器（FIFO），如图所示

 图 资源调度器FIFO

2．容量调度器（Capacity Scheduler），如图28所示

 图 资源调度器之容量调度器

3．公平调度器（Fair Scheduler），如图所示



图29 资源调度器之公平调度器

# 第5章 Hadoop企业优化

## 5.1 MapReduce跑的慢的原因

Mapreduce 程序效率的瓶颈在于两点：

1．计算机性能

CPU、内存、磁盘健康、网络

2．I/O 操作优化

（1）数据倾斜

（2）map和reduce数设置不合理 combineTextinputformat,分区

（3）map运行时间太长，导致reduce等待过久

（4）小文件过多

（5）spill(溢出)次数过多，溢出数据到磁盘，

（6）merge次数过多等。Shuffule,溢出之后会有合并，reduce端也会有合并

## 5.2 MapReduce优化方法

MapReduce优化方法主要从六个方面考虑：数据输入、Map阶段、Reduce阶段、IO传输、数据倾斜问题和常用的调优参数。

### 5.2.1 数据输入

（1）合并小文件：在执行mr任务前将小文件进行合并，大量的小文件会产生大量的map任务，大量节点资源被占用，从而导致mr整体运行较慢。

（2）采用CombineTextInputFormat来作为输入，解决输入端大量小文件场景。

### 5.2.2 Map阶段

**1）减少溢写（spill）次数：**通过调整io.sort.mb及sort.spill.percent参数值（在mapred-default.xml），增大触发spill的内存上限，减少spill次数，从而减少磁盘IO。

**2）减少合并（merge）次数：**通过调整io.sort.factor参数（在mapred-default.xml），增大merge的文件数目，减少merge的次数，从而缩短mr处理时间。

3）在map之后，**不影响业务逻辑前提下，先进行combine处理**，减少 I/O。

### 5.2.3 Reduce阶段

**1）合理设置map和reduce数**：两个都不能设置太少，也不能设置太多。太少，会导致task等待，延长处理时间；太多，会导致 map、reduce任务间竞争资源，造成处理超时等错误。

**2）设置map、reduce共存：**调整slowstart.completedmaps参数，使map运行到一定程度后，reduce也开始运行，减少reduce的等待时间。

**3）合理设置reduce端的buffer：**默认情况下，数据达到一个阈值的时候，buffer中的数据就会写入磁盘，然后reduce会从磁盘中获得所有的数据。也就是说，buffer和reduce是没有直接关联的，中间多个一个写磁盘->读磁盘的过程，既然有这个弊端，那么就可以通过参数来配置，使得buffer中的一部分数据可以直接输送到reduce，从而减少IO开销：mapred.job.reduce.input.buffer.percent，默认为0.0。当值大于0的时候，会保留指定比例的内存读buffer中的数据直接拿给reduce使用。这样一来，设置buffer需要内存，读取数据需要内存，reduce计算也要内存，所以要根据作业的运行情况进行调整。

### 5.2.4 数据倾斜问题

1．数据倾斜现象

数据频率倾斜——某一个区域的数据量要远远大于其他区域。

2．如何收集倾斜数据

在reduce方法中加入记录map输出键的详细情况的功能。

public static final String MAX\_VALUES = "skew.maxvalues";

private int maxValueThreshold;

@Override

public void configure(JobConf job) {

     maxValueThreshold = job.getInt(MAX\_VALUES, 100);

}

@Override

public void reduce(Text key, Iterator<Text> values,

                     OutputCollector<Text, Text> output,

                     Reporter reporter) throws IOException {

     int i = 0;

     while (values.hasNext()) {

         values.next();

         i++;

     }

     if (++i > maxValueThreshold) {

         log.info("Received " + i + " values for key " + key);

     }

}

3．减少数据倾斜的方法

**方法1：自定义分区**

基于输出键的背景知识进行自定义分区。例如，如果map输出键的单词来源于一本书。且其中某几个专业词汇较多。那么就可以自定义分区将这这些专业词汇发送给固定的一部分reduce实例。而将其他的都发送给剩余的reduce实例。

**方法2：Combine**

使用Combine可以大量地减小数据倾斜。在可能的情况下，combine的目的就是聚合并精简数据。

**方法3：采用Map Join，尽量避免Reduce Join。**

# 第6章 常见错误及解决方案

1） 导包容易出错。尤其Text和CombineTextInputFormat。

2） Mapper中第一个输入的参数必须是LongWritable或者NullWritable，不可以是IntWritable. 报的错误是类型转换异常。

3） java.lang.Exception: java.io.IOException: Illegal partition for 13926435656 (4)，说明partition和reducetask个数没对上，调整reducetask个数。

4） 如果分区数不是1，但是reducetask为1，是否执行分区过程。答案是：不执行分区过程。因为在maptask的源码中，执行分区的前提是先判断reduceNum个数是否大于1。不大于1肯定不执行。

5）在Windows环境编译的jar包导入到linux环境中运行，

hadoop jar wc.jar com.bigdata.mapreduce.wordcount.WordCountDriver /user/bigdata/ /user/bigdata/output

报如下错误：

Exception in thread "main" java.lang.UnsupportedClassVersionError: com/bigdata/mapreduce/wordcount/WordCountDriver : Unsupported major.minor version 52.0

原因是Windows环境用的jdk1.7，linux环境用的jdk1.8。

解决方案：统一jdk版本。

6）缓存pd.txt小文件案例中，报找不到pd.txt文件

原因：大部分为路径书写错误。还有就是要检查pd.txt的问题。还有个别电脑写相对路径找不到pd.txt，可以修改为绝对路径。

7）报类型转换异常。

通常都是在驱动函数中设置map输出和最终输出时编写错误。

Map输出的key如果没有排序，也会报类型转换异常。

8）集群中运行wc.jar时出现了无法获得输入文件。

原因：wordcount案例的输入文件不能放在hdfs集群的根目录。