MapReduce Sample code and Explanations

*v1.1 (Sept 2023)*

# Chain 2 MapReduce Jobs

Create two different configurations for the two jobs as “conf” and “conf2” where they also get two different instances.

|  |
| --- |
| *Configuration* conf = **new** Configuration();  Job job1 = Job.getInstance(conf, "Count page views");  *Configuration* conf2 = **new** Configuration();  Job job2 = Job.getInstance(conf2, "Most interesting pages"); |

Job2 can be executed only after job1 is finished.

|  |
| --- |
| job1.waitForCompletion(true); |

**How to get output file from one MapReduce as input file to another MapReduce -**

Output path for first job = Input path for second job

|  |
| --- |
| FileOutputFormat.setOutputPath(job1, new Path(args[1]));  FileInputFormat.addInputPath(job2, new Path(args[1])); |

Here’s an example of setting configurations for 2 MR jobs -



# Input 2 files for MapReduce

You can use:

|  |
| --- |
| FileInputFormat.addInputPaths([Job](https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/Job.html) job, [String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html?is-external=true) commaSeparatedPaths)  *FileInputFormat.addInputPaths(job1, new Path(args[0]) +","+ new Path(args[1]))* |



# Joins in MapReduce

(If you want to know more about Joins in MapReduce, please refer to the book ***MapReduce Design Patterns***.)

## Two types of Joins

There are two major types of joins in MapReduce: Reduce-side Join and Replicated Join (Map-side Join). As far as which pattern to choose, it depends largely on **how large the datasets are and what type of join you want**.

## Reduce-side Join

Reduce-side Join is the most basic join pattern in MapReduce. It can take the longest time to execute compared to other join patterns, but it’s simple to implement and support all different join operations (inner join & outer join).

### Pros & Cons

**Pros:**

1. Easiest to implement
2. Support both inner join and outer joins
3. No limitation on the size of datasets
4. Can join as many data sets together at once as needed

**Cons:**

1. Requires a large amount of network bandwidth
2. Take some longer time to execute

*Note: if all the data sets are large, Reduce-side join may be the only choice.*

### When to use Reduce-side Join

* Multiple large data sets are being joined. If any of data set can be fit into memory, please try using the replicated join (we will talk about it in next section).
* Need flexibility of executing any join operation.

### The structure of Reduce-side join

#### **Mapper**

The mapper takes each input record from each data set and extracting the foreign key.

**Mapper output key**: the extracted foreign key

**Mapper output value**: you can use the entire record as the output value. However, you are free to pick up columns/attributes you need for later computation (you can exclude the foreign key, since we already use it as the mapper output key). Beside the record, we also need a unique identifier for the dataset. For example, using “P” to tag “MyPages” dataset, “F” to tag “Friends”, “A” to tag “AccessLog”.

**Mapper output <key, value> format**

<foreignkey, “dataset tag letter” + attributes in a record needed for later computation>

#### **Reducer**

Collect the values of each input group into a temporary list.

Store record into a list with the same tag.

*Given an example of Project 1, if an input record from myPage dataset in the reducer <14000, P14000 mike canada 10 piano>, we extract the 1st character of input value part “P”, and store the whole record inside a list called myPages. If we have another input record from Friends dataset in the reducer <14000, F230000 15000 2 college-friend >, we extract the 1st character of the input value part “F”, we know this record is originally from the Friends dataset and we will add this record into a list called Friends.*

These lists are then iterated over and the records from both sets are joined together.

**Reducer output <key, value> format:**

<all order info, all matching product info>

### Code example:

Requirement:

In this example, we’ll be using the order table (dataset) and product table (dataset). Using a Reduce-side join, these two data sets can be merged using the product id (*pid*) as the foreign key. In this example, we’ll perform an inner join. The choice of which join to execute is set during job configuration.

*Order.txt*

|  |  |  |
| --- | --- | --- |
| id | pid | amount |
| 1001 | 01 | 1 |
| 1002 | 02 | 2 |
| 1003 | 03 | 3 |
| 1004 | 01 | 4 |
| 1005 | 02 | 5 |
| 1006 | 03 | 6 |

*Product.txt*

|  |  |
| --- | --- |
| pid | pname |
| 01 | iphone |
| 02 | galaxy |
| 03 | pixel |

**Driver code**

We need configure the job to allow multiple input utility and also set the join type in the job configuration to args[2] to make it available to use in the reducer. Part of driver code for this join:

|  |
| --- |
| *// Use MultipleInputs to set which input uses what mapper*  *// The first two elements of the args array are the two inputs*  MultipleInputs.addInputPath(job, new Path(args[0]), TextInputFormat.class,  OrderJoinMapper.class);  MultipleInputs.addInputPath(job, new Path(args[1]), TextInputFormat.class,  ProductJoinMapper.class);  job.getConfiguration()..set("join.type", args[2]) |

**Order mapper code**

This mapper parses each input line of Order.txt. product ID (pid) is being grabbed from each record as the output key and the entre input value will be the output value. It prepends the letter “O” in front of the entire value in the output value part. The tag letter “O” makes reducer know values came from the Order.txt dataset.

|  |
| --- |
| public static class OrderJoinMapper extends Mapper<LongWritable, Text, Text, Text> {  private Text outkey = new Text();  private Text outvalue = new Text();  public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  // get a line of input from Order.txt  String line = value.toString();  String[] split = line.split(" ");  // The foreign join key is the product ID (pid)  outkey.set(split(1));  // Flag this record for the reducer and then output  outvalue.set("O" + value.toString());  context.write(outkey, outvalue);  }  } |

**Product mapper code**

Like the OrderJoinMapper, this mapper parses each input line of Product.txt. It grabs the product ID (pid) from each input record as the output key and the entre input value will be the output value. It prepends the letter “P” in front of the entire value in the output value part. The tag letter “P” makes reducer know values came from the Order.txt dataset.

|  |
| --- |
| public static class ProductJoinMapper extends Mapper<LongWritable, Text, Text, Text> {  private Text outkey = new Text();  private Text outvalue = new Text();  public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  // get a line of input from Product.txt  String line = value.toString();  String[] split = line.split(" ");  // The foreign join key is the product ID (pid)  outkey.set(split(0));  // Flag this record for the reducer and then output  outvalue.set("P" + value.toString());  context.write(outkey, outvalue);  }  } |

**Reducer code**

The reducer iterates through all the values of each group (same mapper output key) and check the data source tag we prepended to the output value of each record. We create a list for records from each dataset (data source): *orderList* & *productList* and put records into these two lists based on the tag “O” or “P”. After binning all values to the list each belongs to, we execute the actual join logic, which differs slight based on the join type (inner join or outer join), iterating through both lists and writing to the Context object. We can retrieve the join type info from the job configuration in the setup method.

|  |
| --- |
| public static class JoinReducer extends Reducer<Text, Text, Text, Text> {  private static final Text EMPTY\_TEXT = Text("");  private Text tag = new Text();  private ArrayList<Text> orderList = new ArrayList<Text>();  private ArrayList<Text> productList = new ArrayList<Text>();  private String joinType = null;  public void setup(Context context) {  // retrieve the type of join from our configuration  joinType = context.getConfiguration().get("join.type");  }  public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {  // clear list  orderList.clear();  productList.clear();  // bin each record from both mapper based on the tag letter "O" or "P". Then, remove the tag.  while (values.hasNext()) {  tag = values.next();  if (tag.charAt(0) == 'O') {  orderList.add(new Text(tag.toString().substring(1)));  } else if (tag.charAt('0') == 'P') {  productList.add(new Text(tag.toString().substring(1)));  }  }  // execute the join logic after both source lists are filled after iterating all values    executeJoinLogic(context);  }  private void executeJoinLogic(Context context) throws IOException, InterruptedException {  // you can change the type of join based on the type you configured in driver class  // here we use inner join as an example  if (joinType.equalsIgnoreCase("inner")) {  if (!orderList.isEmpty() && !productList.isEmpty()) {  for (Text O : orderList) {  for (Text P : productList) {  context.write(O, P);  }  }  }  }  }  } |

*Note: here we only provide the example of inner join, which will be enough for project 1.*

*Note: since the join logic happens inside the reducer, a combiner won’t offer much optimization here.*

## Replicated Join (Map-side Join)

Replicated-side Join is a join pattern in MapReduce between one large and many small data sets. It is done on the map-side only.

### Pros & Cons

**Pros:**

* Only need a map phase

**Cons:**

* Strict data set limit on all data sets (all small data sets must be fit into memory)
* Can only be useful when being used to handle inner join or left-outer join (the large dataset is the “left” data set)

*Note: if all the data sets are large or other types of joins are needed, Reduce-side join may be the one should be considered.*

### When to use Replicated Join

* The type of join we plan to perform is either an inner join or a left outer join (with the large data set to be the left side).
* All data sets except for the large one must be fit into the memory of each individual map task.

### The structure of Replicate Join

#### **Mapper**

Setup phase: the mapper reads all data sets (except for the large data set) from the distributed cache in the phase of setup and store them into in-memory lookup tables.

Map phase: the mapper reads each record from the large data set and joins it with all the data stored inside memory. If the foreign key is not found in the lookup table inside the memory, the record is being omitted (in the case of inner join) or being output (in the case of left-outer join).

### Code example:

Requirement:

In this example, we’ll be using the same example as what we covered in Reduce-side join: Order table (dataset) and Product table (dataset). Order.txt is being regarded as the large data set and Product.txt is being treated as the small data set that could be fit into the main memory.

*Order.txt*

|  |  |  |
| --- | --- | --- |
| id | pid | amount |
| 1001 | 01 | 1 |
| 1002 | 02 | 2 |
| 1003 | 03 | 3 |
| 1004 | 01 | 4 |
| 1005 | 02 | 5 |
| 1006 | 03 | 6 |

*Product.txt*

|  |  |
| --- | --- |
| pid | pname |
| 01 | iphone |
| 02 | galaxy |
| 03 | pixel |

*Expect output format*

|  |  |  |
| --- | --- | --- |
| id | pname | amount |
| 1001 | iphone | 1 |
| 1004 | iphone | 4 |
| 1002 | galaxy | 2 |
| 1005 | galaxy | 5 |
| 1003 | pixel | 3 |
| 1006 | pixel | 6 |

**Driver code**

|  |
| --- |
| *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;*  *import java.io.IOException;*  *import java.net.URI;*  *import java.net.URISyntaxException;*  *public class ReplicatedJoinDriver {*  *public static void main(String[] args) throws IOException, URISyntaxException, ClassNotFoundException, InterruptedException {*  *Configuration conf = new Configuration();*  *Job job = Job.getInstance(conf);*  *job.setJarByClass(ReplicatedJoinDriver.class);*  *job.setMapperClass(ReplicatedJoinMapper.class);*  *job.setMapOutputKeyClass(Text.class);*  *job.setMapOutputValueClass(NullWritable.class);*  *job.setOutputKeyClass(Text.class);*  *job.setOutputValueClass(NullWritable.class);*  *// a file in local file system is being used here as an example*  *job.addCacheFile(new URI("file:///D:/input/tablecache/Product.txt"));*  *// set the number of reduceTask to 0 becuase this is a map-only job*  *job.setNumReduceTasks(0);*  *// change the following input/output path based on your need*  *FileInputFormat.setInputPaths(job, new Path("D:\\input"));*  *FileOutputFormat.setOutputPath(job, new Path("D:\\output"));*    *boolean b = job.waitForCompletion(true);*  *System.exit(b ? 0 : 1);*  *}*  *}* |

**Mapper code**

Setup phase: product data (Product.txt) is being read from the ***DistributedCache*** and store inside the memory. For each record of Product.txt in memory, product ID (pid) is retrieved from the record. Product ID and the record will be added to a HashMap to be ready for being checked up later.

Map phase: Product ID (pid) is retrieved from each record read from the large dataset Order.txt. The retrieved Product ID (pid) will be used to get a value from the HashMap lookup table built in Setup phase.

|  |
| --- |
| import org.apache.commons.lang.StringUtils;  import org.apache.hadoop.fs.FSDataInputStream;  import org.apache.hadoop.fs.FileSystem;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.IOUtils;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  import java.io.BufferedReader;  import java.io.IOException;  import java.io.InputStreamReader;  import java.net.URI;  import java.util.HashMap;  import java.util.Map;  public class ReplicatedJoinMapper extends Mapper<LongWritable, Text, Text, NullWritable> {  private Map<String, String> productMap = new HashMap<>();  private Text text = new Text();  // read the record from Product.txt into the distributed cache  @Override  protected void setup(Context context) throws IOException, InterruptedException {  URI[] cacheFiles = context.getCacheFiles();  Path path = new Path(cacheFiles[0]);  // open the stream  FileSystem fs = FileSystem.get(context.getConfiguration());  FSDataInputStream fis = fs.open(path);  // wrap it into a BufferedReader object which is easy to read a record  BufferedReader reader = new BufferedReader(new InputStreamReader(fis, "UTF-8"));  // read the record line by line  String line;  while (StringUtils.isNotEmpty(line = reader.readLine())) {  String[] split = line.split(" ");  productMap.put(split[0], split[1]);  }  // close the stream  IOUtils.closeStream(reader);  }  @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  // read each line of the large data set (Order.txt)  String[] fields = value.toString().split(" ");  // use the Product ID (pid) pulled from the Order.txt data set  // to retrieve Product name info from the lookup table in memory  String productName = productMap.get(fields[1]);  // replace the Product ID (pid) in Order.txt with the matching Product name productName    text.set(fields[0] + "\t" + productName + "\t" + fields[2]);  // output the mapper key-value pair  context.write(text, NullWritable.get());  }  } |