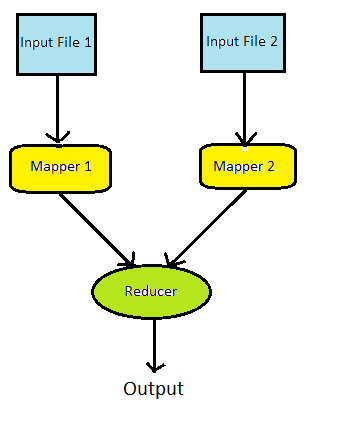
1. How can mapreduce work with multiple small files(1 milion)
2. Distributed cache in mapreduce
3. File permissions
4. Data compression
5. Mapreduce optimization
6. How to compress output in mapreduce program
7. Implement custom writable
8. What is raw comparator
9. What is partitioner n mapreduce
10. How to process hole file as single record
11. Two rows ae record
12. Two rows as a record
13. Multipule inputs
14. Multipule outputs
15. Counters,user define counters, dynamic counters
16. How to kill job in running
17. Sorting in mapreduce
18. Secondary sort
19. Joins in mapreduce
20. Map side joins
21. Reduceside joins
22. Side data distribution
23. Distributed cache
24. Avro
25. Parquet
26. How to make intermediate code compression
27. Secondarysort on values
28. Recordreader and inputsplit, Recordwriter
29. Inputformat and outputformat
30. How to debug
31. Skiping bad records
32. Working with small files(1 million)
33. Identity mapper and reducer
34. Write a MapReduce Program to find a duplicate in dataset?
35. Userdefined Datatypes instead of <LongWritable,Text,Text,IntWritable>
36. Importing the structured data from RDBMS to HDFS can happen using SQOOP.The same importing can happen using java multihreading concept. Then why to use Sqoop rather than java multi-threading?
37. How much data for day,hourly etc
38. How to connect peoplesoft to hdfs
39. Different data formats in HDFS
40. how much time it requires to transfer into hdfs and what would be the volume of data
41. Scheduler: by default uses FIFO scheduler, by using Fair Scheduler / Capacity Scheduler performance can be improved
42. Heap size : how to increase the heap size of the memory
43. How Distributed Cache works, when to use (mapside join)
44. How we can optimize mapper side join? use combiner ( applicable for association /commutation)
45. How to pass multiple records (say 3 records ) to mapper as input
46. Use custom inputformat and custom record reader ...
47. How do you process multiple small size files?
48. Is it possible to provide multiple input to Hadoop? If yes then how can you give multiple directories as input to the Hadoop job?
49. Why map side join and reduce join explain them?
50. How to replicate and which properties are used for the storage?
51. **)**      **How many ways are there to set the block size?**
52. **What is custom partitioner?**
53. Use of Mapper Context in Hadoop?
54. Compressed input to the reducer
55. How can mapreduce work with multiple small files(1 milion)

Answer)

**Mutiple Input Files In MapReduce: The Easy Way**

In the previous issue of this series, we discussed a simple method of using multiple input files : [Side Data Distribution](http://dailyhadoopsoup.blogspot.in/2014/01/reading-from-multiple-input-files-in.html). But it was of limited use as input files can only be of minimal size. In this issue, we’ll use our playground to investigate another approach to facilitate multiple input files offered by Hadoop.  
  
This approach as a matter of fact is very simple and effective. Here we simply need to understand the concept of number of mappers needed. As you may know, mapper extract its input from the input file. When there are more than input file , we need the same number of mapper to read records from input files. For instance, if we are using two input files then we need two mapper classes.

[](http://4.bp.blogspot.com/--kQHhv6ZynA/UupiqicS3mI/AAAAAAAAAK4/7dzDRftUjkk/s1600/xx.png)

We use MultipleInputs class which supports MapReduce jobs that have multiple input paths with a different InputFormat and Mapper for each path. To understand the concept more clearly let us take a case where user want to take input from two input files with similar structure. Also assume that both the input files have 2 columns, first having "Name" and second having "Age". We want to simply combine the data and sort it by "Name". What we need to do? Just two things:

1. Use two mapper classes.
2. Specify the mapper classes in MultipleInputs class object in run/main method.

File 1 File 2

Aman 19 Ash 12

Tom 20 James 21

Tony 15 Punk 21

John 18 Frank 20

Johnny 19

Hugh 17

Here is the code for the same. Notice two mapper classes with same logic and only single reducer.

import java.io.IOException;

import mutipleInput.Join;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.conf.Configured;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

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.Reducer;

import org.apache.hadoop.mapreduce.Mapper.Context;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.input.MultipleInputs;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.util.Tool;

import org.apache.hadoop.util.ToolRunner;

public class multiInputFile extends Configured implements Tool

{

public static class CounterMapper extends Mapper

{

public void map(LongWritable key, Text value, Context context)

throws IOException, InterruptedException

{

String[] line=value.toString().split("\t");

context.write(new Text(line[0]), new Text(line[1]));

}

}

public static class CountertwoMapper extends Mapper

{

public void map(LongWritable key, Text value, Context context)

throws IOException, InterruptedException

{

String[] line=value.toString().split("\t");

context.write(new Text(line[0]), new Text(line[1]));

}

}

public static class CounterReducer extends Reducer

{

String line=null;

public void reduce(Text key, Iterable values, Context context )

throws IOException, InterruptedException

{

for(Text value:values)

{

line = value.toString();

}

context.write(key, new Text(line));

}

}

public int run(String[] args) throws Exception {

Configuration conf = new Configuration();

Job job = new Job(conf, "aggprog");

job.setJarByClass(multiInputFile.class);

MultipleInputs.addInputPath(job,new Path(args[0]),TextInputFormat.class,CounterMapper.class);

MultipleInputs.addInputPath(job,new Path(args[1]),TextInputFormat.class,CountertwoMapper.class);

FileOutputFormat.setOutputPath(job, new Path(args[2]));

job.setReducerClass(CounterReducer.class);

job.setNumReduceTasks(1);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(Text.class);

return (job.waitForCompletion(true) ? 0 : 1);

}

public static void main(String[] args) throws Exception {

int ecode = ToolRunner.run(new multiInputFile(), args);

System.exit(ecode);

}

}

Here is the output.

Ash 12

Tony 15

Hugh 17

John 18

Aman 19

Johnny 19

Frank 20

Tom 20

James 21

Punk 21

In Our basic [**MapReduce Programs**](http://www.hadooptpoint.com/category/mapreduce/mapreduce-programs/) we took a single input file and then we load that particular input file from local path to Mapreduce framework.In this process we just add a single input file and then we got a single output file but coming to the real time scenarios we have to load multiple input files to mapreduce framework at that time this basic program concepts will not work.

So in this kind of situations we are using some Input formats and output formats.For Multiple Input files concept we are using **MultipleInputs**class and for Multiple output files we are using [**MultipleOutputs**](http://www.hadooptpoint.com/multiple-output-files-in-mapreduce-program-using-multipletextoutputformat/)class .

Here is the Main configurations for MultipleOutput class in Driver Class and **Hadoop multiple input files** example In MapReduce .

|  |  |
| --- | --- |
| [Click Here For Source code](http://www.hadooptpoint.com/hadoop-multiple-input-files-example-in-mapreduce/" \l "codesyntax_1" \o "Click to show/hide code block) |  |

MultipleInputs.addInputPath(job, **new** Path(args[0]),TextInputFormat.**class**, CounterMapper.**class**);  
MultipleInputs.addInputPath(job, **new** Path(args[1]),TextInputFormat.**class**, CountertwoMapper.**class**);

We have to import first of all **import org.apache.hadoop.mapreduce.lib.output.MultipleInputs**from hadoop librires for using this **MultipleInputs**class.

## **Hadoop multiple input files**

This approach as a matter of fact is very simple and effective. Here we simply need to understand the concept of number of mappers needed. As you may know, mapper extract its input from the input file. When there are more than input file , we need the same number of mapper to read records from input files. For instance, if we are using two input files then we need two mapper classes.

Hadoop multiple input files example In MapReduce

We use MultipleInputs class which supports MapReduce jobs that have multiple input paths with a different InputFormat and Mapper for each path. To understand the concept more clearly let us take a case where user want to take input from two input files with similar structure.

Also assume that both the input files have 2 columns, first having “Name” and second having “Age”. We want to simply combine the data and sort it by “Name”. What we need to do? Just two things:

1. Use two mapper classes.
2. Specify the mapper classes in **MultipleInputs** class object in run/main method.

Input Files

File1.txt

Aman  19  
Tom   20  
Tony  15  
John  18  
Johnny      19  
Hugh  17

File2.txt

James,21  
Punk,21  
Frank,20

## **Hadoop multiple input files Driver Class**

|  |  |
| --- | --- |
| [Click Here For Source code](http://www.hadooptpoint.com/hadoop-multiple-input-files-example-in-mapreduce/" \l "codesyntax_2" \o "Click to show/hide code block) |  |

**package** com.mcis;  
  
**import** java.io.IOException;  
  
**import** org.apache.hadoop.conf.Configuration;  
**import** org.apache.hadoop.conf.Configured;  
**import** org.apache.hadoop.fs.Path;  
**import** org.apache.hadoop.io.IntWritable;  
**import** org.apache.hadoop.io.LongWritable;  
**import** org.apache.hadoop.io.Text;  
**import** org.apache.hadoop.mapreduce.Job;  
**import** org.apache.hadoop.mapreduce.lib.input.MultipleInputs;  
**import** org.apache.hadoop.mapreduce.lib.input.TextInputFormat;  
**import** org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  
**import** org.apache.hadoop.util.Tool;  
**import** org.apache.hadoop.util.ToolRunner;  
  
**public** **class** MultiInputJob **extends** Configured **implements** Tool {  
  
    **public** **int** run(String[] args) **throws** Exception {  
        Configuration conf = **new** Configuration();  
        Job job = **new** Job(conf);  
        job.setJarByClass(MultiInputJob.**class**);  
        MultipleInputs.addInputPath(job, **new** Path(args[0]),  
                TextInputFormat.**class**, CounterMapper.**class**);  
        MultipleInputs.addInputPath(job, **new** Path(args[1]),  
                TextInputFormat.**class**, CountertwoMapper.**class**);  
  
        FileOutputFormat.setOutputPath(job, **new** Path(args[2]));  
        job.setReducerClass(CounterReducer.**class**);  
        job.setNumReduceTasks(1);  
        job.setMapOutputKeyClass(LongWritable.**class**);  
        job.setMapOutputValueClass(Text.**class**);  
        job.setOutputKeyClass(LongWritable.**class**);  
        job.setOutputValueClass(IntWritable.**class**);  
  
        **return** (job.waitForCompletion(**true**) ? 0 : 1);  
  
    }  
  
    **public** **static** **void** main(String[] args) **throws** Exception {  
  
        ToolRunner.run(**new** Configuration(), **new** MultiInputJob(), args);  
  
    }  
  
}

## **Hadoop multiple input files Counter Mapper Class**

|  |  |
| --- | --- |
| [Click Here For Source code](http://www.hadooptpoint.com/hadoop-multiple-input-files-example-in-mapreduce/" \l "codesyntax_3" \o "Click to show/hide code block) |  |

**package** com.mcis;  
  
**import** java.io.IOException;  
  
**import** org.apache.hadoop.io.LongWritable;  
**import** org.apache.hadoop.io.Text;  
**import** org.apache.hadoop.mapreduce.Mapper;  
  
**class** CounterMapper **extends** Mapper<LongWritable, Text, LongWritable,Text>  
{  
 **public** **void** map(LongWritable key, Text value, Context context)  
 **throws** IOException, InterruptedException  
 {  
  String[] line=value.toString().trim().split("**\t**");   
  
  context.write(**new** LongWritable(Long.parseLong(line[1])), **new** Text(line[0]));  
 }  
}

Several instances of the mapper function are created on the different machines in the cluster. Each instance receives a different input file. If files are bigger than the default dfs block size(128 MB) then files are further split into smaller parts and are then distributed to mappers.

So you can configure the input size being received by each mapper by following 2 ways:

* change the HDFS block size (eg dfs.block.size=1048576)
* set the paramaeter mapred.min.split.size (this can be only set to larger than HDFS block size)

**Note:** These parameters will only be effective if your input format supports splitting the input files. Common compression codecs (such as gzip) don't support splitting the files, so these will be ignored.

1. Distributed cache

# Answer) **Apache Hadoop Distributed Cache Example**

Posted by: [Raman Jhajj](https://examples.javacodegeeks.com/author/raman-jhajj/) in [Apache Hadoop](https://examples.javacodegeeks.com/category/enterprise-java/apache-hadoop/) March 7th, 2016

In this example article, we will go through Apache Hadoop Distributed Cache and will understand how to use it with MapReduce Jobs.

## **1. Introduction**

Distributed Cache as the name indicates is the caching system to store files or data which is required frequently and this mechanism is distributed in nature as all other components of Hadoop are.

It can cache **read-only** text files, archives, jar files etc. Which are needed by the application. So if there is a file which is needed by let us say map tasks. So it needs to be present on all the machines which will run Map tasks, This is what distributed cache is used for.

Top of Form

# **Want to be a Hadoop Master ?**

|  |
| --- |
| **Subscribe to our newsletter and download Hadoop Tutorial right now!****In order to help you master Apache Hadoop, we have compiled a kick-ass guide with all the basic concepts of a Hadoop cluster! Besides studying them online you may download the eBook in PDF format!**   **Email address:** |

Bottom of Form

## **2. Working**

Application which needs to use distributed cache to distribute a file should make sure that the file is available and can be accessed via urls. Urls can be either hdfs:// or http://.

Now once the file is present on the mentioned url and user mention it to be a cache file to the distributed cache API, the Map-Reduce framework will copy the necessary files on all the nodes before initiation of the tasks on those nodes.

**Notes:** In case the files provided are archives, these will be automatically unarchived on the nodes after transfer.

## **3. Implementation**

For understanding how to use the distributed cache API we will see an example in which we will write a modified version of the word count program.

For the basic word count example and if you like to understand the basics of how MapReduce job works, please refer to the article [Apache Hadoop Wordcount Example](http://examples.javacodegeeks.com/enterprise-java/apache-hadoop/apache-hadoop-wordcount-example/)

In this program, we will provide an input file to the Map-Reduce job with the words we need to count but we will also provide another file which contains stop words which we need to remove from the input text before counting the word occurrences.

So let’s start looking into the code:

### 3.1 The Driver Class

The driver class is the main entry point of the system and the class which set up the Map-Reduce job.

|  |  |
| --- | --- |
| 01 | package com.javacodegeeks.examples.distributedcache; |
| 02 |  |

|  |  |
| --- | --- |
| 03 | import org.apache.hadoop.conf.Configured; |
| 04 | import org.apache.hadoop.filecache.DistributedCache; |

|  |  |
| --- | --- |
| 05 | import org.apache.hadoop.fs.Path; |
| 06 | import org.apache.hadoop.io.IntWritable; |

|  |  |
| --- | --- |
| 07 | import org.apache.hadoop.io.Text; |
| 08 | import org.apache.hadoop.mapreduce.Job; |

|  |  |
| --- | --- |
| 09 | import org.apache.hadoop.mapreduce.lib.input.FileInputFormat; |
| 10 | import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat; |

|  |  |
| --- | --- |
| 11 | import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat; |
| 12 | import org.apache.hadoop.util.Tool; |

|  |  |
| --- | --- |
| 13 | import org.apache.hadoop.util.ToolRunner; |
| 14 |  |

|  |  |
| --- | --- |
| 15 | /\*\* |
| 16 | \* The entry point for the WordCount example, |

|  |  |
| --- | --- |
| 17 | \* which setup the Hadoop job with Map and Reduce Class |
| 18 | \* |

|  |  |
| --- | --- |
| 19 | \* @author Raman |
| 20 | \*/ |

|  |  |
| --- | --- |
| 21 | public class Driver extends Configured implements Tool{ |
| 22 |  |

|  |  |
| --- | --- |
| 23 | /\*\* |
| 24 | \* Main function which calls the run method and passes the args using ToolRunner |

|  |  |
| --- | --- |
| 25 | \* @param args Two arguments input and output file paths |
| 26 | \* @throws Exception |

|  |  |
| --- | --- |
| 27 | \*/ |
| 28 | public static void main(String[] args) throws Exception{ |

|  |  |
| --- | --- |
| 29 | int exitCode = ToolRunner.run(new Driver(), args); |
| 30 | System.exit(exitCode); |

|  |  |
| --- | --- |
| 31 | } |
| 32 |  |

|  |  |
| --- | --- |
| 33 | /\*\* |
| 34 | \* Run method which schedules the Hadoop Job |

|  |  |
| --- | --- |
| 35 | \* @param args Arguments passed in main function |
| 36 | \*/ |

|  |  |
| --- | --- |
| 37 | public int run(String[] args) throws Exception { |
| 38 | if (args.length != 3) { |

|  |  |
| --- | --- |
| 39 | System.err.printf("Usage: %s needs two arguments    files\n", |
| 40 | getClass().getSimpleName()); |

|  |  |
| --- | --- |
| 41 | return -1; |
| 42 | } |

|  |  |
| --- | --- |
| 43 |  |
| 44 | //Initialize the Hadoop job and set the jar as well as the name of the Job |

|  |  |
| --- | --- |
| 45 | Job job = new Job(); |
| 46 | job.setJarByClass(Driver.class); |

|  |  |
| --- | --- |
| 47 | job.setJobName("Word Counter With Stop Words Removal"); |
| 48 |  |

|  |  |
| --- | --- |
| 49 | //Add input and output file paths to job based on the arguments passed |
| 50 | FileInputFormat.addInputPath(job, new Path(args[0])); |

|  |  |
| --- | --- |
| 51 | FileOutputFormat.setOutputPath(job, new Path(args[1])); |
| 52 |  |

|  |  |
| --- | --- |
| 53 | job.setOutputKeyClass(Text.class); |
| 54 | job.setOutputValueClass(IntWritable.class); |

|  |  |
| --- | --- |
| 55 | job.setOutputFormatClass(TextOutputFormat.class); |
| 56 |  |

|  |  |
| --- | --- |
| 57 | //Set the MapClass and ReduceClass in the job |
| 58 | job.setMapperClass(MapClass.class); |

|  |  |
| --- | --- |
| 59 | job.setReducerClass(ReduceClass.class); |
| 60 |  |

|  |  |
| --- | --- |
| 61 | DistributedCache.addCacheFile(new Path(args[2]).toUri(), job.getConfiguration()); |
| 62 |  |

|  |  |
| --- | --- |
| 63 | //Wait for the job to complete and print if the job was successful or not |
| 64 | int returnValue = job.waitForCompletion(true) ? 0:1; |

|  |  |
| --- | --- |
| 65 |  |
| 66 | if(job.isSuccessful()) { |

|  |  |
| --- | --- |
| 67 | System.out.println("Job was successful"); |
| 68 | } else if(!job.isSuccessful()) { |

|  |  |
| --- | --- |
| 69 | System.out.println("Job was not successful"); |
| 70 | } |

|  |  |
| --- | --- |
| 71 |  |
| 72 | return returnValue; |

|  |  |
| --- | --- |
| 73 | } |
| 74 | } |

Above is the complete code of the driver class. You can see is the main() method we set up and initialize a Hadoop Job(). First of all this code checks for the arguments passed to the method. Arguments need to be 3 in number:

1. Input text file path which containc the text for word count
2. Output path for storing the output of the program
3. File path and name containing the stop words which we will distribute through the Hadoop Distributed Cache

The code:

|  |  |
| --- | --- |
| 1 | if (args.length != 3) { |
| 2 | System.err.printf("Usage: %s needs two arguments    files\n", |

|  |  |
| --- | --- |
| 3 | getClass().getSimpleName()); |
| 4 | return -1; |

|  |  |
| --- | --- |
| 5 | } |

checks for the number fo arguments and make sure we have the required number of arguments present otherwise it stops the program then and there.

After this the Job is initialized:

|  |  |
| --- | --- |
| 1 | //Initialize the Hadoop job and set the jar as well as the name of the Job |
| 2 | Job job = new Job(); |

followed by all the necessary configuration settings including configuring the jar file, map and reduce classes, input and output methods and input and output paths.

Out main focus here is on the line number 61, which is:

|  |  |
| --- | --- |
| 1 | DistributedCache.addCacheFile(new Path(args[2]).toUri(), job.getConfiguration()); |

This line of code calls the DistributedCache API and adds the cache file URL which we passed as the third argument to the program. Before passing this argument, it need to be converted to the path url. Second argument needs to be the configurations of the Hadoop job we are setting up.

The above code will set up the Hadoop Job and sets up the required file as the cache file in the Hadoop cluster. It is as easy as calling a single function. The main task is how to retrieve this cache file and how to use it to remove stop words from the processing text. That we will see in the map class in the following section.

### 3.2 Map Class

Map class contains the mapper method which is the main focus which contains the code regarding how to use the cache files in the MapReduce Tasks.

|  |  |
| --- | --- |
| 01 | package com.javacodegeeks.examples.distributedcache; |
| 02 |  |

|  |  |
| --- | --- |
| 03 | import java.io.BufferedReader; |
| 04 | import java.io.FileReader; |

|  |  |
| --- | --- |
| 05 | import java.io.IOException; |
| 06 | import java.util.HashSet; |

|  |  |
| --- | --- |
| 07 | import java.util.Set; |
| 08 | import java.util.StringTokenizer; |

|  |  |
| --- | --- |
| 09 |  |
| 10 | import org.apache.hadoop.filecache.DistributedCache; |

|  |  |
| --- | --- |
| 11 | import org.apache.hadoop.fs.Path; |
| 12 | import org.apache.hadoop.io.IntWritable; |

|  |  |
| --- | --- |
| 13 | import org.apache.hadoop.io.LongWritable; |
| 14 | import org.apache.hadoop.io.Text; |

|  |  |
| --- | --- |
| 15 | import org.apache.hadoop.mapreduce.Mapper; |
| 16 |  |

|  |  |
| --- | --- |
| 17 | /\*\* |
| 18 | \* Map Class which extends MaReduce.Mapper class |

|  |  |
| --- | --- |
| 19 | \* Map is passed a single line at a time, it splits the line based on space |
| 20 | \* and generated the token which are output by map with value as one to be consumed |

|  |  |
| --- | --- |
| 21 | \* by reduce class |
| 22 | \* @author Raman |

|  |  |
| --- | --- |
| 23 | \*/ |
| 24 | public class MapClass extends Mapper{ |

|  |  |
| --- | --- |
| 25 |  |
| 26 | private final static IntWritable one = new IntWritable(1); |

|  |  |
| --- | --- |
| 27 | private Text word = new Text(); |
| 28 | private Set stopWords = new HashSet(); |

|  |  |
| --- | --- |
| 29 |  |
| 30 | @Override |

|  |  |
| --- | --- |
| 31 | protected void setup(Context context) throws IOException, InterruptedException { |
| 32 | try{ |

|  |  |
| --- | --- |
| 33 | Path[] stopWordsFiles = DistributedCache.getLocalCacheFiles(context.getConfiguration()); |
| 34 | if(stopWordsFiles != null && stopWordsFiles.length > 0) { |

|  |  |
| --- | --- |
| 35 | for(Path stopWordFile : stopWordsFiles) { |
| 36 | readFile(stopWordFile); |

|  |  |
| --- | --- |
| 37 | } |
| 38 | } |

|  |  |
| --- | --- |
| 39 | } catch(IOException ex) { |
| 40 | System.err.println("Exception in mapper setup: " + ex.getMessage()); |

|  |  |
| --- | --- |
| 41 | } |
| 42 | } |

|  |  |
| --- | --- |
| 43 |  |
| 44 | /\*\* |

|  |  |
| --- | --- |
| 45 | \* map function of Mapper parent class takes a line of text at a time |
| 46 | \* splits to tokens and passes to the context as word along with value as one |

|  |  |
| --- | --- |
| 47 | \*/ |
| 48 | @Override |

|  |  |
| --- | --- |
| 49 | protected void map(LongWritable key, Text value, |
| 50 | Context context) |

|  |  |
| --- | --- |
| 51 | throws IOException, InterruptedException { |
| 52 |  |

|  |  |
| --- | --- |
| 53 | String line = value.toString(); |
| 54 | StringTokenizer st = new StringTokenizer(line," "); |

|  |  |
| --- | --- |
| 55 |  |
| 56 | while(st.hasMoreTokens()){ |

|  |  |
| --- | --- |
| 57 | String wordText = st.nextToken(); |
| 58 |  |

|  |  |
| --- | --- |
| 59 | if(!stopWords.contains(wordText.toLowerCase())) { |
| 60 | word.set(wordText); |

|  |  |
| --- | --- |
| 61 | context.write(word,one); |
| 62 | } |

|  |  |
| --- | --- |
| 63 | } |
| 64 |  |

|  |  |
| --- | --- |
| 65 | } |
| 66 |  |

|  |  |
| --- | --- |
| 67 | private void readFile(Path filePath) { |
| 68 | try{ |

|  |  |
| --- | --- |
| 69 | BufferedReader bufferedReader = new BufferedReader(newFileReader(filePath.toString())); |
| 70 | String stopWord = null; |

|  |  |
| --- | --- |
| 71 | while((stopWord = bufferedReader.readLine()) != null) { |
| 72 | stopWords.add(stopWord.toLowerCase()); |

|  |  |
| --- | --- |
| 73 | } |
| 74 | } catch(IOException ex) { |

|  |  |
| --- | --- |
| 75 | System.err.println("Exception while reading stop words file: " + ex.getMessage()); |
| 76 | } |

|  |  |
| --- | --- |
| 77 | } |
| 78 | } |

Now this is where this code varies significantly from the standard word count MapReduce code. The map class contains a setup method which is the first method called when a node is setup to perform the map task.

|  |  |
| --- | --- |
| 01 | @Override |
| 02 | protected void setup(Context context) throws IOException, InterruptedException { |

|  |  |
| --- | --- |
| 03 | try{ |
| 04 | Path[] stopWordsFiles = DistributedCache.getLocalCacheFiles(context.getConfiguration()); |

|  |  |
| --- | --- |
| 05 | if(stopWordsFiles != null && stopWordsFiles.length > 0) { |
| 06 | for(Path stopWordFile : stopWordsFiles) { |

|  |  |
| --- | --- |
| 07 | readFile(stopWordFile); |
| 08 | } |

|  |  |
| --- | --- |
| 09 | } |
| 10 | } catch(IOException ex) { |

|  |  |
| --- | --- |
| 11 | System.err.println("Exception in mapper setup: " + ex.getMessage()); |
| 12 | } |

|  |  |
| --- | --- |
| 13 | } |

So this is the place where we read the file stored in the distribute cache using the DistributedCache API and getLocalCacheFiles() method as shown in the line number 4 of the above code snippet. If you notice the methods return an array of the type Path. So for each file(we have only one in this case) we will call another method called readFile()and pass the path of the file to this method.

readFile() is the method which reads the content of the file and adds the stop words in the global Set of stopWords. The details of the method are in line numebr 67-77 of the Map class.

Now in the map() method, after splitting the lines into word tokens, we will check if a particular word is present in the stop words set, if it is present we skip that word and move to the next but if it is not a stop word then we pass it on to the context to be executed in the Reduce class as shown in the code snippet below:

|  |  |
| --- | --- |
| 01 | StringTokenizer st = new StringTokenizer(line," "); |
| 02 |  |

|  |  |
| --- | --- |
| 03 | while(st.hasMoreTokens()){ |
| 04 | String wordText = st.nextToken(); |

|  |  |
| --- | --- |
| 05 |  |
| 06 | if(!stopWords.contains(wordText.toLowerCase())) { |

|  |  |
| --- | --- |
| 07 | word.set(wordText); |
| 08 | context.write(word,one); |

|  |  |
| --- | --- |
| 09 | } |
| 10 | } |

### 3.3 Reduce Class

Reduce class in this article is exactly same as it is in the standard word count example, the reduce() method will contain only those words which are not stop words and reduce will count only the good words. Following is the code of the reduce class:

|  |  |
| --- | --- |
| 01 | package com.javacodegeeks.examples.distributedcache; |
| 02 |  |

|  |  |
| --- | --- |
| 03 | import java.io.IOException; |
| 04 | import java.util.Iterator; |

|  |  |
| --- | --- |
| 05 |  |
| 06 | import org.apache.hadoop.io.IntWritable; |

|  |  |
| --- | --- |
| 07 | import org.apache.hadoop.io.Text; |
| 08 | import org.apache.hadoop.mapreduce.Reducer; |

|  |  |
| --- | --- |
| 09 |  |
| 10 | /\*\* |

|  |  |
| --- | --- |
| 11 | \* Reduce class which is executed after the map class and takes |
| 12 | \* key(word) and corresponding values, sums all the values and write the |

|  |  |
| --- | --- |
| 13 | \* word along with the corresponding total occurances in the output |
| 14 | \* |

|  |  |
| --- | --- |
| 15 | \* @author Raman |
| 16 | \*/ |

|  |  |
| --- | --- |
| 17 | public class ReduceClass extends Reducer{ |
| 18 |  |

|  |  |
| --- | --- |
| 19 | /\*\* |
| 20 | \* Method which performs the reduce operation and sums |

|  |  |
| --- | --- |
| 21 | \* all the occurrences of the word before passing it to be stored in output |
| 22 | \*/ |

|  |  |
| --- | --- |
| 23 | @Override |
| 24 | protected void reduce(Text key, Iterable values, |

|  |  |
| --- | --- |
| 25 | Context context) |
| 26 | throws IOException, InterruptedException { |

|  |  |
| --- | --- |
| 27 |  |
| 28 | int sum = 0; |

|  |  |
| --- | --- |
| 29 | Iterator valuesIt = values.iterator(); |
| 30 |  |

|  |  |
| --- | --- |
| 31 | while(valuesIt.hasNext()){ |
| 32 | sum = sum + valuesIt.next().get(); |

|  |  |
| --- | --- |
| 33 | } |
| 34 |  |

|  |  |
| --- | --- |
| 35 | context.write(key, new IntWritable(sum)); |
| 36 | } |

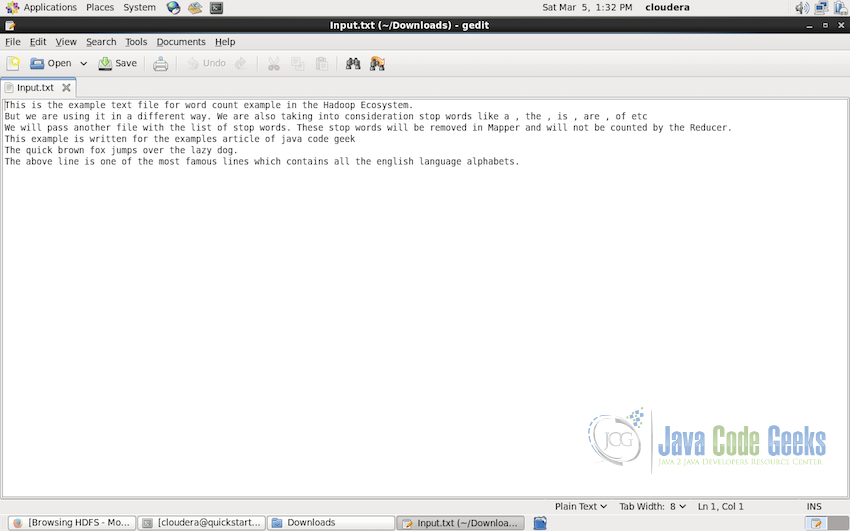
|  |  |
| --- | --- |
| 37 | } |

## **4. Executing the Hadoop Job**

We will execute the MapReduce task we discussed in the previous section on the Hadoop cluster. But before we do so, we need two files

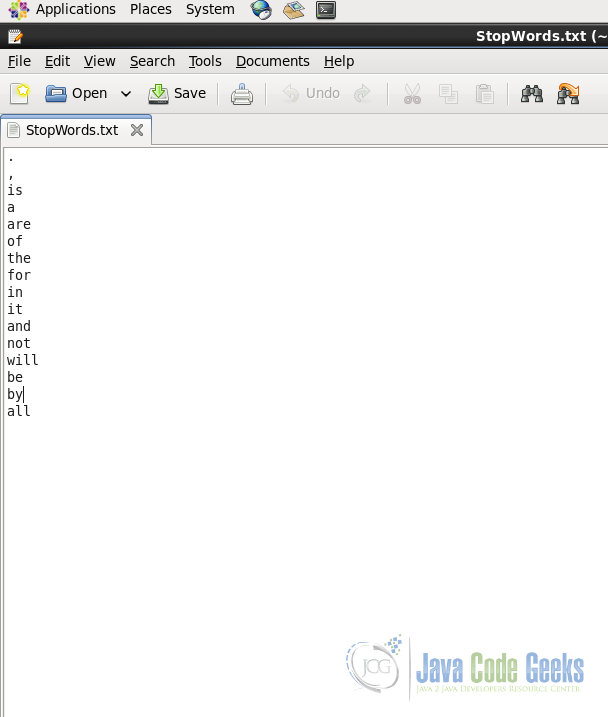
* Input file
* Stop Words file

So following is the dummy text file which we will use for the example:

[](http://examples.javacodegeeks.com/wp-content/uploads/2016/03/2.png)

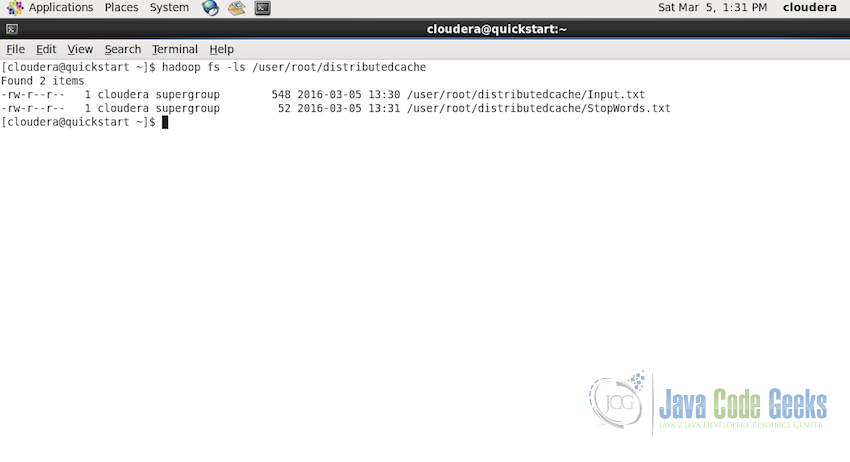
Input.txt file to be processed

and following is the file containing stop words:

[](http://examples.javacodegeeks.com/wp-content/uploads/2016/03/3.png)

StopWords.txt file containing list of stop words

Make sure that both the files are present in the Hadoop Distributed File System. If you would like to read about the basics of HDFS and Hadoop File System including how to put files in HDFS, please refer to the article [Apache Hadoop FS Commands Example](http://examples.javacodegeeks.com/enterprise-java/apache-hadoop/apache-hadoop-fs-commands-example/)

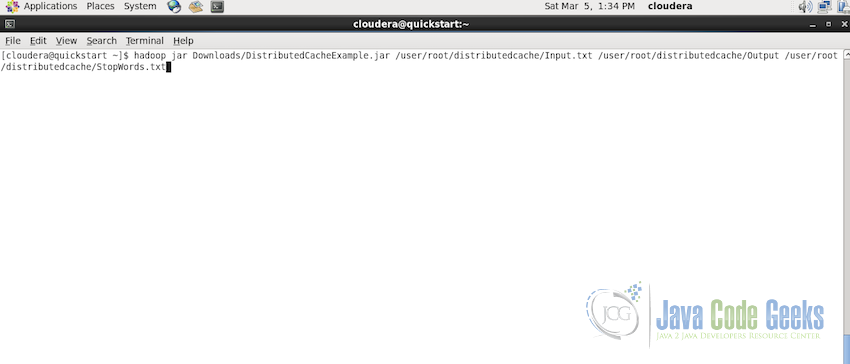
[](http://examples.javacodegeeks.com/wp-content/uploads/2016/03/1.png)

Listing the file present in HDFS

Now, to execute the Distributed Cache Example Task on the Hadoop Cluster, we have to submit the jar file along with the URLs of the input and stopwords files to the Hadoop Cluster. Following is the command to do so:

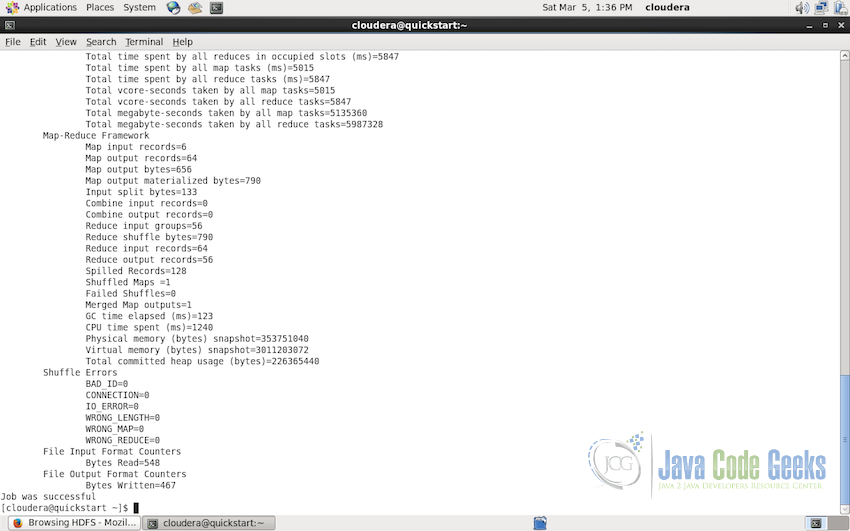
|  |  |
| --- | --- |
| 1 | hadoop jar DistributedCacheExample.jar /user/root/distributedcache/Input.txt /user/root/distributedcache/Output /user/root/distributedcache/StopWords.txt |

First argument mentions the input file to be used, second argument tells about the path where the output should be stored and the third argument tells the path of the stop words file.

[](http://examples.javacodegeeks.com/wp-content/uploads/2016/03/4.png)

Command to submit hadoop job to the cluster

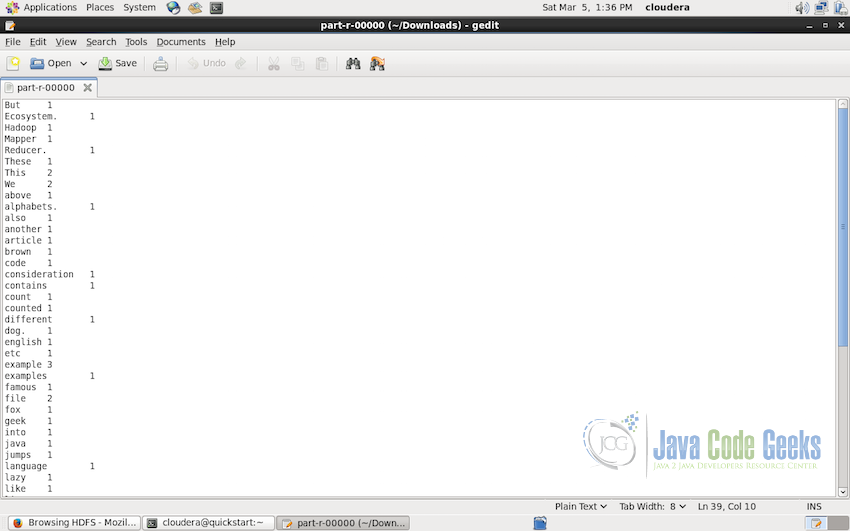
Once the job is successfully executed we will a console output something similar to:

[](http://examples.javacodegeeks.com/wp-content/uploads/2016/03/5.png)

Console Output

Notice the last line which says “Job was successful”. This is the line we printed from the Driver class on successful execution of the job. You can check the other details in the console output to know more about the job execution.

The output of the Hadoop job will be present on the HDFS path /user/root/distributedcache/Output in the Output folder as mentioned in the execution argument, this folder can be downloaded on the system from the HDFS. Following is how the output file looks like:

[](http://examples.javacodegeeks.com/wp-content/uploads/2016/03/6.png)

Output file

## **5. Conclusion**

In this example article, we talked about the Distributed Cache API of Apache Hadoop. We started with the introduction of what exactly distributed cache is and then understood the basic workflow of the distributed cache. Then we dived into the implementation section where we saw how we can use the Distributed Cache API to pass the common files, jars and other archives to the nodes executing the Hadoop Job.

# **DISTRIBUTED CACHE IN HADOOP**

[June 3, 2013](https://pravinchavan.wordpress.com/2013/06/03/distributed-cache-in-hadoop/) · by [Pravin Chavan](https://pravinchavan.wordpress.com/author/pravinchavan/) · in [Hadoop](https://pravinchavan.wordpress.com/category/hadoop/). ·

**Introduction:**  
If you are writing Map Reduce Applications, where you want some files to be shared across all nodes in Hadoop Cluster. It can be simple properties file or can be executable jar file.

Hadoop Map Reduce Project provides us this facility with something called as DistributedCache.  
This Distributed Cache is configured with Job Configuration, What it does is, it provides read only data to all machine on the cluster.

**Step 1 : Put file to HDFS**

# hdfs dfs -put /tmp/file1 /cachefile1

**Step 2: Add cachefile in Job Configuration**

Configuration conf = new Configuration();

Job job = new Job(conf, "wordcount");

DistributedCache.addCacheFile(new URI("/cachefile1"),job.getConfiguration());

**Step 3: Access Cached file**

Path[] cacheFiles = context.getLocalCacheFiles();

FileInputStream fileStream = new FileInputStream(cacheFiles[0].toString());

I,

Welcome all , today i am posting an example for MAP-SIDE JOIN using mapreduce.

we will see Hadoop map side join with Distributed Cache below

first of all we need to prepare data samples for this map side join

**Note:** we need to remember one thing is that we always put the small file in the distributed cache and send large file to map phase.

**Table 1 (Always Small Table):**

Department table with deptno and deptname(tab seperated)

001    hadoop  
002    hive  
003    pig  
004    sqoop  
005    oozie  
006    mapreduce

**Table 2(Always LargeTable):**

Employee table with Employee info(tab separated)

 empid    empname    salary    deptid  
1    name    2000    001  
2    name2    4000    002  
3    name3    5000    005  
4    name4    7000    003  
5    name5    8000    004

**Desired Output**

1    name    2000    001    hadoop  
2    name2    4000    002    hive  
3    name3    5000    005    oozie  
4    name4    7000    003    pig  
5    name5    8000    004    sqoop

**To get the above output we need to join employee table with dept table so that we will get the dept name from the dept table where we join deptID in employee table with dept table.**

## **MapJoinDriver .java**

[code]

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
\*Driver  
\*MapJoinDriver  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

package com.hadoop.mapjoin;

import java.net.URI;

import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.conf.Configured;  
import org.apache.hadoop.filecache.DistributedCache;  
import org.apache.hadoop.fs.Path;  
import org.apache.hadoop.mapreduce.Job;  
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  
import org.apache.hadoop.util.Tool;  
import org.apache.hadoop.util.ToolRunner;

public class MapJoinDriver extends Configured implements Tool {

@Override  
public int run(String[] args) throws Exception {

if (args.length != 2) {  
System.out.printf(“Two parameters are required- <input dir> <output dir>n”);  
return -1;  
}

Job job = new Job(getConf());  
Configuration conf = job.getConfiguration();  
job.setJobName(“Map-side join with text lookup file in DCache”);  
DistributedCache.addCacheFile(new URI(“/user/hadoop/joinProject/data/departments\_txt”),conf);

job.setJarByClass(MapJoinDriver.class);  
FileInputFormat.setInputPaths(job, new Path(args[0]));  
FileOutputFormat.setOutputPath(job, new Path(args[1]));  
job.setMapperClass(MapJoinDistributedCacheMapper.class);

job.setNumReduceTasks(0);

boolean success = job.waitForCompletion(true);  
return success ? 0 : 1;  
}

public static void main(String[] args) throws Exception {  
int exitCode = ToolRunner.run(new Configuration(),new MapJoinDriver(), args);  
System.exit(exitCode);  
}  
}  
[/code]

## **MapJoinDistributedCacheMapper.java**

[code]

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
\*Mapper  
\*MapJoinDistributedCacheMapper  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

package com.hadoop.mapjoin;

import java.io.BufferedReader;  
import java.io.FileNotFoundException;  
import java.io.FileReader;  
import java.io.IOException;  
import java.util.HashMap;

import org.apache.hadoop.filecache.DistributedCache;  
import org.apache.hadoop.fs.Path;  
import org.apache.hadoop.io.LongWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Mapper;

public class MapJoinDistributedCacheMapper extends Mapper<LongWritable, Text, Text, Text> {

private static HashMap<String, String> DepartmentMap = new HashMap<String, String>();  
private BufferedReader brReader;  
private String strDeptName = “”;  
private Text txtMapOutputKey = new Text(“”);  
private Text txtMapOutputValue = new Text(“”);

enum MYCOUNTER {  
RECORD\_COUNT, FILE\_EXISTS, FILE\_NOT\_FOUND, SOME\_OTHER\_ERROR  
}

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

Path[] cacheFilesLocal = DistributedCache.getLocalCacheFiles(context.getConfiguration());

for (Path eachPath : cacheFilesLocal) {  
if (eachPath.getName().toString().trim().equals(“departments\_txt”)) {  
context.getCounter(MYCOUNTER.FILE\_EXISTS).increment(1);  
loadDepartmentsHashMap(eachPath, context);  
}  
}

}

private void loadDepartmentsHashMap(Path filePath, Context context) throws IOException {

String strLineRead = “”;

try {  
brReader = new BufferedReader(new FileReader(filePath.toString()));

// Read each line, split and load to HashMap  
while ((strLineRead = brReader.readLine()) != null) {  
String deptFieldArray[] = strLineRead.split(“t”);  
DepartmentMap.put(deptFieldArray[0].trim(),    deptFieldArray[1].trim());  
}  
} catch (FileNotFoundException e) {  
e.printStackTrace();  
context.getCounter(MYCOUNTER.FILE\_NOT\_FOUND).increment(1);  
} catch (IOException e) {  
context.getCounter(MYCOUNTER.SOME\_OTHER\_ERROR).increment(1);  
e.printStackTrace();  
}finally {  
if (brReader != null) {  
brReader.close();  
}  
}  
}

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

context.getCounter(MYCOUNTER.RECORD\_COUNT).increment(1);

if (value.toString().length() > 0) {  
String arrEmpAttributes[] = value.toString().split(“t”);

try {  
strDeptName = DepartmentMap.get(arrEmpAttributes[3].toString());  
} finally {  
strDeptName = ((strDeptName.equals(null) || strDeptName    .equals(“”)) ? “NOT-FOUND” : strDeptName);  
}

txtMapOutputKey.set(arrEmpAttributes[0].toString());

txtMapOutputValue.set(arrEmpAttributes[0].toString() + “t”  
+ arrEmpAttributes[1].toString() + “t”  
+ arrEmpAttributes[2].toString() + “t”  
+ arrEmpAttributes[3].toString() + “t” + strDeptName);

}  
context.write(txtMapOutputKey, txtMapOutputValue);  
strDeptName = “”;  
}  
}

[/code]

1. File permissions

Answer)

## chmod

Usage: hdfs dfs -chmod [-R] <MODE[,MODE]... | OCTALMODE> URI [URI ...]

Change the permissions of files. With -R, make the change recursively through the directory structure. The user must be the owner of the file, or else a super-user. Additional information is in the [Permissions Guide](http://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-hdfs/HdfsPermissionsGuide.html).

Options

* The -R option will make the change recursively through the directory structure.
* **chmod:** Changes the permissions of files. With -R, makes the change recursively by way of the directory structure. The user must be the file owner or the superuser
* *Usage:* hdfs dfs -chmod [-R] <MODE[,MODE]… | OCTALMODE> URI [URI …]
* *Example:* hdfs dfs -chmod 777 test/data1.txt

1. Data compression

Answer)

Hadoop For Dummies

|  |  |  |
| --- | --- | --- |
| **Codec** | **File Extension** | **Splittable?** |
| Gzip | .gz | No |
| Bzip2 | .bz2 | Yes |
| Snappy | .snappy | No |
| LZO | .lzo | No, unless indexed |

File compression brings two major benefits: it reduces the space needed to store files, and it speeds up data transfer across the network or to or from disk. When dealing with large volumes of data, both of these savings can be significant, so it pays to carefully consider how to use compression in Hadoop.

## 1. What to compress?

**1) Compressing input files**  
If the input file is compressed, then the bytes read in from HDFS is reduced, which means less time to read data. This time conservation is beneficial to the performance of job execution.  
  
If the input files are compressed, they will be decompressed automatically as they are read by MapReduce, using the filename extension to determine which codec to use. For example, a file ending in .gz can be identified as gzip-compressed file and thus read with GzipCodec.  
  
**2) Compressing output files**  
Often we need to store the output as history files. If the amount of output per day is extensive, and we often need to store history results for future use, then these accumulated results will take extensive amount of HDFS space. However, these history files may not be used very frequently, resulting in a waste of HDFS space. Therefore, it is necessary to compress the output before storing on HDFS.   
  
**3) Compressing map output**  
Even if your MapReduce application reads and writes uncompressed data, it may benefit from compressing the intermediate output of the map phase. Since the map output is written to disk and transferred across the network to the reducer nodes, by using a fast compressor such as LZO or Snappy, you can get performance gains simply because the volume of data to transfer is reduced.

## 2. Common input format

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Compression format | Tool | Algorithm | File extention | Splittable |
| gzip | *gzip* | DEFLATE | .gz | No |
| bzip2 | *bizp2* | bzip2 | .bz2 | Yes |
| LZO | *lzop* | LZO | .lzo | Yes if indexed |
| Snappy | N/A | Snappy | .snappy | No |

**gzip:**   
gzip is naturally supported by Hadoop. gzip is based on the DEFLATE algorithm, which is a combination of LZ77 and Huffman Coding.  
  
**bzip2:**   
bzip2 is a freely available, patent free (see below), high-quality data compressor. It typically compresses files to within 10% to 15% of the best available techniques (the PPM family of statistical compressors), whilst being around twice as fast at compression and six times faster at decompression.  
  
**LZO:**   
The LZO compression format is composed of many smaller (~256K) blocks of compressed data, allowing jobs to be split along block boundaries.  Moreover, it was designed with speed in mind: it decompresses about twice as fast as gzip, meaning it’s fast enough to keep up with hard drive read speeds.  It doesn’t compress quite as well as gzip — expect files that are on the order of 50% larger than their gzipped version.  But that is still 20-50% of the size of the files without any compression at all, which means that IO-bound jobs complete the map phase about four times faster.  
  
**Snappy:**   
Snappy is a compression/decompression library. It does not aim for maximum compression, or compatibility with any other compression library; instead, it aims for very high speeds and reasonable compression. For instance, compared to the fastest mode of zlib, Snappy is an order of magnitude faster for most inputs, but the resulting compressed files are anywhere from 20% to 100% bigger. On a single core of a Core i7 processor in 64-bit mode, Snappy compresses at about 250 MB/sec or more and decompresses at about 500 MB/sec or more. Snappy is widely used inside Google, in everything from BigTable and MapReduce to our internal RPC systems.  
  
**Some tradeoffs:**  
All compression algorithms exhibit a space/time trade-off: faster compression and decompression speeds usually come at the expense of smaller space savings. The tools listed in above table typically give some control over this trade-off at compression time by offering nine different options: –1 means optimize for speed and -9 means optimize for space.  
  
The different tools have very different compression characteristics. Gzip is a general purpose compressor, and sits in the middle of the space/time trade-off. Bzip2 compresses more effectively than gzip, but is slower. Bzip2’s decompression speed is faster than its compression speed, but it is still slower than the other formats. LZO and Snappy, on the other hand, both optimize for speed and are around an order of magnitude faster than gzip, but compress less effectively. Snappy is also significantly faster than LZO for decompression.

## 3. Issues about compression and input split

When considering how to compress data that will be processed by MapReduce, it is important to understand whether the compression format supports splitting. Consider an uncompressed file stored in HDFS whose size is 1 GB. With an HDFS block size of 64 MB, the file will be stored as 16 blocks, and a MapReduce job using this file as input will create 16 input splits, each processed independently as input to a separate map task.   
  
Imagine now the file is a gzip-compressed file whose compressed size is 1 GB. As before, HDFS will store the file as 16 blocks. However, creating a split for each block won’t work since it is impossible to start reading at an arbitrary point in the gzip stream and therefore impossible for a map task to read its split independently of the others. The gzip format uses DEFLATE to store the compressed data, and DEFLATE stores data as a series of compressed blocks. The problem is that the start of each block is not distinguished in any way that would allow a reader positioned at an arbitrary point in the stream to advance to the beginning of the next block, thereby synchronizing itself with the stream. For this reason, gzip does not support splitting.  
  
In this case, MapReduce will do the right thing and not try to split the gzipped file, since it knows that the input is gzip-compressed (by looking at the filename extension) and that gzip does not support splitting. This will work, but at the expense of locality: a single map will process the 16 HDFS blocks, most of which will not be local to the map. Also, with fewer maps, the job is less granular, and so may take longer to run.  
  
If the file in our hypothetical example were an LZO file, we would have the same problem since the underlying compression format does not provide a way for a reader to synchronize itself with the stream. However, it is possible to preprocess LZO files using an indexer tool that comes with the Hadoop LZO libraries. The tool builds an index of split points, effectively making them splittable when the appropriate MapReduce input format is used.  
  
A bzip2 file, on the other hand, does provide a synchronization marker between blocks (a 48-bit approximation of pi), so it does support splitting.

## 4. IO-bound and CPU bound

Storing compressed data in HDFS allows your hardware allocation to go further since compressed data is often 25% of the size of the original data.  Furthermore, since MapReduce jobs are nearly always IO-bound, storing compressed data means there is less overall IO to do, meaning jobs run faster.  There are two caveats to this, however: some compression formats cannot be split for parallel processing, and others are slow enough at decompression that jobs become CPU-bound, eliminating your gains on IO.  
  
The gzip compression format illustrates the first caveat. Imagine you have a 1.1 GB gzip file, and your cluster has a 128 MB block size.  This file will be split into 9 chunks of size approximately 128 MB.  In order to process these in parallel in a MapReduce job, a different mapper will be responsible for each chunk. But this means that the second mapper will start on an arbitrary byte about 128MB into the file.  The contextful dictionary that gzip uses to decompress input will be empty at this point, which means the gzip decompressor will not be able to correctly interpret the bytes.  The upshot is that large gzip files in Hadoop need to be processed by a single mapper, which defeats the purpose of parallelism.    
  
Bzip2 compression format illustrates the second caveat in which jobs become CPU-bound. Bzip2 files compress well and are even splittable, but the decompression algorithm is slow and cannot keep up with the streaming disk reads that are common in Hadoop jobs.  While Bzip2 compression has some upside because it conserves storage space, running jobs now spend their time waiting on the CPU to finish decompressing data, which slows them down and offsets the other gains.

## 5. Summary

Reasons to compress:  
a) Data is mostly stored and not frequently processed. It is usual DWH scenario. In this case space saving can be much more significant then processing overhead   
b) Compression factor is very high and thereof we save a lot of IO.   
c) Decompression is very fast (like Snappy) and thereof we have a some gain with little price   
d) Data already arrived compressed  
  
Reasons not to compress  
a) Compressed data is not splittable. Have to be noted that many modern format are built with block level compression to enable splitting and other partial processing of the files. b) Data is created in the cluster and compression takes significant time. Have to be noted that compression usually much more CPU intensive then decompression.   
c) Data has little redundancy and compression gives little gain.

1. Mapreduce optimization

Answer: -

# **7 Tips for Improving MapReduce Performance**

[December 17, 2009](https://blog.cloudera.com/blog/2009/12/7-tips-for-improving-mapreduce-performance/)

[By Todd Lipcon](https://blog.cloudera.com/blog/author/todd/) [(@tlipcon)](https://twitter.com/@tlipcon)

[14 Comments](https://blog.cloudera.com/blog/2009/12/7-tips-for-improving-mapreduce-performance/#comments)

Categories: [General](https://blog.cloudera.com/blog/category/general/)

One service that Cloudera provides for our customers is help with tuning and optimizing MapReduce jobs. Since MapReduce and HDFS are complex distributed systems that run arbitrary user code, there’s no hard and fast set of rules to achieve optimal performance; instead, I tend to think of tuning a cluster or job much like a doctor would treat a sick human being. There are a number of key symptoms to look for, and each set of symptoms leads to a different diagnosis and course of treatment.

In medicine, there’s no automatic process that can replace the experience of a well seasoned doctor. The same is true with complex distributed systems — experienced users and operators often develop a “sixth sense” for common issues. Having worked with Cloudera customers in a number of different industries, each with a different workload, dataset, and cluster hardware, I’ve accumulated a bit of this experience, and would like to share some with you today.

In this blog post, I’ll highlight a few tips for improving MapReduce performance. The first few tips are cluster-wide, and will be useful for operators and developers alike. The latter tips are for developers writing custom MapReduce jobs in Java. For each tip, I’ll also note a few of the “symptoms” or “diagnostic tests” that indicate a particular remedy might bring you some good improvements.

Please note, also, that these tips contain lots of rules of thumb based on my experience across a variety of situations. They may not apply to your particular workload, dataset, or cluster, and you should always benchmark your jobs before and after any changes. For these tips, I’ll show some comparative numbers for a 40GB wordcount job on a small 4-node cluster. Tuned optimally, each of the map tasks in this job runs in about 33 seconds, and the total job runtime is about 8m30s.

## **Tip 1) Configure your cluster correctly**

**Diagnostics/symptoms:**

* top shows slave nodes fairly idle even when all map and reduce task slots are filled up running jobs.
* top shows kernel processes like RAID (mdX\_raid\*) or pdflush taking most of the CPU time.
* Linux load averages are often seen more than twice the number of CPUs on the system.
* Linux load averages stay less than half the number of CPUs on the system, even when running jobs.
* Any swap usage on nodes beyond a few MB.

The first step to optimizing your MapReduce performance is to make sure your cluster configuration has been tuned. For starters, check out our earlier [blog post on configuration parameters](http://blog.cloudera.com/blog/2009/03/30/configuration-parameters-what-can-you-just-ignore/). In addition to those knobs in the Hadoop configuration, here are a few more checklist items you should go through before beginning to tune the performance of an individual job:

* Make sure the mounts you’re using for DFS and MapReduce storage have been mounted with the noatime option. This disables access time tracking and can improve IO performance.
* Avoid RAID and LVM on TaskTracker and DataNode machines – it generally reduces performance.
* Make sure you’ve configured mapred.local.dir and dfs.data.dir to point to one directory on each of your disks to ensure that all of your IO capacity is used. Run iostat -dx 5 from the sysstat package while the cluster is loaded to make sure each disk shows utilization.
* Ensure that you have [SMART](http://en.wikipedia.org/wiki/S.M.A.R.T.) monitoring for the health status of your disk drives. MapReduce jobs are fault tolerant, but dying disks can cause performance to degrade as tasks must be re-executed. If you find that a particular TaskTracker becomes blacklisted on many job invocations, it may have a failing drive.
* Monitor and graph swap usage and network usage with software like [Ganglia](http://ganglia.info/). [Monitoring Hadoop metrics](http://blog.cloudera.com/blog/2009/03/12/hadoop-metrics/) in Ganglia is also a good idea. If you see swap being used, reduce the amount of RAM allocated to each task in mapred.child.java.opts.

**Benchmarks:**

Unfortunately I was not able to perform benchmarks for this tip, as it would involve re-imaging the cluster. If you have had relevant experience, feel free to leave a note in the Comments section below.

## **Tip 2) Use LZO Compression**

**Diagnostics/symptoms:**

* This is almost always a good idea for intermediate data! In the doctor analogy, consider LZO compression your vitamins.
* Output data size of MapReduce job is nontrivial.
* Slave nodes show high iowait utilization in top and iostat when jobs are running.

Almost every Hadoop job that generates an non-negligible amount of map output will benefit from intermediate data compression with LZO. Although LZO adds a little bit of CPU overhead, the reduced amount of disk IO during the shuffle will usually save time overall.

Whenever a job needs to output a significant amount of data, LZO compression can also increase performance on the output side. Since writes are replicated 3x by default, each GB of output data you save will save 3GB of disk writes.  
In order to enable LZO compression, check out our [recent guest blog from Twitter](http://blog.cloudera.com/blog/2009/11/17/hadoop-at-twitter-part-1-splittable-lzo-compression/). Be sure to set mapred.compress.map.output to true.

**Benchmarks:**

Disabling LZO compression on the wordcount example increased the job runtime only slightly on our cluster. The FILE\_BYTES\_WRITTEN counter increased from 3.5GB to 9.2GB, showing that the compression yielded a 62% decrease in disk IO. Since this job was not sharing the cluster, and each node has a high ratio of number of disks to number of tasks, IO is not the bottleneck here, and thus the improvement was not substantial. On clusters where disks are pegged due to a lot of concurrent activity, a 60% reduction in IO can yield a substantial improvement in job completion speed.

## **Tip 3) Tune the number of map and reduce tasks appropriately**

**Diagnostics/symptoms:**

* Each map or reduce task finishes in less than 30-40 seconds.
* A large job does not utilize all available slots in the cluster.
* After most mappers or reducers are scheduled, one or two remains pending and then runs all alone.

Tuning the number of map and reduce tasks for a job is important and easy to overlook. Here are some rules of thumb I use to set these parameters:

* If each task takes less than 30-40 seconds, reduce the number of tasks. The task setup and scheduling overhead is a few seconds, so if tasks finish very quickly, you’re wasting time while not doing work. JVM reuse can also be enabled to solve this problem.
* If a job has more than 1TB of input, consider increasing the block size of the input dataset to 256M or even 512M so that the number of tasks will be smaller. You can change the block size of existing files with a command like hadoop distcp -Ddfs.block.size=$[256\*1024\*1024] /path/to/inputdata /path/to/inputdata-with-largeblocks. After this command completes, you can remove the original data.
* So long as each task runs for at least 30-40 seconds, increase the number of mapper tasks to some multiple of the number of mapper slots in the cluster. If you have 100 map slots in your cluster, try to avoid having a job with 101 mappers – the first 100 will finish at the same time, and then the 101st will have to run alone before the reducers can run. This is more important on small clusters and small jobs.
* Don’t schedule too many reduce tasks – for most jobs, we recommend a number of reduce tasks equal to or a bit less than the number of reduce slots in the cluster.

**Benchmarks:**

To make the wordcount job run with too many tasks, I ran it with the argument -Dmapred.max.split.size=$[16\*1024\*1024]. This yielded 2640 tasks instead of the 360 that the framework chose by default. When running with this setting, each task took about 9 seconds, and watching the Cluster Summary view on the JobTracker showed the number of running maps fluctuating between 0 and 24 continuously throughout the job. The entire job finished in 17m52s, more than twice as slow as the original job.

## **Tip 4) Write a Combiner**

**Diagnostics/symptoms:**

* A job performs aggregation of some sort, and the Reduce input groups counter is significantly smaller than the Reduce input records counter.
* The job performs a large shuffle (e.g. map output bytes is multiple GB per node)
* The number of spilled records is many times larger than the number of map output records as seen in the Job counters.

If your algorithm involves computing aggregates of any sort, chances are you can use a Combiner in order to perform some kind of initial aggregation before the data hits the reducer. The MapReduce framework runs combiners intelligently in order to reduce the amount of data that has to be written to disk and transfered over the network in between the Map and Reduce stages of computation.

**Benchmarks:**

I modified the word count example to remove the call to setCombinerClass, and otherwise left it the same. This changed the average map task run time from 33s to 48s, and increased the amount of shuffled data from 1GB to 1.4GB. The total job runtime increased from 8m30s to 15m42s, nearly a factor of two. Note that this benchmark was run with map output compression enabled – without map output compression, the effect of the combiner would have been even more important.

## **Tip 5) Use the most appropriate and compact Writable type for your data**

**Symptoms/diagnostics:**

* Text objects are used for working with non-textual or complex data
* IntWritable or LongWritable objects are used when most output values tend to be significantly smaller than the maximum value.

When users are new to programming in MapReduce, or are switching from Hadoop Streaming to Java MapReduce, they often use the [Text](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/Text.html) writable type unnecessarily. Although Text can be convenient, converting numeric data to and from UTF8 strings is inefficient and can actually make up a significant portion of CPU time. Whenever dealing with non-textual data, consider using the binary Writables like [IntWritable](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/IntWritable.html), [FloatWritable](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/FloatWritable.html), etc.

In addition to avoiding the text parsing overhead, the binary Writable types will take up less space as intermediate data. Since disk IO and network transfer will become a bottleneck in large jobs, reducing the sheer number of bytes taken up by the intermediate data can provide a substantial performance gain. When dealing with integers, it can also sometimes be faster to use [VIntWritable](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/VIntWritable.html) or [VLongWritable](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/VLongWritable.html) — these implement variable-length integer encoding which saves space when serializing small integers. For example, the value 4 will be serialized in a single byte, whereas the value 10000 will be serialized in two. These variable length numbers can be very effective for data like counts, where you expect that the majority of records will have a small number that fits in one or two bytes.

If the Writable types that ship with Hadoop don’t fit the bill, consider writing your own. It’s pretty simple, and will be significantly faster than parsing text. If you do so, make sure to provide a [RawComparator](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/RawComparator.html) — see the source code for the built in Writables for an example.

Along the same vein, if your MapReduce job is part of a multistage workflow, use a binary format like [SequenceFile](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/SequenceFile.html) for the intermediate steps, even if the last stage needs to output text. This will reduce the amount of data that needs to be materialized along the way.

**Benchmarks:**

For the example word count job, I modified the intermediate count values to be Text type rather than IntWritable. In the reducer, I used Integer.parseString(value.toString()) when accumulating the sum. The performance of the suboptimal version of the WordCount was about 10% slower than the original. The full job ran in a bit over 9 minutes, and each map task took 36 seconds instead of the original 33. Since integer parsing is itself rather fast, this did not represent a large improvement; in the general case, I have seen using more efficient Writables to make as much as a 2-3x difference in performance.

## **Tip 6) Reuse Writables**

**Symptoms/diagnostics:**

* Add -verbose:gc -XX:+PrintGCDetails to mapred.child.java.opts. Then inspect the logs for some tasks. If garbage collection is frequent and represents a lot of time, you may be allocating unnecessary objects.
* grep for “new Text” or “new IntWritable” in your code base. If you find this in an inner loop, or inside the map or reduce functions this tip may help.
* This tip is especially helpful when your tasks are constrained in RAM.

One of the first mistakes that many MapReduce users make is to allocate a new Writable object for every output from a mapper or reducer. For example, one might implement a word-count mapper like this:



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public void map(...) {    ...    for (String word : words) {      output.collect(new Text(word), new IntWritable(1));    }  } |

This implementation causes thousands of very short-lived objects to be allocated. While the Java garbage collector does a reasonable job at dealing with this, it is more efficient to write:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class MyMapper ... {  &#160;Text wordText = new Text();    IntWritable one = new IntWritable(1);    public void map(...) {      ...      for (String word : words) {        wordText.set(word);        output.collect(word, one);      }    }  } |

**Benchmarks:**

When I modified the word count example as described above, I initially found it made no difference in the run time of the job. This is because this cluster’s default settings include a 1GB heap size for each task, so garbage collection never ran. However, running it with each task allocated only 200mb of heap size showed a drastic slowdown in the version that did not reuse Writables — the total job runtime increased from around 8m30s to over 17 minutes. The original version, which does reuse Writables, stayed the same speed even with the smaller heap. Since reusing Writables is an easy fix, I recommend always doing so – it may not bring you a gain for every job, but if you’re low on memory it can make a huge difference.

## **Tip 7) Use “Poor Man’s Profiling” to see what your tasks are doing**

This is a trick I almost always use when first looking at the performance of a MapReduce job. Profiling purists will disagree and say that this won’t work, but you can’t argue with results!

In order to do what I call “poor man’s profiling”, ssh into one of your slave nodes while some tasks from a slow job are running. Then simply run sudo killall -QUIT java 5-10 times in a row, each a few seconds apart. Don’t worry — this doesn’t cause anything to quit, despite the name. Then, use the JobTracker interface to navigate to the stdout logs for one of the tasks that’s running on this node, or look in /var/log/hadoop/userlogs/ for a stdout file of a task that is currently running. You’ll see stack trace output from each time you sent the SIGQUIT signal to the JVM.

It takes a bit of experience to parse this output, but here’s the method I usually use:

1. For each thread in the trace, quickly scan for the name of your Java package (e.g. com.mycompany.mrjobs). If you don’t see any lines in the trace that are part of your code, skip over this thread.
2. When you find a stack trace that has some of your code in it, make a quick mental note what it’s doing. For example, “something NumberFormat-related” is all you need at this point. Don’t worry about specific line numbers yet.
3. Go down to the next dump you took a few seconds later in the logs. Perform the same process here and make a note.
4. After you’ve gone through 4-5 of the traces, you might notice that the same vague thing shows up in every one of them. If that thing is something that you expect to be fast, you probably found your culprit. If you take 10 traces, and 5 of them show NumberFormat in the dump, it means that you’re spending somewhere around 50% of your CPU time formatting numbers, and you might consider doing something differently.

Sure, this method isn’t as scientific as using a real profiler on your tasks, but I’ve found that it’s a surefire way to notice any glaring CPU bottlenecks very quickly and with no setup involved. It’s also a technique that you’ll get better at with practice as you learn what a normal dump looks like and when something jumps out as odd.

Here are a few performance mistakes I often find through this technique:

* NumberFormat is slow – avoid it where possible.
* String.split, as well as encoding or decoding UTF8 are slower than you think – see above tips about using the appropriate Writables
* Concatenating Strings rather than using StringBuffer.append

These are just a few tips for improving MapReduce performance. If you have your own tips and tricks for profiling and optimizing MapReduce jobs, please leave a comment below! If you’d like to look at the code I used for running the benchmarks, I’ve put it online at <http://github.com/toddlipcon/performance-blog-code/>

1. How to compress output in mapreduce program

Answer)

|  |  |  |
| --- | --- | --- |
| ***To enable MapReduce intermediate compression:*** | |  |
| mapred.compress.map.output=true | mapreduce.map.output.compress=true | Should the outputs of the maps be compressed before being sent across the network. Uses SequenceFile compression. |
| mapred.map.output.compression.codec= org.apache.hadoop.io.compress.SnappyCodec | mapreduce.map.output.compress.codec= org.apache.hadoop.io.compress.SnappyCodec | If the map outputs are compressed, how should they be compressed? (i.e. Snappy) |
| ***To compress the final output of a MapReduce job:*** | |  |
| mapred.output.compress=true | mapreduce.output.fileoutputformat.compress=true | Should the job outputs be compressed? |
| mapred.output.compression.type=BLOCK | mapreduce.output.fileoutputformat.compress.type=BLOCK | If the job outputs are to compressed as SequenceFiles, how should they be compressed? Should be one of NONE, RECORD or BLOCK. |
| mapred.output.compression.codec= org.apache.hadoop.io.compress.GzipCodec | mapreduce.output.fileoutputformat.compress.codec=  org.apache.hadoop.io.compress.GzipCodec | If the job outputs are compressed, how should they be compressed? (i.e. Gzip) |
| io.compression.codecs=  org.apache.hadoop.io.compress.DefaultCodec,org.apache.hadoop.io.compress.GzipCodec,  org.apache.hadoop.io.compress.BZip2Codec,org.apache.hadoop.io.compress.SnappyCodec | |  |

As described in the introduction section, if the input files are compressed, they will be decompressed automatically as they are read by MapReduce, using the filename extension to determine which codec to use. This is input compression.  
  
Here we list some code for setting up output compression in Hadoop for some common compression formats.

**Gzip**  
For final output, we can use the static convenience methos on FileOutputFormat to set the properties.  
  
FileOutputFormat.setCompressOutput(job, true);  
FileOutputFormat.setOutputCompressorClass(job, GzipCodec,class);  
  
For map output  
  
Configuration conf = new Configuration();  
conf.setBoolean("mapred.compress.map.output",true);  
conf.setClass("mapred.map.output.compression.codec", GzipCodec.class, CompressionCodec.class);  
Job job=new Job(conf);

**LZO**  
For final output  
  
FileOutputFormat.setCompressOutput(conf, true);  
FileOutputFormat.setOutputCompressorClass(conf, LzoCodec.class);  
  
In addition, to make LZO splittable, we need to make a LZO index file.

**Snappy**  
For final output  
  
conf.setOutputFormat(SequenceFileOutputFormat.class);   
SequenceFileOutputFormat.setOutputCompressionType(conf, CompressionType.BLOCK);  
SequenceFileOutputFormat.setCompressOutput(conf, true); conf.set("mapred.output.compression.codec","org.apache.hadoop.io.compress.SnappyCodec");  
  
For map output  
  
Configuration conf = new Configuration();   
conf.setBoolean("mapred.compress.map.output", true); conf.set("mapred.map.output.compression.codec","org.apache.hadoop.io.compress.SnappyCodec");

### Compressing Hadoop Output usinig Gzip and Lzo

In most of the cases, writing out output files in compressed format is faster - less amount of data will be written. To have a faster computation, compression algorithm should perform well - so time is saved even though there is an extra compression time overhead.  
  
Compressing regular output formats with Gzip, use:

[?](http://yaseminavcular.blogspot.com/2011/04/compressing-hadoop-output-file.html)

|  |  |
| --- | --- |
| 1  2  3  4 | job.setOutputFormatClass(TextOutputFormat.class);  TextOutputFormat.setCompressOutput(job, true);  TextOutputFormat.setOutputCompressorClass(job, GzipCodec.class);  ... |

For Lzo Output compression, download this [package](https://github.com/kevinweil/hadoop-lzo) by [@kevinweil](http://www.twitter.com/kevinweil). Then following should work:

[?](http://yaseminavcular.blogspot.com/2011/04/compressing-hadoop-output-file.html)

|  |  |
| --- | --- |
| 1  2  3  4 | job.setOutputFormatClass(TextOutputFormat.class);  TextOutputFormat.setCompressOutput(job, true);  TextOutputFormat.setOutputCompressorClass(job, LzoCodec.class);  ... |

In terms of space efficiency, Gzip compresses better. However, in terms of time Lzo i smuch faster. Also, it is possible to split Lzo files, splittable Gzip is not available.   
Keep in mind that these two techniques will only compress the final outputs of a Hadoop job. To be able to compress intermediate data, parameters in mapred-site.xml should be [configured](http://yaseminavcular.blogspot.com/2011/04/hadoop-intermediate-data-compression.html).

## Chapter 4. Configuring HDFS Compression

This section describes how to configure HDFS compression on Linux.

Linux supports GzipCodec, DefaultCodec, BZip2Codec, LzoCodec, and SnappyCodec. Typically, GzipCodec is used for HDFS compression. Use the following instructions to use GZipCodec.

* **Option I:** To use GzipCodec with a one-time only job:

hadoop jar hadoop-examples-1.1.0-SNAPSHOT.jar sort sbr"-Dmapred.compress.map.output=true" sbr"-Dmapred.map.output.compression.codec=org.apache.hadoop.io.compress.GzipCodec"sbr "-Dmapred.output.compress=true" sbr"-Dmapred.output.compression.codec=org.apache.hadoop.io.compress.GzipCodec"sbr -outKey org.apache.hadoop.io.Textsbr -outValue org.apache.hadoop.io.Text input output

* **Option II:** To enable GzipCodec as the default compression:
  + Edit the core-site.xml file on the NameNode host machine:
  + <property>
  + <name>io.compression.codecs</name>
  + <value>org.apache.hadoop.io.compress.GzipCodec,
  + org.apache.hadoop.io.compress.DefaultCodec,com.hadoop.compression.lzo.LzoCodec,
  + org.apache.hadoop.io.compress.SnappyCodec</value>
  + <description>A list of the compression codec classes that can be used
  + for compression/decompression.</description>

</property>

* + Edit the mapred-site.xml file on the JobTracker host machine:
  + <property>
  + <name>mapred.compress.map.output</name>
  + <value>true</value>
  + </property>
  + <property>
  + <name>mapred.map.output.compression.codec</name>
  + <value>org.apache.hadoop.io.compress.GzipCodec</value>
  + </property>
  + <property>
  + <name>mapred.output.compression.type</name>
  + <value>BLOCK</value>

</property>

* + (Optional) - Enable the following two configuration parameters to enable job output compression. Edit the mapred-site.xml file on the Resource Manager host machine:
  + <property>
  + <name>mapred.output.compress</name>
  + <value>true</value>
  + </property>
  + <property>
  + <name>mapred.output.compression.codec</name>
  + <value>org.apache.hadoop.io.compress.GzipCodec</value>

</property>

In Mapreduce we need to use below

[view plainprint?](http://www.expertsfollow.com/hadoop/how-to-compress-mapper-output-but-not-reducer-output/questions_answers/forum/16645)

1. conf.set("mapreduce.map.output.compress", **true**)
2. conf.set("mapreduce.output.fileoutputformat.compress", **false**)

For more details, refer: <http://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/mapred-default.xml>

**mapred.compress.map.output**: Is the compression of data between the mapper and the reducer. If you use snappy codec this will most likely increase read write speed and reduce network overhead. Don't worry about spitting here. These files are not stored in hdfs. They are temp files that exist only for the map reduce job.

**mapred.map.output.compression.codec**: I would use snappy

**mapred.output.compress**: This boolean flag will define is the whole map/reduce job will output compressed data. I would always set this to true also. Faster read/write speeds and less disk spaced used.

**mapred.output.compression.type**: I use block. This will make the compression slittable even for all compression formats (gzip, snappy, and bzip2) just make sure your using a splitable file format like sequence, RCFile, or Avro.

**mapred.output.compression.codec**: this is the compression codec for the map/reduce job. I mostly use one of the three: Snappy (Fastest r/w 2x-3x compression), gzip (normal r fast w 5x-8x compression), bzip2 (slow r/w 8x-12x compression)

Also remember when compression mapred output, that because of splitting compression will differ base on your sorting order. The close like data is together the better the compression.  
  
ref link:http://stackoverflow.com/questions/5571156/hadoop-how-to-compress-mapper-output-but-not-the-reducer-output/11336820#11336820

### Compress Map Output

If Map Output is very large, it is always recommended to use compression techniques to reduce the size of intermediate data. By default, Map Output is not compressed but we can enable by setting below properties to true.

|  |  |
| --- | --- |
| mapreduce.map.output.compress | false |

|  |  |
| --- | --- |
| mapreduce.map.output.compress.codec | org.apache.hadoop.io.compress.DefaultCodec |

#### Below is the code snippet to enable gzip map output compression in our job:

Configuration conf = new Configuration();

conf.setBoolean("mapreduce.map.output.compress", true);

conf.setClass("mapreduce.map.output.compress.codec", GzipCodec.class,

CompressionCodec.class);

Job job = new Job(conf);

**Compress Reducer/Final Output**

We can enable compression on Mapreduce job’s output by setting below properties to true at site level for all jobs.

|  |  |  |
| --- | --- | --- |
| **mapreduce.output.fileoutputformat.compress** | False | Compress? |
| **mapreduce.output.fileoutputformat.compress.type** | RECORD | If SequenceFiles, then it Should be one of NONE, RECORD or BLOCK. |
| **mapreduce.output.fileoutputformat.compress.codec** | org.apache.hadoop.io.compress.DefaultCodec |  |

Set the above properties either in Job driver using c**ode snippet like below (Snappy compression with Block Mode)** or in **mapred-site.xml** file.

* **If Output Files are Not Sequence Files**
* FileOutputFormat.setCompressOutput(job, true);

FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);

* **If Output Files are Sequence Files**

job.setOutputFormatClass(SequenceFileOutputFormat.class);

SequenceFileOutputFormat.setCompressOutput(job, true);

SequenceFileOutputFormat.setOutputCompressorClass(job, SnappyCodec.class);

SequenceFileOutputFormat.setOutputCompressionType(job,

CompressionType.BLOCK);

* **To Make Global Changes to cluster**

<property>

<name>mapreduce.output.fileoutputformat.compress</name>

<value>true</value>

</property>

<property>

<name>mapreduce.output.fileoutputformat.compress.codec</name>

<value>SnappyCodec.class</value>

</property>

<property>

<name>mapreduce.output.fileoutputformat.compress.type</name>

<value>BLOCK</value>

</property>

### Use Compression with Mapreduce

                               Hadoop is intended for storing large data volumes, so compression becomes a mandatory requirement here. There are different compression formats available like gzip,Bzip,LZO etc. Of these Bzip(the latest) and LZO are splittable and in that Bzip offers a better compression but the decompression of the same is expensive. When we look at both space and time LZO is more advisable. Also LZO supports indexing which would again help you while using hive on your data. While running mapreduce with compression we need to know at least the following

1.       How to run map reduce on compressed data

2.       How to produce compressed output from mapreduce

**Running Mapreduce on compressed data**

                It is very straight forward, no need to implement any custom input format for the same. You can use any input formats with compression. The only step is to add the compression codec to the value in io.compression.codecs

Suppose if you are using LZO then your value would look something like

**io.compression.codecs**  =  org.apache.hadoop.io.compress.GzipCodec, org.apache.hadoop.io.compress.DefaultCodec, **com.hadoop.compression.lzo.LzopCodec**

Then configure and run your map reduce jobs as you do normally on uncompressed files. When map wants to process a file and if it is compressed it would check for the io.compression.codecs   and use a suitable codec from there to read the file.

**Produce compressed data from map reduce**

                It is again straight forward and you can achieve the same by setting the following parameters. (Using LZO here)

**mapred.output.compress=true**

**mapred.output.compression.codec= com.hadoop.compression.lzo.LzopCodec**

You get your output compressed in LZO. Again here also you can use the same with any normal output formats.

**Index LZO files**

                It is possible with just 2 lines of code as

//Run theLZO indexer on files in hdfs

LzoIndexer indexer = new LzoIndexer(fs.getConf());

indexer.index(filePath);

**Compress Intermediate output (map output)**

                Compressing intermediate output is also a good idea in map reduce. The map outputs have to be copied across nodes to reducers and if compressed it saves network and transfer time. Just specify the following configuration parameters as

**mapred.compress.map.output=true**

**mapred.map.output.compression.codec= hadoop.compression.lzo.LzoCodec**

Posted by [Bejoy KS](https://www.blogger.com/profile/03581597227249520580)at [9:18 AM](http://kickstarthadoop.blogspot.com/2012/02/use-compression-with-mapreduce.html) [https://resources.blogblog.com/img/icon18_email.gif](https://www.blogger.com/email-post.g?blogID=494502301850787022&postID=5684636752366149184)

When you create a Map/Reduce job in Java, you set the [InputFormat](http://hadoop.apache.org/common/docs/current/api/index.html?org/apache/hadoop/mapreduce/InputFormat.html) & [OutputFormat](http://hadoop.apache.org/common/docs/current/api/index.html?org/apache/hadoop/mapreduce/OutputFormat.html) you wish to use. Most people have heard of and used the basic ones like [TextInputFormat](http://hadoop.apache.org/common/docs/current/api/index.html?org/apache/hadoop/mapreduce/lib/input/TextInputFormat.html) and [SequenceFileInputFormat](http://hadoop.apache.org/common/docs/current/api/index.html?org/apache/hadoop/mapreduce/lib/input/SequenceFileInputFormat.html) – but it’s fairly trivial to extend [FileInputFormat](http://hadoop.apache.org/common/docs/current/api/index.html?org/apache/hadoop/mapreduce/lib/input/FileInputFormat.html) to create new ones.

There are two parts to the puzzle:

1. [InputFormat](http://hadoop.apache.org/common/docs/current/api/index.html?org/apache/hadoop/mapreduce/InputFormat.html) which opens the data source and splits the data into chunks,
2. [RecordReader](http://hadoop.apache.org/common/docs/current/api/index.html?org/apache/hadoop/mapreduce/RecordReader.html) which actually parses the chunks into Key/Value pairs.

Once you’ve implemented these two pieces, the rest of the Map/Reduce framework can use your new-fangled [InputFormat](http://hadoop.apache.org/common/docs/current/api/index.html?org/apache/hadoop/mapreduce/InputFormat.html) as normal. This is also true for [OutputFormat](http://hadoop.apache.org/common/docs/current/api/index.html?org/apache/hadoop/mapreduce/OutputFormat.html)‘s exception you need to implement a [RecordWriter](http://hadoop.apache.org/common/docs/current/api/index.html?org/apache/hadoop/mapreduce/RecordWriter.html) instead of a [RecordReader](http://hadoop.apache.org/common/docs/current/api/index.html?org/apache/hadoop/mapreduce/RecordReader.html).

1. Implement custom writable

Ans)

# Implementing Custom Writable

Hadoop MapReduce uses implementations of Writables for interacting with user-provided Mappers and Reducers. Hadoop provides a lot of implementations of Writables which are listed [here](http://hadoop.apache.org/docs/current/api/org/apache/hadoop/io/Writable.html), but sometimes we need to pass custom objects and these custom objects should implement Hadoop's Writable interface. In this post we are going to describe a custom class IntPair. To implement the Writable interface we require two methods:

public interface Writable {

void readFields(DataInput in);

void write(DataOutput out);

}

The code for IntPair is given below:

public class IntPair implements Writable{

private IntWritable first;

private IntWritable second;

public IntPair() {

set(new IntWritable(), new IntWritable());

}

public IntPair(Integer first, Integer second) {

set(new IntWritable(first), new IntWritable(second));

}

public void set(IntWritable first, IntWritable second) {

this.first = first;

this.second = second;

}

public IntWritable getFirst() {

return first;

}

public Integer getFirstInt() {

return new Integer(first.toString());

}

public Integer getSecondInt() {

return new Integer(second.toString());

}

public IntWritable getSecond() {

return second;

}

public void write(DataOutput out) throws IOException {

first.write(out);

second.write(out);

}

public void readFields(DataInput in) throws IOException {

first.readFields(in);

second.readFields(in);

}

}

Now we can use this IntPair class in Hadoop MapReduce as value type. If we want to use IntPair as key in MapReduce then it needs to implement WritableComparable, which we shall cover in a different post.

### Common Rules for creating custom Hadoop Writable Data Type:

* A custom hadoop writable data type which needs to be used as **value**field in Mapreduce programs must implement**Writable** interface **org.apache.hadoop.io.Writable**.
* MapReduce key types should have the ability to compare against each other for sorting purposes. A custom hadoop writable data type that can be used as **key**field in Mapreduce programs must implement WritableComparableinterface which intern extends Writable (**org.apache.hadoop.io.Writable**) and Comparable (**java.lang.Comparable**) interfaces.
* So, i.e. a data type created by implementing **WritableComparable** Interface can be used as either key or value field data type.

Since a data type implementing WritableComparable can be used as data type for key or value fields in mapreduce programs, Lets define a custom data type which can used for both key and value fields.   
  
In this post, Lets create a custom data type to process Web Logs from a server and count the occurrences of each IP address.   
  
In this sample, lets consider a **web log record** with five fields –   
**Request No, Site URL, Request Date, Request Time and IP address**.   
  
 A sample record from web log file  is as shown below.

**127248      /rr.html      2014-03-10      12:32:08      42.416.153.181**

We can treat the entities of the above record as built-in Writable data types forming a new custom data type. We can consider the Request No as IntWritable and other four fields as Text data types. Complete input file **Web\_Log.txt** used in this post is [**attached here**](http://hadooptutorial.info/wp-content/uploads/2014/04/Web_Log.txt)

### Creating Custom Hadoop Writable Data Type:

Lets create a WebLogWritable Data type to serialize and deserialize the above mentioned Web Log record.

###### WebLogWritable.java

public static class WebLogWritable implements WritableComparable<WebLogWritable> {

private Text siteURL, reqDate, timestamp, ipaddress;

private IntWritable reqNo;

//Default Constructor

public WebLogWritable() {

this.siteURL = new Text();

this.reqDate = new Text();

this.timestamp = new Text();

this.ipaddress = new Text();

this.reqNo = new IntWritable();

}

//Custom Constructor

public WebLogWritable(IntWritable reqno, Text url, Text rdate, Text rtime, Text rip) {

this.siteURL = url;

this.reqDate = rdate;

this.timestamp = rtime;

this.ipaddress = rip;

this.reqNo = reqno;

}

//Setter method to set the values of WebLogWritable object

public void set(IntWritable reqno, Text url, Text rdate, Text rtime, Text rip) {

this.siteURL = url;

this.reqDate = rdate;

this.timestamp = rtime;

this.ipaddress = rip;

this.reqNo = reqno;

}

//to get IP address from WebLog Record

public Text getIp(){

return ipaddress;

}

@Override

//overriding default readFields method.

//It de-serializes the byte stream data

public void readFields(DataInput in) throws IOException {

ipaddress.readFields(in);

timestamp.readFields(in);

reqDate.readFields(in);

reqNo.readFields(in);

siteURL.readFields(in);

}

@Override

//It serializes object data into byte stream data

public void write(DataOutput out) throws IOException {

ipaddress.write(out);

timestamp.write(out);

reqDate.write(out);

reqNo.write(out);

siteURL.write(out);

}

@Override

public int compareTo(WebLogWritable o) {

if (ipaddress.compareTo(o.ipaddress)==0)

{

return (timestamp.compareTo(o.timestamp));

}

else return (ipaddress.compareTo(o.ipaddress));

}

@Override

public boolean equals(Object o) {

if (o instanceof WebLogWritable)

{

WebLogWritable other = (WebLogWritable) o;

return ipaddress.equals(other.ipaddress) &amp;&amp; timestamp.equals(other.timestamp);

}

return false;

}

@Override

public int hashCode() {

return ipaddress.hashCode();

}

}

* Let’s briefly discuss  on the methods written in this WebLogWritable class and their purpose.

All Writable implementations must have a **default constructor** so that the MapReduce framework can instantiate them, then populate their fields by calling **readFields()** . Writable instances are mutable and often reused so we have provided **write()** method. We have also provided custom constructor to set the object fields.

**Set()**and**getIP()** methods are setter and getter methods to store or retrieve data. The **compareTo()**method returns a negative integer, 0,  or a positive integer, if our object is less than, equal to, or greater than the object being compared to respectively.

In **equals()**method, we consider the objects equal if both theIP addresses and the time-stamps are the same. If the objects are not equal, we decide the sort order first based on the user IP address and then based on the time-stamp.

The**hashCode()** method is used by the HashPartitioner, the default partitioner in MapReduce, to choose a reduce partition. Usage of IP Address in our hashCode() method ensures that the intermediate WebLogWritable data will be partitioned based on the request host name/IP address.

### Creating Mapper Class With Custom Data Types:

Below is the implementation of **WebLogMapper**class which read Web\_Log.txt records and tokenizes the records and writes each Web Log record with count 1.

Here **Mapper output key is WebLogWritable** and Output value is IntWritable which is a count value.

###### WebLogMapper.java

public static class WebLogMapper extends Mapper <LongWritable, Text, WebLogWritable, IntWritable> {

private static final IntWritable one = new IntWritable(1);

private WebLogWritable wLog = new WebLogWritable();

private IntWritable reqno = new IntWritable();

private Text url = new Text();

private Text rdate = new Text();

private Text rtime = new Text();

private Text rip = new Text();

public void map(LongWritable key, Text value, Context context)

throws IOException, InterruptedException

{

String[] words = value.toString().split("\t") ;

reqno.set(Integer.parseInt(words[0]));

url.set(words[1]);

rdate.set(words[2]);

rtime.set(words[3]);

rip.set(words[4]);

wLog.set(reqno, url, rdate, rtime, rip);

context.write(wLog, one);

}

}

### Creating Reducer Class With Custom Data Types:

Below is the implementation of **WebLogReducer** class which accumulates the count values of each web log record and emits IP Addresses of each web log record along with its total occurrences in the file **Web\_Log.txt**.

Here Reducer**Input key is WebLogWritable** and Input value is IntWritable but output key is Text and output value is IntWritable.

###### WebLogReducer.java

public static class WebLogReducer extends Reducer <WebLogWritable, IntWritable, Text, IntWritable> {

private IntWritable result = new IntWritable();

private Text ip = new Text();

public void reduce(WebLogWritable key, Iterable<IntWritable> values, Context context)

throws IOException, InterruptedException

{

int sum = 0;

ip = key.getIp();

for (IntWritable val : values)

{

sum++ ;

}

result.set(sum);

context.write(ip, result);

}

}

### Creating Mapreduce driver Class With Custom Data Types:

In this example, We have to specify the **mapOutputKeyclass** as **WebLogWritable** in the driver class and rest of the implementation is as usual.

So, Lets create a **WebLogReader.java** file with the custom Writable class, Mapper class and Reducer classes as shown below to test the functionality of custom writable data type WebLogWritable.

###### WebLogReader.java

//Example Driver class for Word count program

import java.io.\*;

import org.apache.hadoop.io.\*;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WebLogReader {

public static class WebLogWritable implements WritableComparable<WebLogWritable> {

private Text siteURL, reqDate, timestamp, ipaddress;

private IntWritable reqNo;

//Default Constructor

public WebLogWritable() {

this.siteURL = new Text();

this.reqDate = new Text();

this.timestamp = new Text();

this.ipaddress = new Text();

this.reqNo = new IntWritable();

}

//Custom Constructor

public WebLogWritable(IntWritable reqno, Text url, Text rdate, Text rtime, Text rip) {

this.siteURL = url;

this.reqDate = rdate;

this.timestamp = rtime;

this.ipaddress = rip;

this.reqNo = reqno;

}

//Setter method to set the values of WebLogWritable object

public void set(IntWritable reqno, Text url, Text rdate, Text rtime, Text rip) {

this.siteURL = url;

this.reqDate = rdate;

this.timestamp = rtime;

this.ipaddress = rip;

this.reqNo = reqno;

}

//to get IP address from WebLog Record

public Text getIp() {

return ipaddress;

}

@Override

//overriding default readFields method.

//It de-serializes the byte stream data

public void readFields(DataInput in) throws IOException {

ipaddress.readFields(in);

timestamp.readFields(in);

reqDate.readFields(in);

reqNo.readFields(in);

siteURL.readFields(in);

}

@Override

//It serializes object data into byte stream data

public void write(DataOutput out) throws IOException {

ipaddress.write(out);

timestamp.write(out);

reqDate.write(out);

reqNo.write(out);

siteURL.write(out);

}

@Override

public int compareTo(WebLogWritable o) {

if (ipaddress.compareTo(o.ipaddress)==0)

{

return (timestamp.compareTo(o.timestamp));

}

else return (ipaddress.compareTo(o.ipaddress));

}

@Override

public boolean equals(Object o) {

if (o instanceof WebLogWritable)

{

WebLogWritable other = (WebLogWritable) o;

return ipaddress.equals(other.ipaddress) &amp;&amp; timestamp.equals(other.timestamp);

}

return false;

}

@Override

public int hashCode() {

return ipaddress.hashCode();

}

}

public static class WebLogMapper extends Mapper <LongWritable, Text, WebLogWritable, IntWritable> {

private static final IntWritable one = new IntWritable(1);

private WebLogWritable wLog = new WebLogWritable();

private IntWritable reqno = new IntWritable();

private Text url = new Text();

private Text rdate = new Text();

private Text rtime = new Text();

private Text rip = new Text();

public void map(LongWritable key, Text value, Context context)

throws IOException, InterruptedException {

String[] words = value.toString().split("\t") ;

System.out.printf("Words[0] - %s, Words[1] - %s, Words[2] - %s, length - %d",

words[0], words[1], words[2], words.length);

reqno.set(Integer.parseInt(words[0]));

url.set(words[1]);

rdate.set(words[2]);

rtime.set(words[3]);

rip.set(words[4]);

wLog.set(reqno, url, rdate, rtime, rip);

context.write(wLog, one);

}

}

public static class WebLogReducer extends Reducer <WebLogWritable, IntWritable, Text, IntWritable> {

private IntWritable result = new IntWritable();

private Text ip = new Text();

public void reduce(WebLogWritable key, Iterable<IntWritable> values, Context context)

throws IOException, InterruptedException {

int sum = 0;

ip = key.getIp();

for (IntWritable val : values)

{

sum++ ;

}

result.set(sum);

context.write(ip, result);

}

}

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

Job job = new Job();

job.setJobName("WebLog Reader");

job.setJarByClass(WebLogReader.class);

job.setMapperClass(WebLogMapper.class);

job.setReducerClass(WebLogReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

job.setMapOutputKeyClass(WebLogWritable.class);

job.setMapOutputValueClass(IntWritable.class);

FileInputFormat.addInputPath(job, new Path(args[0]));

FileOutputFormat.setOutputPath(job, new Path(args[1]));

System.exit(job.waitForCompletion(true) ? 0 : 1);

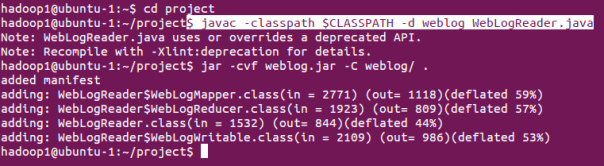
}

}

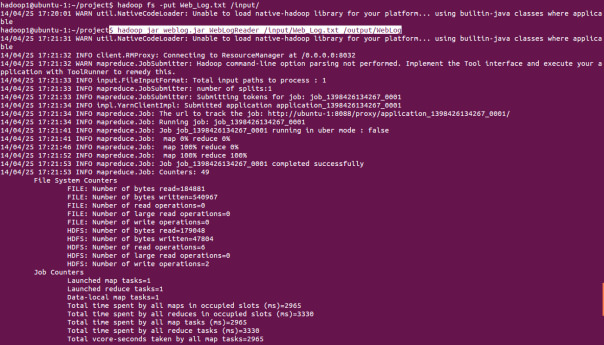
* For easy of compilation and maintenance, we have written all **WebLogWritable, WebLogMapper and WebLogReducer**classes as static and maintained a single **WebLogReader** public main class in our java source file.

### Run WebLogReader.java program:

**Compile** WebLogReader.java file and **create a jar** file with classes.



**Copy** input file **Web\_Log.txt into HDFS** and **execute the jar** file with WebLogReader driver class.



# **Custom Hadoop Writable Data Type**

[Sushanth](https://www.blogger.com/profile/05754574304505876044)

07:10

**Implementing a Custom Hadoop Writable Data Type**

There can be use cases where none of the built-in data types matches the requirements or a custom data type optimized for a  use case may perform better than a Hadoop built-in data type. In such scenarios a custom Writable data type can be written by implementing

the org.apache.hadoop.io.Writable interface to define the serialization format of

your data type.

The Writable interface-based types can be used as value types in Hadoop MapReduce computations.

In this recipe, we implement a sample Hadoop Writable data type for HTTP server log

entries. For the purpose of this sample, we consider that a log entry consists of the five

fields—request host, timestamp, request URL, response size, and the http status code.

The following is a sample log entry:

192.168.0.2 - - [01/Jul/1995:00:00:01 -0400] "GET /history/apollo/

HTTP/1.0" 200 6245

**Program:**

**import** java.io.DataInput;

**import** java.io.DataInputStream;

**import** java.io.DataOutput;

**import** java.io.IOException;

**import** java.io.InputStream;

**import** java.io.InputStreamReader;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

**import** java.util.StringTokenizer;

**import** org.apache.hadoop.fs.Path;

**import** org.apache.hadoop.io.IntWritable;

**import** org.apache.hadoop.io.LongWritable;

**import** org.apache.hadoop.io.Text;

**import** org.apache.hadoop.mapred.FileInputFormat;

**import** org.apache.hadoop.mapred.FileOutputFormat;

**import** org.apache.hadoop.mapred.JobClient;

**import** org.apache.hadoop.mapred.JobConf;

**import** org.apache.hadoop.mapred.MapReduceBase;

**import** org.apache.hadoop.mapred.Mapper;

**import** org.apache.hadoop.mapred.OutputCollector;

**import** org.apache.hadoop.mapred.Reducer;

**import** org.apache.hadoop.mapred.Reporter;

**import** org.apache.hadoop.mapred.TextInputFormat;

**import** org.apache.hadoop.mapred.TextOutputFormat;

**public** **class** CustomDataType {

**public** **static** **class** LogProcessMap **extends** MapReduceBase **implements**

                     Mapper<LongWritable, Text, Text, LogWritable> {

              @Override

**public** **void** map(LongWritable key, Text value,

                           OutputCollector<Text, LogWritable> output, Reporter reporter)

**throws** IOException {

              }

       }

**public** **static** **class** LogProcessReduce **extends** MapReduceBase **implements**

                     Reducer<Text, LogWritable, Text, IntWritable> {

              @Override

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

                           OutputCollector<Text, IntWritable> output, Reporter reporter)

**throws** IOException {

              }

       }

**public** **static** **void** main(String[] args) **throws** Exception {

              JobConf newconf = **new** JobConf(CustomDataType.**class**);

              newconf.setJobName("Custom Data Type");

              newconf.setOutputKeyClass(Text.**class**);

              newconf.setOutputValueClass(IntWritable.**class**);

              newconf.setMapperClass(LogProcessMap.**class**);

              newconf.setReducerClass(LogProcessReduce.**class**);

              newconf.setMapOutputKeyClass(Text.**class**);

              newconf.setMapOutputValueClass(LogWritable.**class**);

              newconf.setInputFormat(TextInputFormat.**class**);

              newconf.setOutputFormat(TextOutputFormat.**class**);

              FileInputFormat.*setInputPaths*(newconf, **new** Path(args[0]));

              FileOutputFormat.*setOutputPath*(newconf, **new** Path(args[1]));

              JobClient.*runJob*(newconf);

       }

}

**import** java.io.DataInput;

**import** java.io.DataOutput;

**import** java.io.IOException;

**import** org.apache.hadoop.io.Text;

**import** org.apache.hadoop.io.IntWritable;

**import** org.apache.hadoop.io.Writable;

**public** **class** LogWritable **implements** Writable{

**private** Text userIP, timestamp, request;

**private** IntWritable responseSize, status;

**public** LogWritable()

       {

**this**.userIP = **new** org.apache.hadoop.io.Text();

**this**.userIP = **new** Text();

**this**.timestamp= **new** Text();

**this**.request = **new** Text();

**this**.responseSize = **new** IntWritable();

**this**.status = **new** IntWritable();

       }

**public** LogWritable(Text userIP,Text timestamp,Text request,IntWritable responseSize,IntWritable status)

       {

**this**.userIP = userIP;

**this**.timestamp = timestamp;

**this**.request = request;

**this**.responseSize = responseSize;

**this**.status = status;

       }

       @Override

**public** **void** readFields(DataInput in) **throws** IOException {

              userIP.readFields(in);

              timestamp.readFields(in);

              request.readFields(in);

              responseSize.readFields(in);

              status.readFields(in);

       }

       @Override

**public** **void** write(DataOutput out) **throws** IOException {

              userIP.write(out);

              timestamp.write(out);

              request.write(out);

              responseSize.write(out);

              status.write(out);

       }

}

- See more at: http://labstrikes.blogspot.in/2012/08/adsense-middle-blog-post.html#sthash.gQgSkqx8.dpuf

1. What is raw comparator

Answer)

# **NTRODUCTION**

Implementing the [org.apache.hadoop.io.RawComparator](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/RawComparator.html) interface will definitely help speed up your Map/Reduce (MR) Jobs. As you may recall, a MR Job is composed of receiving and sending key-value pairs. The process looks like the following.

* (K1,V1) –> Map –> (K2,V2)
* (K2,List[V2]) –> Reduce –> (K3,V3)

The key-value pairs (K2,V2) are called the intermediary key-value pairs. They are passed from the mapper to the reducer. Before these intermediary key-value pairs reach the reducer, a shuffle and sort step is performed. The shuffle is the assignment of the intermediary keys (K2) to reducers and the sort is the sorting of these keys. In this blog, by implementing the RawComparator to compare the intermediary keys, this extra effort will greatly improve sorting. Sorting is improved because the RawComparator will compare the keys by byte. If we did not use RawComparator, the intermediary keys would have to be completely deserialized to perform a comparison.

# **BACKGROUND**

Two ways you may compare your keys is by implementing the [org.apache.hadoop.io.WritableComparable](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/WritableComparable.html) interface or by implementing the RawComparator interface. In the former approach, you will compare (deserialized) objects, but in the latter approach, you will compare the keys using their corresponding raw bytes.

I conducted an empirical test to demonstrate the advantage of RawComparator over WritableComparable. Let’s say we are processing a file that has a list of pairs of indexes {i,j}. These pairs of indexes could refer to the i-th and j-th matrix element. The input data (file) will look something like the following.

1, 2

3, 4

5, 6

...

...

...

0, 0

What we want to do is simply count the occurrences of the {i,j} pair of indexes. Our MR Job will look like the following.

* (LongWritable,Text) –> Map –> ({i,j},IntWritable)
* ({i,j},List[IntWritable]) –> Reduce –> ({i,j},IntWritable)

# **METHOD**

The first thing we have to do is model our intermediary key K2={i,j}. Below is a snippet of the IndexPair. As you can see, it implements WritableComparable. Also, we are sorting the keys ascendingly by the i-th and then j-th indexes.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public class IndexPair implements WritableComparable<IndexPair> {      private IntWritable i;      private IntWritable j;      //....      /\*\*       \* Constructor.       \* @param i i.       \* @param j j.       \*/      public IndexPair(int i, int j) {          this.i = new IntWritable(i);          this.j = new IntWritable(j);      }      //....      @Override      public int compareTo(IndexPair o) {          int cmp = i.compareTo(o.i);          if(0 != cmp)              return cmp;          return j.compareTo(o.j);      }      //....  } |

Below is a snippet of the RawComparator. As you notice, it does not directly implement RawComparator. Rather, it extends WritableComparator (which implements RawComparator). We could have directly implemented RawComparator, but by extending WritableComparator, depending on the complexity of our intermediary key, we may use some of the utility methods of WritableComparator.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class IndexPairComparator extends WritableComparator {      protected IndexPairComparator() {          super(IndexPair.class);      }        @Override      public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {          int i1 = readInt(b1, s1);          int i2 = readInt(b2, s2);            int comp = (i1 < i2) ? -1 : (i1 == i2) ? 0 : 1;          if(0 != comp)              return comp;            int j1 = readInt(b1, s1+4);          int j2 = readInt(b2, s2+4);          comp = (j1 < j2) ? -1 : (j1 == j2) ? 0 : 1;            return comp;      }  } |

As you can see the above code, for the two objects we are comparing, there are two corresponding byte arrays (b1 and b2), the starting positions of the objects in the byte arrays, and the length of the bytes they occupy. Please note that the byte arrays themselves represent other things and not only the objects we are comparing. That is why the starting position and length are also passed in as arguments. Since we want to sort ascendingly by i then j, we first compare the bytes representing the i-th indexes and if they are equal, then we compare the j-th indexes. You can also see that we use the util method, readInt(byte[], start), inherited from WritableComparator. This method simply converts the 4 consecutive bytes beginning at start into a primitive int (the primitive int in Java is 4 bytes). If the i-th indexes are equal, then we shift the starting point by 4, read in the j-th indexes and then compare them.

A snippet of the mapper is shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {      String[] tokens = value.toString().split(",");      int i = Integer.parseInt(tokens[0].trim());      int j = Integer.parseInt(tokens[1].trim());        IndexPair indexPair = new IndexPair(i, j);      context.write(indexPair, ONE);  } |

A snippet of the reducer is shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public void reduce(IndexPair key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {      int sum = 0;      for(IntWritable value : values) {          sum += value.get();      }        context.write(key, new IntWritable(sum));  } |

The snippet of code below shows how I wired up the MR Job that does NOT use raw byte comparison.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public int run(String[] args) throws Exception {      Configuration conf = getConf();      Job job = new Job(conf, "raw comparator example");        job.setJarByClass(RcJob1.class);        job.setMapOutputKeyClass(IndexPair.class);      job.setMapOutputValueClass(IntWritable.class);        job.setOutputKeyClass(IndexPair.class);      job.setOutputValueClass(IntWritable.class);        job.setMapperClass(RcMapper.class);      job.setReducerClass(RcReducer.class);        job.waitForCompletion(true);        return 0;  } |

The snippet of code below shows how I wired up the MR Job using the raw byte comparator.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public int run(String[] args) throws Exception {      Configuration conf = getConf();      Job job = new Job(conf, "raw comparator example");        job.setJarByClass(RcJob1.class);      job.setSortComparatorClass(IndexPairComparator.class);        job.setMapOutputKeyClass(IndexPair.class);      job.setMapOutputValueClass(IntWritable.class);        job.setOutputKeyClass(IndexPair.class);      job.setOutputValueClass(IntWritable.class);        job.setMapperClass(RcMapper.class);      job.setReducerClass(RcReducer.class);        job.waitForCompletion(true);        return 0;  } |

As you can see, the only difference is that in the MR Job using the raw comparator, we explicitly set its sort comparator class.

# **RESULTS**

I ran the MR Jobs (without and with raw byte comparisons) 10 times on a dataset of 4 million rows of {i,j} pairs. The runs were against Hadoop v0.20 in standalone mode on Cygwin. The average running time for the MR Job without raw byte comparison is 60.6 seconds, and the average running time for the job with raw byte comparison is 31.1 seconds. A two-tail paired t-test showed p < 0.001, meaning, there is a statistically significant difference between the two implementations in terms of empirical running time.

I then ran each implementation on datasets of increasing record sizes from 1, 2, …, and 10 million records. At 10 million records, without using raw byte comparison took 127 seconds (over 2 minutes) to complete, while using raw byte comparison took 75 seconds (1 minute and 15 seconds) to complete. Below is a line graph.

8)What is partitioner n mapreduce

Answer:-

A partitioner works like a condition in processing an input dataset. The partition phase takes place after the Map phase and before the Reduce phase.

The number of partitioners is equal to the number of reducers. That means a partitioner will divide the data according to the number of reducers. Therefore, the data passed from a single partitioner is processed by a single Reducer.

## **Partitioner**

A partitioner partitions the key-value pairs of intermediate Map-outputs. It partitions the data using a user-defined condition, which works like a hash function. The total number of partitions is same as the number of Reducer tasks for the job. Let us take an example to understand how the partitioner works.

## **MapReduce Partitioner Implementation**

For the sake of convenience, let us assume we have a small table called Employee with the following data. We will use this sample data as our input dataset to demonstrate how the partitioner works.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Id** | **Name** | **Age** | **Gender** | **Salary** |
| 1201 | gopal | 45 | Male | 50,000 |
| 1202 | manisha | 40 | Female | 50,000 |
| 1203 | khalil | 34 | Male | 30,000 |
| 1204 | prasanth | 30 | Male | 30,000 |
| 1205 | kiran | 20 | Male | 40,000 |
| 1206 | laxmi | 25 | Female | 35,000 |
| 1207 | bhavya | 20 | Female | 15,000 |
| 1208 | reshma | 19 | Female | 15,000 |
| 1209 | kranthi | 22 | Male | 22,000 |
| 1210 | Satish | 24 | Male | 25,000 |
| 1211 | Krishna | 25 | Male | 25,000 |
| 1212 | Arshad | 28 | Male | 20,000 |
| 1213 | lavanya | 18 | Female | 8,000 |

We have to write an application to process the input dataset to find the highest salaried employee by gender in different age groups (for example, below 20, between 21 to 30, above 30).

### Input Data

The above data is saved as **input.txt** in the “/home/hadoop/hadoopPartitioner” directory and given as input.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1201 | gopal | 45 | Male | 50000 |
| 1202 | manisha | 40 | Female | 51000 |
| 1203 | khaleel | 34 | Male | 30000 |
| 1204 | prasanth | 30 | Male | 31000 |
| 1205 | kiran | 20 | Male | 40000 |
| 1206 | laxmi | 25 | Female | 35000 |
| 1207 | bhavya | 20 | Female | 15000 |
| 1208 | reshma | 19 | Female | 14000 |
| 1209 | kranthi | 22 | Male | 22000 |
| 1210 | Satish | 24 | Male | 25000 |
| 1211 | Krishna | 25 | Male | 26000 |
| 1212 | Arshad | 28 | Male | 20000 |
| 1213 | lavanya | 18 | Female | 8000 |

Based on the given input, following is the algorithmic explanation of the program.

### Map Tasks

The map task accepts the key-value pairs as input while we have the text data in a text file. The input for this map task is as follows −

**Input** − The key would be a pattern such as “any special key + filename + line number” (example: key = @input1) and the value would be the data in that line (example: value = 1201 \t gopal \t 45 \t Male \t 50000).

**Method** − The operation of this map task is as follows −

* Read the **value** (record data), which comes as input value from the argument list in a string.
* Using the split function, separate the gender and store in a string variable.

String[] str = value.toString().split("\t", -3);

String gender=str[3];

* Send the gender information and the record data **value** as output key-value pair from the map task to the **partition task**.

context.write(new Text(gender), new Text(value));

* Repeat all the above steps for all the records in the text file.

**Output** − You will get the gender data and the record data value as key-value pairs.

### Partitioner Task

The partitioner task accepts the key-value pairs from the map task as its input. Partition implies dividing the data into segments. According to the given conditional criteria of partitions, the input key-value paired data can be divided into three parts based on the age criteria.

**Input** − The whole data in a collection of key-value pairs.

key = Gender field value in the record.

value = Whole record data value of that gender.

**Method** − The process of partition logic runs as follows.

* Read the age field value from the input key-value pair.

String[] str = value.toString().split("\t");

int age = Integer.parseInt(str[2]);

* Check the age value with the following conditions.
  + Age less than or equal to 20
  + Age Greater than 20 and Less than or equal to 30.
  + Age Greater than 30.

if(age<=20)

{

return 0;

}

else if(age>20 && age<=30)

{

return 1 % numReduceTasks;

}

else

{

return 2 % numReduceTasks;

}

**Output** − The whole data of key-value pairs are segmented into three collections of key-value pairs. The Reducer works individually on each collection.

### Reduce Tasks

The number of partitioner tasks is equal to the number of reducer tasks. Here we have three partitioner tasks and hence we have three Reducer tasks to be executed.

**Input** − The Reducer will execute three times with different collection of key-value pairs.

key = gender field value in the record.

value = the whole record data of that gender.

**Method** − The following logic will be applied on each collection.

* Read the Salary field value of each record.

String [] str = val.toString().split("\t", -3);

Note: str[4] have the salary field value.

* Check the salary with the max variable. If str[4] is the max salary, then assign str[4] to max, otherwise skip the step.

if(Integer.parseInt(str[4])>max)

{

max=Integer.parseInt(str[4]);

}

* Repeat Steps 1 and 2 for each key collection (Male & Female are the key collections). After executing these three steps, you will find one max salary from the Male key collection and one max salary from the Female key collection.

context.write(new Text(key), new IntWritable(max));

**Output** − Finally, you will get a set of key-value pair data in three collections of different age groups. It contains the max salary from the Male collection and the max salary from the Female collection in each age group respectively.

After executing the Map, the Partitioner, and the Reduce tasks, the three collections of key-value pair data are stored in three different files as the output.

All the three tasks are treated as MapReduce jobs. The following requirements and specifications of these jobs should be specified in the Configurations −

* Job name
* Input and Output formats of keys and values
* Individual classes for Map, Reduce, and Partitioner tasks

Configuration conf = getConf();

//Create Job

Job job = new Job(conf, "topsal");

job.setJarByClass(PartitionerExample.class);

// File Input and Output paths

FileInputFormat.setInputPaths(job, new Path(arg[0]));

FileOutputFormat.setOutputPath(job,new Path(arg[1]));

//Set Mapper class and Output format for key-value pair.

job.setMapperClass(MapClass.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(Text.class);

//set partitioner statement

job.setPartitionerClass(CaderPartitioner.class);

//Set Reducer class and Input/Output format for key-value pair.

job.setReducerClass(ReduceClass.class);

//Number of Reducer tasks.

job.setNumReduceTasks(3);

//Input and Output format for data

job.setInputFormatClass(TextInputFormat.class);

job.setOutputFormatClass(TextOutputFormat.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(Text.class);

### Example Program

The following program shows how to implement the partitioners for the given criteria in a MapReduce program.

package partitionerexample;

import java.io.\*;

import org.apache.hadoop.io.\*;

import org.apache.hadoop.mapreduce.\*;

import org.apache.hadoop.conf.\*;

import org.apache.hadoop.conf.\*;

import org.apache.hadoop.fs.\*;

import org.apache.hadoop.mapreduce.lib.input.\*;

import org.apache.hadoop.mapreduce.lib.output.\*;

import org.apache.hadoop.util.\*;

public class PartitionerExample extends Configured implements Tool

{

//Map class

public static class MapClass extends Mapper<LongWritable,Text,Text,Text>

{

public void map(LongWritable key, Text value, Context context)

{

try{

String[] str = value.toString().split("\t", -3);

String gender=str[3];

context.write(new Text(gender), new Text(value));

}

catch(Exception e)

{

System.out.println(e.getMessage());

}

}

}

//Reducer class

public static class ReduceClass extends Reducer<Text,Text,Text,IntWritable>

{

public int max = -1;

public void reduce(Text key, Iterable <Text> values, Context context) throws IOException, InterruptedException

{

max = -1;

for (Text val : values)

{

String [] str = val.toString().split("\t", -3);

if(Integer.parseInt(str[4])>max)

max=Integer.parseInt(str[4]);

}

context.write(new Text(key), new IntWritable(max));

}

}

//Partitioner class

public static class CaderPartitioner extends

Partitioner < Text, Text >

{

@Override

public int getPartition(Text key, Text value, int numReduceTasks)

{

String[] str = value.toString().split("\t");

int age = Integer.parseInt(str[2]);

if(numReduceTasks == 0)

{

return 0;

}

if(age<=20)

{

return 0;

}

else if(age>20 && age<=30)

{

return 1 % numReduceTasks;

}

else

{

return 2 % numReduceTasks;

}

}

}

@Override

public int run(String[] arg) throws Exception

{

Configuration conf = getConf();

Job job = new Job(conf, "topsal");

job.setJarByClass(PartitionerExample.class);

FileInputFormat.setInputPaths(job, new Path(arg[0]));

FileOutputFormat.setOutputPath(job,new Path(arg[1]));

job.setMapperClass(MapClass.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(Text.class);

//set partitioner statement

job.setPartitionerClass(CaderPartitioner.class);

job.setReducerClass(ReduceClass.class);

job.setNumReduceTasks(3);

job.setInputFormatClass(TextInputFormat.class);

job.setOutputFormatClass(TextOutputFormat.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(Text.class);

System.exit(job.waitForCompletion(true)? 0 : 1);

return 0;

}

public static void main(String ar[]) throws Exception

{

int res = ToolRunner.run(new Configuration(), new PartitionerExample(),ar);

System.exit(0);

}

}

Save the above code as **PartitionerExample.java** in “/home/hadoop/hadoopPartitioner”. The compilation and execution of the program is given below.

### Compilation and Execution

Let us assume we are in the home directory of the Hadoop user (for example, /home/hadoop).

Follow the steps given below to compile and execute the above program.

**Step 1** − Download Hadoop-core-1.2.1.jar, which is used to compile and execute the MapReduce program. You can download the jar from [mvnrepository.com](http://mvnrepository.com/artifact/org.apache.hadoop/hadoop-core/1.2.1).

Let us assume the downloaded folder is “/home/hadoop/hadoopPartitioner”

**Step 2** − The following commands are used for compiling the program **PartitionerExample.java** and creating a jar for the program.

$ javac -classpath hadoop-core-1.2.1.jar -d ProcessUnits.java

$ jar -cvf PartitionerExample.jar -C .

**Step 3** − Use the following command to create an input directory in HDFS.

$HADOOP\_HOME/bin/hadoop fs -mkdir input\_dir

**Step 4** − Use the following command to copy the input file named **input.txt** in the input directory of HDFS.

$HADOOP\_HOME/bin/hadoop fs -put /home/hadoop/hadoopPartitioner/input.txt input\_dir

**Step 5** − Use the following command to verify the files in the input directory.

$HADOOP\_HOME/bin/hadoop fs -ls input\_dir/

**Step 6** − Use the following command to run the Top salary application by taking input files from the input directory.

$HADOOP\_HOME/bin/hadoop jar PartitionerExample.jar partitionerexample.PartitionerExample input\_dir/input.txt output\_dir

Wait for a while till the file gets executed. After execution, the output contains a number of input splits, map tasks, and Reducer tasks.

15/02/04 15:19:51 INFO mapreduce.Job: Job job\_1423027269044\_0021 completed successfully

15/02/04 15:19:52 INFO mapreduce.Job: Counters: 49

File System Counters

FILE: Number of bytes read=467

FILE: Number of bytes written=426777

FILE: Number of read operations=0

FILE: Number of large read operations=0

FILE: Number of write operations=0

HDFS: Number of bytes read=480

HDFS: Number of bytes written=72

HDFS: Number of read operations=12

HDFS: Number of large read operations=0

HDFS: Number of write operations=6

Job Counters

Launched map tasks=1

Launched reduce tasks=3

Data-local map tasks=1

Total time spent by all maps in occupied slots (ms)=8212

Total time spent by all reduces in occupied slots (ms)=59858

Total time spent by all map tasks (ms)=8212

Total time spent by all reduce tasks (ms)=59858

Total vcore-seconds taken by all map tasks=8212

Total vcore-seconds taken by all reduce tasks=59858

Total megabyte-seconds taken by all map tasks=8409088

Total megabyte-seconds taken by all reduce tasks=61294592

Map-Reduce Framework

Map input records=13

Map output records=13

Map output bytes=423

Map output materialized bytes=467

Input split bytes=119

Combine input records=0

Combine output records=0

Reduce input groups=6

Reduce shuffle bytes=467

Reduce input records=13

Reduce output records=6

Spilled Records=26

Shuffled Maps =3

Failed Shuffles=0

Merged Map outputs=3

GC time elapsed (ms)=224

CPU time spent (ms)=3690

Physical memory (bytes) snapshot=553816064

Virtual memory (bytes) snapshot=3441266688

Total committed heap usage (bytes)=334102528

Shuffle Errors

BAD\_ID=0

CONNECTION=0

IO\_ERROR=0

WRONG\_LENGTH=0

WRONG\_MAP=0

WRONG\_REDUCE=0

File Input Format Counters

Bytes Read=361

File Output Format Counters

Bytes Written=72

**Step 7** − Use the following command to verify the resultant files in the output folder.

$HADOOP\_HOME/bin/hadoop fs -ls output\_dir/

You will find the output in three files because you are using three partitioners and three Reducers in your program.

**Step 8** − Use the following command to see the output in **Part-00000** file. This file is generated by HDFS.

$HADOOP\_HOME/bin/hadoop fs -cat output\_dir/part-00000

**Output in Part-00000**

Female 15000

Male 40000

Use the following command to see the output in **Part-00001** file.

$HADOOP\_HOME/bin/hadoop fs -cat output\_dir/part-00001

**Output in Part-00001**

Female 35000

Male 31000

Use the following command to see the output in **Part-00002** file.

$HADOOP\_HOME/bin/hadoop fs -cat output\_dir/part-00002

**Output in Part-00002**

Female 51000

Male 50000

# **Custom Partitioner**

This example illustrates how to use customized partitioner in a MapReduce program. The partitioning phase takes place after the map phase and before the reduce phase. The number of partitions is equal to the number of reducers. The data gets partitioned across the reducers according to the partitioning function[[1]](https://hadooptutorial.wikispaces.com/Custom+partitioner#cite_note-1) . The difference between a partitioner and a [combiner](https://hadooptutorial.wikispaces.com/Custom+combiner) is that the partitioner divides the data according to the number of reducers so that all the data in a single partition gets executed by a single reducer. However, the combiner functions similar to the reducer and processes the data in each partition. The combiner is an optimization to the reducer. The default partitioning function is the hash partitioning function where the hashing is done on the key. However it might be useful to partition the data according to some other function of the key or the value.  
  
For this example we do not consider a graph problem. Let's consider the data that has input in the following format:  
  
**Input Format:**  
name<tab>age<tab>gender<tab>score  
  
We will use custom partitioning in MapReduce program to find the maximum scorer in each gender and three age categories: less than 20, 20 to 50, greater than 50.  
  
**Input**  
  
Alice<tab>23<tab>female<tab>45  
Bob<tab>34<tab>male<tab>89  
Chris<tab>67<tab>male<tab>97  
Kristine<tab>38<tab>female<tab>53  
Connor<tab>25<tab>male<tab>27  
Daniel<tab>78<tab>male<tab>95  
James<tab>34<tab>male<tab>79  
Alex<tab>52<tab>male<tab>69  
Nancy<tab>7<tab>female<tab>98  
Adam<tab>9<tab>male<tab>37  
Jacob<tab>7<tab>male<tab>23  
Mary<tab>6<tab>female<tab>93  
Clara<tab>87<tab>female<tab>72  
Monica<tab>56<tab>female<tab>92  
  
  
**Output:**  
  
***Partition - 0:*** (this partition contains the maximum scorers for each gender whose age is less than 20)  
Nancy<tab>age- 7<tab>female<tab>score-98  
Adam<tab>age- 9<tab>male<tab>score-37  
  
***Partition - 1:*** (this partition contains the maximum scorers for each gender whose age is between 20 and 50)  
  
Kristine<tab>age- 38<tab>female<tab>score-53  
Bob<tab>age- 34<tab>male<tab>score-89  
  
**Partition - 2:** (this partition contains the maximum scorers for each gender whose age is greater than 50)  
  
Monica<tab>age- 56<tab>female<tab>score-92  
Chris<tab>age- 67<tab>male<tab>score-97  
  
  
  
**PartitionMapper**  
  
PartitionMapper prepares the data for the partitioner and the reducer. It parses the input records and emits key-value pairs where the key is the gender and the value is the other information associated with a person.

1. *//mapper output format : gender is the key, the value is formed by concatenating the name, age and the score*
3. *// the type parameters are the input keys type, the input values type, the*
4. *// output keys type, the output values type*
5. @Override
6. **public** **static** **class** PartitionMapper **extends**
7. Mapper<Object, Text, Text, Text> {

10. **public** **void** map([Object](http://www.google.com/search?hl=en&q=allinurl%3Aobject+java.sun.com&btnI=I%27m%20Feeling%20Lucky) key, Text value, [Context](http://www.google.com/search?hl=en&q=allinurl%3Acontext+java.sun.com&btnI=I%27m%20Feeling%20Lucky) context)
11. **throws** [IOException](http://www.google.com/search?hl=en&q=allinurl%3Aioexception+java.sun.com&btnI=I%27m%20Feeling%20Lucky), [InterruptedException](http://www.google.com/search?hl=en&q=allinurl%3Ainterruptedexception+java.sun.com&btnI=I%27m%20Feeling%20Lucky) {
13. [String](http://www.google.com/search?hl=en&q=allinurl%3Astring+java.sun.com&btnI=I%27m%20Feeling%20Lucky)[] tokens = value.toString().split("**\t**");
15. [String](http://www.google.com/search?hl=en&q=allinurl%3Astring+java.sun.com&btnI=I%27m%20Feeling%20Lucky) gender = tokens[2].toString();
16. [String](http://www.google.com/search?hl=en&q=allinurl%3Astring+java.sun.com&btnI=I%27m%20Feeling%20Lucky) nameAgeScore = tokens[0]+"**\t**"+tokens[1]+"**\t**"+tokens[3];
18. *//the mapper emits key, value pair where the key is the gender and the value is the other information which includes name, age and score*
19. context.write(**new** Text(gender), **new** Text(nameAgeScore));
20. }
21. }

**AgePartitioner**  
  
Lines 6-33 implement a customized partitioner called the AgePartitioner that extends the Partitioner class. It overrides the getPartition function (lines 9-32), which has three parameters. The key and value are the intermediate key and value produced by the map function. The numReduceTasks is the number of reducers used in the MapReduce program and it is specified in the driver program. Here we parse the value and get the age information in lines 12-13. Then we assign them to the partition 0, 1, or 2, depending on the age categories [lines 20-30]. Lines 16-17 return partition 0 in case the number of reducers is set to 0, to avoid divide by zero exception. It is possible to have empty partitions with no data. We do the assigned partition number modulo numReduceTasks to avoid illegal partitions if the system has a lesser number of possible reducers than the assigned number.

1. *//AgePartitioner is a custom Partitioner to partition the data according to age.*
2. *//The age is a part of the value from the input file.*
3. *//The data is partitioned based on the range of the age.*
4. *//In this example, there are 3 partitions, the first partition contains the information where the age is less than 20*
5. *//The second partition contains data with age ranging between 20 and 50 and the third partition contains data where the age is >50.*
6. **public** **static** **class** AgePartitioner **extends** Partitioner<Text, Text> {
8. @Override
9. **public** **int** getPartition(Text key, Text value, **int** numReduceTasks) {
11. [String](http://www.google.com/search?hl=en&q=allinurl%3Astring+java.sun.com&btnI=I%27m%20Feeling%20Lucky) [] nameAgeScore = value.toString().split("**\t**");
12. [String](http://www.google.com/search?hl=en&q=allinurl%3Astring+java.sun.com&btnI=I%27m%20Feeling%20Lucky) age = nameAgeScore[1];
13. **int** ageInt = [Integer](http://www.google.com/search?hl=en&q=allinurl%3Ainteger+java.sun.com&btnI=I%27m%20Feeling%20Lucky).parseInt(age);
15. *//this is done to avoid performing mod with 0*
16. **if**(numReduceTasks == 0)
17. **return** 0;
19. *//if the age is <20, assign partition 0*
20. **if**(ageInt <=20){
21. **return** 0;
22. }
23. *//else if the age is between 20 and 50, assign partition 1*
24. **if**(ageInt >20 && ageInt <=50){
26. **return** 1 % numReduceTasks;
27. }
28. *//otherwise assign partition 2*
29. **else**
30. **return** 2 % numReduceTasks;
32. }
33. }

**PartitionReducer**  
  
PartitionReducer finds the maximum scorer in each age category. That is, it finds the maximum scorer in each partition for both male and female.  
  
Lines 10-34 overrides the reduce function of the Reducer class. Lines 20-32 iterate over all the values and finds the maximum scorer for male and female in each age category. This information is emitted from the reducer in the form of key-value pair in line 33. The key is the name and the value is the rest of the person's information.

1. *//The data belonging to the same partition go to the same reducer. In a particular partition, all the values with the same key are iterated and the person with the maximum score is found.*
2. *//Therefore the output of the reducer will contain the male and female maximum scorers in each of the 3 age categories.*
4. *// the type parameters are the input keys type, the input values type, the*
5. *// output keys type, the output values type*
7. **static** **class** ParitionReducer **extends** Reducer<Text, Text, Text, Text> {
9. @Override
10. **public** **void** reduce(Text key, Iterable<Text> values, [Context](http://www.google.com/search?hl=en&q=allinurl%3Acontext+java.sun.com&btnI=I%27m%20Feeling%20Lucky) context)
11. **throws** [IOException](http://www.google.com/search?hl=en&q=allinurl%3Aioexception+java.sun.com&btnI=I%27m%20Feeling%20Lucky), [InterruptedException](http://www.google.com/search?hl=en&q=allinurl%3Ainterruptedexception+java.sun.com&btnI=I%27m%20Feeling%20Lucky) {
13. **int** maxScore = [Integer](http://www.google.com/search?hl=en&q=allinurl%3Ainteger+java.sun.com&btnI=I%27m%20Feeling%20Lucky).MIN\_VALUE;
15. [String](http://www.google.com/search?hl=en&q=allinurl%3Astring+java.sun.com&btnI=I%27m%20Feeling%20Lucky) name = " ";
16. [String](http://www.google.com/search?hl=en&q=allinurl%3Astring+java.sun.com&btnI=I%27m%20Feeling%20Lucky) age = " ";
17. [String](http://www.google.com/search?hl=en&q=allinurl%3Astring+java.sun.com&btnI=I%27m%20Feeling%20Lucky) gender = " ";
18. **int** score = 0;
19. *//iterating through the values corresponding to a particular key*
20. **for**(Text val: values){
22. [String](http://www.google.com/search?hl=en&q=allinurl%3Astring+java.sun.com&btnI=I%27m%20Feeling%20Lucky) [] valTokens = val.toString().split("**\\**t");
23. score = [Integer](http://www.google.com/search?hl=en&q=allinurl%3Ainteger+java.sun.com&btnI=I%27m%20Feeling%20Lucky).parseInt(valTokens[2]);
25. *//if the new score is greater than the current maximum score, update the fields as they will be the output of the reducer after all the values are processed for a particular key*
26. **if**(score > maxScore){
27. name = valTokens[0];
28. age = valTokens[1];
29. gender = key.toString();
30. maxScore = score;
31. }
32. }
33. context.write(**new** Text(name), **new** Text("age- "+age+"**\t**"+gender+"**\t**score-"+maxScore));
34. }
35. }

**driver**  
  
The [driver](https://hadooptutorial.wikispaces.com/Single-source+all+pairs+shortest+path+using+parallel+Breadth-first+search#driver) code is similar to the driver code of the single-source shortest-path problem except that the classes are set according to the classes of this program. In addition it has the following line:

job.setPartitionerClass(AgePartitioner.**class**);

The above line specifies the partitioner class========

**Why we need to do partitioning in map reduce ??**

As you must be aware that a map reduce job takes an input data set and produces the list of key value paire(Key,value) which is a result of map phase in which the input data set is split and each map task processs the split and each map output the list of key value pairs.

The output from map are then feed to reduce tasks which processes the user defined reduce function on map outputs. But before the reduce phase is another process that partition the map outputs based on the key and it keeps the record of same key into the same partitions.

Again why we are doing partitioning before providing them to reduce tasks.

**Consider an example**

We have a word count example, you must have analysed the outputs of such well known example let'say our input dataset by two map tasks gives results as -

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Map output1  i 20  we 15  they 12  an 25    Map output2  their 12  them 10  we 15  to 18 |

So if you observe the output from two map tasks you should have noticed that the count of word 'we' is in outputs of both map tasks and it will be processed twice if sent to two different reducers So here partitioning plays the role.

Before it sends outputs to reducers it will partition the intermediate key value pairs based on key and send the same key to the same partition.

**How the number of partitions are decided ??**

Hadoop decides it at the time when the map reduce job starts that how may partitions will be there which is controlled by the JobConf.setNumReduceTasks()) method, suppose if decide 20 reduce tasks, the 20 partitions will be there and must be filled.

**What is happening by default ??**

By default the partitioner implementation is called HashPartitioner. It uses the hashCode() method of the key objects modulo the number of partitions total to determine which partition to send a given (key, value) pair to.

Partitioner provides the getPartition() method that you can implement yourself if you want to declare the custom partition for your job.  
The getPartition() method receives a key and a value and the number of partitions to split the data, a number in the range [0, numPartitions) must be returned by this method, indicating which partition to send the key and value to. For any two keys k1 and k2, k1.equals(k2) implies getPartition(k1, \*, n) == getPartition(k2, \*, n).

|  |  |
| --- | --- |
| 1  2  3  4 | public int getPartition(K key, V value,                           int numReduceTasks) {     return (key.hashCode() & Integer.MAX\_VALUE) % numReduceTasks;   } |

So you have seen why partitioning is necessary, now let me shade a light on what is poor partitioning and why it happens ??

**Case of Poor partitioning and how to overcome it ??**

Suppose you know that one of the key in your data input will appear more than any other key so you may want to send all your key (large number) to one partition and then distribute the other keys over all other partition by their hasCode(). So now if you have two mechanism of sending data to partitions

1. First,the key appearing more will be send to one partition
2. Second, all other keys will be send to partitions according to their hashCode().
3. Now suppose if your hashCode() method does not uniformly distribute other keys data over partitions range. So the data is not evenly distributed in partitions as well as reducers.Since each partition is equivalent to a reducer.So here some reducers will have more data than other reducers.So other reducers will wait for one reducer(one with user defined keys) due to the work load it shares.

So here we should take an approach that its work load may be shared across many different reducers.

**Example Scenario**

Let's say i have an data set of the form person information,Our dataset contains the name,country, sports liked(To keep it simple i have used three fields).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | PersonA,India,Cricket  PersonB,Brazil,Soccer  PersonC,Australia,Baseball  PersonD,India,Cricket  PersonE,England,Cricket  PersonF,Australia,Cricket  PersonG,India,Cricket  PersonH,England,Cricket  PersonI,India,Cricket  PersonJ,India,Cricket  PersonK,India,Cricket  .. |

We need to count the person for each of the game in the list.

So our key becomes the third field i.e., the game .

Observe the above example and let's suppose we have a large set of data like this where the frequency of data is in direction of country india.

I have used the country frequency factor to show when u need the partitioning.

**How it works ??**

Our map function will take the inputs and generate the intermediate key value pair. so what it ll do it send the output to reducers with default partitioning using the hashPartitioner which uses the hashCode() to partition the data.

And what does the default partitioner does it will send out all values with the same keys to same reducer.

So all values with same key(cricket) send to same reducer. but since our data contains a large number of such key value pair due the fact the country frequency of dataset is india.

And also data key (cricket) is also present for other country map output. So we have a lots of key value data to send to same reducer for cricket key. Here we will write our custom partitioner. And follows the approach below.

Our custom partitioner will send all key value by country india to one partition and other key value with countries like(England,Australia) to other partition so that work load one reducer that should process key cricket is divided into two reducers.

**Implementation of our custom Partitone**r(you can clone the project at<https://github.com/roanjain/hadoop-partitioner>)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108 | package com.mapred.partitioner;    import java.io.IOException;    import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.conf.Configured;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.Partitioner;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;  import org.apache.hadoop.util.Tool;  import org.apache.hadoop.util.ToolRunner;  import org.apache.hadoop.mapreduce.Reducer;  import org.apache.hadoop.mapreduce.Mapper;  /\*\*   \*   \* @author Rohan Jain   \*   \*/  public class PartitionerDemo extends Configured implements Tool {      /\*\*       \* Mapper class generating key value pair of game,country as intermediate keys       \*/      public static class PartitionerMap extends Mapper<LongWritable, Text, Text, Text> {          public void map(LongWritable key,Text value,Context context) throws IOException, InterruptedException {              String[] words = value.toString().split(" ");              try{              context.write(new Text(words[2]),new Text(words[1]));              }              catch(Exception e){                  System.err.println(e);              }          }      }      /\*\*       \* Each partition processed by different reducer tasks as defined in our custom partitioner       \*/      public static class PartitionerReduce extends Reducer<Text,Text,Text,IntWritable> {          public void reduce(Text key,Iterable<Text> values,Context context) throws IOException, InterruptedException {              int gameCount=0;              for(Text val:values){                  gameCount++;              }              context.write(new Text(key),new IntWritable(gameCount));          }      }      /\*\*       \* Our custom Partitioner class will divide the dataset into three partitions one with key as cricket and value as       \* india, second partition with key as cricket and value other than india, and third partition with game(key) other       \* than cricket       \*/      public static class customPartitioner extends Partitioner<Text,Text>{          public int getPartition(Text key, Text value, int numReduceTasks){          if(numReduceTasks==0)              return 0;          if(key.equals(new Text("Cricket")) && !value.equals(new Text("India")))              return 0;          if(key.equals(new Text("Cricket")) && value.equals(new Text("India")))              return 1;          else              return 2;          }      }      public static void main(String[] args) throws Exception {          int res= ToolRunner.run(new Configuration(),new PartitionerDemo(),args);          System.exit(res);      }      public int run(String[] args) throws Exception {          if(args.length!=2){              System.out.print("Run as -- hadoop jar /path/to/partitioner.jar /inputdataset /output");              System.exit(-1);          }            Configuration conf = this.getConf();          Job job = Job.getInstance(conf);          job.setJarByClass(PartitionerDemo.class);            //Set number of reducer tasks          job.setNumReduceTasks(3);            job.setOutputKeyClass(Text.class);          job.setOutputValueClass(IntWritable.class);            job.setMapOutputKeyClass(Text.class);          job.setMapOutputValueClass(Text.class);            job.setMapperClass(PartitionerMap.class);          job.setReducerClass(PartitionerReduce.class);            //Set Partitioner Class          job.setPartitionerClass(customPartitioner.class);            job.setInputFormatClass(TextInputFormat.class);          job.setOutputFormatClass(TextOutputFormat.class);            FileInputFormat.setInputPaths(job, new Path(args[0]));          FileOutputFormat.setOutputPath(job, new Path(args[1]));            return job.waitForCompletion(true) ? 0 : 1;      }    } |

You can check your output for your partitioned dataset at output directory, you will see the output as below :

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | hduser@rohan-Vostro-3446:/usr/local/hadoop/bin$ hadoop fs -ls /partitionerOutput/  14/12/01 17:50:52 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable  Found 4 items  -rw-r--r--   1 hduser supergroup          0 2014-12-01 17:49 /partitionerOutput/\_SUCCESS  -rw-r--r--   1 hduser supergroup         10 2014-12-01 17:48 /partitionerOutput/part-r-00000  -rw-r--r--   1 hduser supergroup         10 2014-12-01 17:48 /partitionerOutput/part-r-00001  -rw-r--r--   1 hduser supergroup          9 2014-12-01 17:49 /partitionerOutput/part-r-00002 |

Hello All,

In last class we seen the simple word-count Example.Now we will learn hadoop mapreduce example with partitioner and how to create multiple output files those have the same info i.e. nothing but the data is break down in groups and send them to different reducers and create separate outputs.

**Input Format:**  
name<tab>age<tab>gender<tab>salary

We will use custom [partitioning in MapReduce](http://www.myhadoopexamples.com/2015/06/10/partitioning-in-mapreduce/) program to find the maximum salary in each gender and three age categories: less than 20, 20 to 50, greater than 50.

**Input**  
Rajee<tab>23<tab>female<tab>5000  
Rama<tab>34<tab>male<tab>7000  
Arjun<tab>67<tab>male<tab>900000  
Keerthi<tab>38<tab>female<tab>100000  
Kishore<tab>25<tab>male<tab>23000  
Daniel<tab>78<tab>male<tab>7600  
James<tab>34<tab>male<tab>86000  
Alex<tab>52<tab>male<tab>6900  
Nancy<tab>7<tab>female<tab>9800  
Adam<tab>9<tab>male<tab>3700  
Jacob<tab>7<tab>male<tab>2390  
Mary<tab>6<tab>female<tab>9300  
Clara<tab>87<tab>female<tab>72000  
Monica<tab>56<tab>female<tab>92000

**Output:**

***Partition – 0:*** (this partition contains the maximum salaries for each gender whose age is less than 20)  
Nancy<tab>age- 7<tab>female<tab>salary-9800  
Adam<tab>age- 9<tab>male<tab>salary-3700

***Partition – 1:*** (this partition contains the maximum salaries for each gender whose age is between 20 and 50)

Kristine<tab>age- 38<tab>female<tab>salary-100000  
Bob<tab>age- 34<tab>male<tab>salary-86000

**Partition – 2:** (this partition contains the maximum salaries for each gender whose age is greater than 50)

Monica<tab>age- 56<tab>female<tab>salary-92000  
Chris<tab>age- 67<tab>male<tab>salary-900000

## **PartitionerDriver.java**

[code]

package com.hadoop;

import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.conf.Configured;  
import org.apache.hadoop.fs.Path;  
//import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Job;  
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;  
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;  
import org.apache.hadoop.util.Tool;  
import org.apache.hadoop.util.ToolRunner;

public class PartitionerDriver extends Configured implements Tool{

@Override  
public int run(String[] args) throws Exception {  
Configuration conf = new Configuration();  
Job job = new Job(conf, “partitioner”);

job.setJarByClass(getClass());

// configure output and input source  
TextInputFormat.addInputPath(job, new Path(args[0]));  
job.setInputFormatClass(TextInputFormat.class);

job.setMapperClass(PartitionerMapper.class);  
job.setPartitionerClass(AgePartitioner.class);  
job.setReducerClass(PartitionerReducer.class);

// the number of reducers is set to 3, this can be altered according to  
// the program’s requirements  
job.setNumReduceTasks(3);

// configure output  
TextOutputFormat.setOutputPath(job, new Path(args[1]));  
job.setOutputFormatClass(TextOutputFormat.class);  
job.setOutputKeyClass(Text.class);  
job.setOutputValueClass(Text.class);

return job.waitForCompletion(true) ? 0 : 1;  
}  
public static void main(String[] args) throws Exception {  
int exitCode = ToolRunner.run(new PartitionerDriver(), args);  
System.exit(exitCode);  
}  
}

[/code]

## **PartitionerMapper.Java**

[code]

package com.hadoop;

import java.io.IOException;  
//import java.util.StringTokenizer;

//import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.LongWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Mapper;

public class PartitionerMapper extends Mapper<LongWritable, Text, Text, Text> {

//private final static IntWritable countOne = new IntWritable(1);  
//private final Text reusableText = new Text();

@Override  
protected void map(LongWritable key, Text value, Context context) throws IOException,  
InterruptedException {  
//sample record  
//name<tab>age<tab>gender<tab>salary  
//Raju<tab>23<tab>male<tab>5000  
//Rani<tab>21<tab>female<tab>50000  
String[] tokens = value.toString().split(“t”);  
String gender = tokens[2].toString();  
String nameAgeSalary = tokens[0]+”t”+tokens[1]+”t”+tokens[3];

//the mapper emits key, value pair where the key is the gender and the value is the other information which includes name, age and score

context.write(new Text(gender), new Text(nameAgeSalary));  
}  
}

[/code]

## **AgePartitioner.Java**

[code]

package com.hadoop;

import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Partitioner;

//AgePartitioner is a custom Partitioner to partition the data according to age.  
//The age is a part of the value from the input file.  
//The data is partitioned based on the range of the age.  
//In this example, there are 3 partitions, the first partition contains the information where the age is less than 20  
//The second partition contains data with age ranging between 20 and 50 and the third partition contains data where the age is >50.

public class AgePartitioner extends Partitioner<Text, Text> {  
@Override  
public int getPartition(Text key, Text value, int numReduceTasks) {

String [] nameAgeSalary = value.toString().split(“t”);

String age = nameAgeSalary[1];

int ageInt = Integer.parseInt(age);  
//this is done to avoid performing mod with 0  
if(numReduceTasks == 0)  
return 0;  
//if the age is <20, assign partition 0  
if(ageInt <=20){  
return 0;  
}  
//else if the age is between 20 and 50, assign partition 1  
if(ageInt >20 && ageInt <=50){  
return 1 % numReduceTasks;  
}  
//otherwise assign partition 2  
else  
return 2 % numReduceTasks;

}

}

[/code]

## **PartitionerReducer.Java**

[code]

package com.hadoop;

import java.io.IOException;

//import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Reducer;

public class PartitionerReducer extends Reducer<Text, Text, Text, Text> {  
@Override  
protected void reduce(Text key, Iterable<Text> values, Context context)  
throws IOException, InterruptedException {

int maxSalary = Integer.MIN\_VALUE;  
String name = ” “;  
String age = ” “;  
String gender = ” “;  
int salary = 0;

//iterating through the values corresponding to a particular key  
for(Text val: values){  
String [] valTokens = val.toString().split(“t”);  
salary = Integer.parseInt(valTokens[2]);  
//if the new salary is greater than the current maximum salary, update the fields as they will be the output of the reducer after all the values are processed for a particular key

if(salary > maxSalary){  
name = valTokens[0];  
age = valTokens[1];  
gender = key.toString();  
maxSalary = salary;  
}  
}  
context.write(new Text(name), new Text(“age- “+age+”t”+gender+”tscore-“+maxSalary));  
}  
}

[/code]

Hope this helps some one looking for Custom Partitioner Example.

10)How to process hole file as single record

Answer:--

# [**Reading Complete File in Mapreduce - WholeFileInputFormat.**](http://bigdatathinker.blogspot.com/2014/01/reading-complete-file-in-mapreduce.html)

Lately I have come across a requirement where I had to read complete file in hdfs to process them. I did not find it so easily on google to complete my task. So I thought this might help you all if you required to do same thing.  
  
Definitely we need to write a custom input format to meet our requirement. In our input format below we return isSplitable() method to return false, which implies that we don't want to split a file. So one file will be processed by only one mapper. We also ensure that our custom recordreader is used, which does actual job or reading a complete file content at a time.

import java.io.IOException;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.BytesWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.InputSplit;

import org.apache.hadoop.mapreduce.JobContext;

import org.apache.hadoop.mapreduce.RecordReader;

import org.apache.hadoop.mapreduce.TaskAttemptContext;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

public class WholeFileInputFormat extends FileInputFormat<Text, BytesWritable>{

@Override

protected boolean isSplitable(JobContext context, Path file){

System.out.println("set isplitable");

return false;

}

@Override

public RecordReader<Text, BytesWritable> createRecordReader

(InputSplit split,TaskAttemptContext context) throws IOException,

InterruptedException {

wholefileRecordReader reader = new wholefileRecordReader();

reader.initialize(split,context);

return reader;

}

}

We need to customise RecordReader of our inputformat. Below is the the class wholefileRecordReader which extends RecordReader. In this case my my key to mapper was filename i.e text and value was complete content of file in BytesWritable type.  
  
Obser in nextKeyValue() method I am using IOUtils.readFully(in, contents, 0, contents.length) line which actually copies all the content in file to 'contents' variable which is our value to mapper. In mapper we can process the content in file as we wish to.

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.FSDataInputStream;

import org.apache.hadoop.fs.FileSystem;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.BytesWritable;

import org.apache.hadoop.io.IOUtils;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.InputSplit;

import org.apache.hadoop.mapreduce.RecordReader;

import org.apache.hadoop.mapreduce.TaskAttemptContext;

import org.apache.hadoop.mapreduce.lib.input.FileSplit;

class wholefileRecordReader extends RecordReader<Text, BytesWritable>

{

private FileSplit fileSplit;

private Configuration conf;

private byte[] value;

private boolean processed = false;

public void initialize(InputSplit split, TaskAttemptContext context)

throws IOException, InterruptedException

{

this.fileSplit = (FileSplit) split;

this.conf = context.getConfiguration();

System.out.println("initialize in whole record reader");

}

@Override

public boolean nextKeyValue() throws IOException, InterruptedException {

if (!processed)

{

System.out.println("next key value");

byte[] contents = new byte[(int) fileSplit.getLength()];

Path file = fileSplit.getPath();

FileSystem fs = file.getFileSystem(conf);

FSDataInputStream in = null;

//FileInputStream in = null;

try

{

in = fs.open(file);

IOUtils.readFully(in, contents, 0, contents.length);

value = contents;

}

finally

{

IOUtils.closeStream(in);

}

processed = true;

return true;

}

return false;

}

@Override

public Text getCurrentKey() throws IOException, InterruptedException {

return new Text(this.fileSplit.getPath().getName());

}

@Override

public BytesWritable getCurrentValue() throws IOException,

InterruptedException

{

return new BytesWritable(value);

}

@Override

public float getProgress() throws IOException

{

return processed ? 1.0f : 0.0f;

}

@Override

public void close() throws IOException {

// do nothing

}

}

# **Process Small Files on Hadoop Using CombineFileInputFormat (1)**

SEP 22ND, 2013

Processing small files is an old typical problem in hadoop; On [Stack Overflow](http://stackoverflow.com/questions/14541759/how-can-i-work-with-large-number-of-small-files-in-hadoop) it suggested people to use [CombineFileInputFormat](http://hadoop.apache.org/docs/r1.1.1/api/org/apache/hadoop/mapred/lib/CombineFileInputFormat.html), but I haven’t found a good step-to-step article that teach you how to use it. So, I decided to write one myself.

From [Cloudera’s blog](http://blog.cloudera.com/blog/2009/02/the-small-files-problem/):

*A small file is one which is significantly smaller than the HDFS block size (default 64MB). If you’re storing small files, then you probably have lots of them (otherwise you wouldn’t turn to Hadoop), and the problem is that HDFS can’t handle lots of files.*

In my benchmark, just using a custom CombineFileInputFormat can speedup the program from 3 hours to 23 minutes, and after some further tuning, the same task can be run in 6 minutes!

## **Benchmark Setup**

To test the raw performance of different approaches to solve small problems, I setup a map only hadoop job that basically just do grep and perform a small binary search. The binary search part is to generate the reduce side keys that I’ll use in further data processing; it took only a little resource (8MB index) to run, so it does not affect the result of the benchmark.

The data to process is some server log data, 53.1 GB in total. The hadoop clusters consist 6 nodes, using hadoop version 1.1.2. In this benchmark I implemented CombineFileInputFormat to shrink the map jobs; I also tested the difference of reusing JVM or not, and different number of block sizes to combine files.

## **CombineFileInputFormat**

The code listed here is modified from [Hadoop example code](https://svn.apache.org/repos/asf/hadoop/common/trunk/hadoop-mapreduce-project/hadoop-mapreduce-examples/src/main/java/org/apache/hadoop/examples/MultiFileWordCount.java). To use CombineFileInputFormat you need to implement three classes. The class CombineFileInputFormat is an abstract class with no implementation, so you must create a subclass to support it; we’ll name the subclass CFInputFormat. The subclass will initiate a delegate CFRecordReader that extends RecordReader; this is the code that does the file processing logic. We’ll also need a class for FileLineWritable, which replaces LongWritable normally used as a key to file lines.

### CFInputFormat.java

The CFInputFormat.java doesn’t do much. You implement createRecordReader to pass in the record reader that does the combine file logic, that’s all. Note that you can call setMaxSplitSize in the initializer to control the size of each chunk of files; if you don’t want to split files into half, remember to return false in isSplitable method, which defaults to true.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | package org.idryman.combinefiles;  import java.io.IOException;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.InputSplit;  import org.apache.hadoop.mapreduce.JobContext;  import org.apache.hadoop.mapreduce.RecordReader;  import org.apache.hadoop.mapreduce.TaskAttemptContext;  import org.apache.hadoop.mapreduce.lib.input.CombineFileInputFormat;  import org.apache.hadoop.mapreduce.lib.input.CombineFileRecordReader;  import org.apache.hadoop.mapreduce.lib.input.CombineFileSplit;  import org.idryman.combinefiles.CFRecordReader;  import org.idryman.combinefiles.FileLineWritable;  public class CFInputFormat extends CombineFileInputFormat<FileLineWritable, Text> {  public CFInputFormat(){  super();  setMaxSplitSize(67108864); // 64 MB, default block size on hadoop  }  public RecordReader<FileLineWritable, Text> createRecordReader(InputSplit split, TaskAttemptContext context) throws IOException{  return new CombineFileRecordReader<FileLineWritable, Text>((CombineFileSplit)split, context, CFRecordReader.class);  }  @Override  protected boolean isSplitable(JobContext context, Path file){  return false;  }  } |

### CFRecordReader.java

CFRecordReader is a delegate class of CombineFileRecordReader, a built in class that pass each split (typically a whole file in this case) to our class CFRecordReader. When the hadoop job starts, CombineFileRecordReader reads all the file sizes in HDFS that we want it to process, and decides how many splits base on the MaxSplitSize we defined in CFInputFormat. For every split (must be a file, because we set isSplitabe to false), CombineFileRecordReader creates a CFRecrodReader instance via a custom constructor, and pass in CombineFileSplit, context, and index for CFRecordReader to locate the file to process with.

When processing the file, the CFRecordReader creates a FileLineWritable as the key for hadoop mapper class. With each line a FileLineWritable consists the file name and the offset length of that line. The difference between FileLineWritable and the normally used LongWritable in mapper is LongWritable only denote the offset of a line in a file, while FileLineWritable adds the file information into the key.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91 | package org.idryman.combinefiles;  import java.io.IOException;  import org.idryman.combinefiles.FileLineWritable;  import org.apache.hadoop.fs.FSDataInputStream;  import org.apache.hadoop.fs.FileSystem;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.InputSplit;  import org.apache.hadoop.mapreduce.RecordReader;  import org.apache.hadoop.mapreduce.TaskAttemptContext;  import org.apache.hadoop.mapreduce.lib.input.CombineFileSplit;  import org.apache.hadoop.util.LineReader;  public class CFRecordReader extends RecordReader<FileLineWritable, Text>{  private long startOffset;  private long end;  private long pos;  private FileSystem fs;  private Path path;  private FileLineWritable key;  private Text value;  private FSDataInputStream fileIn;  private LineReader reader;  public CFRecordReader(CombineFileSplit split, TaskAttemptContext context, Integer index) throws IOException{  this.path = split.getPath(index);  fs = this.path.getFileSystem(context.getConfiguration());  this.startOffset = split.getOffset(index);  this.end = startOffset + split.getLength(index);  fileIn = fs.open(path);  reader = new LineReader(fileIn);  this.pos = startOffset;  }  @Override  public void initialize(InputSplit arg0, TaskAttemptContext arg1)  throws IOException, InterruptedException {  // Won't be called, use custom Constructor  // `CFRecordReader(CombineFileSplit split, TaskAttemptContext context, Integer index)`  // instead  }  @Override  public void close() throws IOException {}  @Override  public float getProgress() throws IOException{  if (startOffset == end) {  return 0;  }  return Math.min(1.0f, (pos - startOffset) / (float) (end - startOffset));  }  @Override  public FileLineWritable getCurrentKey() throws IOException, InterruptedException {  return key;  }  @Override  public Text getCurrentValue() throws IOException, InterruptedException {  return value;  }  @Override  public boolean nextKeyValue() throws IOException{  if (key == null) {  key = new FileLineWritable();  key.fileName = path.getName();  }  key.offset = pos;  if (value == null){  value = new Text();  }  int newSize = 0;  if (pos < end) {  newSize = reader.readLine(value);  pos += newSize;  }  if (newSize == 0) {  key = null;  value = null;  return false;  } else{  return true;  }  }  } |

The reason to use a custom constructor is not documented anywhere in hadoop api nor document. You can only find it in [hadoop source code](http://grepcode.com/file/repo1.maven.org/maven2/com.ning/metrics.collector/1.2.1/org/apache/hadoop/mapreduce/lib/input/CombineFileRecordReader.java#40), line 40:

|  |  |
| --- | --- |
| 1  2  3  4 | static final Class [] constructorSignature = new Class []  {CombineFileSplit.class,  TaskAttemptContext.class,  Integer.class}; |

### FileLineWritable.java

This file is very simple: store the file name and offset, and override the compareTo method to compare the file name first, then compare the offset.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57 | package org.idryman.combinefiles;  import java.io.DataInput;  import java.io.DataOutput;  import java.io.IOException;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.io.WritableComparable;  public class FileLineWritable implements WritableComparable<FileLineWritable>{  public long offset;  public String fileName;  public void readFields(DataInput in) throws IOException {  this.offset = in.readLong();  this.fileName = Text.readString(in);  }  public void write(DataOutput out) throws IOException {  out.writeLong(offset);  Text.writeString(out, fileName);  }  public int compareTo(FileLineWritable that) {  int cmp = this.fileName.compareTo(that.fileName);  if (cmp != 0) return cmp;  return (int)Math.signum((double)(this.offset - that.offset));  }  @Override  public int hashCode() { // generated hashCode()  final int prime = 31;  int result = 1;  result = prime \* result + ((fileName == null) ? 0 : fileName.hashCode());  result = prime \* result + (int) (offset ^ (offset >>> 32));  return result;  }  @Override  public boolean equals(Object obj) { // generated equals()  if (this == obj)  return true;  if (obj == null)  return false;  if (getClass() != obj.getClass())  return false;  FileLineWritable other = (CFFileLineWritableInputFormat) obj;  if (fileName == null) {  if (other.fileName != null)  return false;  } else if (!fileName.equals(other.fileName))  return false;  if (offset != other.offset)  return false;  return true;  }  } |

## **job setup**

Finally is the job setup for hadoop cluster to run. We just need to assign the classes to job:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | import org.apache.hadoop.mapreduce.Job;  // standard hadoop conf  Job job = new Job(getConf());  FileInputFormat.addInputPath(job, new Path(args[0]));  job.setInputFormatClass(CFInputFormat.class);  job.setMapperClass(MyMapper.class);  job.setNumReduceTasks(0); // map only  FileOutputFormat.setOutputPath(job, new Path(args[1]));  job.submit(); |

The benchmark result is in the next post.

# Hadoop: RecordReader and FileInputFormat

[27 MAY 2013](https://hadoopi.wordpress.com/2013/05/27/understand-recordreader-inputsplit/) / [ANTOINE AMEND](https://hadoopi.wordpress.com/author/aamend/)

Today’s new challenge…  
I want to create a custom MapReduce job that can handle more than 1 single line at a time. Actually, it took me some time to understand the implementation of default **LineRecordReader** class, not because of its implementation Vs. my Java skill set, but rather that I was not familiar with its concept. I am describing in this article my understanding on this implementation.

As **InputSplit** is nothing more than a chunk of 1 or several blocks, it should be pretty rare to get a block boundary ending up at the exact location of a end of line (EOL). Some of my records located around block boundaries should be therefore split in 2 different blocks. This triggers the following issues:

1. How Hadoop can guarantee lines read are 100% complete ?
2. How Hadoop can consolidate a line that is starting on block B and that ends up on B+1 ?
3. How Hadoop can guarantee we do not miss any line ?
4. Is there a limitation in term of line’s size ? Can a line be greater than a block (i.e. spanned over more than 2 blocks) ? If so, is there any consequence in term of MapReduce performance ?

## Definitions

### InputFormat

Definition taken from

Hadoop relies on the input format of the job to do three things:  
1. Validate the input configuration for the job (i.e., checking that the data is there).  
2. Split the input blocks and files into logical chunks of type InputSplit, each of which is assigned to a map task for processing.  
3. Create the RecordReader implementation to be used to create key/value pairs from the raw InputSplit. These pairs are sent one by one to their mapper.

### RecordReader

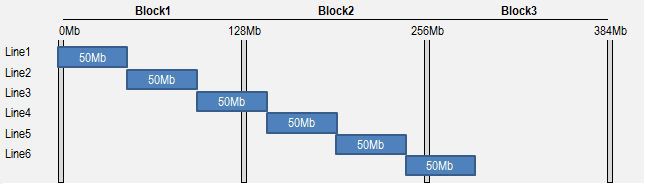
Definition taken from

A RecordReader uses the data within the boundaries created by the input split to generate key/value pairs. In the context of file-based input, the “start” is the byte position in the file where the RecordReader should start generating key/value pairs. The “end” is where it should stop reading records. These are not hard boundaries as far as the API is concerned—there is nothing stopping a developer from reading the entire file for each map task. While reading the entire file is not advised, reading outside of the boundaries it often necessary to ensure that a complete record is generated

## Example

I jumped right into the code of **LineRecordReader** and found it not that obvious to understand. Let’s get an example first that will hopefully make the code slightly more readable.  
Suppose my data set is composed on a single 300Mb file, spanned over 3 different blocks (blocks of 128Mb), and suppose that I have been able to get 1 InputSplit for each block. Let’s imagine now 3 different scenarios.

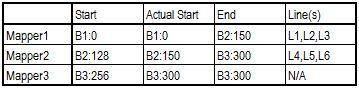
### File is composed on 6 lines of 50Mb each

[](https://hadoopi.files.wordpress.com/2013/05/inputsplit1.jpg)

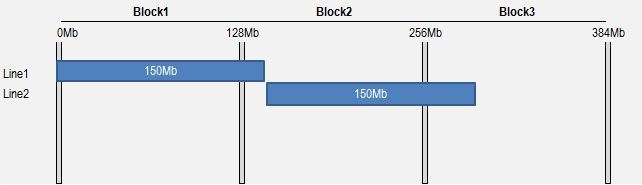
* The first Reader will start reading bytes from Block B1, position 0. The first two EOL will be met at respectively 50Mb and 100Mb. 2 lines (L1 & L2) will be read and sent as key / value pairs to Mapper 1 instance. Then, starting from byte 100Mb, we will reach end of our Split (128Mb) before having found the third EOL. This incomplete line will be completed by reading the bytes in Block B2 until position 150Mb. First part of Line L3 will be read locallyfrom Block B1, second part will be read remotelyfrom Block B2 (by the mean of **FSDataInputStream**), and a complete record will be finally sent as key / value to Mapper 1.
* The second Reader starts on Block B2, at position 128Mb. Because 128Mb is not the start of a file, there are strong chance our pointer is located somewhere in an existing record that has been already processed by previous Reader. We need to skip this record by jumping out to the next available EOL, found at position 150Mb. Actual start of RecordReader 2 will be at 150Mb instead of 128Mb.

We can wonder what happens in case a block starts exactly **on** a EOL. By jumping out until the next available record (through readLine method), we might miss 1 record. Before jumping to next EOL, we actually need to decrement initial “start” value to “start – 1”. Being located at at least 1 offset before EOL, we ensure no record is skipped !

Remaining process is following same logic, and everything is summarized in below table.

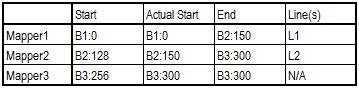
[](https://hadoopi.files.wordpress.com/2013/05/inputsplit_meta1.jpg)

### File composed on 2 lines of 150Mb each

[](https://hadoopi.files.wordpress.com/2013/05/inputsplit2.jpg)

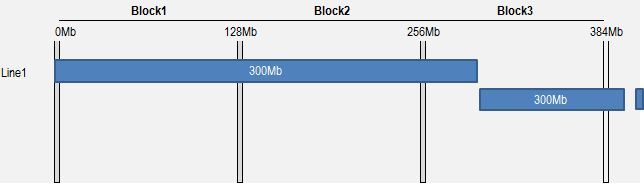
Same process as before:

* Reader 1 will start reading from block B1, position 0. It will read line L1 locallyuntil end of its split (128Mb), and will then continue reading remotelyon B2 until EOL (150Mb)
* Reader 2 will not start reading from 128Mb, but from 150Mb, and until B3:300

[](https://hadoopi.files.wordpress.com/2013/05/inputsplit_meta2.jpg)

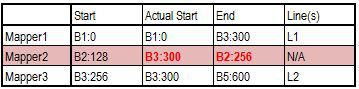
### File composed on 2 lines of 300Mb each

OK, this one is a tricky and perhaps unrealistic example, but I was wondering what happens in case a record is larger than 2 blocks (spanned over at least 3 blocks).

[](https://hadoopi.files.wordpress.com/2013/05/inputsplit5.jpg)

* Reader 1 will start reading locallyfrom B1:0 until B1:128, then remotelyall bytes available on B2, and finally remotely on B3 until EOL is reached (300Mb). There is here some overhead as we’re trying to read a lot of data that is not locally available
* Reader 2 will start reading from B2:128 and will jump out to next available record located at B3:300. Its new start position (B3:300) is actually greater than its maximum position (B2:256). This reader will therefore not provide Mapper 2 with any key / value. I understand it somehow as a kind of security feature ensuring data locality (that makes Hadoop so efficient in data processing) is preserved (i.e. Do not process a line that is not starting in the chunk I’m responsible for).
* Reader 3 will start reading from B3:300 to B5:600

This is summarized in below table

[](https://hadoopi.files.wordpress.com/2013/05/inputsplit_meta51.jpg)

#### Maximum size for a single record

There is a maximum size allowed for a single record to be processed. This value can be set using below parameter.

|  |  |
| --- | --- |
| 1 | conf.setInt("mapred.linerecordreader.maxlength", Integer.MAX\_VALUE); |

A line with a size greater than this maximum value (default is 2,147,483,647) will be ignored.

I hope these 3 examples gives you a high level understanding on **RecordReader** and **InputFormat**. If so, let’s jump to the code, else, let me know.

I doubt a single record is hundreds of Mb large (300Mb in my example) in a real environment… With hundreds of Kb for a single record, the overhead due to a line spanning over different blocks should not be that significant, and overall performance should not be really affected

## Implementation

### RecordReader

I added some (a tons of) comments in the code in order to point out what has been previously said in the example section. Hopefully this makes it slightly clearer. A new Reader must extends class **RecordReader** and override several methods.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188 | public class CustomLineRecordReader      extends RecordReader<LongWritable, Text> {        private long start;      private long pos;      private long end;      private LineReader in;      private int maxLineLength;      private LongWritable key = new LongWritable();      private Text value = new Text();        private static final Log LOG = LogFactory.getLog(              CustomLineRecordReader.class);        /\*\*       \* From Design Pattern, O'Reilly...       \* This method takes as arguments the map task’s assigned InputSplit and       \* TaskAttemptContext, and prepares the record reader. For file-based input       \* formats, this is a good place to seek to the byte position in the file to       \* begin reading.       \*/      @Override      public void initialize(              InputSplit genericSplit,              TaskAttemptContext context)              throws IOException {            // This InputSplit is a FileInputSplit          FileSplit split = (FileSplit) genericSplit;            // Retrieve configuration, and Max allowed          // bytes for a single record          Configuration job = context.getConfiguration();          this.maxLineLength = job.getInt(                  "mapred.linerecordreader.maxlength",                  Integer.MAX\_VALUE);            // Split "S" is responsible for all records          // starting from "start" and "end" positions          start = split.getStart();          end = start + split.getLength();            // Retrieve file containing Split "S"          final Path file = split.getPath();          FileSystem fs = file.getFileSystem(job);          FSDataInputStream fileIn = fs.open(split.getPath());            // If Split "S" starts at byte 0, first line will be processed          // If Split "S" does not start at byte 0, first line has been already          // processed by "S-1" and therefore needs to be silently ignored          boolean skipFirstLine = false;          if (start != 0) {              skipFirstLine = true;              // Set the file pointer at "start - 1" position.              // This is to make sure we won't miss any line              // It could happen if "start" is located on a EOL              --start;              fileIn.seek(start);          }            in = new LineReader(fileIn, job);            // If first line needs to be skipped, read first line          // and stores its content to a dummy Text          if (skipFirstLine) {              Text dummy = new Text();              // Reset "start" to "start + line offset"              start += in.readLine(dummy, 0,                      (int) Math.min(                              (long) Integer.MAX\_VALUE,                              end - start));          }            // Position is the actual start          this.pos = start;        }        /\*\*       \* From Design Pattern, O'Reilly...       \* Like the corresponding method of the InputFormat class, this reads a       \* single key/ value pair and returns true until the data is consumed.       \*/      @Override      public boolean nextKeyValue() throws IOException {            // Current offset is the key          key.set(pos);            int newSize = 0;            // Make sure we get at least one record that starts in this Split          while (pos < end) {                // Read first line and store its content to "value"              newSize = in.readLine(value, maxLineLength,                      Math.max((int) Math.min(                              Integer.MAX\_VALUE, end - pos),                              maxLineLength));                // No byte read, seems that we reached end of Split              // Break and return false (no key / value)              if (newSize == 0) {                  break;              }                // Line is read, new position is set              pos += newSize;                // Line is lower than Maximum record line size              // break and return true (found key / value)              if (newSize < maxLineLength) {                  break;              }                // Line is too long              // Try again with position = position + line offset,              // i.e. ignore line and go to next one              // TODO: Shouldn't it be LOG.error instead ??              LOG.info("Skipped line of size " +                      newSize + " at pos "                      + (pos - newSize));          }              if (newSize == 0) {              // We've reached end of Split              key = null;              value = null;              return false;          } else {              // Tell Hadoop a new line has been found              // key / value will be retrieved by              // getCurrentKey getCurrentValue methods              return true;          }      }        /\*\*       \* From Design Pattern, O'Reilly...       \* This methods are used by the framework to give generated key/value pairs       \* to an implementation of Mapper. Be sure to reuse the objects returned by       \* these methods if at all possible!       \*/      @Override      public LongWritable getCurrentKey() throws IOException,              InterruptedException {          return key;      }        /\*\*       \* From Design Pattern, O'Reilly...       \* This methods are used by the framework to give generated key/value pairs       \* to an implementation of Mapper. Be sure to reuse the objects returned by       \* these methods if at all possible!       \*/      @Override      public Text getCurrentValue() throws IOException, InterruptedException {          return value;      }        /\*\*       \* From Design Pattern, O'Reilly...       \* Like the corresponding method of the InputFormat class, this is an       \* optional method used by the framework for metrics gathering.       \*/      @Override      public float getProgress() throws IOException, InterruptedException {          if (start == end) {              return 0.0f;          } else {              return Math.min(1.0f, (pos - start) / (float) (end - start));          }      }        /\*\*       \* From Design Pattern, O'Reilly...       \* This method is used by the framework for cleanup after there are no more       \* key/value pairs to process.       \*/      @Override      public void close() throws IOException {          if (in != null) {              in.close();          }      }    } |

### FileInputFormat

Now that you have created a custom Reader, you need to use it from a class extending **FileInputFormat**, as reported below …

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class CustomFileInputFormat extends FileInputFormat<LongWritable,Text>{        @Override      public RecordReader<LongWritable, Text> createRecordReader(              InputSplit split, TaskAttemptContext context) throws IOException,              InterruptedException {          return new CustomLineRecordReader();      }  } |

### MapReduce

… and to use this new **CustomFileInputFormat** in your MapReduce driver code when specifying Input format.

|  |  |
| --- | --- |
| 1  2  3  4 | .../...  FileInputFormat.addInputPath(job, inputPath);  job.setInputFormatClass(CustomFileInputFormat.class);  .../... |

Congratulations, if you followed this article you have just re-invented the wheel. We did not do anything more that re-implementing **LineRecordReader**and **FileInputFormat**, default implementations for Text file. However, I hope you now understand a bit better how these 2 classes works, allowing you to create your custom Reader and therefore being able to handle specific file format.

I hope you liked this article, that it was not too high-level and therefore not a waste of time..  
Should you have any question / remarks / suggestions, feel free to comment. Feel also free to share it !

Cheers !

12) two rwos as input

Anser)

You'll need to write your own input format and record reader to ensure proper file splitting around your record delimiter.

Basically your record reader will need to seek to it's split byte offset, scan forward (read lines) until it finds either:

* the Begin ... line
  + Read lines upto the next end ... line and provide these lines between the begin and end as input for the next record
* It scans pasts the end of the split or finds EOF

This is similar in algorithm to how Mahout's [XMLInputFormat](https://github.com/apache/mahout/blob/mahout-0.1/examples/src/main/java/org/apache/mahout/classifier/bayes/XmlInputFormat.java) handles multi line XML as input - in fact you might be able to amend this source code directly to handle your situation.

As mentioned in @irW's answer, NLineInputFormat is another option if your records have a fixed number of lines per record, but is really inefficient for larger files as it has to open and read the entire file to discover the line offsets in the input format's getSplits() method.

# Hadoop: Custom RecordReader – Processing String / Pattern delimited records

[31 MAY 2013](https://hadoopi.wordpress.com/2013/05/31/custom-recordreader-processing-string-pattern-delimited-records/) / [ANTOINE AMEND](https://hadoopi.wordpress.com/author/aamend/)

Now that both **InputFormat** and **RecordReader** are familiar concepts for you (if not, you can still refer to article [Hadoop RecordReader and FileInputFormat](https://hadoopi.wordpress.com/2013/05/27/understand-recordreader-inputsplit/)), it is time to enter into the heart of the subject.

The default implementation of **TextInputFormat** is based on a Line-by-Line approach. Each line found in data set will be supplied to MapReduce framework as a set of key / value. Should you need to handle more than 1 line at a time, you can quite easily implement your own NLinesRecordReader (refer to this good article – [bigdatacircus](http://bigdatacircus.com/2012/08/01/wordcount-with-custom-record-reader-of-textinputformat/)), but…

* What if all your records do not have the same number of lines ?
* How to process Record-by-Record instead of Line-by-Line ?

Should you need to process your data set based on a Record-by-Record approach, distinct records must be obviously separated by a common delimiter. This delimiter could be either a line (common String) or a common pattern.

## String delimited records

### Data set example

Take the below example of a (dummy) data set where all your records are separated by a same String.

**----------**

pleff lorem monaq morel plaff lerom baple merol pliff ipsum ponaq mipsu ploff pimsu

caple supim pluff sumip qonaq issum daple ussum ronaq ossom fap25 abcde tonaq fghij

merol pliff ipsum ponaq mipsu ploff pimsu caple supim pluff sumip qonaq issum daple

ussum ronaq ossom faple abc75 tonaq fghij gaple klmno vonaq pqrst haple uvwxy nonaq

**----------**

zzzzz laple pleff lorem monaq morel plaff sumip qonaq issum daple ussum ponaq gapl

Klmno pm100 pleff lorem monaq morel plaff lerom baple merol pliff ipsum ponaq mipsu

ploff pimsu caple supim pluff sumip qonaq issum daple ussum ronaq ossom fa125 abcde

**----------**

lerom baple merol pliff ipsum ponaq mipsu ploff pimsu caple supim pluff sumih Qonaq

### Implementation

In that case (records are always separated by a same “10-dash” String), the implementation is somehow out of the box. Indeed, default **LineReader** can take as an argument a recordDelimiterBytesbyte array that can be retrieved / set directly from the Hadoop configuration. This parameter will be used as a String delimiter to separate distinct records.

Just make sure to set it up in your MapReduce driver code

|  |  |
| --- | --- |
| 1  2 | Configuration conf = new Configuration(true);  conf.set("textinputformat.record.delimiter","------------"); |

…and to specify the default **TextInputFormat** for your MapReduce job’s InputFormat.

|  |  |
| --- | --- |
| 1  2 | Job job = new Job(conf);  job.setInputFormat(TextInputFormat.class); |

Instead of processing 1 given line at a time, you should be able to process a full NLines record. Will be supplied to your mappers instances the following keys / values:

* **Key**is the offset (location of your record’s first line)
* **Value**is the record itself

Note that the default delimiter is CRLF (additionally CR) character. Using the Hadoop default configuration, **LineReader** can be seen as a Record-by-Record reader that uses a CRLF delimiter, thus equivalent to a Line-by-Line reader actually.

## Important update

Contrary to what is stated there on [JIRA](https://issues.apache.org/jira/browse/HADOOP-7096), custom delimiter (provided by “textinputformat.record.delimiter” parameter) is not supported on version 1.2.1 of Hadoop. However, you can still create your own record reader to handle that particular case. Have a look on my github account ([hadoop-recordreader](https://github.com/aamend/hadoop-recordreader)). See Delimiter.java that uses CustomFileInputFormat.java

## Pattern delimited records

### Data set example

Take now the following data set structure. Records are not separated by a common String anymore, but rather by a common pattern (DateTime). A String cannot be used here, so you will have to create your own RecordReader that splits records using a Regular Expression.

**Sat, 25 May 2013 22:29:30**

pleff lorem monaq morel plaff lerom baple merol pliff ipsum ponaq mipsu ploff pimsu

caple supim pluff sumip qonaq issum daple ussum ronaq ossom fap25 abcde tonaq fghij

merol pliff ipsum ponaq mipsu ploff pimsu caple supim pluff sumip qonaq issum daple

ussum ronaq ossom faple abc75 tonaq fghij gaple klmno vonaq pqrst haple uvwxy nonaq

**Sat, 25 May 2013 22:30:30**

zzzzz laple pleff lorem monaq morel plaff sumip qonaq issum daple ussum ponaq gapl

Klmno pm100 pleff lorem monaq morel plaff lerom baple merol pliff ipsum ponaq mipsu

ploff pimsu caple supim pluff sumip qonaq issum daple ussum ronaq ossom fa125 abcde

**Sat, 25 May 2013 22:31:30**

lerom baple merol pliff ipsum ponaq mipsu ploff pimsu caple supim pluff sumih Qonaq

### PatternRecordReader

The first thing to do here is to implement a custom reader that extends the default **RecordReader** and to implement its abstract methods. Should you need to get more details on how these methods work, please refer to my previous post ([Hadoop RecordReader and FileInputFormat](https://hadoopi.wordpress.com/2013/05/27/understand-recordreader-inputsplit/)) as I will describe here only the delta compared to the default implementation.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59 | public class PatternRecordReader          extends RecordReader<LongWritable, Text> {        private LineReader in;      private final static Text EOL = new Text("\n");      private Pattern delimiterPattern;      private String delimiterRegex;      private int maxLengthRecord;        @Override      public void initialize(InputSplit split,                          TaskAttemptContext context)              throws IOException, InterruptedException {            Configuration job = context.getConfiguration();          this.delimiterRegex = job.get("record.delimiter.regex");          this.maxLengthRecord = job.getInt(                                  "mapred.linerecordreader.maxlength",                  Integer.MAX\_VALUE);            delimiterPattern = Pattern.compile(delimiterRegex);          ../..      }        private int readNext(Text text,                          int maxLineLength,                          int maxBytesToConsume)              throws IOException {            int offset = 0;          text.clear();          Text tmp = new Text();            for (int i = 0; i < maxBytesToConsume; i++) {                int offsetTmp = in.readLine(                                       tmp,                                       maxLineLength,                                       maxBytesToConsume);              offset += offsetTmp;              Matcher m = delimiterPattern.matcher(tmp.toString());                // End of File              if (offsetTmp == 0) {                  break;              }                if (m.matches()) {                  // Record delimiter                  break;              } else {                  // Append value to record                  text.append(EOL.getBytes(), 0, EOL.getLength());                  text.append(tmp.getBytes(), 0, tmp.getLength());              }          }          return offset;      }  } |

Note the following points that differs from default implementation:

* line 16: Retrieve regular expression from Hadoop configuration
* line 21: Compile this regular expression only once per InputSplit

**The actual logic is located in the readNext private method:**  
We simply get into a loop (limited by default with Integer.MAX\_VALUE value) and append every line found together with a EOL character into a final Text() until current line matches our Regular Expression delimiter. We finally return the number of bytes we have read.

In the default implementation we were reading lines by using

|  |  |
| --- | --- |
| 1  2  3  4 | newSize = in.readLine(value, maxLineLength,          Math.max((int) Math.min(                  Integer.MAX\_VALUE, end - pos),                  maxLineLength)); |

it becomes now

|  |  |
| --- | --- |
| 1  2  3  4 | newSize = readNext(value, maxLineLength,          Math.max((int) Math.min(                  Integer.MAX\_VALUE, end - pos),                  maxLineLength)); |

### PatternInputFormat

Next step is to create a custom **InputFormat**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class PatternInputFormat          extends FileInputFormat<LongWritable,Text>{        @Override      public RecordReader<LongWritable, Text> createRecordReader(              InputSplit split,              TaskAttemptContext context)                             throws IOException,                        InterruptedException {            return new PatternRecordReader();      }    } |

### Driver code

In your driver code you need to provide Hadoop framework with the regular expression you have chosen

|  |  |
| --- | --- |
| 1  2  3  4 | // regex matching pattern "Sat, 25 May 2013"  String regex = "^[A-Za-z]{3},\\s\\d{2}\\s[A-Za-z]{3}.\*";  Configuration conf = new Configuration(true);  conf.set("record.delimiter.regex", regex); |

and to use this new InputFormat

|  |  |
| --- | --- |
| 1  2 | Job job = new Job(conf);  job.setInputFormatClass(PatternInputFormat.class); |

## Mapper

I’m doing here a simple Map-only job in order to make sure all my records have been correctly separated

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public static class RecordMapper extends          Mapper<LongWritable, Text, Text, NullWritable> {        private Text out = new Text();        @Override      public void map(LongWritable key, Text value, Context context)              throws IOException, InterruptedException {            out.set(key + " -------------\n" + value);          context.write(out, NullWritable.get());      }  } |

Given the same data set as before, the Map-only job’s output is the following

10000 -------------

pleff lorem monaq morel plaff lerom baple merol pliff ipsum ponaq mipsu ploff pimsu

caple supim pluff sumip qonaq issum daple ussum ronaq ossom fap25 abcde tonaq fghij

merol pliff ipsum ponaq mipsu ploff pimsu caple supim pluff sumip qonaq issum daple

ussum ronaq ossom faple abc75 tonaq fghij gaple klmno vonaq pqrst haple uvwxy nonaq

13221 -------------

zzzzz laple pleff lorem monaq morel plaff sumip qonaq issum daple ussum ponaq gapl

Klmno pm100 pleff lorem monaq morel plaff lerom baple merol pliff ipsum ponaq mipsu

ploff pimsu caple supim pluff sumip qonaq issum daple ussum ronaq ossom fa125 abcde

15224 -------------

lerom baple merol pliff ipsum ponaq mipsu ploff pimsu caple supim pluff sumih Qonaq

## Conclusion

You are now able to process a “pseudo-unstructured” data set by reading Record-by-Record instead of Line-by-Line. This implementation might be really helpful if you need to convert rough log files into a more readable format (e.g. CSV). Instead of getting an external script that pre-process your data (e.g. Perl script) before uploading them on HDFS, you can take full benefit of the distributing computing, parsing your data set using the MapReduce framework.

I hope this article was interesting. Don’t hesitate to let me know if you have any questions.

In this hadoop tutorial we will have a look at the modification to our previous program wordcount with our own custom mapper and reducer by implementing a concept called as custom record reader. Before we attack the problem let us look at some theory required to understand the topic.

So far in the series of articles we have seen [how to create a mapreduce program without writing explicit mapper or reducer](http://analyticspro.org/2012/06/15/hadoop-a-wordcount-without-explicit-mapperreducer/) also in the second part we wrote the [wordcount with our own custom mapper and reducer](http://analyticspro.org/2012/07/24/word-count-with-custom-mapper-and-reducer/)

(Input format theory information **reference** from yahoo tutorial )

The *InputFormat* defines how to read data from a file into the Mapper instances. Hadoop comes with several implementations of *InputFormat*; some work with text files and describe different ways in which the text files can be interpreted. Others, like SequenceFileInputFormat, are purpose-built for reading particular binary file formats.

More powerfully, you can define your own *InputFormat* implementations to format the input to your programs however you want. For example, the default TextInputFormat reads lines of text files. The key it emits for each record is the byte offset of the line read (as a LongWritable), and the value is the contents of the line up to the terminating '\n' character (as a Text object). If you have multi-line records each separated by a $ character, you could write your own *InputFormat* that parses files into records split on this character instead.

Another important job of the *InputFormat* is to divide the input data sources (e.g., input files) into fragments that make up the inputs to individual map tasks. These fragments are called “splits” and are encapsulated in instances of the *InputSplit* interface. Most files, for example, are split up on the boundaries of the underlying blocks in HDFS, and are represented by instances of the FileInputSplitclass. Other files may be unsplittable, depending on application-specific data. Dividing up other data sources (e.g., tables from a database) into splits would be performed in a different, application-specific fashion. When dividing the data into input splits, it is important that this process be quick and cheap. The data itself should not need to be accessed to perform this process (as it is all done by a single machine at the start of the MapReduce job).

So in nutshell InputFormat does 2 tasks :

1. Divide the data source ( the data files ) into fragments or blocks which are sent to a mapper. These are called splits.
2. These splits are further divided into records and these records are provided one at a time to the mapper for processing. This is achieved through a class called as Record Reader

We will concentrate on customizing #2 above customizing #1 will be left for one of the next articles. By customizing record reader as in #2 above we get immense power of sending any kind of records / xml sections / JSON objects to the mapper after reading it from the source text files

Okey. Now that we understand how mapper is fed data from source files lets look at what we will try to achieve in the example program in this article.

**Problem :** We want our mapper to receive 3 records ( 3 lines ) from the source file at a time instead on 1 line as provided by default by the TextInputFormat.

**Approach :**

1. We will extend from  **TextInputFormat** class to create our own **NLinesInputFormat** .
2. We will also create our own **RecordReader** class called **NLinesRecordReader** where we will implement the logic of feeding 3 lines/records at a time.
3. We will make a change in our driver program to use our new **NLinesInputFormat** class.
4. To prove that we are really getting 3 lines at a time, instead of actually counting words ( which we already know now how to do ) , we will emit out number of lines we get in the input at a time as a key and 1 as a value , which after going through reducer will give us frequency of  each unique number of lines to the mappers.

**Example :**

**Step 1 : Creating NLinesInputFormat class as  a custom inputformat class.**

This is really straightforward, we will inherit our class from **TextInputFormat** and override **createInputFormat( )** function to use our custom record reader class **NLinesRecordReader**which we will soon write.   The sourcelisting for this follows :

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class NLinesInputFormat extends TextInputFormat{      @Override      public RecordReader<LongWritable, Text> createRecordReader(InputSplit split, TaskAttemptContext context) {          return new NLinesRecordReader();      }  } |

Now that we have our new inputformat ready lets look at creating custom record reader. This is little complicated and the source code is a modified version of hadoop’s own LineInputFormat.

**Step 2:  Creating NLinesRecordReader class as a custom RecordReader class.**

We will inherit from RecordReader class. RecordReader has 6 abstract methods which we will have to implement.

* close ( )
* getCurrentKey ( )
* getCurrentValue ( )
* getProgress ( )
* initialize ( )
* nextKeyValue ( )

The most important ones for our discussion are the **initialize** and **nextKeyvalue** functions which we will override. The initialize function will be called only once for each split so we will do setup in this function and the nextKeyValue function is called for providing records, here we will write logic so that we send 3 records in the **value**instead of default 1. Here is the source listing for the class :

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97 | public class NLinesRecordReader extends RecordReader<LongWritable, Text>{      private final int NLINESTOPROCESS = 3;      private LineReader in;      private LongWritable key;      private Text value = new Text();      private long start =0;      private long end =0;      private long pos =0;      private int maxLineLength;    @Override      public void close() throws IOException {          if (in != null) {              in.close();          }      }    @Override      public LongWritable getCurrentKey() throws IOException,InterruptedException {          return key;      }    @Override      public Text getCurrentValue() throws IOException, InterruptedException {          return value;      }    @Override      public float getProgress() throws IOException, InterruptedException {          if (start == end) {              return 0.0f;          }          else {              return Math.min(1.0f, (pos - start) / (float)(end - start));          }      }    @Override      public void initialize(InputSplit genericSplit, TaskAttemptContext context)throws IOException, InterruptedException {          FileSplit split = (FileSplit) genericSplit;          final Path file = split.getPath();          Configuration conf = context.getConfiguration();          this.maxLineLength = conf.getInt(&quot;mapred.linerecordreader.maxlength&quot;,Integer.MAX\_VALUE);          FileSystem fs = file.getFileSystem(conf);          start = split.getStart();          end= start + split.getLength();          boolean skipFirstLine = false;          FSDataInputStream filein = fs.open(split.getPath());            if (start != 0){              skipFirstLine = true;              --start;              filein.seek(start);          }          in = new LineReader(filein,conf);          if(skipFirstLine){              start += in.readLine(new Text(),0,(int)Math.min((long)Integer.MAX\_VALUE, end - start));          }          this.pos = start;      }    @Override      public boolean nextKeyValue() throws IOException, InterruptedException {          if (key == null) {              key = new LongWritable();          }          key.set(pos);          if (value == null) {              value = new Text();          }          value.clear();          final Text endline = new Text(&quot;\n&quot;);          int newSize = 0;          for(int i=0;i<NLINESTOPROCESS;i++){              Text v = new Text();              while (pos < end) {                  newSize = in.readLine(v, maxLineLength,Math.max((int)Math.min(Integer.MAX\_VALUE, end-pos),maxLineLength));                  value.append(v.getBytes(),0, v.getLength());                  value.append(endline.getBytes(),0, endline.getLength());                  if (newSize == 0) {                      break;                  }                  pos += newSize;                  if (newSize < maxLineLength) {                      break;                  }              }          }          if (newSize == 0) {              key = null;              value = null;              return false;          } else {              return true;          }      }  } |

**Step 3 : Change in driver to use new Inputformat**

Now that we have the custom record reader ready lets modify our driver to use the new input format by adding following line of code

|  |  |
| --- | --- |
| 1 | job.setInputFormatClass(NLinesInputFormat.class); |

**Step 4 : Change the mapper to emit number of lines it gets each time**  
Here is the listing; its pretty self explanatory. I am only putting listing of map function here for the listing here.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public void map(LongWritable key, Text value,Context context) throws java.io.IOException ,InterruptedException  {      String lines = value.toString();      String []lineArr = lines.split(&quot;\n&quot;);      int lcount = lineArr.length;      context.write(new Text(new Integer(lcount).toString()),new IntWritable(1));   } |

**Sample Data Input :**

I have used sample data input files of 10000 lines of following format

*Shantanu , Deo*

*Suruchi, Bhide*

*Shamika, Deo*

*…*

*Mujtaba, Ahmed*

**Sample Output from Reducer:**

*1        1*

*3        3333*

This is because our mapper got 3333 records of 3 lines each and 1 last record of 1 line.

I hope you understood the article. If you liked it please feel free to share this . Also comment.

### NLineInputFormat in Java MapReduce - Sample code

With NLineInputFormat each mapper receives fixed number of lines of input, unlike TextInputFormat and KeyValueTextInputFormat. The number of lines of input to each mapper can be controlled by setting the property, mapreduce.input.lineinputformat.linespermap in new API and mapred.line.input.format.linespermap in old API. The default value is 1.  
  
Alternatively, we can  also set as  
conf.setInt(NLineInputFormat.LINES\_PER\_MAP, 1000); // sets N value to 1000   
  
NLineInputFormat is used in applications that take a small amount of input data and run an extensive (that is, CPU-intensive) computation for it, then emit their output.   
  
Lets try this with a simple MapReduce code that takes as input Employees file with 32161 records. In the code I have set N to 1000. So each mapper will have 1000 records and hence we will have total 33 mappers launched.  
  
  
$ hadoop dfs -cat Employees | wc -l  
32161  
  
Data in the input file looks like this..  
  
Name,Position Title,Department,Employee Annual Salary  
"AARON,  ELVIA J",WATER RATE TAKER,WATER MGMNT,$87228.00  
"AARON,  JEFFERY M",POLICE OFFICER,POLICE,$75372.00  
"AARON,  KARINA",POLICE OFFICER,POLICE,$75372.00  
"AARON,  KIMBERLEI R",CHIEF CONTRACT EXPEDITER,GENERAL SERVICES,$80916.00  
"ABAD JR,  VICENTE M",CIVIL ENGINEER IV,WATER MGMNT,$99648.00  
Here is the simple program  code would look like this  
  
import java.io.IOException;  
  
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.Reducer;  
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  
import org.apache.hadoop.mapreduce.lib.input.NLineInputFormat;  
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  
import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.conf.Configured;  
import org.apache.hadoop.util.Tool;  
import org.apache.hadoop.util.ToolRunner;  
  
  
public class NlineEmp extends Configured implements Tool{  
      
    public static class NLineMapper extends Mapper<LongWritable, Text, Text, Text>{  
        @Override  
        public void map(LongWritable key, Text value,Context context) throws IOException, InterruptedException{  
            String a = value.toString();  
            String[] b = a.split(",(?=([^\"]\*\"[^\"]\*\")\*[^\"]\*$)");  
              
            if (!(b[0].equals("Name")))  
                context.write(new Text(b[0]), new Text(b[2]));  
        }  
    }  
  
    public static class NLineReducer extends Reducer<Text, Text, Text, Text>{  
        @Override  
        public void reduce(Text key, Iterable<Text> value, Context context) throws IOException, InterruptedException{  
            for (Text values : value){  
                context.write(key, values);  
            }  
        }  
    }  
      
/\*    public static class NLineEmpInputFormat extends FileInputFormat<LongWritable,Text>{  
        public static final String  LINES\_PER\_MAP = "mapreduce.input.lineinputformat.linespermap";  
          
        public RecordReader<LongWritable, Text> getRecordReader(InputSplit split, TaskAttemptContext context) throws IOException{  
            context.setStatus(split);  
            return new LineRecordReader();   
        }  
    } \*/  
      
    public int run(String[] args) throws Exception{  
        Configuration conf = new Configuration();  
        conf.setInt(NLineInputFormat.LINES\_PER\_MAP, 1000);  
          
        Job job = new Job(conf,"NLine Input Format");  
        job.setJarByClass(NlineEmp.class);  
  
        job.setMapperClass(NLineMapper.class);  
        job.setReducerClass(NLineReducer.class);  
        job.setInputFormatClass(NLineInputFormat.class);  
          
        FileInputFormat.addInputPath(job, new Path(args[0]));  
        FileOutputFormat.setOutputPath(job, new Path(args[1]));  
          
        job.setOutputKeyClass(Text.class);  
        job.setOutputValueClass(Text.class);  
          
        return job.waitForCompletion(true) ? 0:1;  
    }  
  
    public static void main(String[] args) throws Exception{  
        int exitcode = ToolRunner.run(new NlineEmp(), args);  
        System.exit(exitcode);  
    }  
}

15) Counters

Answer)

## **MapReduce Counter**

Hadoop MapReduce Counter provides a way to measure the progress or the number of operations that occur within MapReduce programs. Basically, MapReduce framework provides a number of built-in counters to measure basic I/O operations, such as FILE\_BYTES\_READ/WRITTEN and Map/Combine/Reduce input/output records. These counters are very useful especially when you evaluate some MapReduce programs. Besides, the MapReduce Counter allows users to employ your own counters. Since MapReduce Counters are automatically aggregated over Map and Reduce phases, it is one of the easiest way to investigate internal behaviors of MapReduce programs. In this post, I’m going to introduce how to use your own MapReduce Counter. The example sources described in this post are based on Hadoop 0.21 API.

## **Incrementing your counter**

For your own MapReduce counter, you first define a enum type as follow:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public static enum MATCH\_COUNTER {    INCOMING\_GRAPHS,    PRUNING\_BY\_NCV,    PRUNING\_BY\_COUNT,    PRUNING\_BY\_ISO,    ISOMORPHIC  }; |

And then, when you want to increment your own counter, you should call the incrementmethod as follows:

|  |  |
| --- | --- |
| 1 | context.getCounter(MATCH\_COUNTER.INCOMING\_GRAPHS).increment(1); |

You can accesscontext instance within setup, cleanup, map, and reduce method in Mapper or Reducer class. You can get a desired counter via calling context.getCounter method with some enum value.

## **Finding your counter**

You can get some Counters from a finished job as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Configuration conf = new Configuration();  Cluster cluster = new Cluster(conf);  Job job = Job.getInstance(cluster,conf);  result = job.waitForCompletion(true);  ...  Counters counters = job.getCounters(); |

The instance of Counters class contains all of the counters obtained from a job. So, when you want to get your own counter, you should call findCounter method with a enum type as follows:

|  |  |
| --- | --- |
| 1  2 | Counter c1 = counters.findCounter(MATCH\_COUNTER.INCOMING\_GRAPHS);  System.out.println(c1.getDisplayName()+":"+c1.getValue()); |

The below example shows how to get built-in counter groups that Hadoop provides basically.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | for (CounterGroup group : counters) {    System.out.println("\* Counter Group: " + group.getDisplayName() + " (" + group.getName() + ")");    System.out.println("  number of counters in this group: " + group.size());    for (Counter counter : group) {      System.out.println("  - " + counter.getDisplayName() + ": " + counter.getName() + ": "+counter.getValue());    }  } |

While developing a Hadoop application, we might be interested in gathering information about the data we are analyzing, like how many types of records were processed, how many invalid records were found while running the job, etc. This functionality is provided by ***Counters***in Hadoop, which are lightweight objects helping us in tracking the progress of Map and reduce tasks. 

### Built In Counters In Hadoop

Hadoop provides some inbuilt counters for every job, for purposes like counting the number of records or bytes processed. These built in counters are divided into various groups. The basic types of inbuilt counters in Hadoop are :-

**1) Map Reduce Task Counters -**These counters are used for collecting information about the various tasks in a job. An example of this type of counter is '*MAP\_INPUT\_RECORDS*'  which is used to count the number of input records read by each map task. Th output is aggregated over all the tasks in a particular job.

**2) Job Counters -**Job Counters are maintained by the JobTracker to collect statistics about the entire job. Example of this counter is '*TOTAL\_LAUNCHED\_MAPS*' which is used to count the number of map tasks that were launched over the course of a job.

### 

### Custom Java Counters

Map Reduce allows users to specify their own counters for performing their own counting operation. A custom counter is defined by a Java enum, that groups related counters. The syntax for defining a Custom Counter is:-

*enum MyCounter {*

*MISSING,*

*TOTAL*

*}*

where the counter MISSING is used to maintain the count of the missing records and the counter TOTAL is used to maintain the count of the total number of records . MyCounter represents the group of these two counters. A user can define any number of enums to group related counters.

***Note***:- Counters are global. The counter values are shared by all the map and reduce tasks across the MapReduce framework and aggregated at the end of the job across all the tasks.

### 

### Using Counters

Usage of a counter is simple. Once a counter is defined, it can be incremented.

*public void reduce(Text key, Iterable values, Context context)****throws****IOException,*  
 *InterruptedException {*

*if(missing)       //if missing condition*

*context.getCounter(MyCounter.MISSING).increment(1);*

*}*

### Dynamic Counters

Using enums, we cannot create new counters at run time. To do so, we use dynamic counter which is not defined at compile time using Java enum.

*void incCounter (Context context,Text key){*  
*context.getCounter("DynamicCounter",key.toString()).increment(1);*  
*}*

which is used to create and increment a dynamic counter by the name of "DynamicCounter".

### 

### Viewing The Counter Values

We can you the following syntax to view the counter values :-

*Counter missingCounter = job.getCounters().findCounter(MyCounter.MISSING);*

*System.out.println("Missing Counter = "+missingCounter.getValue());*

Counters are an important feature provided by the Hadoop framework. They can be used to track job progress in a very easy and flexible manner.

**Introduction to Counters**

Hadoop counters are a way to report basic statistics of a job in Hadoop.

While developing a Hadoop application, we might be interested in gathering information about the data we are analysing, like how many types of records were processed, how many invalid records were found while running the job, etc. This functionality is provided by Counters in Hadoop, which are lightweight objects helping us in tracking the progress of Map and reduce tasks.

**Built In Counters In Hadoop**  
Hadoop provides some inbuilt counters for every job, for purposes like counting the number of records or bytes processed. These built in counters are divided into various groups. The basic types of inbuilt counters in Hadoop are –

**1) Map Reduce Task Counters** – These counters are used for collecting information about the various tasks in a job. An example of this type of counter is ‘MAP\_INPUT\_RECORDS’  which is used to count the number of input records read by each map task. The output is aggregated over all the tasks in a particular job.

**2) Job Counters** – Job Counters are maintained by the JobTracker to collect statistics about the entire job. Example of this counter is ‘TOTAL\_LAUNCHED\_MAPS’ which is used to count the number of map tasks that were launched over the course of a job.

**Custom Java Counters**

Map Reduce allows users to specify their own counters for performing their own counting operation. A custom counter is defined by a Java enum, that groups related counters. The syntax for defining a Custom Counter is –

enum MyCounter {

MISSING,

TOTAL

}

where the counter MISSING is used to maintain the count of the missing records and the counter TOTAL is used to maintain the count of the total number of records . MyCounter represents the group of these two counters. A user can define any number of enums to group related counters.

Note:- Counters are global. The counter values are shared by all the map and reduce tasks across the MapReduce framework and aggregated at the end of the job across all the tasks.

**Using Counters**  
Usage of a counter is simple. Once a counter is defined, it can be incremented.

public void reduce(Text key, Iterable values, Context context) throws IOException, InterruptedException {

if(missing)       //if missing condition

context.getCounter(MyCounter.MISSING).increment(1);

}

Counters are saved in the name node for all the data nodes. It can be accessed by the context in the map as well as reduce methods. When multiple data nodes are trying to access the counter, hadoop takes care of locking mechanism automatically. Whichever job is creating the counter can only use the counter. Counter is unique for job id.

We can also use counters through the Reporter.incrCounter() method. The Reporter object is passed to the map() and reduce() methods. You call incrCounter() with the name of the counter and the amount to increment. You use uniquely named counters for each different event. When you call incrCounter() with a new counter name, that counter is initialized and takes on the increment value.

The Reporter.incrCounter()method has two signatures, depending on how you want to specify a counter’s name:

**public void incrCounter(String group, String counter, long amount)**

**public void incrCounter(Enum key, long amount)**

The first form is more general in that it allows you to specify the counter name with dynamic strings at run time. The combination of two strings, group and counter, uniquely defines a counter. When counters are reported (in the Web UI or as text at the end of a job run), counters of the same group are reported together.

The second form uses a Java enum to specify counter names, which forces you to have them defined at compile time, but it also allows for type checking. The enum’s name is used as the group string, whereas the enum’s field is used as the counter string.

### [Hadoop Fundamentals - Custom Counters](http://www.wmanalytics.io/blog/hadoop-fundamentals-custom-counters)

Written by WebbMason Analytics

**Introduction**

The Hadoop framework keeps track of a set of built-in counters that it uses to track job progress. As an example, some of the ones displayed in the summary web page in the JobTracker are: the number of bytes read by the input format, the number of launched reduce tasks, the number of data local map tasks, and the number of bytes written by the output format. There is a mechanism that allows user-defined custom counters to be tracked alongside the built-in ones using the existing counter framework.

## Creating Custom Counters

In order to create and use a custom counter, the first step is to create an Enum that will contain the names of all custom counters for a particular job. This can be achieved as follows:

enum CustomCounters {RED, BLUE}  
  
  
Inside the map or reduce task, the counter can be adjusted using the following:

if(red)  
    context.getCounter(CustomCounters.RED).increment(1); // increase the counter by 1  
else if(blue)  
    context.getCounter(CustomCounters.BLUE).increment(-1); // decrease the counter by 1

## Programmatically Retrieving Custom Counter Values

The custom counter values will be displayed alongside the built-in counter values on the summary web page for a job viewed through the JobTracker. In addition to this, the values can be accessed programmatically with the following:

long redCounterValue = job.getCounters().findCounter(CustomCounters.RED);

## Increasing the Counter Limit

The Hadoop framework does impose an upper bound on the number of counters that can be tracked per application. This is in place in order to ensure that no single job will accidentally use all available JobTracker memory during the course of its execution. While this may not be an issue for individual MapReduce jobs, some frameworks also make use of this method to keep track of their custom counters. Cascading is one such example, and the details of its Counter logic is described in [a previous blog post](http://blog.spryinc.com/2013/12/cascading-implemeting-counter-logic.html). Another example is the graph framework Giraph. In either case, if the number of Counters the framework attempts to track exceeds the limit on the number of counters (120 by default), the job will fail.

Fortunately, the property that limits the number of counters is configurable through mapred-site.xml.

In Hadoop 1.0: "mapreduce.job.counters.max".  
In Hadoop 2.0: "mapreduce.job.counters.limit".

## Where can I go for more information?

[Giraph FAQ](http://giraph.apache.org/faq.html)  
[Map/Reduce JIRA ticket to make counter limits configurable](https://issues.apache.org/jira/browse/MAPREDUCE-2835)  
[External Blog - Limiting Usage Counters In Hadoop](http://www.thecloudavenue.com/2011/12/limiting-usage-counters-in-hadoop.html)  
Or feel free to leave your question or comment below!

**Hadoop MapReduce Features:**  
They are more advanced features are available in [Hadoop](http://www.hadooptpoint.com/category/hadoop/) [MapReduce](http://www.hadooptpoint.com/category/mapreduce/) such as counters and sorting and joining datasets.

**Hadoop MapReduce Counters:**  
Counters are a useful channel for gathering statistics about the job which means it show for quality control or for application level-statistics.They are also useful for problem diagnosis.Counter values being much easier to retrieve than log output for large distributed jobs.  
                        If you aretempted to put a log message into your map or reduce task, then it is often better to see whether you can use a counter instead to record that a particular condition occurred.

**Built-in Counters:**

Hadoop should maintain a built-in counters for eveyjob,which report various metrics for your job.for example there are counters for the number of input files and records processed.

Hadoop MapReduce **Counters** are divided into**two groups**:  
1)Task Counters  
2)Job Counters

Hadoop MapReduce Counters

There are several **groups** for the **built-in counters**  
1)MapReduceTask Counters  
2)Filesystem Counters  
3)FileInput-Format Counters  
4)FileOutput-Format Counters  
5)Job Counters

**Task counters:**  
Task counters gather information about tasks over the course of their execution, and the results are aggregated over all the tasks in a job.**Task counters** are maintained by each task attempt, and periodically sent to the **tasktracker** and then to the **jobtracker**. Counter values are definitive only once a job has successfully completed. However,some counters provide useful diagnostic information as a task is progressing, and it can  
be useful to monitor them with the web UI.   
                  For example,**PHYSICAL\_MEMORY\_BYTES,VIRTUAL\_MEMORY\_BYTES,** and **COMMITTED\_HEAP\_BYTES** provide an indication of how memory usage varies over the course of a particular task attempt.

**Job counters:**

Job counters are maintained by jobtracker,which measures the job level statistics.For example, **TOTAL\_LAUNCHED\_MAPS** counts the number of map tasks that were launched over the course of a job.

**User-Defined Java Counters:**

MapReduce can allow the userdefined java counters by using java “enum” keyword.A job may define an arbitrary number of enums, each with an arbitrary number of fields. The name of the enum is the group name, and the enum’s fields are the counter names.  
**Example:**  
public class MaxTemperatureWithCounters extends Configured implements Tool {  
enum Temperature {  
MISSING,  
MALFORMED  
}  
.  
.  
System.err.println(“Ignoring possibly corrupt input: ” + value);  
reporter.incrCounter(Temperature.MALFORMED, 1);  
} else if (parser.isMissingTemperature()) {  
reporter.incrCounter(Temperature.MISSING, 1);}  
Results:  
2014/11/20 06:33:36 INFO mapred.JobClient: Air Temperature Records  
2014/11/20 06:33:36 INFO mapred.JobClient: Malformed=3——-> userdefined counters  
2014/11/20 06:33:36 INFO mapred.JobClient: Missing=661368—-> userdefined counters

16) How to kill job in running

Answer)

# **Listing current running Hadoop Jobs and Killing running Jobs**

★★★★★

★★★★

★★★

★★

★

[Avkash Chauhan - MSFT](https://social.msdn.microsoft.com/profile/Avkash+Chauhan+-+MSFT)April 1, 2012[1](https://blogs.msdn.microsoft.com/avkashchauhan/2012/04/01/listing-current-running-hadoop-jobs-and-killing-running-jobs/#comments)

* [0](https://blogs.msdn.microsoft.com/avkashchauhan/2012/04/01/listing-current-running-hadoop-jobs-and-killing-running-jobs/)
* [0](https://blogs.msdn.microsoft.com/avkashchauhan/2012/04/01/listing-current-running-hadoop-jobs-and-killing-running-jobs/)

When you have jobs running in Hadoop, you can use the map/reduce web view to list the current running jobs however what if you would need to kill any current running job because the submitted jobs started malfunctioning or in worst case scenario, the job is stuck in infinite loops. I have seen several scenarios when a submitted job got stuck in problematic states due to code defect in map/reduce job or the Hadoop cluster itself. In any of such situation, you would need to manually kill the job which is already started.

To kill a currently running Hadoop job first you need Job ID and then Kill the job using the as following commands:

* Hadoop job -list
* Hadoop job –kill <JobID>

To list current running job in Hadoop Command shell please use below command:

On Linux:      $ bin/hadoop job –list

On Windows:    HADOOP\_HOME = C:\Apps\Dist\

               HADOOP\_HOME\bin\Hadoop job list

Above command will return job details as below:

[Linux]

1 jobs currently running

JobId                  State          StartTime       UserName

job\_201203293423\_0001   1             1334506474312   avkash

[Windows]

c:\apps\dist>hadoop job -list

1 jobs currently running

JobId                  State   StartTime       UserName        Priority        SchedulingInfo

job\_201204011859\_0002   1       1333307249654   avkash       NORMAL          NA

Once you have the JobID you can use the following command to kill the job:

On Linux:      $ bin/hadoop job -kill *jobid*

On Windows:    HADOOP\_HOME = C:\Apps\Dist\

               HADOOP\_HOME\bin\Hadoop job –kill <Job\_ID>

[Windows]

**c:\apps\dist>hadoop job -kill job\_201204011859\_0002**

Killed job job\_201204011859\_0002

n fact, achieve kill the specified user job is very simple, hadoop job command itself already carries a lot of useful job management function.

+

List all the jobs on Jobtracer

hadoop job - List

Use hadoop job -kill kill jobid specified

hadoop job - kill job\_id

Combinations of the above two commands can be achieved kill off the specified user's job

for i in `hadoop job -list | grep -w username | awk '{Print $ 1}' | grep job\_` ; do hadoop job -

17) Sorting in Mareduce

Answer)

ometimes, we would like to sort the values coming into the Reducer of a Hadoop Map/Reduce (MR) Job. You can indirectly sort the values by using a combination of implementations. They are as follows.

1. Use a composite key.
2. Extend org.apache.hadoop.mapreduce.Partitioner.
3. Extend org.apache.hadoop.io.WritableComparator.

Other tutorials that explains this approach on sorting values going into a Reducer are explained in the links below. In this blog, I summarize what I have learned from the links below and also provide a self-contained example. The main difference between this blog and the links below is that I will show how to do this using the new M/R API (i.e. org.apache.hadoop.mapreduce.\*).

* <http://www.cloudera.com/blog/2011/04/simple-moving-average-secondary-sort-and-mapreduce-part-3/>
* <http://pkghosh.wordpress.com/2011/04/13/map-reduce-secondary-sort-does-it-all/>
* <http://www.riccomini.name/Topics/DistributedComputing/Hadoop/SortByValue/>

# **THE PROBLEM**

Imagine we have stock data that looks like the following. Each line represents the value of a stock at a particular time. Each value in a line is delimited by a comma. The first value is the stock symbol (i.e. GOOG), the second value is the timestamp (i.e. the number of milliseconds since January 1, 1970, 00:00:00 GMT), and the third value is the stock’s price. The data below is a toy data set. As you can see, there are 3 stock symbols: a, b, and c. The timestamps are also simple: 1, 2, 3, 4. The values are fake as well: 1.0, 2.0, 3.0, and 4.0.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | a, 1, 1.0  b, 1, 1.0  c, 1, 1.0  a, 2, 2.0  b, 2, 2.0  c, 2, 2.0  a, 3, 3.0  b, 3, 3.0  c, 3, 3.0  a, 4, 4.0  b, 4, 4.0  c, 4, 4.0 |

Let’s say we want for each stock symbol (the reducer key input, or alternatively, the mapper key output), to order the values descendingly by timestamp when they come into the reducer. How do we sort the timestamp descendingly? This problem is known as secondary sorting. Hadoop’s M/R platform sorts the keys, but not the values. (Note, Google’s M/R platform explicitly supports secondary sorting, see Lin and Dyer 2010).

# **A SOLUTION FOR SECONDARY SORTING**

## **USE A COMPOSITE KEY**

A solution for secondary sorting involves doing multiple things. First, instead of simply emitting the stock symbol as the key from the mapper, we need to emit a composite key, a key that has multiple parts. The key will have the stock symbol and timestamp. If you remember, the process for a M/R Job is as follows.

* (K1,V1) –> Map –> (K2,V2)
* (K2,List[V2]) –> Reduce –> (K3,V3)

The notation here is a little bit different from what you may be accustomed to. Instead of angles (i.e. < and >), I use parentheses to describe a key-value pair. Also, for an array, I use List[]. In actuality, it’s an Iterator.

In the toy data above, K1 will be of type LongWritable, and V1 will be of type Text. Without secondary sorting, K2 will be of type Text and V2 will be of type DoubleWritable (we simply emit the stock symbol and price from the mapper to the reducer). So, K2=symbol, and V2=price, or (K2,V2) = (symbol,price). However, if we emit such an intermediary key-value pair, secondary sorting is not possible. We have to emit a composite key, K2={symbol,timestamp}. So the intermediary key-value pair is (K2,V2) = ({symbol,timestamp},price). Note that composite data structures, such as the composite key, is held within the curly braces. Our reducer simply outputs a K3 of type Text and V3 of type Text; (K3,V3) = (symbol, price). The complete M/R job with the new composite key is shown below.

* (LongWritable,Text) –> Map –> ({symbol,timestamp},price)
* ({symbol,timestamp},List[price]) –> Reduce –> (symbol,price)

K2 is a composite key, but inside it, the symbol part/component is referred to as the “natural” key. It is the key which values will be grouped by.

## **USE A COMPOSITE KEY COMPARATOR**

The composite key comparator is where the secondary sorting takes place. It compares composite key by symbol ascendingly and timestamp descendingly. It is shown below. Notice here we sort based on symbol and timestamp. All the components of the composite key is considered.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class CompositeKeyComparator extends WritableComparator {      protected CompositeKeyComparator() {          super(StockKey.class, true);      }      @SuppressWarnings("rawtypes")      @Override      public int compare(WritableComparable w1, WritableComparable w2) {          StockKey k1 = (StockKey)w1;          StockKey k2 = (StockKey)w2;            int result = k1.getSymbol().compareTo(k2.getSymbol());          if(0 == result) {              result = -1\* k1.getTimestamp().compareTo(k2.getTimestamp());          }          return result;      }  } |

## **USE A NATURAL KEY GROUPING COMPARATOR**

The natural key group comparator “groups” values together according to the natural key. Without this component, each K2={symbol,timestamp} and its associated V2=price may go to different reducers. Notice here, we only consider the “natural” key.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class NaturalKeyGroupingComparator extends WritableComparator {      protected NaturalKeyGroupingComparator() {          super(StockKey.class, true);      }      @SuppressWarnings("rawtypes")      @Override      public int compare(WritableComparable w1, WritableComparable w2) {          StockKey k1 = (StockKey)w1;          StockKey k2 = (StockKey)w2;            return k1.getSymbol().compareTo(k2.getSymbol());      }  } |

## **USE A NATURAL KEY PARTITIONER**

The natural key partitioner uses the natural key to partition the data to the reducer(s). Again, note that here, we only consider the “natural” key.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public class NaturalKeyPartitioner extends Partitioner<StockKey, DoubleWritable> {        @Override      public int getPartition(StockKey key, DoubleWritable val, int numPartitions) {          int hash = key.getSymbol().hashCode();          int partition = hash % numPartitions;          return partition;      }    } |

## **THE M/R JOB**

Once we define the Mapper, Reducer, natural key grouping comparator, natural key partitioner, composite key comparator, and composite key, in Hadoop’s new M/R API, we may configure the Job as follows.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | public class SsJob extends Configured implements Tool {      public static void main(String[] args) throws Exception {          ToolRunner.run(new Configuration(), new SsJob(), args);      }      @Override      public int run(String[] args) throws Exception {          Configuration conf = getConf();          Job job = new Job(conf, "secondary sort");            job.setJarByClass(SsJob.class);          job.setPartitionerClass(NaturalKeyPartitioner.class);          job.setGroupingComparatorClass(NaturalKeyGroupingComparator.class);          job.setSortComparatorClass(CompositeKeyComparator.class);            job.setMapOutputKeyClass(StockKey.class);          job.setMapOutputValueClass(DoubleWritable.class);            job.setOutputKeyClass(Text.class);          job.setOutputValueClass(Text.class);            job.setInputFormatClass(TextInputFormat.class);          job.setOutputFormatClass(TextOutputFormat.class);            job.setMapperClass(SsMapper.class);          job.setReducerClass(SsReducer.class);            job.waitForCompletion(true);            return 0;      }  } |

# **SUMMARY AND CONCLUSION**

The full source code of this blog may be [downloaded here](http://www.box.com/s/22df998009068391c380). Implementing secondary sorting in Hadoop’s M/R programming paradigm is not trivial. As you can see, you need at least 4 new classes. The composite key class needs to hold the natural key and other data that you will sort on. The composite key comparator will perform the sorting of the keys (and thus values). The natural key grouping comparator will group values based on the natural key. The natural key partitioner will send values with the same natural key to the same reducer.

Have fun programming and learning! Sib ntsib dua nawb mog. Hasta luego!

I am writing this answer assuming that the reader is familiar with Map Reduce in general,   
  
I find it easier to understand non trivial concepts with the help of diagrams, so I am going to use a few diagrams to explain what secondary sorting is.  
  
**Just to run through a definition, like Sandy says**  
  
**Secondary sort is a technique that allows the MapReduce programmer to control the order that the values show up within a reduce function call.**  
  
Lets assume that our secondary sorting is on a composite key made out of Last Name and First Name.

Now lets look at the steps involved in secondary sorting 

The partitioner and the group comparator use only **natural key**, the partitioner uses it to channel all records with the same natural key to a single reducer. *This partitioning happens in the Map Phase, data from various Map tasks are received by reducers where they are****grouped****and then sent to the reduce method*. This grouping is where the group comparator comes into picture, if we would **not** have specified a custom group comparator then Hadoop would have used the default implementation which would have considered the entire composite key, which would have lead to incorrect results.  
  
Finally just reviewing the steps involved in a MR Job and relating it to secondary sorting should help us clear out the lingering doubts

# MapReduce Algorithms - Secondary Sorting

JAN 14TH, 2013

We continue with our series on implementing MapReduce algorithms found in Data-Intensive Text Processing with MapReduce book. Other posts in this series:

1. [Working Through Data-Intensive Text Processing with MapReduce](http://codingjunkie.net/text-processing-with-mapreduce-part1/)
2. [Working Through Data-Intensive Text Processing with MapReduce – Local Aggregation Part II](http://codingjunkie.net/text-processing-with-mapreduce-part-2/)
3. [Calculating A Co-Occurrence Matrix with Hadoop](http://codingjunkie.net/cooccurrence/)
4. [MapReduce Algorithms – Order Inversion](http://codingjunkie.net/order-inversion/)

This post covers the pattern of secondary sorting, found in chapter 3 of [Data-Intensive Text Processing with MapReduce](http://www.amazon.com/Data-Intensive-Processing-MapReduce-Synthesis-Technologies/dp/1608453421). While Hadoop automatically sorts data emitted by mappers before being sent to reducers, what can you do if you also want to sort by value? You use secondary sorting of course. With a slight manipulation to the format of the key object, secondary sorting gives us the ability to take the value into account during the sort phase. There are two possible approaches here. The first approach involves having the reducer buffer all of the values for a given key and do an in-reducer sort on the values. Since the reducer will be receiving all values for a given key, this approach could possibly cause the reducer to run out of memory. The second approach involves creating a composite key by adding a part of, or the entire value to the natural key to achieve your sorting objectives. The trade off between these two approaches is doing an explicit sort on values in the reducer would most likely be faster(at the risk of running out of memory) but implementing a “value to key” conversion approach, is offloading the sorting the MapReduce framework, which lies at the heart of what Hadoop/MapReduce is designed to do. For the purposes of this post, we will consider the “value to key” approach. We will need to write a custom partitioner to ensure all the data with same key (the natural key not including the composite key with the value) is sent to the same reducer and a custom Comparator so the data is grouped by the natural key once it arrives at the reducer.

### Value to Key Conversion

Creating a composite key is straight forward. What we need to do is analyze what part(s) of the value we want to account for during the sort and add the appropriate part(s) to the natural key. Then we need to work on the compareTo method either in key class, or comparator class to make sure the composite key is accounted. We will be re-visiting the weather data set and include the temperature as part of the natural key (the natural key being the year and month concatenated together). The result will be a listing of the coldest day for a given month and year. This example was inspired from the secondary sorting example found in [Hadoop, The Definitive Guide](http://www.amazon.com/Hadoop-Definitive-Guide-Tom-White/dp/1449311520/ref=dp_ob_title_bk) book. While there are probably better ways to achieve this objective, but it will be good enough to demonstrate how secondary sorting works.

### Mapper Code

Our mapper code already concatenates the year and month together, but we will also include the temperature as part of the key. Since we have included the value in the key itself, the mapper will emit a NullWritable, where in other cases we would emit the temperature.

public class SecondarySortingTemperatureMapper extends Mapper<LongWritable, Text, TemperaturePair, NullWritable> {

private TemperaturePair temperaturePair = new TemperaturePair();

private NullWritable nullValue = NullWritable.get();

private static final int MISSING = 9999;

@Override

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

String line = value.toString();

String yearMonth = line.substring(15, 21);

int tempStartPosition = 87;

if (line.charAt(tempStartPosition) == '+') {

tempStartPosition += 1;

}

int temp = Integer.parseInt(line.substring(tempStartPosition, 92));

if (temp != MISSING) {

temperaturePair.setYearMonth(yearMonth);

temperaturePair.setTemperature(temp);

context.write(temperaturePair, nullValue);

}

}

}

Now we have added the temperature to the key, we set the stage for enabling secondary sorting. What’s left to do is write code taking temperature into account when necessary. Here we have two choices, write a Comparator or adjust the compareTo method on the TemperaturePair class (TemperaturePair implements WritableComparable). In most cases I would recommend writing a separate Comparator, but the TemperaturePair class was written specifically to demonstrate secondary sorting, so we will modify the TemperaturePair class compareTo method.

@Override

public int compareTo(TemperaturePair temperaturePair) {

int compareValue = this.yearMonth.compareTo(temperaturePair.getYearMonth());

if (compareValue == 0) {

compareValue = temperature.compareTo(temperaturePair.getTemperature());

}

return compareValue;

}

If we wanted to sort in descending order, we could simply multiply the result of the temperature comparison by a -1. Now that we have completed the part necessary for sorting, we need to write a custom partitioner.

### Partitoner Code

To ensure only the natural key is considered when determining which reducer to send the data to, we need to write a custom partitioner. The code is straight forward and only considers the yearMonth value of the TemperaturePair class when calculating the reducer the data will be sent to.

public class TemperaturePartitioner extends Partitioner<TemperaturePair, NullWritable>{

@Override

public int getPartition(TemperaturePair temperaturePair, NullWritable nullWritable, int numPartitions) {

return temperaturePair.getYearMonth().hashCode() % numPartitions;

}

}

While the custom partitioner guarantees that all of the data for the year and month arrive at the same reducer, we still need to account for the fact the reducer will group records by key.

### Grouping Comparator

Once the data reaches a reducer, all data is grouped by key. Since we have a composite key, we need to make sure records are grouped solely by the natural key. This is accomplished by writing a custom GroupPartitioner. We have a Comparator object only considering the yearMonth field of the TemperaturePair class for the purposes of grouping the records together.

public class YearMonthGroupingComparator extends WritableComparator {

public YearMonthGroupingComparator() {

super(TemperaturePair.class, true);

}

@Override

public int compare(WritableComparable tp1, WritableComparable tp2) {

TemperaturePair temperaturePair = (TemperaturePair) tp1;

TemperaturePair temperaturePair2 = (TemperaturePair) tp2;

return temperaturePair.getYearMonth().compareTo(temperaturePair2.getYearMonth());

}

}

### Results

Here are the results of running our secondary sort job:

new-host-2:sbin bbejeck$ hdfs dfs -cat secondary-sort/part-r-00000

190101 -206

190102 -333

190103 -272

190104 -61

190105 -33

190106 44

190107 72

190108 44

190109 17

190110 -33

190111 -217

190112 -300

### Conclusion

While sorting data by value may not be a common need, it’s a nice tool to have in your back pocket when needed. Also, we have been able to take a deeper look at the inner workings of Hadoop by working with custom partitioners and group partitioners. Thank you for your time.

## orting Reducer Input Values in Hadoop

13 Nov 2009

I recently found the need to sort by value (intead of key) in Hadoop. I’ve seen some comments that call this a “secondary sort”. Essentially, I wanted the reducer’s values iterator to be sorted. There seem to be almost **no** docs, tutorials, or examples (that I could find) on the net for this.

I HIGHLY recommend that you read the email thread by [Owen O’Malley](http://markmail.org/message/7gonm3kiasyh2xnf#query:setOutputKeyComparatorClass+page:3+mid:esn3lgzyx3ag26cy+state:results) that describes this technique in brief. I should also note that this example is using the 0.18 Hadoop API.

##### PROBLEM STATEMENT

Suppose we have a file with a bunch of comma/line separated letters:

l,f,a,e,a,a,l

f,g,b,c,b,d,f

x,i,t,u,f,e,h

...etc

We want our reducer to receive bigrams (lf, fa, ae, ea, aa, al, etc), but partitioned by the first letter, and sorted (ascending) by the second. For example, for the letter a, the reducer should receive:

<a, [l,e,a]>

This is actually somewhat difficult to do, since we want to partition by key, but sort the reducer’s values iterator. The trick is to have the mapper output the bigram in the key, and only the second letter in the value. For the example above, the mapper would emit:

<ae, e>

<aa, a>

<al, l>

...

We can then use a custom partitioner/sorter to partition and sort according to our needs.

##### SORTING BY VALUE

To sort Hadoop’s mapper output by value, you need to set three settings in your JobConf:

* setPartitionerClass
* setOutputValueGroupingComparator
* setOutputKeyComparatorClass

There are many threads that say that you can’t sort by value in Hadoop. This is true. What you can do, instead, is have your mapper output all data in the key, rather than the value. Then you can use a specialized Partitioner classes and two RawComparator classes to sort and partition your map output properly.

##### PARTITIONER

The first class that you need to set is a class that extends org.apache.hadoop.mapred.Partitioner. This class has a single function that determines which partition your map output should go to. This means that you can’t go below 0, or above numPartitions – 1. Mostly, you’ll want to hashCode() some portion of your key and mod it by numPartitions.

In our example, the partitioner will partition by the first letter of the key.

##### OUTPUT VALUE GROUPING COMPARATOR

The OutputValueGroupingComparator JobConf setting takes in a org.apache.hadoop.io.RawComparator. This RawComparator is used to determine which reducer the mapper output row should go to. This RawComparator does not sort a reducer’s value iterator. Instead, it’s used to sort reducer input, so that the reducer knows when a new grouping starts.

In our example, the value grouping comparator will sort by the first letter of the key.

##### OUTPUT KEY COMPARATOR

The OutputKeyComparatorClass JobConf setting also takes in a org.apache.hadoop.io.RawComparator. This RawComparator is used to sort the values iterator that the reducer gets, which is what we want. It should be noted, that although the RawComparator is used to sort the values iterator, the data that gets passed into the comparator is the mapper key output. This is the reason that we must put all data in the key as well as the value.

A **very** important thing to note is that they key compartor must also enforce the value grouping comparator’s rules. In our example, this means that it must first check if the first letter is equal. If it’s not equal, it should return the same ruls as the value comparator. Only if the first letter of the key is equal should we apply our value-level sorting (comparing the second letter). If you do not do this, you will break your grouping.

In our example, the key comparator will sort by the second letter of the key.

##### RUNNING THE JOB

Now, all we need to do is run the job.

|  |  |
| --- | --- |
|  | public class SortReducerByValues { |
|  | public static final String INPUT = "/tmp/data\_in"; |
|  | public static final String OUTPUT = "/tmp/data\_out"; |
|  |  |
|  | public static void main(String[] args) throws IOException { |
|  | new SortReducerByValues().run(); |
|  | } |
|  |  |
|  | public void run() throws IOException { |
|  | JobConf conf = new JobConf(); |
|  |  |
|  | conf.setInputFormat(SequenceFileInputFormat.class); |
|  | conf.setOutputFormat(SequenceFileOutputFormat.class); |
|  |  |
|  | conf.setMapOutputKeyClass(Text.class); |
|  | conf.setMapOutputValueClass(Text.class); |
|  |  |
|  | conf.setOutputKeyClass(Text.class); |
|  | conf.setOutputValueClass(Text.class); |
|  |  |
|  | conf.setMapperClass(SortReducerByValuesMapper.class); |
|  | conf.setReducerClass(SortReducerByValuesReducer.class); |
|  |  |
|  | conf.setOutputKeyComparatorClass(SortReducerByValuesKeyComparator.class); |
|  | conf.setOutputValueGroupingComparator(SortReducerByValuesValueGroupingComparator.class); |
|  | conf.setPartitionerClass(SortReducerByValuesPartitioner.class); |
|  |  |
|  | FileInputFormat.addInputPath(conf, new Path(INPUT)); |
|  | FileOutputFormat.setOutputPath(conf, new Path(OUTPUT)); |
|  |  |
|  | conf.getWorkingDirectory().getFileSystem(conf).delete(new Path(INPUT), true); |
|  | conf.getWorkingDirectory().getFileSystem(conf).delete(new Path(OUTPUT), true); |
|  |  |
|  | loadFakeData(INPUT); |
|  |  |
|  | JobClient.runJob(conf).waitForCompletion(); |
|  | } |
|  |  |
|  | public static final class SortReducerByValuesKeyComparator implements RawComparator { |
|  | public int compare(byte[] text1, int start1, int length1, byte[] text2, int start2, int length2) { |
|  | // hadoop gives you an extra byte before text data. get rid of it. |
|  | byte[] trimmed1 = new byte[2]; |
|  | byte[] trimmed2 = new byte[2]; |
|  | System.arraycopy(text1, start1+1, trimmed1, 0, 2); |
|  | System.arraycopy(text2, start2+1, trimmed2, 0, 2); |
|  |  |
|  | char char10 = (char)trimmed1[0]; |
|  | char char20 = (char)trimmed2[0]; |
|  | char char11 = (char)trimmed1[1]; |
|  | char char21 = (char)trimmed2[1]; |
|  |  |
|  | // first enforce the same rules as the value grouping comparator |
|  | // (first letter of key) |
|  | int compare = new Character(char10).compareTo(char20); |
|  |  |
|  | if(compare == 0) { |
|  | // ONLY if we're in the same reduce aggregate should we try and |
|  | // sort by value (second letter of key) |
|  | return -1 \* new Character(char11).compareTo(char21); |
|  | } |
|  |  |
|  | return compare; |
|  | } |
|  |  |
|  | public int compare(Text o1, Text o2) { |
|  | // reverse the +1 since the extra text byte is not passed into |
|  | // compare() from this function |
|  | return compare(o1.getBytes(), 0, o1.getLength() - 1, o2.getBytes(), 0, o2.getLength() - 1); |
|  | } |
|  | } |
|  |  |
|  | public static final class SortReducerByValuesPartitioner implements Partitioner { |
|  | public int getPartition(Text key, Text value, int numPartitions) { |
|  | // just partition by the first character of each key since that's |
|  | // how we are grouping for the reducer |
|  | return key.toString().charAt(0) % numPartitions; |
|  | } |
|  |  |
|  | public void configure(JobConf conf) { } |
|  | } |
|  |  |
|  | public static final class SortReducerByValuesValueGroupingComparator implements RawComparator { |
|  | public int compare(byte[] text1, int start1, int length1, byte[] text2, int start2, int length2) { |
|  | // look at first character of each text byte array |
|  | return new Character((char)text1[0]).compareTo((char)text2[0]); |
|  | } |
|  |  |
|  | public int compare(Text o1, Text o2) { |
|  | return compare(o1.getBytes(), 0, o1.getLength(), o2.getBytes(), 0, o2.getLength()); |
|  | } |
|  | } |
|  |  |
|  | protected void loadFakeData(String path) throws IOException { |
|  | JobConf conf = new JobConf(); |
|  | Writer writer = SequenceFile.createWriter(FileSystem.get(conf), conf, new Path(path), Text.class, Text.class); |
|  |  |
|  | for(int i = 0; i < 100; ++i) { |
|  | String letterCSV = ""; |
|  |  |
|  | for(int j = 0; j < 10; ++j) { |
|  | letterCSV += (char)(65 + (int)(Math.random() \* 26)) + ","; |
|  | } |
|  |  |
|  | writer.append(new Text(), new Text(letterCSV.substring(0, letterCSV.length() - 1))); |
|  | } |
|  |  |
|  | writer.close(); |
|  | } |
|  |  |
|  | public static final class SortReducerByValuesMapper implements Mapper { |
|  | public void map(Text key, Text val, |
|  | OutputCollector collector, Reporter reporter) |
|  | throws IOException { |
|  | String[] chars = val.toString().split(","); |
|  |  |
|  | for(int i = 0; i < chars.length - 1; ++i) { |
|  | collector.collect(new Text(chars[i] + chars[i+1]), new Text(chars[i+1])); |
|  | } |
|  | } |
|  |  |
|  | public void configure(JobConf conf) { } |
|  | public void close() throws IOException { } |
|  | } |
|  |  |
|  | public static final class SortReducerByValuesReducer implements Reducer { |
|  |  |
|  | @Override |
|  | public void reduce(Text key, Iterator values, |
|  | OutputCollector collector, Reporter reporter) |
|  | throws IOException { |
|  | // values should now be in order |
|  | String check = key + ": "; |
|  |  |
|  | while(values.hasNext()) { |
|  | check += values.next(); |
|  | } |
|  |  |
|  | System.err.println(check); |
|  | } |
|  |  |
|  | public void configure(JobConf conf) { } |
|  | public void close() throws IOException { } |
|  | } |
|  | } |

<https://gist.github.com/airawat/6604175>

20) Joind in Mapreuce, mapside joins, reduce side joins

Answer)

# MapReduce Algorithms - Understanding Data Joins Part 1

JUN 26TH, 2013

In this post we continue with our series of implementing the algorithms found in the [Data-Intensive Text Processing with MapReduce](http://www.amazon.com/Data-Intensive-Processing-MapReduce-Synthesis-Technologies/dp/1608453421) book, this time discussing data joins. While we are going to discuss the techniques for joining data in Hadoop and provide sample code, in most cases you probably won’t be writing code to perform joins yourself. Instead, joining data is better accomplished using tools that work at a higher level of abstraction such as Hive or Pig. Why take the time to learn how to join data if there are tools that can take care of it for you? Joining data is arguably one of the biggest uses of Hadoop. Gaining a full understanding of how Hadoop performs joins is critical for deciding which join to use and for debugging when trouble strikes. Also, once you fully understand how different joins are performed in Hadoop, you can better leverage tools like Hive and Pig. Finally, there might be the one off case where a tool just won’t get you what you need and you’ll have to roll up your sleeves and write the code yourself.

### The Need for Joins

When processing large data sets the need for joining data by a common key can be very useful, if not essential. By joining data you can further gain insight such as joining with timestamps to correlate events with a time a day. The need for joining data are many and varied. We will be covering 3 types of joins, Reduce-Side joins, Map-Side joins and the Memory-Backed Join over 3 separate posts. This installment we will consider working with Reduce-Side joins.

### Reduce Side Joins

Of the join patterns we will discuss, reduce-side joins are the easiest to implement. What makes reduce-side joins straight forward is the fact that Hadoop sends identical keys to the same reducer, so by default the data is organized for us. To perform the join, we simply need to cache a key and compare it to incoming keys. As long as the keys match, we can join the values from the corresponding keys. The trade off with reduce-side joins is performance, since all of the data is shuffled across the network. Within reduce-side joins there are two different scenarios we will consider: one-to-one and one-to-many. We’ll also explore options where we don’t need to keep track of the incoming keys; all values for a given key will be grouped together in the reducer.

### One-To-One Joins

A one-to-one join is the case where a value from dataset ‘X’ shares a common key with a value from dataset ‘Y’. Since Hadoop guarantees that equal keys are sent to the same reducer, mapping over the two datasets will take care of the join for us. Since sorting only occurs for keys, the order of the values is unknown. We can easily fix the situation by using [secondary sorting](http://codingjunkie.net/secondary-sort/). Our implementation of secondary sorting will be to tag keys with either a “1” or a “2” to determine order of the values. We need to take a couple extra steps to implement our tagging strategy.

##### Implementing a WritableComparable

First we need to write a class that implements the WritableComparable interface that will be used to wrap our key.

public class TaggedKey implements Writable, WritableComparable<TaggedKey> {

private Text joinKey = new Text();

private IntWritable tag = new IntWritable();

@Override

public int compareTo(TaggedKey taggedKey) {

int compareValue = this.joinKey.compareTo(taggedKey.getJoinKey());

if(compareValue == 0 ){

compareValue = this.tag.compareTo(taggedKey.getTag());

}

return compareValue;

}

//Details left out for clarity

}

When our TaggedKey class is sorted, keys with the same joinKey value will have a secondary sort on the value of the tag field, ensuring the order we want.

##### Writing a Custom Partitioner

Next we need to write a custom partitioner that will only consider the join key when determining which reducer the composite key and data are sent to:

public class TaggedJoiningPartitioner extends Partitioner<TaggedKey,Text> {

@Override

public int getPartition(TaggedKey taggedKey, Text text, int numPartitions) {

return taggedKey.getJoinKey().hashCode() % numPartitions;

}

}

At this point we have what we need to join the data and ensure the order of the values. But we don’t want to keep track of the keys as they come into the reduce() method. We want all the values grouped together for us. To accomplish this we will use a Comparator that will consider only the join key when deciding how to group the values.

##### Writing a Group Comparator

Our Comparator used for grouping will look like this:

public class TaggedJoiningGroupingComparator extends WritableComparator {

public TaggedJoiningGroupingComparator() {

super(TaggedKey.class,true);

}

@Override

public int compare(WritableComparable a, WritableComparable b) {

TaggedKey taggedKey1 = (TaggedKey)a;

TaggedKey taggedKey2 = (TaggedKey)b;

return taggedKey1.getJoinKey().compareTo(taggedKey2.getJoinKey());

}

}

### Structure of the data

Now we need to determine what we will use for our key to join the data. For our sample data we will be using a CSV file generated from the [Fakenames Generator](http://www.fakenamegenerator.com/order.php). The first column is a GUID and that will serve as our join key. Our sample data contains information like name, address, email, job information, credit cards and automobiles owned. For the purposes of our demonstration we will take the GUID, name and address fields and place them in one file that will be structured like this:

cdd8dde3-0349-4f0d-b97a-7ae84b687f9c,Esther,Garner,4071 Haven Lane,Okemos,MI 81a43486-07e1-4b92-b92b-03d0caa87b5f,Timothy,Duncan,753 Stadium Drive,Taunton,MA aef52cf1-f565-4124-bf18-47acdac47a0e,Brett,Ramsey,4985 Shinn Street,New York,NY

Then we will take the GUID, email address, username, password and credit card fields and place then in another file that will look like:

cdd8dde3-0349-4f0d-b97a-7ae84b687f9c,517-706-9565,EstherJGarner@teleworm.us,Waskepter38,noL2ieghie,MasterCard,5305687295670850 81a43486-07e1-4b92-b92b-03d0caa87b5f,508-307-3433,TimothyDDuncan@einrot.com,Conerse,Gif4Edeiba,MasterCard,5265896533330445 aef52cf1-f565-4124-bf18-47acdac47a0e,212-780-4015,BrettMRamsey@dayrep.com,Subjecall,AiKoiweihi6,MasterCard,5243379373546690

Now we need to have a Mapper that will know how to work with our data to extract the correct key for joining and also set the proper tag.

##### Creating the Mapper

Here is our Mapper code:

public class JoiningMapper extends Mapper<LongWritable, Text, TaggedKey, Text> {

private int keyIndex;

private Splitter splitter;

private Joiner joiner;

private TaggedKey taggedKey = new TaggedKey();

private Text data = new Text();

private int joinOrder;

@Override

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

keyIndex = Integer.parseInt(context.getConfiguration().get("keyIndex"));

String separator = context.getConfiguration().get("separator");

splitter = Splitter.on(separator).trimResults();

joiner = Joiner.on(separator);

FileSplit fileSplit = (FileSplit)context.getInputSplit();

joinOrder = Integer.parseInt(context.getConfiguration().get(fileSplit.getPath().getName()));

}

@Override

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

List<String> values = Lists.newArrayList(splitter.split(value.toString()));

String joinKey = values.remove(keyIndex);

String valuesWithOutKey = joiner.join(values);

taggedKey.set(joinKey, joinOrder);

data.set(valuesWithOutKey);

context.write(taggedKey, data);

}

}

Let’s review what is going on in the setup() method.

1. First we get the index of our join key and the separator used in the text from values set in the Configuration when the job was launched.
2. Then we create a [Guava Splitter](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/base/Splitter.html) used to split the data on the separator we retrieved from the call to context.getConfiguration().get("separator"). We also create a [Guava Joiner](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/base/Joiner.html) used to put the data back together once the key has been extracted.
3. Next we get the name of the file that this mapper will be processing. We use the filename to pull the join order for this file that was stored in the configuration.

We should also discuss what’s going on in the map() method:

1. Spitting our data and creating a List of the values
2. Remove the join key from the list
3. Re-join the data back into a single String
4. Set the join key, join order and the remaining data
5. Write out the data

So we have read in our data, extracted the key, set the join order and written our data back out. Let’s take a look how we will join the data.

### Joining the Data

Now let’s look at how the data is joined in the reducer:

public class JoiningReducer extends Reduce<TaggedKey, Text, NullWritable, Text> {

private Text joinedText = new Text();

private StringBuilder builder = new StringBuilder();

private NullWritable nullKey = NullWritable.get();

@Override

protected void reduce(TaggedKey key, Iterable<Text> values, Context context) throws IOException, InterruptedException {

builder.append(key.getJoinKey()).append(",");

for (Text value : values) {

builder.append(value.toString()).append(",");

}

builder.setLength(builder.length()-1);

joinedText.set(builder.toString());

context.write(nullKey, joinedText);

builder.setLength(0);

}

}

Since the key with the tag of “1” reached the reducer first, we know that the name and address data is the first value and the email,username,password and credit card data is second. So we don’t need to keep track of any keys. We simply loop over the values and concatenate them together.

### One-To-One Join results

Here are the results from running our One-To-One MapReduce job: cdd8dde3-0349-4f0d-b97a-7ae84b687f9c,Esther,Garner,4071 Haven Lane,Okemos,MI,517-706-9565,EstherJGarner@teleworm.us,Waskepter38,noL2ieghie,MasterCard,5305687295670850 81a43486-07e1-4b92-b92b-03d0caa87b5f,Timothy,Duncan,753 Stadium Drive,Taunton,MA,508-307-3433,TimothyDDuncan@einrot.com,Conerse,Gif4Edeiba,MasterCard,5265896533330445 aef52cf1-f565-4124-bf18-47acdac47a0e,Brett,Ramsey,4985 Shinn Street,New York,NY,212-780-4015,BrettMRamsey@dayrep.com,Subjecall,AiKoiweihi6,MasterCard,5243379373546690As we can see the two records from our sample data above have been merged into a single record. We have successfully joined the GUID, name,address,email address, username, password and credit card fields together into one file.

##### Specifying Join Order

At this point we may be asking how do we specify the join order for multiple files? The answer lies in our ReduceSideJoinDriver class that serves as the driver for our MapReduce program.

public class ReduceSideJoinDriver {

public static void main(String[] args) throws Exception {

Splitter splitter = Splitter.on('/');

StringBuilder filePaths = new StringBuilder();

Configuration config = new Configuration();

config.set("keyIndex", "0");

config.set("separator", ",");

for(int i = 0; i< args.length - 1; i++) {

String fileName = Iterables.getLast(splitter.split(args[i]));

config.set(fileName, Integer.toString(i+1));

filePaths.append(args[i]).append(",");

}

filePaths.setLength(filePaths.length() - 1);

Job job = Job.getInstance(config, "ReduceSideJoin");

job.setJarByClass(ReduceSideJoinDriver.class);

FileInputFormat.addInputPaths(job, filePaths.toString());

FileOutputFormat.setOutputPath(job, new Path(args[args.length-1]));

job.setMapperClass(JoiningMapper.class);

job.setReducerClass(JoiningReducer.class);

job.setPartitionerClass(TaggedJoiningPartitioner.class);

job.setGroupingComparatorClass(TaggedJoiningGroupingComparator.class);

job.setOutputKeyClass(TaggedKey.class);

job.setOutputValueClass(Text.class);

System.exit(job.waitForCompletion(true) ? 0 : 1);

}

}

1. First we create a Guava Splitter on line 5 that will split strings by a “/”.
2. Then on lines 8-10 we are setting the index of our join key and the separator used in the files.
3. In lines 12-17 we setting the tags for the input files to be joined. The order of the file names on the command line determines their position in the join. As we loop over the file names from the command line, we split the whole file name and retrieve the last value (the base filename) via the Guava [Iterables.getLast()](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/collect/Iterables.html) method. We then call config.set() with the filename as the key and we use i + 1 as the value, which sets the tag or join order. The last value in the args array is skipped in the loop, as that is used for the output path of our MapReduce job on line 23. On the last line of the loop we append each file path in a StringBuilder which is used later (line 22) to set the input paths for the job.
4. We only need to use one mapper for all files, the JoiningMapper, which is set on line 25.
5. Lines 27 and 28 set our custom partitioner and group comparator (respectively) which ensure the arrival order of keys and values to the reducer and properly group the values with the correct key.

By using the partitioner and the grouping comparator we know the first value belongs to first key and can be used to join with every other value contained in the Iterable sent to the reduce() method for a given key. Now it’s time to consider the one-to-many join.

### One-To-Many Join

The good news is with all the work that we have done up to this point, we can actually use the code as it stands to perform a one-to-many join. There are 2 approaches we can consider for the one-to-many join: 1) A small file with the single records and a second file with many records for the same key and 2) Again a smaller file with the single records, but N number of files each containing a record that matches to the first file. The main difference is that with the first approach the order of the values beyond the join of the first two keys will be unknown. With the second approach however, we will “tag” each join file so we can can control the order of all the joined values. For our example the first file will remain our GUID-name-address file, and we will have 3 additional files that will contain automobile, employer and job description records. This is probably not the most realistic scenario but it will serve for the purposes of demonstration. Here’s a sample of how the data will look before we do the join: //The single person records cdd8dde3-0349-4f0d-b97a-7ae84b687f9c,Esther,Garner,4071 Haven Lane,Okemos,MI 81a43486-07e1-4b92-b92b-03d0caa87b5f,Timothy,Duncan,753 Stadium Drive,Taunton,MA aef52cf1-f565-4124-bf18-47acdac47a0e,Brett,Ramsey,4985 Shinn Street,New York,NY //Automobile records cdd8dde3-0349-4f0d-b97a-7ae84b687f9c,2003 Holden Cruze 81a43486-07e1-4b92-b92b-03d0caa87b5f,2012 Volkswagen T5 aef52cf1-f565-4124-bf18-47acdac47a0e,2009 Renault Trafic //Employer records cdd8dde3-0349-4f0d-b97a-7ae84b687f9c,Creative Wealth 81a43486-07e1-4b92-b92b-03d0caa87b5f,Susie's Casuals aef52cf1-f565-4124-bf18-47acdac47a0e,Super Saver Foods //Job Description records cdd8dde3-0349-4f0d-b97a-7ae84b687f9c,Data entry clerk 81a43486-07e1-4b92-b92b-03d0caa87b5f,Precision instrument and equipment repairer aef52cf1-f565-4124-bf18-47acdac47a0e,Gas and water service dispatcher

##### One-To-Many Join results

Now let’s look at a sample of the results of our one-to-many joins (using the same values from above to aid in the comparison): cdd8dde3-0349-4f0d-b97a-7ae84b687f9c,Esther,Garner,4071 Haven Lane,Okemos,MI,2003 Holden Cruze,Creative Wealth,Data entry clerk 81a43486-07e1-4b92-b92b-03d0caa87b5f,Timothy,Duncan,753 Stadium Drive,Taunton,MA,2012 Volkswagen T5,Susie's Casuals,Precision instrument and equipment repairer aef52cf1-f565-4124-bf18-47acdac47a0e,Brett,Ramsey,4985 Shinn Street,New York,NY,2009 Renault Trafic,Super Saver Foods,Gas and water service dispatcherAs the results show, we have been able to successfully join several values in a specified order.

### Conclusion

We have successfully demonstrated how we can perform reduce-side joins in MapReduce. Even though the approach is not overly complicated, we can see that performing joins in Hadoop can involve writing a fair amount of code. While learning how joins work is a useful exercise, in most cases we are much better off using tools like Hive or Pig for joining data. Thanks for your time.

### Resources

\* [Data-Intensive Processing with MapReduce](http://www.amazon.com/Data-Intensive-Processing-MapReduce-Synthesis-Technologies/dp/1608453421) by Jimmy Lin and Chris Dyer

\* [Hadoop: The Definitive Guide](http://www.amazon.com/Hadoop-Definitive-Guide-Tom-White/dp/1449311520/ref=tmm\_pap\_title\_0?ie=UTF8&qid=1347589052&sr=1-1) by Tom White

\* [Source Code and Tests](https://github.com/bbejeck/hadoop-algorithms) from blog

\* [Programming Hive](http://www.amazon.com/Programming-Hive-Edward-Capriolo/dp/1449319335) by Edward Capriolo, Dean Wampler and Jason Rutherglen

\* [Programming Pig](http://www.amazon.com/Programming-Pig-Alan-Gates/dp/1449302645) by Alan Gates

\* [Hadoop API](http://hadoop.apache.org/docs/r0.20.2/api/index.html)

\* [MRUnit](http://mrunit.apache.org/) for unit testing Apache Hadoop map reduce jobs

# Hadoop MapReduce Tutorial: Counters & Joins with Example

A counter in MapReduce is a mechanism used for collecting statistical information about the MapReduce job. This information could be useful for diagnosis of a problem in MapReduce job processing. Counters are similar to putting log message in the code for map or reduce.

In this tutorial we will learn,

* [Two types of counters](http://www.guru99.com/introduction-to-counters-joins-in-map-reduce.html#1)
* [MapReduce Join](http://www.guru99.com/introduction-to-counters-joins-in-map-reduce.html#2)
* [MapReduce Hadoop Program To Join Data](http://www.guru99.com/introduction-to-counters-joins-in-map-reduce.html#3)

Typically, these counters are defined in a program (map or reduce) and are incremented during execution when a particular event or condition (specific to that counter) occurs. A very good application of counters is to track valid and invalid records from an input dataset.

### Two types of counters:

**1. Hadoop Built-In counters:**There are some built-in counters which exist per job. Below are built-in counter groups-

* **MapReduce Task Counters** - Collects task specific information (e.g., number of input records) during its execution time.
* **FileSystem Counters** - Collects information like number of bytes read or written by a task
* **FileInputFormat Counters** - Collects information of number of bytes read through FileInputFormat
* **FileOutputFormat Counters** - Collects information of number of bytes written through FileOutputFormat
* **Job Counters -** These counters are used by JobTracker. Statistics collected by them include e.g., number of task launched for a job.

**2. User Defined Counters**

In addition to built-in counters, user can define his own counters using similar functionalities provided by programming languages. For example, in[Java](http://www.guru99.com/java-tutorial.html)'enum' are used to define user defined counters.

**An example MapClass with Counters to count the number of missing and invalid values:**

public static class MapClass

extends MapReduceBase

implements Mapper<LongWritable, Text, Text, Text>

{

static enum SalesCounters { MISSING, INVALID };

public void map ( LongWritable key, Text value,

OutputCollector<Text, Text> output,

Reporter reporter) throws IOException

{

//Input string is split using ',' and stored in 'fields' array

String fields[] = value.toString().split(",", -20);

//Value at 4th index is country. It is stored in 'country' variable

String country = fields[4];

//Value at 8th index is sales data. It is stored in 'sales' variable

String sales = fields[8];

if (country.length() == 0) {

reporter.incrCounter(SalesCounters.MISSING, 1);

} else if (sales.startsWith("\"")) {

reporter.incrCounter(SalesCounters.INVALID, 1);

} else {

output.collect(new Text(country), new Text(sales + ",1"));

}

}

}

Above code snippet shows an example implementation of counters in Map Reduce.

Here, **SalesCounters** is a counter defined using **'enum'**. It is used to count **MISSING** and **INVALID** input records.

In the code snippet, if **'country'** field has zero length then its value is missing and hence corresponding counter **SalesCounters.MISSING**is incremented.

Next, if **'sales'**field starts with a **"** then the record is considered INVALID. This is indicated by incrementing counter **SalesCounters.INVALID.**

### ****MapReduce Join****

Joining two large dataset can be achieved using MapReduce Join. However, this process involves writing lots of code to perform actual join operation.

Joining of two datasets begin by comparing size of each dataset. If one dataset is smaller as compared to the other dataset then smaller dataset is distributed to every datanode in the cluster. Once it is distributed, either Mapper or Reducer uses smaller dataset to perform lookup for matching records from large dataset and then combine those records to form output records.

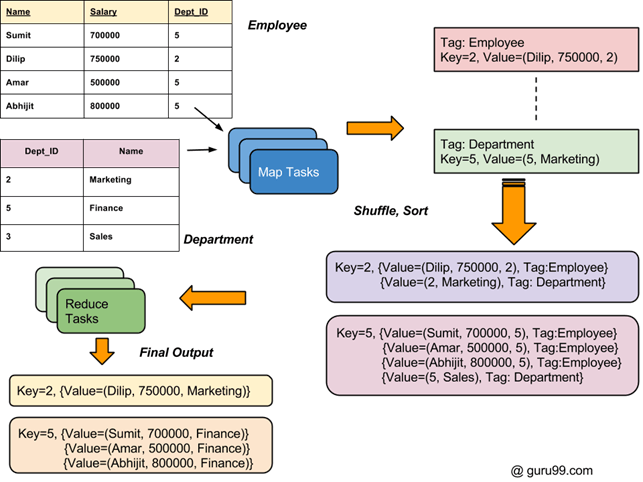
Depending upon the place where actual join is performed, this join is classified into-

**1. Map-side join -** When the join is performed by the mapper, it is called as map-side join. In this type, the join is performed before data is actually consumed by the map function. It is mandatory that the input to each map is in the form of a partition and is in sorted order. Also, there must be an equal number of partitions and it must be sorted by the join key.

**2. Reduce-side join -** When the join is performed by the reducer, it is called as reduce-side join. There is no necessity in this join to have dataset in a structured form (or partitioned).

Here, map side processing emits join key and corresponding tuples of both the tables. As an effect of this processing, all the tuples with same join key fall into the same reducer which then joins the records with same join key.

Overall process flow is depicted in below diagram.

[](http://cdn.guru99.com/images/Big_Data/061114_1003_Introductio1.png)

### MapReduce Hadoop Program To Join Data

**Problem Statement:**

There are 2 Sets of Data in 2 Different Files

|  |  |
| --- | --- |
| [Hadoop MapReduce Tutorial: Counters & Joins with Example](http://cdn.guru99.com/images/Big_Data/061114_1032_MapReduceHa1.png) | [Hadoop MapReduce Tutorial: Counters & Joins with Example](http://cdn.guru99.com/images/Big_Data/061114_1032_MapReduceHa2.png) |

The Key Dept\_ID is common in both files.

The goal is to use MapReduce Join to combine these files

**Input:** Our input data set is a txt file, **DeptName.txt & DepStrength.txt**

[Download Input Files From Here](https://drive.google.com/uc?export=download&id=0B_vqvT0ovzHcLVNFLWQwWUZsc3M)

**Prerequisites:**

* This tutorial is developed on [**Linux - Ubuntu**](http://www.guru99.com/ubuntu-installation-on-virtual-box.html) operating System.
* You should have **Hadoop** (**version 2.2.0** used for this tutorial) already installed.
* You should have [Java](http://www.guru99.com/how-to-install-java-on-ubuntu.html)(**version 1.8.0** used for this tutorial) already installed on the system.

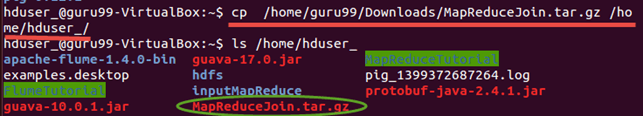
Before we start with the actual process, change user to 'hduser' (user used for Hadoop ).

**su - hduser\_**

[Hadoop MapReduce Tutorial: Counters & Joins with Example](http://cdn.guru99.com/images/Big_Data/061114_1032_MapReduceHa3.png)

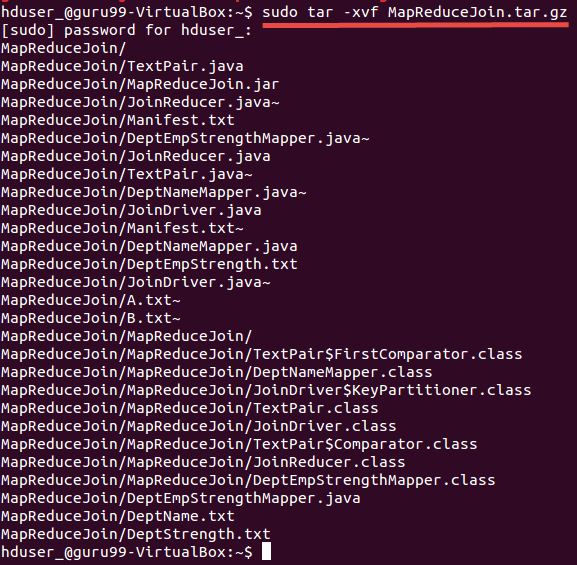
**Steps:**

**Step 1) Copy the zip file to location of your choice**

[](http://cdn.guru99.com/images/Big_Data/061114_1032_MapReduceHa4.png)

**Step 2) Uncompress the Zip File**

**sudo tar -xvf MapReduceJoin.tar.gz**

[](http://cdn.guru99.com/images/Big_Data/061114_1032_MapReduceHa5.png)

**Step 3)**

**Go to directory MapReduceJoin/**

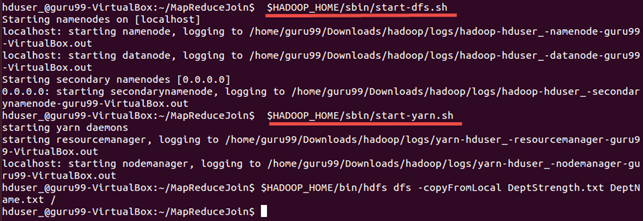
**cd MapReduceJoin/**

[Hadoop MapReduce Tutorial: Counters & Joins with Example](http://cdn.guru99.com/images/Big_Data/061114_1032_MapReduceHa6.png)

**Step 4)**Start Hadoop

**$HADOOP\_HOME/sbin/start-dfs.sh**

**$HADOOP\_HOME/sbin/start-yarn.sh**

[](http://cdn.guru99.com/images/Big_Data/061114_1032_MapReduceHa7.png)

**Step 5) DeptStrength.txt and DeptName.txt are the input files used for this program.**

**These file needs to be copied to HDFS using below command-**

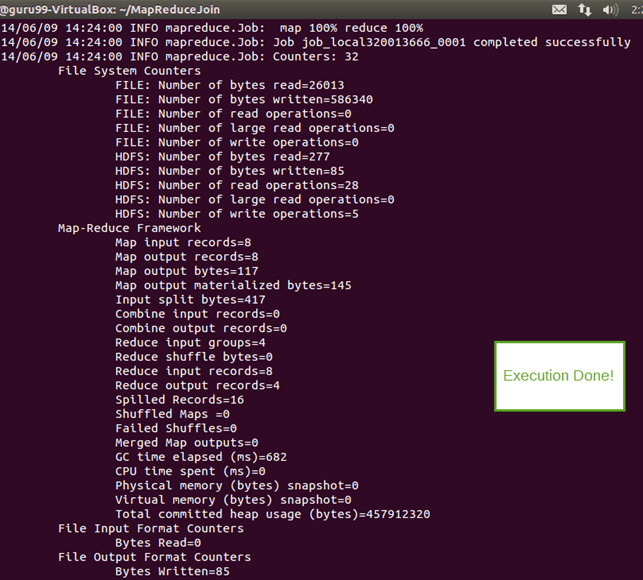
**$HADOOP\_HOME/bin/hdfs dfs -copyFromLocal DeptStrength.txt DeptName.txt /**

[Hadoop MapReduce Tutorial: Counters & Joins with Example](http://cdn.guru99.com/images/Big_Data/061114_1032_MapReduceHa8.png)

**Step 6) Run the program using below command-**

**$HADOOP\_HOME/bin/hadoop jar MapReduceJoin.jar /DeptStrength.txt /DeptName.txt /output\_mapreducejoin**

[Hadoop MapReduce Tutorial: Counters & Joins with Example](http://cdn.guru99.com/images/Big_Data/061114_1032_MapReduceHa9.png)

[](http://cdn.guru99.com/images/Big_Data/061114_1032_MapReduceHa10.png)

**Step 7)**

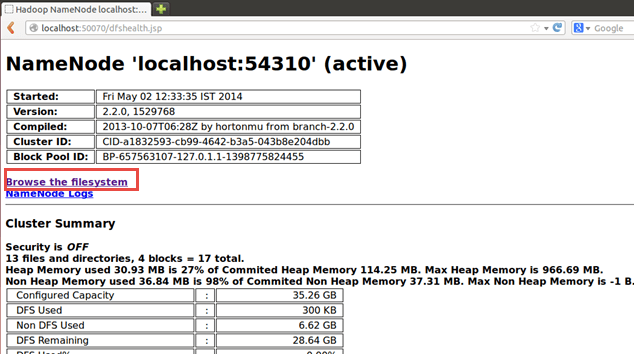
**After execution, output file (named 'part-00000') will stored in the directory /output\_mapreducejoin on HDFS**

**Results can be seen using the command line interface**

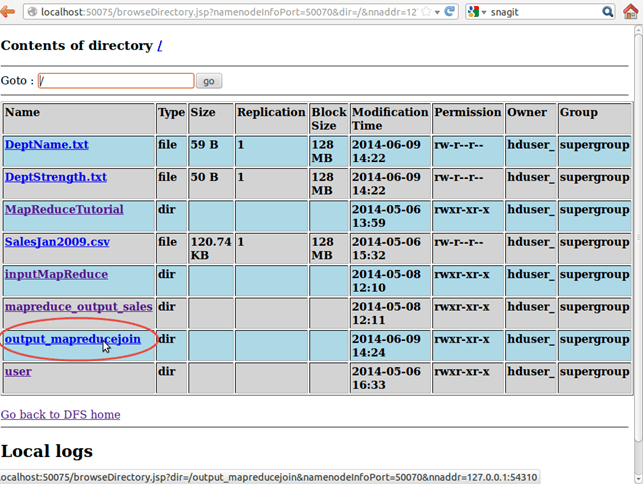
**$HADOOP\_HOME/bin/hdfs dfs -cat /output\_mapreducejoin/part-00000**

[](http://cdn.guru99.com/images/Big_Data/061114_1032_MapReduceHa11.png)

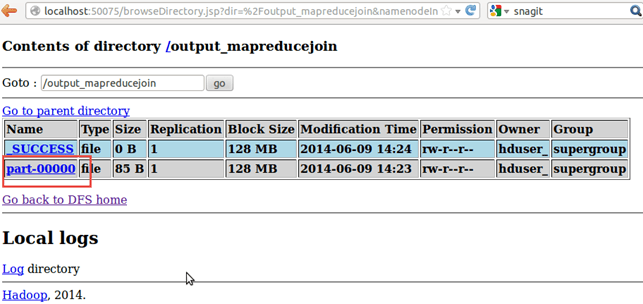
Results can also be seen via web interface as-

[](http://cdn.guru99.com/images/Big_Data/061114_1032_MapReduceHa12.png)

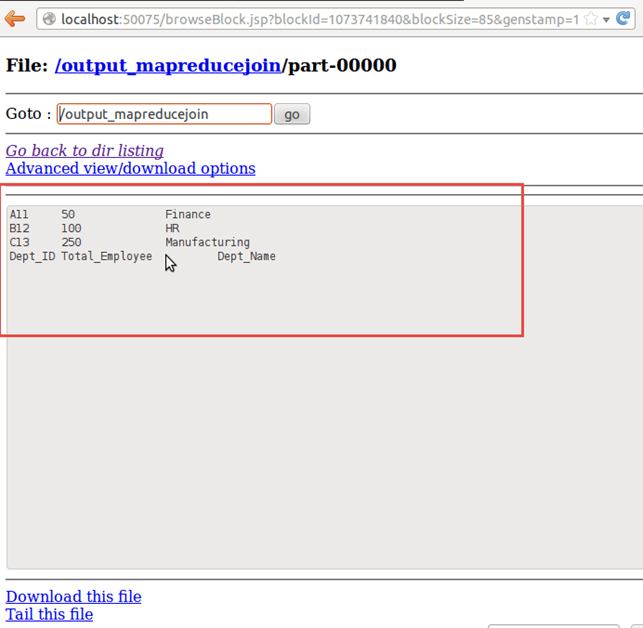
Now select **'Browse the filesystem'**and navigate upto **/output\_mapreducejoin**

[](http://cdn.guru99.com/images/Big_Data/061114_1032_MapReduceHa13.png)

Open **part-r-00000**

[](http://cdn.guru99.com/images/Big_Data/061114_1032_MapReduceHa14.png)

Results are shown

[](http://cdn.guru99.com/images/Big_Data/061114_1032_MapReduceHa15.png)

**NOTE: Please note that before running this program for the next time, you will need to delete output directory /output\_mapreducejoin**

**$HADOOP\_HOME/bin/hdfs dfs -rm -r /output\_mapreducejoin**

**Alternative is to use different name for output directory.**

### Joins with plain Map Reduce or MultipleInputs

Being a map reduce developer I’d never recommend to write joins of data sets using custom map reduce code. You have very intelligent and powerful tools handy in hadoop like hive and pig that can easily join huge data sets with the choice of join like inner, outer etc. But if such a scenario arises where you need to do join using map reduce you should be able to accomplish that with your knowledge on basic map reduce programming.

                Let us look into a mocked up example for the same.

**Problem Statement**: A retailer has a customer data base and he need to do some promotions based on the same. He chooses bulk sms as the choice of his promotion which is done by a third part for him. And once the sms is pushed the sms provider returns the delivery status back to the retailer. Now let us look into more details which makes things a little complicated

We have 3 input files as follows

1.       UserDetails.txt

                                 i.            Every record is of the format ‘mobile number , consumer name’

2.       DeliveryDetails.txt

                                 i.            Every record is of the format ‘mobile number, delivery status code’

3.       DeliveryStatusCodes.txt

                                 i.            Every record is of the format ‘delivery status code, status message’

The retailer has a consumer data base(UserDetails.txt)  from which only the mobile number are provided to a bulk sms provider. He can’t reveal the customer name due to security reasons. Once the messages are pushed the sms provider sends back a report of the mobile numbers with status code (DeliveryDetails.txt) and also a look up file that relates every status code to the corresponding Status message (DeliveryStatusCodes.txt).

 The requirement is that for meaningful information we need the consumer name along with its corresponding status message. And we need to obtain the same from these 3 files.

**Sample Inputs**

**File 1 – UserDetails.txt**

123 456, Jim

456 123, Tom

789 123, Harry

789 456, Richa

**File 2 – DeliveryDetails.txt**

123 456, 001

456 123, 002

789 123, 003

789 456, 004

**File 3 – DeliveryStatusCodes.txt**

001, Delivered

002, Pending

003, Failed

004, Resend

**Expected Output**

Jim, Delivered

Tom, Pending

Harry, Failed

Richa, Resend

**Solution : Using core MapReduce**

1.       Use two different mapper classes for both processing the  initial inputs from UserDetails.txt and DeliveryDetails.txt, The Key value output from the mappers should be as follows

a)      UserDetails.txt

                                                         i.            Key(Text) – mobile number

                                                       ii.            Value(Text) – An identifier to indicate the source of input(using ‘CD’ for the customer details file) + Customer Name

b)      DeliveryDetails.txt

                                                         i.            Key(Text) – mobile number

                                                       ii.            Value(Text) – An identifier to indicate the source of input(using ‘DR’ for the delivery report file) + Status Code

So here since the two files needs to be parsed separately using two mappers. I’m using

UserFile Mapper.java to process UserDetails.txt and

DeliveryFileMapper.java to process DeliveryDetails.txt

In map reduce API, I’m using MulipleInputFormat to specify which input to go into which mapper. But the ouput key value pairs from the mapper go into the same reducer, for the Reducer to identify the source of the value we are prepending the values ‘CD’ or ‘DR’.

2.       On the reducer end use distributed cache to distribute the DeliveryStatusCodes.txt. Parse the file and load the contents into HashMap with Key being the status code and value being the status message

3.       On the reducer every key would be having two values one with prefix ‘CD’ and other ‘DR’. (For simplicity let us assume only 2 values, in real time it can be more). Identify the records and from CD get the customer name corresponding to the cell number (input key) and from DR get the status code. On obtaining the status code do a look up on the HashMap to get the status message. So finally the output Key values from the reducer would be as follows

a)      Key : Customer Name

b)      Value : Status Message

Let’s just look at the source code

**Mapper Class1: UserFileMapper.java**

**import** java.io.IOException;

**import** org.apache.hadoop.io.LongWritable;

**import** org.apache.hadoop.io.Text;

**import** org.apache.hadoop.mapred.~~MapReduceBase~~;

**import** org.apache.hadoop.mapred.~~Mapper~~;

**import** org.apache.hadoop.mapred.OutputCollector;

**import** org.apache.hadoop.mapred.Reporter;

**public** **class** UserFileMapper **extends** ~~MapReduceBase~~ **implements**~~Mapper~~<LongWritable, Text, Text, Text>

{

    //variables to process Consumer Details

**private** String cellNumber,customerName,fileTag="CD~";

    /\* map method that process ConsumerDetails.txt and frames the initial key value pairs

       Key(Text) – mobile number

       Value(Text) – An identifier to indicate the source of input(using ‘CD’ for the customer details file) + Customer Name

     \*/

**public** **void** map(LongWritable key, Text value, OutputCollector<Text, Text> output, Reporter reporter) **throws**IOException

    {

       //taking one line/record at a time and parsing them into key value pairs

        String line = value.toString();

        String splitarray[] = line.split(",");

        cellNumber = splitarray[0].trim();

        customerName = splitarray[1].trim();

      //sending the key value pair out of mapper

        output.collect(**new** Text(cellNumber), **new**Text(fileTag+customerName));

     }

}

**Mapper Class2:DeliverFileMapper.java**

**import** java.io.IOException;

**import** java.util.StringTokenizer;

**import** org.apache.hadoop.io.IntWritable;

**import** org.apache.hadoop.io.LongWritable;

**import** org.apache.hadoop.io.Text;

**import** org.apache.hadoop.mapred.~~MapReduceBase~~;

**import** org.apache.hadoop.mapred.~~Mapper~~;

**import** org.apache.hadoop.mapred.OutputCollector;

**import** org.apache.hadoop.mapred.Reporter;

**public** **class** DeliveryFileMapper **extends** ~~MapReduceBase~~ **implements**~~Mapper~~<LongWritable, Text, Text, Text>

{

    //variables to process delivery report

**private** String cellNumber,deliveryCode,fileTag="DR~";

   /\* map method that process DeliveryReport.txt and frames the initial key value pairs

    Key(Text) – mobile number

    Value(Text) – An identifier to indicate the source of input(using ‘DR’ for the delivery report file) + Status Code\*/

**public** **void** map(LongWritable key, Text value, OutputCollector<Text, Text> output, Reporter reporter) **throws**IOException

    {

       //taking one line/record at a time and parsing them into key value pairs

        String line = value.toString();

        String splitarray[] = line.split(",");

        cellNumber = splitarray[0].trim();

        deliveryCode = splitarray[1].trim();

        //sending the key value pair out of mapper

        output.collect(**new** Text(cellNumber), **new**Text(fileTag+deliveryCode));

     }

}

**Reducer Class:SmsReducer.java**

**import** java.io.BufferedReader;

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**import** java.io.IOException;

**import** java.util.HashMap;

**import** java.util.Iterator;

**import** java.util.Map;

**import** org.apache.hadoop.io.Text;

**import** org.apache.hadoop.mapred.~~JobConf~~;

**import** org.apache.hadoop.mapred.~~MapReduceBase~~;

**import** org.apache.hadoop.mapred.OutputCollector;

**import** org.apache.hadoop.mapred.~~Reducer~~;

**import** org.apache.hadoop.mapred.Reporter;

**public** **class** SmsReducer **extends** ~~MapReduceBase~~ **implements** ~~Reducer~~<Text, Text, Text, Text> {

       //Variables to aid the join process

**private** String customerName,deliveryReport;

       /\*Map to store Delivery Codes and Messages

       Key being the status code and vale being the status message\*/

**private** **static** Map<String,String> *DeliveryCodesMap*= **new**HashMap<String,String>();

**public** **void** configure(~~JobConf~~ job)

       {

              //To load the Delivery Codes and Messages into a hash map

              loadDeliveryStatusCodes();

       }

**public** **void** reduce(Text key, Iterator<Text> values, OutputCollector<Text, Text> output, Reporter reporter) **throws**IOException

    {

**while** (values.hasNext())

        {

             String currValue = values.next().toString();

             String valueSplitted[] = currValue.split("~");

             /\*identifying the record source that corresponds to a cell number

             and parses the values accordingly\*/

**if**(valueSplitted[0].equals("CD"))

             {

               customerName=valueSplitted[1].trim();

             }

**else** **if**(valueSplitted[0].equals("DR"))

             {

              //getting the delivery code and using the same to obtain the Message

               deliveryReport =*DeliveryCodesMap*.get(valueSplitted[1].trim());

             }

        }

        //pump final output to file

**if**(customerName!=**null** && deliveryReport!=**null**)

        {

               output.collect(**new** Text(customerName), **new**Text(deliveryReport));

        }

**else** **if**(customerName==**null**)

               output.collect(**new** Text("customerName"), **new**Text(deliveryReport));

**else** **if**(deliveryReport==**null**)

               output.collect(**new** Text(customerName), **new**Text("deliveryReport"));

    }

       //To load the Delivery Codes and Messages into a hash map

**private** **void** loadDeliveryStatusCodes()

    {

       String strRead;

**try** {

              //read file from Distributed Cache

                     BufferedReader reader = **new** BufferedReader(**new**FileReader("DeliveryStatusCodes.txt"));

**while** ((strRead=reader.readLine() ) != **null**)

                     {

                           String splitarray[] = strRead.split(",");

                           //parse record and load into HahMap

*DeliveryCodesMap*.put(splitarray[0].trim(), splitarray[1].trim());

                     }

              }

**catch** (FileNotFoundException e) {

              e.printStackTrace();

              }**catch**( IOException e ) {

                       e.printStackTrace();

                }

       }

}

**Driver Class: SmsDriver.java**

**import** org.apache.hadoop.conf.Configuration;

**import** org.apache.hadoop.conf.Configured;

**import** org.apache.hadoop.fs.Path;

**import** org.apache.hadoop.io.Text;

**import** org.apache.hadoop.mapred.FileOutputFormat;

**import** org.apache.hadoop.mapred.JobClient;

**import** org.apache.hadoop.mapred.~~JobConf~~;

**import** org.apache.hadoop.mapred.~~TextInputFormat~~;

**import** org.apache.hadoop.mapred.lib.MultipleInputs;

**import** org.apache.hadoop.util.Tool;

**import** org.apache.hadoop.util.ToolRunner;

**public** **class** SmsDriver **extends** Configured **implements** Tool

{

**public** **int** run(String[] args) **throws** Exception {

              //get the configuration parameters and assigns a job name

~~JobConf~~ conf = **new** JobConf(getConf(), SmsDriver.**class**);

              conf.setJobName("SMS Reports");

              //setting key value types for mapper and reducer outputs

              conf.setOutputKeyClass(Text.**class**);

              conf.setOutputValueClass(Text.**class**);

              //specifying the custom reducer class

              conf.setReducerClass(SmsReducer.**class**);

              //Specifying the input directories(@ runtime) andMappers independently for inputs from multiple sources

              MultipleInputs.*addInputPath*(conf, **new** Path(args[0]),~~TextInputFormat~~.**class**, UserFileMapper.**class**);

              MultipleInputs.*addInputPath*(conf, **new** Path(args[1]),~~TextInputFormat~~.**class**, DeliveryFileMapper.**class**);

              //Specifying the output directory @ runtime

              FileOutputFormat.*setOutputPath*(conf, **new** Path(args[2]));

              JobClient.*runJob*(conf);

**return** 0;

       }

**public** **static** **void** main(String[] args) **throws** Exception {

**int** res = ToolRunner.*run*(**new** Configuration(), **new**SmsDriver(),

                           args);

              System.*exit*(res);

       }

}

Let us go in for a small code walk through, I’m not going in for each and every line of code as it is commented and is bit self-explanatory. The few points to keep in mind

1.       The only difference in the code is that we are using MultipleInputFormat instead of FileInputFormat. This is necessary as we use two mappers and we need the output of the two mappers to be processed by a single reducer

2.       When we normally execute our map reduce with the hadoop jar command the last two arguments on the command line represent the input and output dir in hdfs. But here instead of two we’d have three input locations and one output location.

3.       The second key thing to be noted here is that in place of input locations don’t provide the full path with file names. Provide the input directories instead. Load the two files in two separate directories and provide the corresponding paths to mappers.

4.       Since my driver is getting the arguments from command line, the order of arguments is also very critical. Make sure that the input directories always point to their corresponding mappers itself.

You can run the above example with the following command on CLI as

hadoop jar /home/bejoys/samples/ smsMarketing.jar com.bejoy.samples.smsmarketing.SmsDriver  -files /home/bejoys/samples/ DeliveryStatusCodes.txt /userdata/bejoys/samples/sms/consumerdata /userdata/bejoys/samples/sms/deliveryinformation /userdata/bejoys/samples/sms/output

Note:

                     i.            Since the join the happening on reduce, it is termed as a reduce side join.

                   ii.            This is a very basic approach to implement joins in map reduce and is for those who have a basic knowledge on map reduce programming. You can implement it in more sophisticated manner in mapreduce frame work using DataJoin Mappers and Reducers with TaggedMap Output Types.

But if it is a join then I’d strongly recommend you to go in with Pig or Hive as both of these are highly optimized for implementing joins. Also you can eliminate the coding effort you need to put in. It is not exaggerating if I say I can implement the same in a single step using hive. Let us just check it out

# MapReduce Algorithms - Understanding Data Joins Part II

[**[](https://dzone.com/users/719987/bbejeck.html)**](https://dzone.com/users/719987/bbejeck.html)**by**

[**Bill Bejeck**](https://dzone.com/users/719987/bbejeck.html)

**·**

**May. 28, 14 ·**[**Big Data Zone**](https://dzone.com/big-data-analytics-tutorials-tools-news)

**Like** [**(0)**](https://dzone.com/articles/mapreduce-algorithms-0)

**Comment (0)**

**Save**

[**Tweet**](https://dzone.com/articles/mapreduce-algorithms-0)

**6,325 Views**

Join the DZone community and get the full member experience.

[**JOIN FOR FREE**](https://dzone.com/articles/mapreduce-algorithms-0)

*Learn how you can*[*maximize big data in the cloud*](https://dzone.com/go?i=177153&u=http%3A%2F%2Fhortonworks.com%2Finfo%2Fmaximize-big-data-cloud-aws-ebook%2F%3Futm_medium%3Dsponsored-content%26utm_source%3Ddzone%26utm_campaign%3Daws)*with Apache Hadoop. Download this eBook now. Brought to you in partnership with*[*Hortonworks.*](https://dzone.com/go?i=177153&u=http%3A%2F%2Fhortonworks.com%2Finfo%2Fmaximize-big-data-cloud-aws-ebook%2F%3Futm_medium%3Dsponsored-content%26utm_source%3Ddzone%26utm_campaign%3Daws)

It’s been awhile since I last posted, and like last time I took a big break, I was taking some classes on Coursera. This time it was [Functional Programming Principals in Scala](https://www.coursera.org/course/progfun) and [Principles of Reactive Programming](https://www.coursera.org/course/reactive). I found both of them to be great courses and would recommend taking either one if you have the time. In this post we resume our series on implementing the algorithms found in [Data-Intensive Text Processing with MapReduce](http://www.amazon.com/Data-Intensive-Processing-MapReduce-Synthesis-Technologies/dp/1608453421), this time covering map-side joins. As we can guess from the name, map-side joins join data exclusively during the mapping phase and completely skip the reducing phase. In the last post on data joins we covered [reduce side joins](http://codingjunkie.net/mapreduce-reduce-joins/). Reduce-side joins are easy to implement, but have the drawback that all data is sent across the network to the reducers. Map-side joins offer substantial gains in performance since we are avoiding the cost of sending data across the network. However, unlike reduce-side joins, map-side joins require very specific criteria be met. Today we will discuss the requirements for map-side joins and how we can implement them.

### Map-Side Join Conditions

To take advantage of map-side joins our data must meet one of following criteria:

1. The datasets to be joined are already sorted by the same key and have the same number of partitions
2. Of the two datasets to be joined, one is small enough to fit into memory

We are going to consider the first scenario where we have two (or more) datasets that need to be joined, but are too large to fit into memory. We will assume the worst case scenario, the files aren’t sorted or partitioned the same.

### Data Format

Before we start, let’s take a look at the data we are working with. We will have two datasets:

1. The first dataset consists of a GUID, First Name, Last Name, Address, City and State
2. The second dataset consists of a GUID and Employer information

Both datasets are comma delimited and the join-key (GUID) is in the first position. After the join we want the employer information from dataset two to be appended to the end of dataset one. Additionally, we want to keep the GUID in the first position of dataset one, but remove the GUID from dataset two. Dataset 1:

aef9422c-d08c-4457-9760-f2d564d673bc,Linda,Narvaez,3253 Davis Street,Atlanta,GA

08db7c55-22ae-4199-8826-c67a5689f838,John,Gregory,258 Khale Street,Florence,SC

de68186a-1004-4211-a866-736f414eac61,Charles,Arnold,1764 Public Works Drive,Johnson City,TN

6df1882d-4c81-4155-9d8b-0c35b2d34284,John,Schofield,65 Summit Park Avenue,Detroit,MI

Dataset 2:

de68186a-1004-4211-a866-736f414eac61,Jacobs

6df1882d-4c81-4155-9d8b-0c35b2d34284,Chief Auto Parts

aef9422c-d08c-4457-9760-f2d564d673bc,Earthworks Yard Maintenance

08db7c55-22ae-4199-8826-c67a5689f838,Ellman's Catalog Showrooms

Joined Results:

08db7c55-22ae-4199-8826-c67a5689f838,John,Gregory,258 Khale Street,Florence,SC,Ellman's Catalog Showrooms

6df1882d-4c81-4155-9d8b-0c35b2d34284,John,Schofield,65 Summit Park Avenue,Detroit,MI,Chief Auto Parts

aef9422c-d08c-4457-9760-f2d564d673bc,Linda,Narvaez,3253 Davis Street,Atlanta,GA,Earthworks Yard Maintenance

de68186a-1004-4211-a866-736f414eac61,Charles,Arnold,1764 Public Works Drive,Johnson City,TN,Jacobs

Now we move on to how we go about joining our two datasets.

### Map-Side Joins with Large Datasets

To be able to perform map-side joins we need to have our data sorted by the same key and have the same number of partitions, implying that all keys for any record are in the same partition. While this seems to be a tough requirement, it is easily fixed. Hadoop sorts all keys and guarantees that keys with the same value are sent to the same reducer. So by simply running a MapReduce job that does nothing more than output the data by the key you want to join on and specifying the exact same number of reducers for all datasets, we will get our data in the correct form. Considering the gains in efficiency from being able to do a map-side join, it may be worth the cost of running additional MapReduce jobs. It bears repeating at this point it is crucial all datasets specify the exact same number of reducers during the “preparation” phase when the data will be sorted and partitioned. In this post we will take two data-sets and run an initial MapReduce job on both to do the sorting and partitioning and then run a final job to perform the map-side join. First let’s cover the MapReduce job to sort and partition our data in the same way.

#### Step One: Sorting and Partitioning

First we need to create a Mapper that will simply choose the key for sorting by a given index:

public class SortByKeyMapper extends Mapper<LongWritable, Text, Text, Text> {

private int keyIndex;

private Splitter splitter;

private Joiner joiner;

private Text joinKey = new Text();

@Override

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

String separator = context.getConfiguration().get("separator");

keyIndex = Integer.parseInt(context.getConfiguration().get("keyIndex"));

splitter = Splitter.on(separator);

joiner = Joiner.on(separator);

}

@Override

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

Iterable<String> values = splitter.split(value.toString());

joinKey.set(Iterables.get(values,keyIndex));

if(keyIndex != 0){

value.set(reorderValue(values,keyIndex));

}

context.write(joinKey,value);

}

private String reorderValue(Iterable<String> value, int index){

List<String> temp = Lists.newArrayList(value);

String originalFirst = temp.get(0);

String newFirst = temp.get(index);

temp.set(0,newFirst);

temp.set(index,originalFirst);

return joiner.join(temp);

}

}

The SortByKeyMapper simply sets the value of the joinKey by extracting the value from the given line of text found at the position given by the configuration parameter keyIndex. Also, if the keyIndex is not equal to zero, we swap the order of the values found in the first position and the keyIndexposition. Although this is a questionable feature, We’ll discuss why we are doing this later. Next we need a Reducer:

public class SortByKeyReducer extends Reducer<Text,Text,NullWritable,Text> {

private static final NullWritable nullKey = NullWritable.get();

@Override

protected void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {

for (Text value : values) {

context.write(nullKey,value);

}

}

}

The SortByKeyReducer writes out all values for the given key, but throws out the key and writes a NullWritable instead. In the next section we will explain why we are not using the key.

#### Step Two: The Map-Side join

When performing a map-side join the records are merged before they reach the mapper. To achieve this, we use the [CompositeInputFormat](http://hadoop.apache.org/docs/current2/api/org/apache/hadoop/mapreduce/lib/join/CompositeInputFormat.html). We will also need to set some configuration properties. Let’s look at how we will configure our map-side join:

private static Configuration getMapJoinConfiguration(String separator, String... paths) {

Configuration config = new Configuration();

config.set("mapreduce.input.keyvaluelinerecordreader.key.value.separator", separator);

String joinExpression = CompositeInputFormat.compose("inner", KeyValueTextInputFormat.class, paths);

config.set("mapred.join.expr", joinExpression);

config.set("separator", separator);

return config;

}

First, we are specifying the character that separates the key and values by setting the mapreduce.input.keyvaluelinerecordreader.key.value.separator property. Next we use the CompositeInputFormat.compose method to create a “join expression” specifying an inner join by using the word “inner”, then specifying the input format to use, the [KeyValueTextInput](http://hadoop.apache.org/docs/current2/api/org/apache/hadoop/mapreduce/lib/input/KeyValueTextInputFormat.html)class and finally a String varargs representing the paths of the files to join (which are the output paths of the map-reduce jobs ran to sort and partition the data). The KeyValueTextInputFormat class will use the separator character to set the first value as the key and the rest will be used for the value.

#### Mapper for the join

Once the values from the source files have been joined, the Mapper.map method is called, it will receive a Text object for the key (the same key across joined records) and a TupleWritable that is composed of the values joined from our input files for a given key. Remember we want our final output to have the join-key in the first position, followed by all of joined values in one delimited String. To achieve this we have a custom mapper to put our data in the correct format:

public class CombineValuesMapper extends Mapper<Text, TupleWritable, NullWritable, Text> {

private static final NullWritable nullKey = NullWritable.get();

private Text outValue = new Text();

private StringBuilder valueBuilder = new StringBuilder();

private String separator;

@Override

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

separator = context.getConfiguration().get("separator");

}

@Override

protected void map(Text key, TupleWritable value, Context context) throws IOException, InterruptedException {

valueBuilder.append(key).append(separator);

for (Writable writable : value) {

valueBuilder.append(writable.toString()).append(separator);

}

valueBuilder.setLength(valueBuilder.length() - 1);

outValue.set(valueBuilder.toString());

context.write(nullKey, outValue);

valueBuilder.setLength(0);

}

}

In the CombineValuesMapper we are appending the key and all the joined values into one delimited String. Here we can finally see the reason why we threw the join-key away in the previous MapReduce jobs. Since the key is the first position in the values for all the datasets to be joined, our mapper naturally eliminates the duplicate keys from the joined datasets. All we need to do is insert the given key into a StringBuilder, then append the values contained in the TupleWritable.

### Putting It All Together

Now we have all the code in place to run a map-side join on large datasets. Let’s take a look at how we will run all the jobs together. As was stated before, we are assuming that our data is not sorted and partitioned the same, so we will need to run N (2 in this case) MapReduce jobs to get the data in the correct format. After the initial sorting/partitioning jobs run, the final job performing the actual join will run.

public class MapSideJoinDriver {

public static void main(String[] args) throws Exception {

String separator = ",";

String keyIndex = "0";

int numReducers = 10;

String jobOneInputPath = args[0];

String jobTwoInputPath = args[1];

String joinJobOutPath = args[2];

String jobOneSortedPath = jobOneInputPath + "\_sorted";

String jobTwoSortedPath = jobTwoInputPath + "\_sorted";

Job firstSort = Job.getInstance(getConfiguration(keyIndex, separator));

configureJob(firstSort, "firstSort", numReducers, jobOneInputPath, jobOneSortedPath, SortByKeyMapper.class, SortByKeyReducer.class);

Job secondSort = Job.getInstance(getConfiguration(keyIndex, separator));

configureJob(secondSort, "secondSort", numReducers, jobTwoInputPath, jobTwoSortedPath, SortByKeyMapper.class, SortByKeyReducer.class);

Job mapJoin = Job.getInstance(getMapJoinConfiguration(separator, jobOneSortedPath, jobTwoSortedPath));

configureJob(mapJoin, "mapJoin", 0, jobOneSortedPath + "," + jobTwoSortedPath, joinJobOutPath, CombineValuesMapper.class, Reducer.class);

mapJoin.setInputFormatClass(CompositeInputFormat.class);

List<Job> jobs = Lists.newArrayList(firstSort, secondSort, mapJoin);

int exitStatus = 0;

for (Job job : jobs) {

boolean jobSuccessful = job.waitForCompletion(true);

if (!jobSuccessful) {

System.out.println("Error with job " + job.getJobName() + " " + job.getStatus().getFailureInfo());

exitStatus = 1;

break;

}

}

System.exit(exitStatus);

}

The MapSideJoinDriver does the basic configuration for running MapReduce jobs. One interesting point is the sorting/partitioning jobs specify 10 reducers each, while the final job explicitly sets the number of reducers to 0, since we are joining on the map-side and don’t need a reduce phase. Since we don’t have any complicated dependencies, we put the jobs in an ArrayList and run the jobs in linear order (lines 24-33).

### Results

Initially we had 2 files; name and address information in the first file and employment information in the second. Both files had a unique id in the first column. File one:

....

08db7c55-22ae-4199-8826-c67a5689f838,John,Gregory,258 Khale Street,Florence,SC

...

File two:

....

08db7c55-22ae-4199-8826-c67a5689f838,Ellman's Catalog Showrooms

....

Results:

08db7c55-22ae-4199-8826-c67a5689f838,John,Gregory,258 Khale Street,Florence,SC,Ellman's Catalog Showrooms

As we can see here, we’ve successfully joined the records together and maintained the format of the files without duplicate keys in the results.

#### Conclusion

In this post we’ve demonstrated how to perform a map-side join when both data sets are large and can’t fit into memory. If you get the feeling this takes a lot of work to pull off, you are correct. While in most cases we would want to use higher level tools like Pig or Hive, it’s helpful to know the mechanics of performing map-side joins with large datasets. This especially true on those occasions when you need to write a solution from scratch. Thanks for your time.

### *Joining Two Files Using MultipleInput In Hadoop MapReduce - MapSide Join*

There are cases where we need to get 2 files as input and join them based on id or something like that.  
Two different large data can be joined in map reduce programming also. Joins in Map phase refers as Map side join, while join at reduce side called as reduce side join.    
MapSide can be achieved using **MultipleInputFormat** in Hadoop.  
  
Say I have 2 files ,One file with EmployeeID,Name,Designation and another file with EmployeeID,Salary,Department.  
  
File1.txt  
1 Anne,Admin  
2 Gokul,Admin  
3 Janet,Sales  
4 Hari,Admin  
  
AND  
  
File2.txt  
1 50000,A  
2 50000,B  
3 60000,A  
4 50000,C  
  
We will try to join these files into one based on **EmployeeID**  
The result we aim at is   
  
1 Anne,Admin,50000,A  
2 Gokul,Admin,50000,B  
3 Janet,Sales,60000,A  
4 Hari,Admin,50000,C  
  
Here in both file File1.txt,File2.txt we can see that we need to join the records based on id.  So the employeeId's are common.  
We will write 2 map jobs to process these files.  
  
**Processing File1.txt**

public void map(LongWritable k, Text value, Context context) throws IOException, InterruptedException

{

String line=value.toString();

String[] words=line.split("\t");

keyEmit.set(words[0]);

valEmit.set(words[1]);

context.write(keyEmit, valEmit);

}

The above map job process File1.txt  
String[] words=line.split("\t");  
splits each line with \t space so words[0] will be the employeeId which we pass it as key and the rest as value.  
  
eg: 1 Anne,Admin  
words[0] = 1  
words[1] = Anne,Admin  
  
Or else you can also use KeyValueTextInputFormat.class as InputFormat. This class gives key as employeeId and the rest as value.  
You dont need to split it.  
  
**Processing File2.txt**

public void map(LongWritable k, Text v, Context context) throws IOException, InterruptedException

{

String line=v.toString();

String[] words=line.split(" ");

keyEmit.set(words[0]);

valEmit.set(words[1]);

context.write(keyEmit, valEmit);

}

The above map job process File2.txt  
  
eg: 1 50000,A  
words[0] = 1  
words[1] = 50000,A  
  
If the files are of same delimiter and ID comes first you can **resuse** the same map job  
  
Lets write a commomn Reducer task to join the data using key.

String merge = "";

public void reduce(Text key, Iterable<Text> values, Context context)

{

int i =0;

for(Text value:values)

{

if(i == 0){

merge = value.toString()+",";

}

else{

merge += value.toString();

}

i++;

}

valEmit.set(merge);

context.write(key, valEmit);

}

Here we will be caching 1 data from a mapper and appends it to string "merge".  
And emit employeeId as key and merge as value.  
  
Now we need to furnish our Driver class to take 2 inputs and use **MultipleInputFormat**as InputFormat

public int run(String[] args) throws Exception {

Configuration c=new Configuration();

String[] files=new GenericOptionsParser(c,args).getRemainingArgs();

Path p1=new Path(files[0]);

Path p2=new Path(files[1]);

Path p3=new Path(files[2]);

FileSystem fs = FileSystem.get(c);

if(fs.exists(p3)){

fs.delete(p3, true);

}

Job job = new Job(c,"Multiple Job");

job.setJarByClass(MultipleFiles.class);

MultipleInputs.addInputPath(job, p1, TextInputFormat.class, MultipleMap1.class);

MultipleInputs.addInputPath(job,p2, TextInputFormat.class, MultipleMap2.class);

job.setReducerClass(MultipleReducer.class);

.

.

}

**MultipleInputs.addInputPath(job, p1, TextInputFormat.class, MultipleMap1.class);**

**MultipleInputs.addInputPath(job,p2, TextInputFormat.class, MultipleMap2.class);**

p1,p2 are the Path variable holding 2 input files.  
You can find the code in [Github](https://github.com/studhadoop/Mapside-Join)

There is one more case where we can make our output in a sequential manner.

Say if we need to get the output as below

1 Anne,Admin,50000,A

2 Gokul,Admin,50000,B

3 Janet,Sales,60000,A

4 Hari,Admin,50000,C

Inorder to achieve the same we can make use of **TextPair Writable** concepts in Hadoop.

You can find the working code [in github](https://github.com/studhadoop/MapSide-Join-In-order) . Thanks to one of  my blog reader Ravi Kumar who sorted out the sequence in order.

**Joins in Hadoop MapReduce**  
Hadoop MapReduce supports two types of joins-

* Map Side Join
* Reduce Side Join([Reduce side join example](http://www.javamakeuse.com/2016/03/mapreduce-reduce-side-join-example-in.html))

In this tutorial, I am going to explain you the usage of Map side join.  
**Map side Join**  
You can use Map side join using two different ways based on your datasets, and those depends on below conditions -

1. Both datasets are must be divided into the same number of partitions, and must be already sorted by the same key.
2. From the two datasets one must be small(something like master dataset) and able to fit into the memory of each nodes.

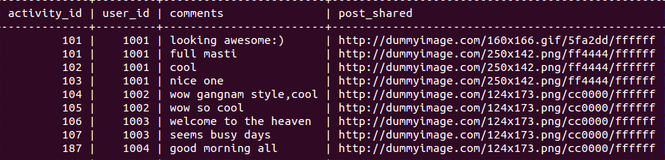
In this tutorial, I will show you the second one, and to use the second one you need distributed cache to keep the small(master) dataset into the memory of each nodes.  
  
OK, Let's find the user activity on social media, what are the actions user performed on popular social media like commenting on post, shared something, like something etc.  
And for these we have two different log files -

* user.log
* user\_activity.log

Here is the tabular view of these datasets,  
1. user.log

[](https://3.bp.blogspot.com/-5iYboihUV9I/VvA-rAEeRfI/AAAAAAAAAmA/6pC-1E9omf4gmSeXM_p23DSsvQDlA8IPw/s1600/user.png)

2. user\_activity.log

[](https://1.bp.blogspot.com/-eYOWuGfU03U/VvA_YSzaDwI/AAAAAAAAAmI/dKlaxCLs_OUS6FjZd4JjPA8I1p5ZoLnMA/s1600/user_activity_log.png)

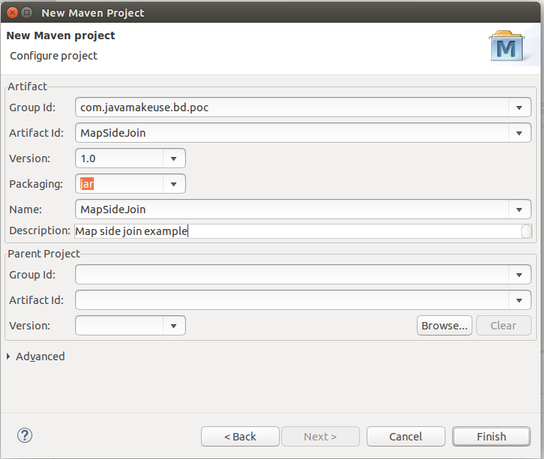
3. Expected output

[](https://1.bp.blogspot.com/-3qGVMi5vXsE/VvA_-fSv7gI/AAAAAAAAAmM/1ZkTX2K-5hkMq8uYUDkKR_SfkFJMuxsig/s1600/joined.png)

#### **Tools and Technologies we are using here:**

* Java 8
* Eclipse Mars
* Hadoop 2.7.1
* Maven 3.3
* Ubuntu 14(Linux OS)

**Step 1. Create a new maven project**  
Go to File Menu then New->Maven Project, and provide the required details, see the below attached screen.

[](https://3.bp.blogspot.com/-Mx_2rcR6hzk/VvBCsSM86OI/AAAAAAAAAmc/c3faML2dnVkj5GW6RSHjXHTOZzUbywQnQ/s1600/map_sidejoin.png)

**Step 2. Edit pom.xml**  
Double click on your project's pom.xml file, it will looks like this with very limited information.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | <project xmlns="<http://maven.apache.org/POM/4.0.0>"xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"xsi:schemaLocation="<http://maven.apache.org/POM/4.0.0><http://maven.apache.org/xsd/maven-4.0.0.xsd>"> | | |
| 2 | | <modelVersion>4.0.0</modelVersion> |

|  |  |  |
| --- | --- | --- |
| 3 | <groupId>com.javamakeuse.bd.poc</groupId> | |
| 4 | <artifactId>MapSideJoin</artifactId> |

|  |  |
| --- | --- |
| 5 | <version>1.0</version> |
| 6 | <name>MapSideJoin</name> | |

|  |  |  |
| --- | --- | --- |
| 7 | <description>MapSideJoin Example in Map Reduce</description> | |
| 8 | </project> |

Now edit this pom.xml file and add Hadoop dependencies, below is the complete pom.xml file, just copy and paste, it will work.  
**pom.xml**

|  |  |  |  |
| --- | --- | --- | --- |
| 01 | <project xmlns="<http://maven.apache.org/POM/4.0.0>"xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"xsi:schemaLocation="<http://maven.apache.org/POM/4.0.0><http://maven.apache.org/xsd/maven-4.0.0.xsd>"> | | |
| 02 | | <modelVersion>4.0.0</modelVersion> |

|  |  |  |
| --- | --- | --- |
| 03 | <groupId>com.javamakeuse.bd.poc</groupId> | |
| 04 | <artifactId>MapSideJoin</artifactId> |

|  |  |
| --- | --- |
| 05 | <version>1.0</version> |
| 06 | <name>MapSideJoin</name> | |

|  |  |  |
| --- | --- | --- |
| 07 | <description>MapSideJoin Example</description> | |
| 08 | <dependencies> |

|  |  |
| --- | --- |
| 09 | <dependency> |
| 10 | <groupId>org.apache.hadoop</groupId> | |

|  |  |  |
| --- | --- | --- |
| 11 | <artifactId>hadoop-client</artifactId> | |
| 12 | <version>2.7.1</version> |

|  |  |  |
| --- | --- | --- |
| 13 | </dependency> | |
| 14 | <dependency> |

|  |  |
| --- | --- |
| 15 | <groupId>org.apache.hadoop</groupId> |
| 16 | <artifactId>hadoop-mapreduce-client-core</artifactId> | |

|  |  |  |
| --- | --- | --- |
| 17 | <version>2.7.1</version> | |
| 18 | </dependency> |

|  |  |  |
| --- | --- | --- |
| 19 | </dependencies> | |
| 20 | <build> |

|  |  |
| --- | --- |
| 21 | <plugins> |
| 22 | <plugin> |

|  |  |
| --- | --- |
| 23 | <groupId>org.apache.maven.plugins</groupId> |
| 24 | <artifactId>maven-compiler-plugin</artifactId> | |

|  |  |
| --- | --- |
| 25 | <configuration> |
| 26 | <source>1.7</source> | |

|  |  |  |
| --- | --- | --- |
| 27 | <target>1.7</target> | |
| 28 | </configuration> |

|  |  |
| --- | --- |
| 29 | </plugin> |
| 30 | </plugins> |

|  |  |
| --- | --- |
| 31 | </build> |
| 32 | </project> | |

**Step 3. UserActivityVO.java**  
This is our value object class, which will contains the fields needs to be written as an output of the project.

|  |  |  |
| --- | --- | --- |
| 01 | package com.javamakeuse.bd.poc.vo; | |
| 02 |  |

|  |  |
| --- | --- |
| 03 | import java.io.DataInput; |
| 04 | import java.io.DataOutput; | |

|  |  |
| --- | --- |
| 05 | import java.io.IOException; |
| 06 | import org.apache.hadoop.io.Writable; | |

|  |  |
| --- | --- |
| 07 |  |
| 08 | public class UserActivityVO implements Writable { | |

|  |  |
| --- | --- |
| 09 |  |
| 10 | private int userId; | |

|  |  |
| --- | --- |
| 11 | private String userName; |
| 12 | private String comments; |

|  |  |  |
| --- | --- | --- |
| 13 | private String postShared; | |
| 14 |  |

|  |  |  |
| --- | --- | --- |
| 15 | public int getUserId() { | |
| 16 | return userId; |

|  |  |
| --- | --- |
| 17 | } |
| 18 | public void setUserId(int userId) { | |

|  |  |  |
| --- | --- | --- |
| 19 | this.userId = userId; | |
| 20 | } |

|  |  |  |
| --- | --- | --- |
| 21 | public String getUserName() { | |
| 22 | return userName; |

|  |  |
| --- | --- |
| 23 | } |
| 24 | public void setUserName(String userName) { | |

|  |  |  |
| --- | --- | --- |
| 25 | this.userName = userName; | |
| 26 | } |

|  |  |  |
| --- | --- | --- |
| 27 | public String getComments() { | |
| 28 | return comments; |

|  |  |
| --- | --- |
| 29 | } |
| 30 | public void setComments(String comments) { | |

|  |  |  |
| --- | --- | --- |
| 31 | this.comments = comments; | |
| 32 | } |

|  |  |  |
| --- | --- | --- |
| 33 | public String getPostShared() { | |
| 34 | return postShared; |

|  |  |
| --- | --- |
| 35 | } |
| 36 | public void setPostShared(String postShared) { | |

|  |  |  |
| --- | --- | --- |
| 37 | this.postShared = postShared; | |
| 38 | } |

|  |  |
| --- | --- |
| 39 | @Override |
| 40 | public void write(DataOutput out) throws IOException { | |

|  |  |
| --- | --- |
| 41 | out.writeInt(userId); |
| 42 | out.writeUTF(userName); | |

|  |  |
| --- | --- |
| 43 | out.writeUTF(comments); |
| 44 | out.writeUTF(postShared); | |

|  |  |
| --- | --- |
| 45 | } |
| 46 | @Override | |

|  |  |  |
| --- | --- | --- |
| 47 | public void readFields(DataInput in) throws IOException { | |
| 48 | userId = in.readInt(); |

|  |  |
| --- | --- |
| 49 | userName = in.readUTF(); |
| 50 | comments = in.readUTF(); |

|  |  |  |
| --- | --- | --- |
| 51 | postShared = in.readUTF(); | |
| 52 | } |

|  |  |
| --- | --- |
| 53 |  |
| 54 | @Override | |

|  |  |
| --- | --- |
| 55 | public String toString() { |
| 56 | return "UserActivityVO [userId=" + userId + ", userName=" + userName + ",   comments=" + comments | |

|  |  |  |
| --- | --- | --- |
| 57 | + ", postShared=" + postShared + "]"; | |
| 58 | } |

|  |  |
| --- | --- |
| 59 |  |
| 60 | } | |

**Step 4. UserActivityMapper.java (Mapper)**  
Inside this mapper class, we are setting the properties of UserActivityVO class, user.log file is in from distributed cache.

|  |  |  |
| --- | --- | --- |
| 01 | package com.javamakeuse.bd.poc.mapper; | |
| 02 |  |

|  |  |  |
| --- | --- | --- |
| 03 | import java.io.BufferedReader; | |
| 04 | import java.io.FileReader; |

|  |  |  |
| --- | --- | --- |
| 05 | import java.io.IOException; | |
| 06 | import java.util.HashMap; |

|  |  |
| --- | --- |
| 07 | import java.util.Map; |
| 08 | import org.apache.hadoop.io.IntWritable; | |

|  |  |  |
| --- | --- | --- |
| 09 | import org.apache.hadoop.io.LongWritable; | |
| 10 | import org.apache.hadoop.io.Text; |

|  |  |
| --- | --- |
| 11 | import org.apache.hadoop.mapreduce.Mapper; |
| 12 | import com.javamakeuse.bd.poc.vo.UserActivityVO; | |

|  |  |
| --- | --- |
| 13 |  |
| 14 | public class UserActivityMapper extends Mapper<LongWritable, Text, IntWritable, UserActivityVO> { | |

|  |  |
| --- | --- |
| 15 |  |
| 16 | // user map to keep the userId-userName | |

|  |  |  |
| --- | --- | --- |
| 17 | private Map<Integer, String> userMap = new HashMap<>(); | |
| 18 |  |

|  |  |
| --- | --- |
| 19 | @Override |
| 20 | protected void map(LongWritable key, Text value, | |

|  |  |  |
| --- | --- | --- |
| 21 | Mapper<LongWritable, Text, IntWritable, UserActivityVO>.Context context) | |
| 22 | throws IOException, InterruptedException { |

|  |  |
| --- | --- |
| 23 |  |
| 24 | String[] columns = value.toString().split("\t"); | |

|  |  |
| --- | --- |
| 25 | if (columns != null && columns.length > 2) { |
| 26 | UserActivityVO userActivityVO = new UserActivityVO(); | |

|  |  |  |
| --- | --- | --- |
| 27 | userActivityVO.setUserId(Integer.parseInt(columns[1])); | |
| 28 | userActivityVO.setComments(columns[2]); |

|  |  |
| --- | --- |
| 29 | userActivityVO.setPostShared(columns[3]); |
| 30 | userActivityVO.setUserName(userMap.get(userActivityVO.getUserId())); | |

|  |  |
| --- | --- |
| 31 | // writing into context |
| 32 | context.write(new IntWritable(userActivityVO.getUserId()), userActivityVO); | |

|  |  |  |
| --- | --- | --- |
| 33 | } | |
| 34 | } |

|  |  |
| --- | --- |
| 35 |  |
| 36 | @Override | |

|  |  |  |
| --- | --- | --- |
| 37 | protected void setup(Mapper<LongWritable, Text, IntWritable, UserActivityVO>.Context context) | |
| 38 | throws IOException, InterruptedException { |

|  |  |  |
| --- | --- | --- |
| 39 | // loading user map in context | |
| 40 | loadUserInMemory(context); |

|  |  |  |
| --- | --- | --- |
| 41 | } | |
| 42 |  |

|  |  |  |
| --- | --- | --- |
| 43 | private void loadUserInMemory(Mapper<LongWritable, Text, IntWritable, UserActivityVO>.Context context) { | |
| 44 | // user.log is in distributed cache |

|  |  |  |
| --- | --- | --- |
| 45 | try (BufferedReader br = new BufferedReader(newFileReader("user.log"))) { | |
| 46 | String line; |

|  |  |  |
| --- | --- | --- |
| 47 | while ((line = br.readLine()) != null) { | |
| 48 | String columns[] = line.split("\t"); |

|  |  |  |
| --- | --- | --- |
| 49 | userMap.put(Integer.parseInt(columns[0]), columns[1]); | |
| 50 | } |

|  |  |  |
| --- | --- | --- |
| 51 | } catch (IOException e) { | |
| 52 | e.printStackTrace(); |

|  |  |  |
| --- | --- | --- |
| 53 | } | |
| 54 |  |

|  |  |  |
| --- | --- | --- |
| 55 | } | |
| 56 | } |

**Step 5. UserActivityReducer.java (Reducer)**  
The reducer class just iterating the values and writing into the context.

|  |  |  |
| --- | --- | --- |
| 01 | package com.javamakeuse.bd.poc.reducer; | |
| 02 |  |

|  |  |
| --- | --- |
| 03 | import java.io.IOException; |
| 04 | import org.apache.hadoop.io.IntWritable; | |

|  |  |
| --- | --- |
| 05 | import org.apache.hadoop.io.NullWritable; |
| 06 | import org.apache.hadoop.mapreduce.Reducer; | |

|  |  |  |
| --- | --- | --- |
| 07 | import com.javamakeuse.bd.poc.vo.UserActivityVO; | |
| 08 |  |

|  |  |  |
| --- | --- | --- |
| 09 | public class UserActivityReducer extends Reducer<IntWritable, UserActivityVO, UserActivityVO, NullWritable> { | |
| 10 |  |

|  |  |  |
| --- | --- | --- |
| 11 | NullWritable value = NullWritable.get(); | |
| 12 |  |

|  |  |
| --- | --- |
| 13 | @Override |
| 14 | protected void reduce(IntWritable key, Iterable<UserActivityVO> values, | |

|  |  |  |
| --- | --- | --- |
| 15 | Reducer<IntWritable, UserActivityVO, UserActivityVO, NullWritable>.Context context) | |
| 16 | throws IOException, InterruptedException { |

|  |  |  |
| --- | --- | --- |
| 17 | for (UserActivityVO userActivityVO : values) { | |
| 18 | context.write(userActivityVO, value); |

|  |  |  |
| --- | --- | --- |
| 19 | } | |
| 20 | } |

|  |  |
| --- | --- |
| 21 | } |

**Step 6. UserActivityDriver.java (Driver)**  
Here we are adding user.log file into the distributed cache.

|  |  |  |
| --- | --- | --- |
| 01 | package com.javamakeuse.bd.poc.driver; | |
| 02 |  |

|  |  |
| --- | --- |
| 03 | import java.net.URI; |
| 04 | import org.apache.hadoop.conf.Configured; | |

|  |  |
| --- | --- |
| 05 | import org.apache.hadoop.fs.Path; |
| 06 | import org.apache.hadoop.io.IntWritable; | |

|  |  |  |
| --- | --- | --- |
| 07 | import org.apache.hadoop.io.NullWritable; | |
| 08 | import org.apache.hadoop.mapreduce.Job; |

|  |  |
| --- | --- |
| 09 | import org.apache.hadoop.mapreduce.lib.input.FileInputFormat; |
| 10 | import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat; | |

|  |  |
| --- | --- |
| 11 | import org.apache.hadoop.util.Tool; |
| 12 | import org.apache.hadoop.util.ToolRunner; | |

|  |  |
| --- | --- |
| 13 | import com.javamakeuse.bd.poc.mapper.UserActivityMapper; |
| 14 | import com.javamakeuse.bd.poc.reducer.UserActivityReducer; | |

|  |  |  |
| --- | --- | --- |
| 15 | import com.javamakeuse.bd.poc.vo.UserActivityVO; | |
| 16 |  |

|  |  |  |
| --- | --- | --- |
| 17 | public class UserActivityDriver extends Configured implements Tool { | |
| 18 |  |

|  |  |  |
| --- | --- | --- |
| 19 | public static void main(String[] args) { | |
| 20 | try { |

|  |  |  |
| --- | --- | --- |
| 21 | int status = ToolRunner.run(new UserActivityDriver(), args); | |
| 22 | System.exit(status); |

|  |  |  |
| --- | --- | --- |
| 23 | } catch (Exception e) { | |
| 24 | e.printStackTrace(); |

|  |  |  |
| --- | --- | --- |
| 25 | } | |
| 26 | } |

|  |  |
| --- | --- |
| 27 |  |
| 28 | public int run(String[] args) throws Exception { | |

|  |  |
| --- | --- |
| 29 | if (args.length != 2) { |
| 30 | System.err.printf("Usage: %s [generic options] <input1> <output>\n", getClass().getSimpleName()); | |

|  |  |  |
| --- | --- | --- |
| 31 | ToolRunner.printGenericCommandUsage(System.err); | |
| 32 | return -1; |

|  |  |
| --- | --- |
| 33 | } |
| 34 | Job job = Job.getInstance(); | |

|  |  |
| --- | --- |
| 35 | job.setJarByClass(getClass()); |
| 36 | job.setJobName("MapSideJoin Example"); | |

|  |  |
| --- | --- |
| 37 | // input path |
| 38 | FileInputFormat.addInputPath(job, new Path(args[0])); | |

|  |  |
| --- | --- |
| 39 |  |
| 40 | // output path | |

|  |  |  |
| --- | --- | --- |
| 41 | FileOutputFormat.setOutputPath(job, new Path(args[1])); | |
| 42 |  |

|  |  |
| --- | --- |
| 43 | job.setMapperClass(UserActivityMapper.class); |
| 44 | job.setReducerClass(UserActivityReducer.class); | |

|  |  |
| --- | --- |
| 45 |  |
| 46 | job.setMapOutputKeyClass(IntWritable.class); | |

|  |  |  |
| --- | --- | --- |
| 47 | job.setMapOutputValueClass(UserActivityVO.class); | |
| 48 |  |

|  |  |  |
| --- | --- | --- |
| 49 | job.addCacheFile(new URI("<hdfs://localhost:9000/input/user.log>")); | |
| 50 | job.setOutputKeyClass(UserActivityVO.class); |

|  |  |  |
| --- | --- | --- |
| 51 | job.setOutputValueClass(NullWritable.class); | |
| 52 |  |

|  |  |  |
| --- | --- | --- |
| 53 | return job.waitForCompletion(true) ? 0 : 1; | |
| 54 | } |

|  |  |
| --- | --- |
| 55 |  |
| 56 | } | |

Done, next to run this program, you can run it using any eclipse also, below are the steps to run using terminal.  
  
**Step 7. Steps to execute MapSideJoin project**  
**i. Start Hadoop**components,open your terminal and type

|  |  |
| --- | --- |
| 1 | subodh@subodh-Inspiron-3520:~/software$ start-all.sh |

**ii. Verify Hadoop** started or not with jps command

|  |  |  |
| --- | --- | --- |
| 1 | subodh@subodh-Inspiron-3520:~/software$ jps | |
| 2 | 8385 NameNode |

|  |  |
| --- | --- |
| 3 | 8547 DataNode |
| 4 | 5701 org.eclipse.equinox.launcher\_1.3.100.v20150511-1540.jar | |

|  |  |
| --- | --- |
| 5 | 9446 Jps |
| 6 | 8918 ResourceManager | |

|  |  |
| --- | --- |
| 7 | 9054 NodeManager |
| 8 | 8751 SecondaryNameNode | |

You can verify with web-ui also using "*http://localhost:50070/explorer.html#/*" url.  
**iii. Create input**folder on HDFS with below command.

|  |  |
| --- | --- |
| 1 | subodh@subodh-Inspiron-3520:~/software$ hadoop fs -mkdir /input |

The above command will create an input folder on HDFS, you can verify it using web UI, Now time to move input file which we need to process, below is the command to copy the user.log and user\_activity.log input file on HDFS inside input folder.

|  |  |
| --- | --- |
| 1 | subodh@subodh-Inspiron-3520:~$ hadoop fs -copyFromLocal /home/subodh/programs/input/user.log user\_activity.log /input/ |

Note - user.log and user\_activity.log file is available inside this project source code, you would be able to download it from our downloadable link, you will find downloadable link at the end of this tutorial.  
  
**Step 8. Create & Execute jar file**  
We almost done,now create jar file of MapSideJoin source code. You can create jar file using eclipse or by using **mvn package** command.  
To execute MapSideJoin-1.0.jar file use below command

|  |  |
| --- | --- |
| 1 | hadoop jar /home/subodh/MapSideJoin-1.0.jar com.javamakeuse.bd.poc.driver.UserActivityDriver /input/user\_activity.log /output |

Above will generate below output and also create an output folder with output of the MapSideJoin project.

|  |  |
| --- | --- |
| 001 | 16/03/24 17:49:05 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable |
| 002 | 16/03/24 17:49:06 INFO Configuration.deprecation: session.id is deprecated. Instead, use dfs.metrics.session-id |

|  |  |
| --- | --- |
| 003 | 16/03/24 17:49:06 INFO jvm.JvmMetrics: Initializing JVM Metrics with processName=JobTracker, sessionId= |
| 004 | 16/03/24 17:49:06 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with ToolRunner to remedy this. |

|  |  |  |
| --- | --- | --- |
| 005 | 16/03/24 17:49:06 INFO input.FileInputFormat: Total input paths to process : 1 | |
| 006 | 16/03/24 17:49:06 INFO mapreduce.JobSubmitter: number of splits:1 |

|  |  |
| --- | --- |
| 007 | 16/03/24 17:49:06 INFO mapreduce.JobSubmitter: Submitting tokens for job: job\_local1445861261\_0001 |
| 008 | 16/03/24 17:49:07 INFO mapred.LocalDistributedCacheManager: Creating symlink: /tmp/hadoop-subodh/mapred/local/1458821947020/user.log <- /home/subodh/user.log |

|  |  |
| --- | --- |
| 009 | 16/03/24 17:49:07 INFO mapred.LocalDistributedCacheManager: Localized <hdfs://localhost:9000/input/user.log> as file:/tmp/hadoop-subodh/mapred/local/1458821947020/user.log |
| 010 | 16/03/24 17:49:07 INFO mapreduce.Job: The url to track the job: <http://localhost:8080/> |

|  |  |
| --- | --- |
| 011 | 16/03/24 17:49:07 INFO mapreduce.Job: Running job: job\_local1445861261\_0001 |
| 012 | 16/03/24 17:49:07 INFO mapred.LocalJobRunner: OutputCommitter set in config null |

|  |  |
| --- | --- |
| 013 | 16/03/24 17:49:07 INFO output.FileOutputCommitter: File Output Committer Algorithm version is 1 |
| 014 | 16/03/24 17:49:07 INFO mapred.LocalJobRunner: OutputCommitter is org.apache.hadoop.mapreduce.lib.output.FileOutputCommitter |

|  |  |
| --- | --- |
| 015 | 16/03/24 17:49:07 INFO mapred.LocalJobRunner: Waiting for map tasks |
| 016 | 16/03/24 17:49:07 INFO mapred.LocalJobRunner: Starting task: attempt\_local1445861261\_0001\_m\_000000\_0 | |

|  |  |
| --- | --- |
| 017 | 16/03/24 17:49:07 INFO output.FileOutputCommitter: File Output Committer Algorithm version is 1 |
| 018 | 16/03/24 17:49:07 INFO mapred.Task:  Using ResourceCalculatorProcessTree : [ ] |

|  |  |
| --- | --- |
| 019 | 16/03/24 17:49:07 INFO mapred.MapTask: Processing split: <hdfs://localhost:9000/input/user_activity.log:0>+282 |
| 020 | 16/03/24 17:49:07 INFO mapred.MapTask: (EQUATOR) 0 kvi 26214396(104857584) |

|  |  |  |
| --- | --- | --- |
| 021 | 16/03/24 17:49:07 INFO mapred.MapTask: mapreduce.task.io.sort.mb: 100 | |
| 022 | 16/03/24 17:49:07 INFO mapred.MapTask: soft limit at 83886080 |

|  |  |
| --- | --- |
| 023 | 16/03/24 17:49:07 INFO mapred.MapTask: bufstart = 0; bufvoid = 104857600 |
| 024 | 16/03/24 17:49:07 INFO mapred.MapTask: kvstart = 26214396; length = 6553600 | |

|  |  |  |
| --- | --- | --- |
| 025 | 16/03/24 17:49:07 INFO mapred.MapTask: Map output collector class = org.apache.hadoop.mapred.MapTask$MapOutputBuffer | |
| 026 | 16/03/24 17:49:07 INFO mapred.LocalJobRunner: |

|  |  |  |
| --- | --- | --- |
| 027 | 16/03/24 17:49:07 INFO mapred.MapTask: Starting flush of map output | |
| 028 | 16/03/24 17:49:07 INFO mapred.MapTask: Spilling map output |

|  |  |
| --- | --- |
| 029 | 16/03/24 17:49:07 INFO mapred.MapTask: bufstart = 0; bufend = 339; bufvoid = 104857600 |
| 030 | 16/03/24 17:49:07 INFO mapred.MapTask: kvstart = 26214396(104857584); kvend = 26214384(104857536); length = 13/6553600 |

|  |  |
| --- | --- |
| 031 | 16/03/24 17:49:07 INFO mapred.MapTask: Finished spill 0 |
| 032 | 16/03/24 17:49:07 INFO mapred.Task: Task:attempt\_local1445861261\_0001\_m\_000000\_0 is done. And is in the process of committing | |

|  |  |
| --- | --- |
| 033 | 16/03/24 17:49:07 INFO mapred.LocalJobRunner: map |
| 034 | 16/03/24 17:49:07 INFO mapred.Task: Task 'attempt\_local1445861261\_0001\_m\_000000\_0' done. | |

|  |  |  |
| --- | --- | --- |
| 035 | 16/03/24 17:49:07 INFO mapred.LocalJobRunner: Finishing task: attempt\_local1445861261\_0001\_m\_000000\_0 | |
| 036 | 16/03/24 17:49:07 INFO mapred.LocalJobRunner: map task executor complete. |

|  |  |
| --- | --- |
| 037 | 16/03/24 17:49:07 INFO mapred.LocalJobRunner: Waiting for reduce tasks |
| 038 | 16/03/24 17:49:07 INFO mapred.LocalJobRunner: Starting task: attempt\_local1445861261\_0001\_r\_000000\_0 | |

|  |  |
| --- | --- |
| 039 | 16/03/24 17:49:07 INFO output.FileOutputCommitter: File Output Committer Algorithm version is 1 |
| 040 | 16/03/24 17:49:07 INFO mapred.Task:  Using ResourceCalculatorProcessTree : [ ] |

|  |  |
| --- | --- |
| 041 | 16/03/24 17:49:07 INFO mapred.ReduceTask: Using ShuffleConsumerPlugin: org.apache.hadoop.mapreduce.task.reduce.Shuffle@396989af |
| 042 | 16/03/24 17:49:07 INFO reduce.MergeManagerImpl: MergerManager: memoryLimit=334338464, maxSingleShuffleLimit=83584616, mergeThreshold=220663392, ioSortFactor=10, memToMemMergeOutputsThreshold=10 |

|  |  |
| --- | --- |
| 043 | 16/03/24 17:49:07 INFO reduce.EventFetcher: attempt\_local1445861261\_0001\_r\_000000\_0 Thread started: EventFetcher for fetching Map Completion Events |
| 044 | 16/03/24 17:49:07 INFO reduce.LocalFetcher: localfetcher#1 about to shuffle output of map attempt\_local1445861261\_0001\_m\_000000\_0 decomp: 349 len: 353 to MEMORY |

|  |  |
| --- | --- |
| 045 | 16/03/24 17:49:07 INFO reduce.InMemoryMapOutput: Read 349 bytes from map-output for attempt\_local1445861261\_0001\_m\_000000\_0 |
| 046 | 16/03/24 17:49:07 INFO reduce.MergeManagerImpl: closeInMemoryFile -> map-output of size: 349, inMemoryMapOutputs.size() -> 1, commitMemory -> 0, usedMemory ->349 |

|  |  |  |
| --- | --- | --- |
| 047 | 16/03/24 17:49:07 INFO reduce.EventFetcher: EventFetcher is interrupted.. Returning | |
| 048 | 16/03/24 17:49:07 INFO mapred.LocalJobRunner: 1 / 1 copied. |

|  |  |  |
| --- | --- | --- |
| 049 | 16/03/24 17:49:07 INFO reduce.MergeManagerImpl: finalMerge called with 1 in-memory map-outputs and 0 on-disk map-outputs | |
| 050 | 16/03/24 17:49:07 INFO mapred.Merger: Merging 1 sorted segments |

|  |  |
| --- | --- |
| 051 | 16/03/24 17:49:07 INFO mapred.Merger: Down to the last merge-pass, with 1 segments left of total size: 343 bytes |
| 052 | 16/03/24 17:49:07 INFO reduce.MergeManagerImpl: Merged 1 segments, 349 bytes to disk to satisfy reduce memory limit |

|  |  |
| --- | --- |
| 053 | 16/03/24 17:49:07 INFO reduce.MergeManagerImpl: Merging 1 files, 353 bytes from disk |
| 054 | 16/03/24 17:49:07 INFO reduce.MergeManagerImpl: Merging 0 segments, 0 bytes from memory into reduce |

|  |  |
| --- | --- |
| 055 | 16/03/24 17:49:07 INFO mapred.Merger: Merging 1 sorted segments |
| 056 | 16/03/24 17:49:07 INFO mapred.Merger: Down to the last merge-pass, with 1 segments left of total size: 343 bytes | |

|  |  |
| --- | --- |
| 057 | 16/03/24 17:49:07 INFO mapred.LocalJobRunner: 1 / 1 copied. |
| 058 | 16/03/24 17:49:07 INFO Configuration.deprecation: mapred.skip.on is deprecated. Instead, use mapreduce.job.skiprecords | |

|  |  |  |
| --- | --- | --- |
| 059 | 16/03/24 17:49:07 INFO mapred.Task: Task:attempt\_local1445861261\_0001\_r\_000000\_0 is done. And is in the process of committing | |
| 060 | 16/03/24 17:49:07 INFO mapred.LocalJobRunner: 1 / 1 copied. |

|  |  |  |
| --- | --- | --- |
| 061 | | 16/03/24 17:49:07 INFO mapred.Task: Task attempt\_local1445861261\_0001\_r\_000000\_0 is allowed to commit now |
| 062 | 16/03/24 17:49:07 INFO output.FileOutputCommitter: Saved output of task 'attempt\_local1445861261\_0001\_r\_000000\_0' to <hdfs://localhost:9000/ua6/_temporary/0/task_local1445861261_0001_r_000000> | |

|  |  |
| --- | --- |
| 063 | 16/03/24 17:49:07 INFO mapred.LocalJobRunner: reduce > reduce |
| 064 | 16/03/24 17:49:07 INFO mapred.Task: Task 'attempt\_local1445861261\_0001\_r\_000000\_0' done. | |

|  |  |
| --- | --- |
| 065 | 16/03/24 17:49:07 INFO mapred.LocalJobRunner: Finishing task: attempt\_local1445861261\_0001\_r\_000000\_0 |
| 066 | 16/03/24 17:49:07 INFO mapred.LocalJobRunner: reduce task executor complete. |

|  |  |  |
| --- | --- | --- |
| 067 | 16/03/24 17:49:08 INFO mapreduce.Job: Job job\_local1445861261\_0001 running in uber mode : false | |
| 068 | 16/03/24 17:49:08 INFO mapreduce.Job:  map 100% reduce 100% |

|  |  |  |
| --- | --- | --- |
| 069 | 16/03/24 17:49:08 INFO mapreduce.Job: Job job\_local1445861261\_0001 completed successfully | |
| 070 | 16/03/24 17:49:08 INFO mapreduce.Job: Counters: 35 |

|  |  |
| --- | --- |
| 071 | File System Counters |
| 072 | FILE: Number of bytes read=17570 | |

|  |  |  |
| --- | --- | --- |
| 073 | FILE: Number of bytes written=576233 | |
| 074 | FILE: Number of read operations=0 |

|  |  |  |
| --- | --- | --- |
| 075 | FILE: Number of large read operations=0 | |
| 076 | FILE: Number of write operations=0 |

|  |  |
| --- | --- |
| 077 | HDFS: Number of bytes read=854 |
| 078 | HDFS: Number of bytes written=527 | |

|  |  |
| --- | --- |
| 079 | HDFS: Number of read operations=31 |
| 080 | HDFS: Number of large read operations=0 | |

|  |  |  |
| --- | --- | --- |
| 081 | HDFS: Number of write operations=4 | |
| 082 | Map-Reduce Framework |

|  |  |
| --- | --- |
| 083 | Map input records=4 |
| 084 | Map output records=4 | |

|  |  |
| --- | --- |
| 085 | Map output bytes=339 |
| 086 | Map output materialized bytes=353 | |

|  |  |
| --- | --- |
| 087 | Input split bytes=110 |
| 088 | Combine input records=0 | |

|  |  |  |
| --- | --- | --- |
| 089 | Combine output records=0 | |
| 090 | Reduce input groups=4 |

|  |  |  |
| --- | --- | --- |
| 091 | Reduce shuffle bytes=353 | |
| 092 | Reduce input records=4 |

|  |  |  |
| --- | --- | --- |
| 093 | Reduce output records=4 | |
| 094 | Spilled Records=8 |

|  |  |
| --- | --- |
| 095 | Shuffled Maps =1 |
| 096 | Failed Shuffles=0 | |

|  |  |
| --- | --- |
| 097 | Merged Map outputs=1 |
| 098 | GC time elapsed (ms)=0 | |

|  |  |  |
| --- | --- | --- |
| 099 | Total committed heap usage (bytes)=534773760 | |
| 100 | Shuffle Errors |

|  |  |
| --- | --- |
| 101 | BAD\_ID=0 |
| 102 | CONNECTION=0 | |

|  |  |
| --- | --- |
| 103 | IO\_ERROR=0 |
| 104 | WRONG\_LENGTH=0 | |

|  |  |
| --- | --- |
| 105 | WRONG\_MAP=0 |
| 106 | WRONG\_REDUCE=0 | |

|  |  |  |
| --- | --- | --- |
| 107 | File Input Format Counters | |
| 108 | Bytes Read=282 |

|  |  |  |
| --- | --- | --- |
| 109 | File Output Format Counters | |
| 110 | Bytes Written=527 |

**Step 9. Verify output**

[view source](http://www.javamakeuse.com/2016/03/mapreduce-map-side-join-example-hadoop.html#viewSource)

[print](http://www.javamakeuse.com/2016/03/mapreduce-map-side-join-example-hadoop.html#printSource)[?](http://www.javamakeuse.com/2016/03/mapreduce-map-side-join-example-hadoop.html#about)

|  |  |
| --- | --- |
| 1 | subodh@subodh-Inspiron-3520:~/Downloads$ hadoop fs -cat /output/par\* |
| 2 | 16/03/24 17:49:18 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable | |

|  |  |
| --- | --- |
| 3 | UserActivityVO [userId=1, userName=Susan, comments=looking awesome:), postShared=<http://dummyimage.com/160x166.gif/5fa2dd/ffffff>] |
| 4 | UserActivityVO [userId=2, userName=Kathleen, comments=full masti, postShared=<http://dummyimage.com/250x142.png/ff4444/ffffff>] |

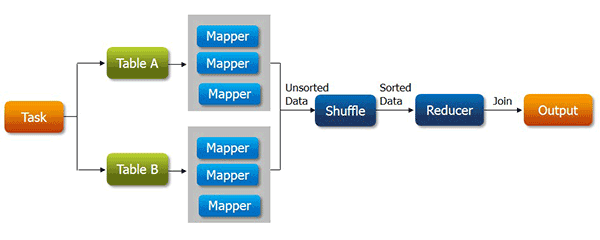
|  |  |
| --- | --- |
| 5 | UserActivityVO [userId=3, userName=Marilyn, comments=wow gangnam style,cool, postShared=<http://dummyimage.com/124x173.png/cc0000/ffffff>] |
| 6 | UserActivityVO [userId=4, userName=Craig, comments=welcome to the heaven, postShared=<http://dummyimage.com/148x156.png/ff4444/ffffff>] |

In this blog we shall discuss about **Map side join** and its advantages over the normal join operation in **Hive**. But before knowing about this, we should first understand the concept of**‘Join’** and what happens internally when we perform the join in **Hive**.

**Join** **is a clause that combines the records of two tables (or Data-Sets).**Assume that we have two tables A and B. When we perform join operation on them, it will return the records which are the combination of all columns o f A and B.

***Now let us understand the functionality of normal join with an example..***

Whenever, we apply join operation, the job will be assigned to a Map Reduce task which consists of two stages- a ***‘Map stage***’ and a ‘***Reduce stage***’. A mapper’s job during Map Stage is to “read” the data from join tables and to “return” the **‘join key’** and **‘join value’** pair into an intermediate file. Further, in the shuffle stage, this intermediate file is then sorted and merged. The reducer’s job during reduce stage is to take this sorted result as input and complete the task of join.

[](http://cdn.edureka.co/blog/wp-content/uploads/2013/11/joins1.png)

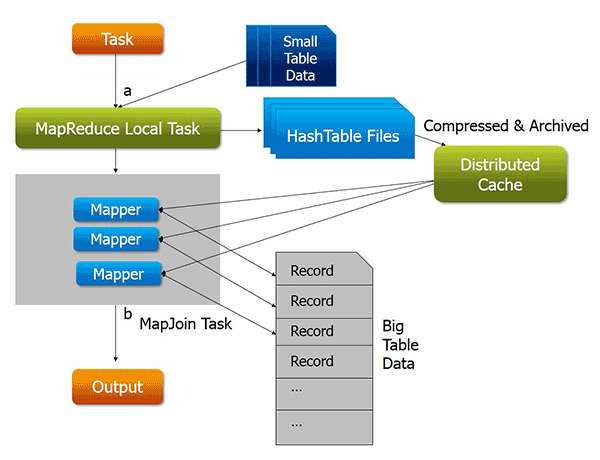
* Map-side Join is similar to a join but  all the task will be performed by the mapper alone.
* The Map-side Join will be mostly suitable for small tables to optimize the task.

[Master-Map-Side-Join](http://www.edureka.co/big-data-and-hadoop?utm_source=blog&utm_medium=button&utm_campaign=map-side-join-vs-join-master-map-side-join)

#### **How will the map-side join optimize the task?**

Assume that we have two tables of which one of them is a small table. When we submit a map reduce task, a Map Reduce local task will be created before the original join Map Reduce task which will read data of the small table from HDFS and store it into an in-memory hash table. After reading, it serializes the in-memory hash table into a hash table file.

***In the next stage,*** when the original join Map Reduce task is running, it moves the data in the hash table file to the Hadoop distributed cache, which populates these files to each mapper’s local disk. So all the mappers can load this persistent hash table file back into the memory and do the join work as before. The execution flow of the optimized map join is shown in the figure below. After optimization, the small table needs to be read just once. Also if multiple mappers are running on the same machine, the distributed cache only needs to push one copy of the hash table file to this machine.

[](http://cdn.edureka.co/blog/wp-content/uploads/2013/11/joins2.png)

## **Advantages of using map side join:**

* Map-side join helps in minimizing the cost that is incurred for sorting and merging in the shuffle and reduce stages.
* Map-side join also helps in improving the performance of the task by decreasing the time to finish the task.

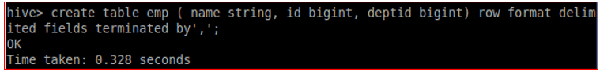
#### **Disadvantages of Map-side join:**

* Map side join is adequate only when one of the tables on which you perform map-side join operation is small enough to fit into the memory.  Hence it is not suitable to perform map-side join on the tables which are huge data in both of them.

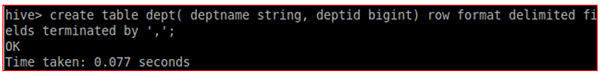
**Simple Example for Map Reduce Joins:**

Let us create two tables:

* **Emp**: contains details of an Employee such as Employee name, Employee ID and the Department she belongs to.

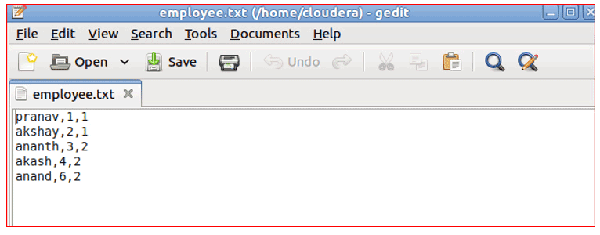
[](http://cdn.edureka.co/blog/wp-content/uploads/2013/11/Untitled-11.png)

* **Dept:**contains the details like the Name of the Department, Department ID and so on.

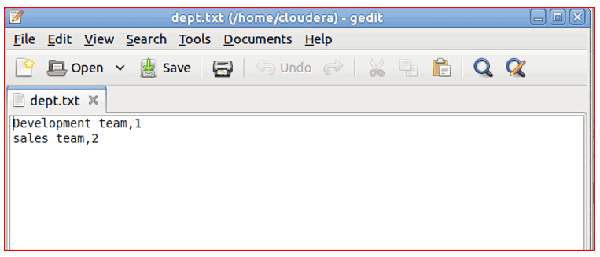
[](http://cdn.edureka.co/blog/wp-content/uploads/2013/11/Creating-table-using-Mapreduce-joins.png)

**Create two input files as shown in the following image to load the data into the tables created.**

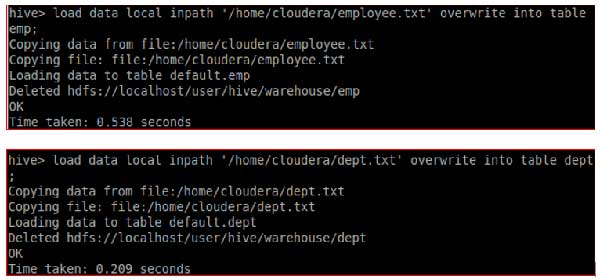
**employee.txt**

[](http://cdn.edureka.co/blog/wp-content/uploads/2013/11/Untitled-21.png)

**dept.txt**

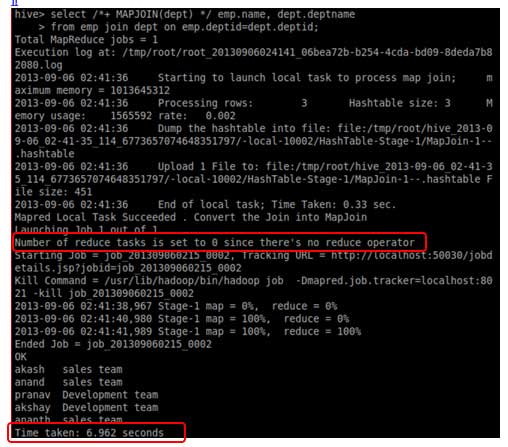
[](http://cdn.edureka.co/blog/wp-content/uploads/2013/11/Untitled-31.png)

**Now, let us load the data into the tables.**

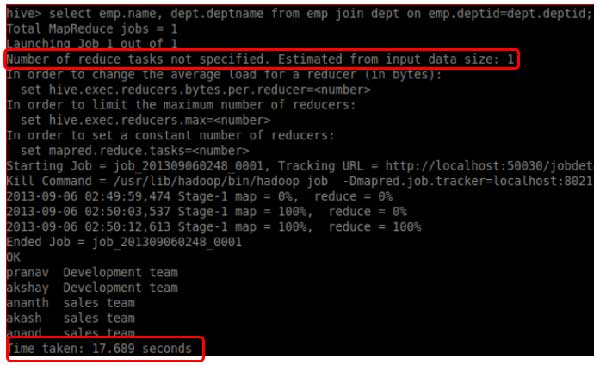
[](http://cdn.edureka.co/blog/wp-content/uploads/2013/11/Untitled-51.jpg)

Let us perform the **Map-side** **Join** on the two tables to extract the list of departments in which each employee is working.

Here, the **second table dept** is a small table. Remember, always the number of department will be less than the number of employees in an organization.

[](http://cdn.edureka.co/blog/wp-content/uploads/2013/11/Untitled-101.jpg)

Now let’s perform the same task with the help of normal Reduce-side join.

[](http://cdn.edureka.co/blog/wp-content/uploads/2013/11/Untitled-81.jpg)

**While executing both the joins, you can find the two differences:**

* Map-reduce join has completed the job in less time when compared with the time taken in normal join.
* Map-reduce join has completed its job without the help of any reducer whereas normal join executed this job with the help of one reducer.

Hence, **Map-side Join** is your best bet when one of the tables is small enough to fit in memory to complete the job in a short span of time.

In **Real-time environment**, you will be have data-sets with huge amount of data. So performing analysis and retrieving the data will be time consuming if one of the data-sets is of a smaller size. In such cases Map-side join will help to complete the job in less time.

There has never been a better time to master Hadoop! Get started now with the specially curated Big Data and Hadoop

## **MapReduce Example: Reduce Side Join in Hadoop MapReduce**

## **Introduction:**

In this blog, I am going to explain you how a reduce side join is performed in Hadoop MapReduce using a MapReduce example. Here, I am assuming that you are already familiar with MapReduce framework and know how to write a basic MapReduce program. In case you don’t, I would suggest you to go through my previous blog on [***MapReduce Tutorial***](http://www.edureka.co/blog/mapreduce-tutorial/?utm_source=blog&utm_medium=content-link&utm_campaign=mapreduce-example-reduce-side-join) so that you can grasp the concepts discussed here without facing any difficulties. The topics discussed in this blog are as follows:

* What is a Join?
* Joins in MapReduce
* What is a Reduce side join?
* MapReduce Example on Reduce side join
* Conclusion

## **What is a Join?**

The join operation is used to combine two or more database tables based on foreign keys. In general, companies maintain separate tables for the customer and the transaction records in their database. And, many times these companies need to generate analytic reports using the data present in such separate tables. Therefore, they perform a join operation on these separate tables using a common column (foreign key), like customer id, etc., to generate a combined table. Then, they analyze this combined table to get the desired analytic reports.

## **Joins in MapReduce**

Just like SQL join, we can also perform join operations in MapReduce on different data sets. There are two types of join operations in MapReduce:

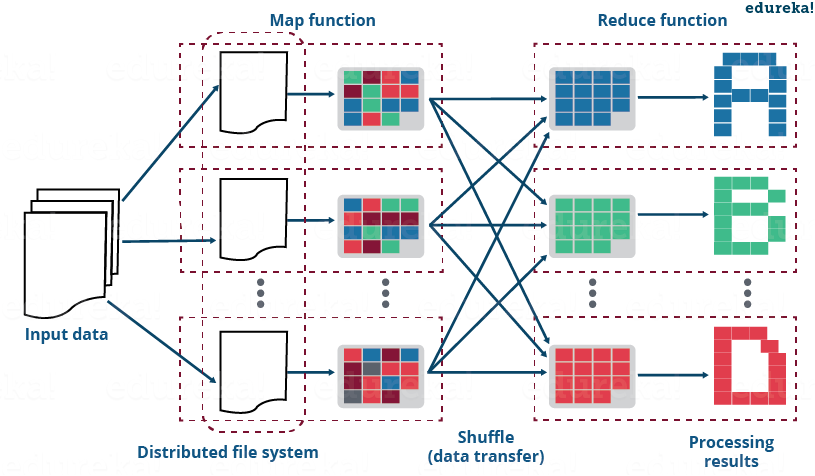
* **Map Side Join:** As the name implies, the join operation is performed in the map phase itself. Therefore, in the map side join, the mapper performs the join and it is mandatory that the input to each map is partitioned and sorted according to the keys.

The map side join has been covered in a separate blog with an example. [**Click Here**](http://www.edureka.co/blog/map-side-join-vs-join/?utm_source=blog&utm_medium=content-link&utm_campaign=mapreduce-example-reduce-side-join) to go through that blog to understand how the map side join works and what are its advantages.

* **Reduce Side Join:**As the name suggests, in the reduce side join, the reducer is responsible for performing the join operation. It is comparatively simple and easier to implement than the map side join as the sorting and shuffling phase sends the values having identical keys to the same reducer and therefore, by default, the data is organized for us.

Now, let us understand the reduce side join in detail.

## **What is Reduce Side Join?**



As discussed earlier, the reduce side join is a process where the join operation is performed in the reducer phase. Basically, the reduce side join takes place in the following manner:

* Mapper reads the input data which are to be combined based on common column or join key.
* The mapper processes the input and adds a tag to the input to distinguish the input belonging from different sources or data sets or databases.
* The mapper outputs the intermediate key-value pair where the key is nothing but the join key.
* After the sorting and shuffling phase, a key and the list of values is generated for the reducer.
* Now, the reducer joins the values present in the list with the key to give the final aggregated output.

*Meanwhile, you may go through this MapReduce Tutorial video where various MapReduce Use Cases has been clearly explained and practically demonstrated:*

## **MapReduce Example | MapReduce Programming | Hadoop MapReduce Tutorial | Edureka**

[**Learn MapReduce From Industry Experts**](http://www.edureka.co/big-data-and-hadoop?utm_source=blog&utm_medium=blog-cta&utm_campaign=mapreduce-example-reduce-side-join)

Now, let us take a MapReduce example to understand the above steps in the reduce side join.

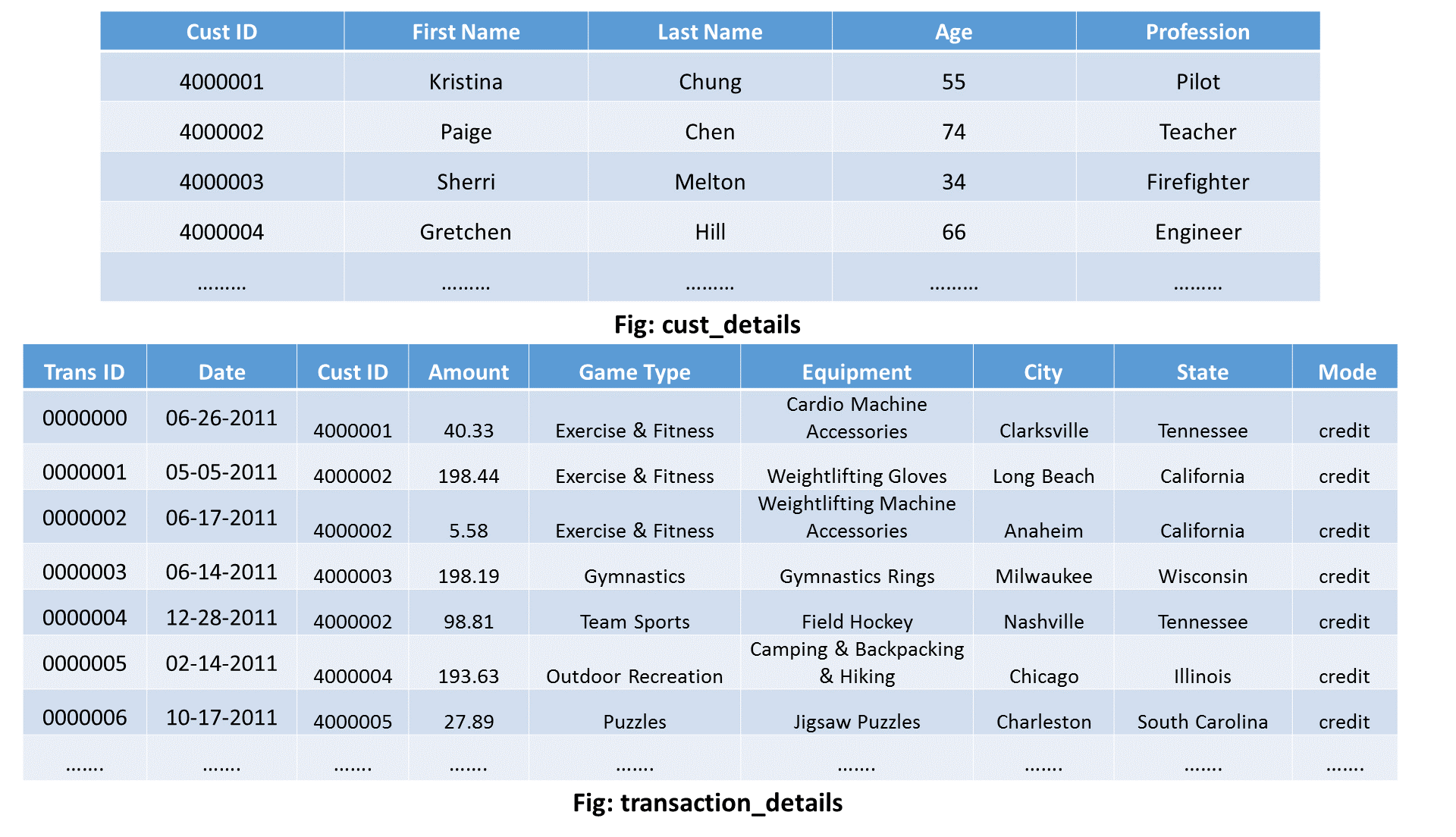
## **https://ssl.gstatic.com/ui/v1/icons/mail/images/cleardot.gifMapReduce Example of Reduce Side Join**

Suppose that I have two separate datasets of a sports complex:

* **cust\_details:** It contains the details of the customer.
* **transaction\_details:** It contains the transaction record of the customer.

Using these two datasets, I want to know the lifetime value of each customer. In doing so, I will be needing the following things:

* The person’s name along with the frequency of the visits by that person.
* The total amount spent by him/her for purchasing the equipment.



The above figure is just to show you the schema of the two datasets on which we will perform the reduce side join operation. Click on the button below to download the whole project containing the source code and the input files for this MapReduce example:

[**Download**](https://goo.gl/IeZZmo?utm_source=blog&utm_medium=button&utm_campaign=mapreduce-example-reduce-side-join-download)

Kindly, keep the following things in mind while importing the above MapReduce example project on reduce side join into Eclipse:

* The input files are in input\_files directory of the project. Load these into your HDFS.
* Don’t forget to build the path of Hadoop Reference Jars (present in reduce side join project lib directory)  according to your system or VM.

Now, let us understand what happens inside the map and reduce phases in this MapReduce example on reduce side join:

### ****1. Map Phase:****

I will have a separate mapper for each of the two datasets i.e. One mapper for cust\_details input and other for transaction\_details input.

**Mapper for cust\_details:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public static class CustsMapper extends Mapper <Object, Text, Text, Text>  {  public void map(Object key, Text value, Context context) throws IOException, InterruptedException  {String record = value.toString();  String[] parts = record.split(",");  context.write(new Text(parts[0]), new Text("cust\t" + parts[1]));  }  } |

* I will read the input taking one tuple at a time.
* Then, I will tokenize each word in that tuple and fetch the cust ID along with the name of theperson.
* The cust ID will be my key of the key-value pair that my mapper will generate eventually.
* I will also add a tag “cust” to indicate that this input tuple is of cust\_details type.
* Therefore, my mapper for cust\_details will produce following intermediate key-value pair:

**Key – Value pair: [cust ID, cust        name]**

Example: [4000001, cust    Kristina], [4000002, cust   Paige], etc.

**Mapper for transaction\_details:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public static class TxnsMapper extends Mapper <Object, Text, Text, Text>  {  public void map(Object key, Text value, Context context) throws IOException, InterruptedException  {  String record = value.toString();  String[] parts = record.split(",");  context.write(new Text(parts[2]), new Text("tnxn\t" + parts[3]));  }  } |

* Like mapper for cust\_details, I will follow the similar steps here. Though, there will be a few differences:
  + I will fetch the amount value instead of name of the person.
  + In this case, we will use “tnxn” as a tag.
* Therefore, the cust ID will be my key of the key-value pair that the mapper will generate eventually.
* Finally, the output of my mapper for transaction\_details will be of the following format:

**Key, Value Pair: [cust ID, tnxn   amount]**

Example: [4000001, tnxn   40.33], [4000002, tnxn   198.44], etc.

### ****2. Sorting and Shuffling Phase****

The sorting and shuffling phase will generate an array list of values corresponding to each key. In other words, it will put together all the values corresponding to each unique key in the intermediate key-value pair. The output of sorting and shuffling phase will be of the following format:

**Key – list of Values:**

* {cust ID1 – [(cust    name1), (tnxn    amount1), (tnxn    amount2), (tnxn    amount3),…..]}
* {cust ID2 – [(cust    name2), (tnxn    amount1), (tnxn    amount2), (tnxn    amount3),…..]}
* ……

**Example:**

* {4000001 – [(cust    kristina), (tnxn    40.33), (tnxn    47.05),…]};
* {4000002 – [(cust    paige), (tnxn    198.44), (tnxn     5.58),…]};
* ……

Now, the framework will call reduce() method (reduce(Text key, Iterable<Text> values, Context context)) for each unique join key (cust id) and the corresponding list of values. Then, the reducer will perform the join operation on the values present in the respective list of values to calculate the desired output eventually. Therefore, the number of reducer task performed will be equal to the number of unique cust ID.

Let us now understand how the reducer performs the join operation in this MapReduce example.

### ****3. Reducer Phase****

If you remember, the primary goal to perform this reduce-side join operation was to find out that how many times a particular customer has visited sports complex and the total amount spent by that very customer on different sports. Therefore, my final output should be of the following format:

**Key – Value pair: [Name of the customer] (Key) – [total amount, frequency of the visit] (Value)**

### Reducer Code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | public static class ReduceJoinReducer extends Reducer <Text, Text, Text, Text>  {  public void reduce(Text key, Iterable<Text> values, Context context)  throws IOException, InterruptedException  {  String name = "";  double total = 0.0;  int count = 0;  for (Text t : values)  {  String parts[] = t.toString().split("\t");  if (parts[0].equals("tnxn"))  {  count++;  total += Float.parseFloat(parts[1]);  }  else if (parts[0].equals("cust"))  {  name = parts[1];  }  }  String str = String.format("%d\t%f", count, total);  context.write(new Text(name), new Text(str));  }  } |

So, following steps will be taken in each of the reducers to achieve the desired output:

* In each of the reducer I will have a key & list of values where the key is nothing but the cust ID. The list of values will have the input from both the datasets i.e. Amount from transaction\_details and name from cust\_details.
* Now, I will loop through the values present in the list of values in the reducer.
* Then, I will split the list of values and check whether the value is of transaction\_details type or cust\_details type.
* If it is of the transaction\_details type, I will perform the following steps:
  + I will increase the counter value by one to calculate the frequency of visit by the very person.
  + I will cumulatively update the amount value to calculate the total amount spent by that person.
* On the other hand, if the value is of cust\_details type, I will store it in a string variable. Later, I will assign the name as my key  in my output key-value pair.
* Finally, I will write the output key-value pair in the output folder in my HDFS.

Hence, the final output that my reducer will generate is given below:

**Kristina, 651.05 8**

**Paige, 706.97  6**

**…..**

And, this whole process that we did above is called **Reduce Side Join** in MapReduce.

### ****Source Code:****

The source code for the above MapReduce example of the reduce side join is given below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77 | import java.io.IOException;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.Mapper;  import org.apache.hadoop.mapreduce.Reducer;  import org.apache.hadoop.mapreduce.lib.input.MultipleInputs;  import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;     public class ReduceJoin {   public static class CustsMapper extends Mapper <Object, Text, Text, Text>   {   public void map(Object key, Text value, Context context)   throws IOException, InterruptedException   {   String record = value.toString();   String[] parts = record.split(",");   context.write(new Text(parts[0]), new Text("cust\t" + parts[1]));   }   }     public static class TxnsMapper extends Mapper <Object, Text, Text, Text>   {   public void map(Object key, Text value, Context context)   throws IOException, InterruptedException   {   String record = value.toString();   String[] parts = record.split(",");   context.write(new Text(parts[2]), new Text("tnxn\t" + parts[3]));   }   }     public static class ReduceJoinReducer extends Reducer <Text, Text, Text, Text>   {   public void reduce(Text key, Iterable<Text> values, Context context)   throws IOException, InterruptedException   {   String name = "";   double total = 0.0;   int count = 0;   for (Text t : values)   {   String parts[] = t.toString().split("\t");   if (parts[0].equals("tnxn"))   {   count++;   total += Float.parseFloat(parts[1]);   }   else if (parts[0].equals("cust"))   {   name = parts[1];   }   }   String str = String.format("%d\t%f", count, total);   context.write(new Text(name), new Text(str));   }   }     public static void main(String[] args) throws Exception {   Configuration conf = new Configuration();   Job job = new Job(conf, "Reduce-side join");   job.setJarByClass(ReduceJoin.class);   job.setReducerClass(ReduceJoinReducer.class);   job.setOutputKeyClass(Text.class);   job.setOutputValueClass(Text.class);     MultipleInputs.addInputPath(job, new Path(args[0]),TextInputFormat.class, CustsMapper.class);   MultipleInputs.addInputPath(job, new Path(args[1]),TextInputFormat.class, TxnsMapper.class);   Path outputPath = new Path(args[2]);     FileOutputFormat.setOutputPath(job, outputPath);   outputPath.getFileSystem(conf).delete(outputPath);   System.exit(job.waitForCompletion(true) ? 0 : 1);   }   } |

**Run this Program**

Finally, the command to run the above MapReduce example program on reduce side join is given below:

hadoop jar reducejoin.jar ReduceJoin /sample/input/cust\_details /sample/input/transaction\_details /sample/output

## **Conclusion:**

The reduce side join procedure generates a huge network I/O traffic in the sorting and reducer phase where the values of the same key are brought together. So, if you have a large number of different data sets having millions of values, there is a high chance that you will encounter an OutOfMemory Exception i.e. Your RAM is full and therefore, overflown. In my opinion, the advantages of using reduce side join are:

* It is very easy to implement as we are taking advantage of the inbuilt sorting and shuffling algorithm in the MapReduce framework which combines values of the same key and send it to the same reducer.
* In the reduce side join, your input does not require to follow any strict format and therefore, you can perform the join operation on unstructured data as well.

In general, people prefer Apache Hive, which is a part of the Hadoop ecosystem, to perform the join operation. So, if you are from the SQL background, you don’t need to worry about writing the MapReduce Java code for performing a join operation. You can use Hive as an alternative.

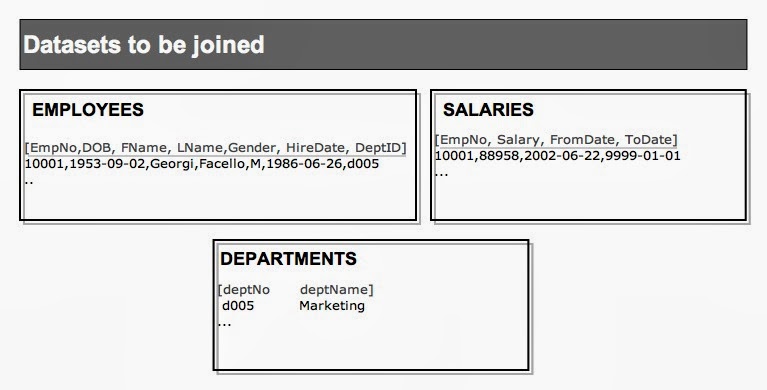
### Reduce-side joins in Java map-reduce

### 1.0. About reduce side joins

Joins of datasets done in the reduce phase are called reduce side joins.  Reduce side joins are easier to implement as they are less stringent than map-side joins that require the data to be sorted and partitioned the same way.  They are less efficient than maps-side joins because  the datasets have to go through the sort and shuffle phase.   
  
What's involved..  
1.  The key of the map output, of datasets being joined, has to be the join key - so they reach the same reducer  
2.  Each dataset has to be tagged with its identity, in the mapper- to help differentiate between the datasets in the reducer, so they can be processed accordingly.  
3.  In each reducer, the data values from both datasets, for keys assigned to the reducer, are available, to be processed as required.  
4.  A secondary sort needs to be done to ensure the ordering of the values sent to the reducer  
5.  If the input files are of different formats, we would need separate mappers, and we would need to use MultipleInputs class in the driver to add the inputs and associate the specific mapper to the same.  
[MultipleInputs.addInputPath( job, (input path n), (inputformat class), (mapper class n));]  
  
Note:  The join between the datasets (employee, current salary - cardinality of 1..1) in the sample program below has been demonstrated in my blog on map side joins of large datasets, as well.  I have used the same datasets here...as the purpose of this blog is to demonstrate the concept.  Whenever possible, reduce-side joins should be avoided.  
  
[Update - 10/15/2013]  
I have added a pig equivalent in the final section.

### 2.0. Sample datasets used in this gist

The datasets used are employees and salaries.  For salary data, there are two files - one file with  current salary (1..1), and one with historical salary data (1..many). Then there is the department data, a small reference dataset, that we will add to distributed cache and look up in the reducer.

[](http://4.bp.blogspot.com/-4kxXL-XStQU/UkHnbVnGw0I/AAAAAAAAAqU/MADbxjebpa0/s1600/reducesidejoinsampledata1.jpg)

### 3.0. Implementation a reduce-side join

The sample code is common for a 1..1 as well as 1..many join for the sample datasets.  
The mapper is common for both datasets, as the format is the same.

### 3.0.1. Components/steps/tasks:

**1.  Map output key**

The key will be the empNo as it is the join key for the datasets employee and salary

[Implementation: in the mapper]

**2.  Tagging the data with the dataset identity**

Add an attribute called srcIndex to tag the identity of the data (1=employee, 2=salary, 3=salary history)

[Implementation: in the mapper]

**3.  Discarding unwanted atributes**

[Implementation: in the mapper]

**4. Composite key**

Make the map output key a composite of empNo and srcIndex

[Implementation: create custom writable]

**5.  Partitioner**

Partition the data on natural key of empNo

[Implementation: create custom partitioner class]

**5.  Sorting**

Sort the data on empNo first, and then source index

[Implementation: create custom sorting comparator class]

**6.  Grouping**

Group the data based on natural key

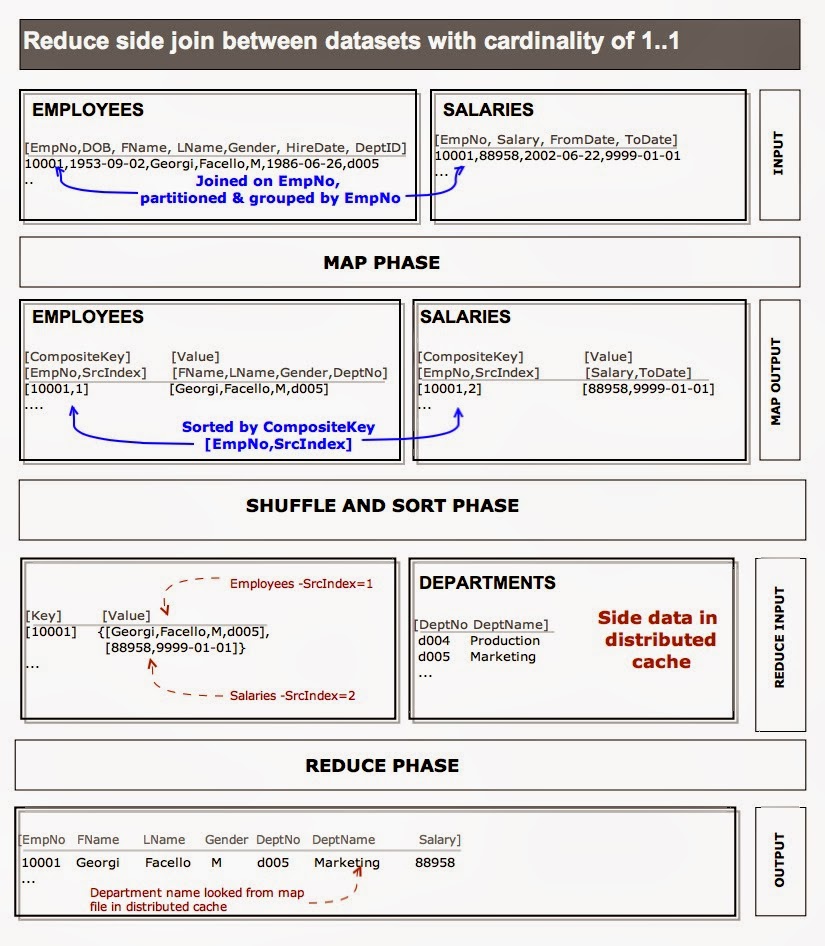
[Implementation: create custom grouping comparator class]

**7. Joining**

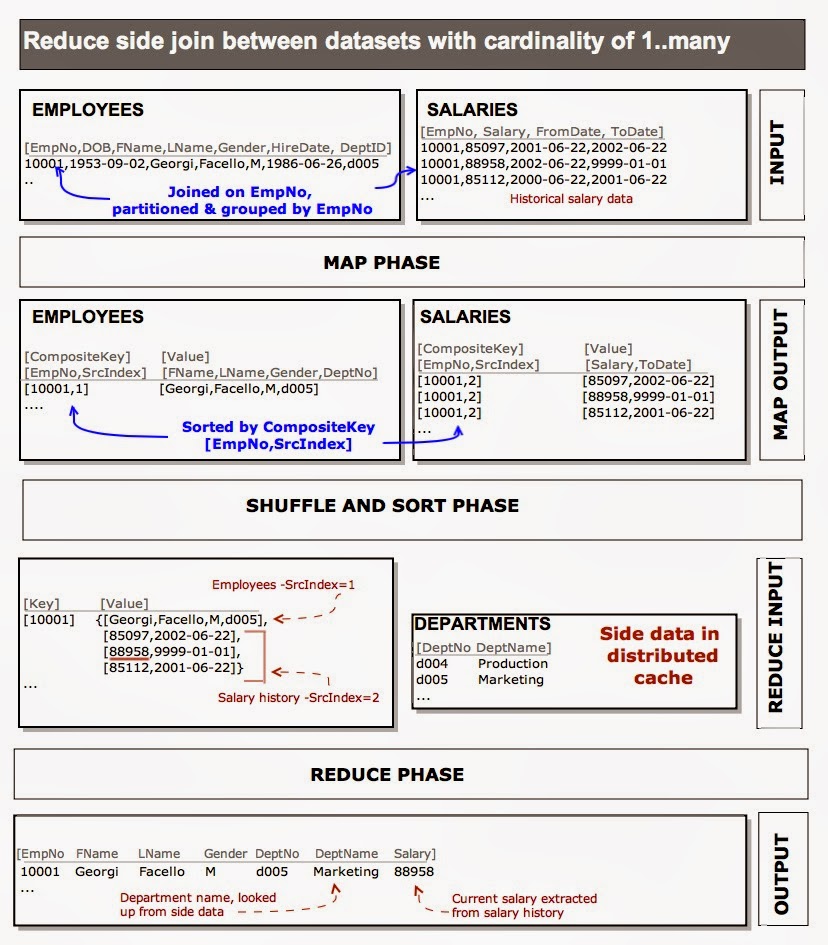
Iterate through the values for a key and complete the join for employee and salary data, perform lookup of department to include department name in the output

[Implementation: in the reducer]

### 3.0.2a. Data pipeline for cardinality of 1..1 between employee and salary data:

[](http://2.bp.blogspot.com/-XaqIE6tB2Xg/UkHsRCGoTfI/AAAAAAAAAqk/28TyHSKF8nQ/s1600/reducesidejoin.jpg)

### 3.0.2b. Data pipeline for cardinality of 1..many between employee and salary data:

[](http://1.bp.blogspot.com/-LEeoFzSCJUA/UkH-xm9c9WI/AAAAAAAAAq0/CpCfDyTutpE/s1600/reducesidejoin1_many.jpg)

### 3.0.3. The Composite key

The composite key is a combination of the joinKey empNo, and the source Index (1=employee file.., 2=salary file...)

|  |  |
| --- | --- |
|  | //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
|  | //Class: CompositeKeyWritableRSJ |
|  | //Purpose: Custom Writable that serves as composite key |
|  | // with attributes joinKey and sourceIndex |
|  | //Author: Anagha Khanolkar |
|  | //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
|  |  |
|  | package khanolkar.mapreduce.join.samples.reducesidejoin; |
|  |  |
|  | import java.io.DataInput; |
|  | import java.io.DataOutput; |
|  | import java.io.IOException; |
|  |  |
|  | import org.apache.hadoop.io.Writable; |
|  | import org.apache.hadoop.io.WritableComparable; |
|  | import org.apache.hadoop.io.WritableUtils; |
|  |  |
|  | public class CompositeKeyWritableRSJ implements Writable, |
|  | WritableComparable<CompositeKeyWritableRSJ> { |
|  |  |
|  | // Data members |
|  | private String joinKey;// EmployeeID |
|  | private int sourceIndex;// 1=Employee data; 2=Salary (current) data; 3=Salary historical data |
|  |  |
|  | public CompositeKeyWritableRSJ() { |
|  | } |
|  |  |
|  | public CompositeKeyWritableRSJ(String joinKey, int sourceIndex) { |
|  | this.joinKey = joinKey; |
|  | this.sourceIndex = sourceIndex; |
|  | } |
|  |  |
|  | @Override |
|  | public String toString() { |
|  |  |
|  | return (new StringBuilder().append(joinKey).append("\t") |
|  | .append(sourceIndex)).toString(); |
|  | } |
|  |  |
|  | public void readFields(DataInput dataInput) throws IOException { |
|  | joinKey = WritableUtils.readString(dataInput); |
|  | sourceIndex = WritableUtils.readVInt(dataInput); |
|  | } |
|  |  |
|  | public void write(DataOutput dataOutput) throws IOException { |
|  | WritableUtils.writeString(dataOutput, joinKey); |
|  | WritableUtils.writeVInt(dataOutput, sourceIndex); |
|  | } |
|  |  |
|  | public int compareTo(CompositeKeyWritableRSJ objKeyPair) { |
|  |  |
|  | int result = joinKey.compareTo(objKeyPair.joinKey); |
|  | if (0 == result) { |
|  | result = Double.compare(sourceIndex, objKeyPair.sourceIndex); |
|  | } |
|  | return result; |
|  | } |
|  |  |
|  | public String getjoinKey() { |
|  | return joinKey; |
|  | } |
|  |  |
|  | public void setjoinKey(String joinKey) { |
|  | this.joinKey = joinKey; |
|  | } |
|  |  |
|  | public int getsourceIndex() { |
|  | return sourceIndex; |
|  | } |
|  |  |
|  | public void setsourceIndex(int sourceIndex) { |
|  | this.sourceIndex = sourceIndex; |
|  | } |
|  | } |

[**view raw**](https://gist.github.com/airawat/6666608/raw/e0e3d7d47ec32195632d66957a3c9e26316d0091/05CompositeKey)[**05CompositeKey**](https://gist.github.com/airawat/6666608#file-05compositekey) hosted with ❤ by [**GitHub**](https://github.com/)

### 3.0.4. The mapper

In the setup method of the mapper-  
1. Get the filename from the input split, cross reference it against the configuration (set in driver), to derive the source index.  [Driver code: Add configuration [key=filename of employee,value=1], [key=filename of current salary dataset,value=2], [key=filename of historical salary dataset,value=3]  
2. Build a list of attributes we cant to emit as map output for each data entity  
  
The setup method is called only once, at the beginning of a map task.  So it is the logical place to to identify the source index.  
  
In the map method of the mapper:  
3. Build the map output based on attributes required, as specified in the list from #2  
  
Note:  For salary data, we are including the "effective till" date, even though it is not required in the final output because this is common code for a 1..1 as well as 1..many join to salary data.  If the salary data is historical, we want the current salary only, that is "effective till date= 9999-01-01".

|  |  |
| --- | --- |
|  | //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
|  | //Class: MapperRSJ |
|  | //Purpose: Mapper |
|  | //Author: Anagha Khanolkar |
|  | //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
|  |  |
|  | package khanolkar.mapreduce.join.samples.reducesidejoin; |
|  |  |
|  | import java.io.IOException; |
|  | import java.util.ArrayList; |
|  | import java.util.List; |
|  |  |
|  | 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 MapperRSJ extends |
|  | Mapper<LongWritable, Text, CompositeKeyWritableRSJ, Text> { |
|  |  |
|  | CompositeKeyWritableRSJ ckwKey = new CompositeKeyWritableRSJ(); |
|  | Text txtValue = new Text(""); |
|  | int intSrcIndex = 0; |
|  | StringBuilder strMapValueBuilder = new StringBuilder(""); |
|  | List<Integer> lstRequiredAttribList = new ArrayList<Integer>(); |
|  |  |
|  | @Override |
|  | protected void setup(Context context) throws IOException, |
|  | InterruptedException { |
|  |  |
|  | // {{ |
|  | // Get the source index; (employee = 1, salary = 2) |
|  | // Added as configuration in driver |
|  | FileSplit fsFileSplit = (FileSplit) context.getInputSplit(); |
|  | intSrcIndex = Integer.parseInt(context.getConfiguration().get( |
|  | fsFileSplit.getPath().getName())); |
|  | // }} |
|  |  |
|  | // {{ |
|  | // Initialize the list of fields to emit as output based on |
|  | // intSrcIndex (1=employee, 2=current salary, 3=historical salary) |
|  | if (intSrcIndex == 1) // employee |
|  | { |
|  | lstRequiredAttribList.add(2); // FName |
|  | lstRequiredAttribList.add(3); // LName |
|  | lstRequiredAttribList.add(4); // Gender |
|  | lstRequiredAttribList.add(6); // DeptNo |
|  | } else // salary |
|  | { |
|  | lstRequiredAttribList.add(1); // Salary |
|  | lstRequiredAttribList.add(3); // Effective-to-date (Value of |
|  | // 9999-01-01 indicates current |
|  | // salary) |
|  |  |
|  | } |
|  | // }} |
|  |  |
|  | } |
|  |  |
|  | private String buildMapValue(String arrEntityAttributesList[]) { |
|  | // This method returns csv list of values to emit based on data entity |
|  |  |
|  | strMapValueBuilder.setLength(0);// Initialize |
|  |  |
|  | // Build list of attributes to output based on source - employee/salary |
|  | for (int i = 1; i < arrEntityAttributesList.length; i++) { |
|  | // If the field is in the list of required output |
|  | // append to stringbuilder |
|  | if (lstRequiredAttribList.contains(i)) { |
|  | strMapValueBuilder.append(arrEntityAttributesList[i]).append( |
|  | ","); |
|  | } |
|  | } |
|  | if (strMapValueBuilder.length() > 0) { |
|  | // Drop last comma |
|  | strMapValueBuilder.setLength(strMapValueBuilder.length() - 1); |
|  | } |
|  |  |
|  | return strMapValueBuilder.toString(); |
|  | } |
|  |  |
|  | @Override |
|  | public void map(LongWritable key, Text value, Context context) |
|  | throws IOException, InterruptedException { |
|  |  |
|  | if (value.toString().length() > 0) { |
|  | String arrEntityAttributes[] = value.toString().split(","); |
|  |  |
|  | ckwKey.setjoinKey(arrEntityAttributes[0].toString()); |
|  | ckwKey.setsourceIndex(intSrcIndex); |
|  | txtValue.set(buildMapValue(arrEntityAttributes)); |
|  |  |
|  | context.write(ckwKey, txtValue); |
|  | } |
|  |  |
|  | } |
|  | } |

[**view raw**](https://gist.github.com/airawat/6666608/raw/e0e3d7d47ec32195632d66957a3c9e26316d0091/06-Mapper)[**06-Mapper**](https://gist.github.com/airawat/6666608#file-06-mapper) hosted with ❤ by [**GitHub**](https://github.com/)

### 3.0.5. The partitioner

Even though the map output key is composite, we want to partition by the natural join key of empNo, therefore a custom partitioner is in order.

|  |  |
| --- | --- |
|  | //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
|  | //Class: PartitionerRSJ |
|  | //Purpose: Custom partitioner |
|  | //Author: Anagha Khanolkar |
|  | //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
|  |  |
|  | package khanolkar.mapreduce.join.samples.reducesidejoin; |
|  |  |
|  | import org.apache.hadoop.io.Text; |
|  | import org.apache.hadoop.mapreduce.Partitioner; |
|  |  |
|  | public class PartitionerRSJ extends Partitioner<CompositeKeyWritableRSJ, Text> { |
|  |  |
|  | @Override |
|  | public int getPartition(CompositeKeyWritableRSJ key, Text value, |
|  | int numReduceTasks) { |
|  | // Partitions on joinKey (EmployeeID) |
|  | return (key.getjoinKey().hashCode() % numReduceTasks); |
|  | } |
|  | } |

[**view raw**](https://gist.github.com/airawat/6666608/raw/e0e3d7d47ec32195632d66957a3c9e26316d0091/07-Partitioner)[**07-Partitioner**](https://gist.github.com/airawat/6666608#file-07-partitioner) hosted with ❤ by [**GitHub**](https://github.com/)

### 3.0.6. The sort comparator

To ensure that the input to the reducer is sorted on empNo, then on sourceIndex, we need a sort comparator.  This will guarantee that the employee data is the first set in the values list for a key, then the salary data.

|  |  |
| --- | --- |
|  | package khanolkar.mapreduce.join.samples.reducesidejoin; |
|  |  |
|  | import org.apache.hadoop.io.WritableComparable; |
|  | import org.apache.hadoop.io.WritableComparator; |
|  |  |
|  | //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
|  | //Class: SortingComparatorRSJ |
|  | //Purpose: Sorting comparator |
|  | //Author: Anagha Khanolkar |
|  | //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
|  |  |
|  | public class SortingComparatorRSJ extends WritableComparator { |
|  |  |
|  | protected SortingComparatorRSJ() { |
|  | super(CompositeKeyWritableRSJ.class, true); |
|  | } |
|  |  |
|  | @Override |
|  | public int compare(WritableComparable w1, WritableComparable w2) { |
|  | // Sort on all attributes of composite key |
|  | CompositeKeyWritableRSJ key1 = (CompositeKeyWritableRSJ) w1; |
|  | CompositeKeyWritableRSJ key2 = (CompositeKeyWritableRSJ) w2; |
|  |  |
|  | int cmpResult = key1.getjoinKey().compareTo(key2.getjoinKey()); |
|  | if (cmpResult == 0)// same joinKey |
|  | { |
|  | return Double.compare(key1.getsourceIndex(), key2.getsourceIndex()); |
|  | } |
|  | return cmpResult; |
|  | } |
|  | } |

[**view raw**](https://gist.github.com/airawat/6666608/raw/e0e3d7d47ec32195632d66957a3c9e26316d0091/08-SortComparator)[**08-SortComparator**](https://gist.github.com/airawat/6666608#file-08-sortcomparator) hosted with ❤ by [**GitHub**](https://github.com/)

### 3.0.7. The grouping comparator

This class is needed to indicate the group by attribute - the natural join key of empNo

|  |  |
| --- | --- |
|  | package khanolkar.mapreduce.join.samples.reducesidejoin; |
|  |  |
|  | import org.apache.hadoop.io.WritableComparable; |
|  | import org.apache.hadoop.io.WritableComparator; |
|  |  |
|  | //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
|  | //Class: GroupingComparatorRSJ |
|  | //Purpose: For use as grouping comparator |
|  | //Author: Anagha Khanolkar |
|  | //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
|  |  |
|  | public class GroupingComparatorRSJ extends WritableComparator { |
|  | protected GroupingComparatorRSJ() { |
|  | super(CompositeKeyWritableRSJ.class, true); |
|  | } |
|  |  |
|  | @Override |
|  | public int compare(WritableComparable w1, WritableComparable w2) { |
|  | // The grouping comparator is the joinKey (Employee ID) |
|  | CompositeKeyWritableRSJ key1 = (CompositeKeyWritableRSJ) w1; |
|  | CompositeKeyWritableRSJ key2 = (CompositeKeyWritableRSJ) w2; |
|  | return key1.getjoinKey().compareTo(key2.getjoinKey()); |
|  | } |
|  | } |

[**view raw**](https://gist.github.com/airawat/6666608/raw/e0e3d7d47ec32195632d66957a3c9e26316d0091/09-GroupingComparator)[**09-GroupingComparator**](https://gist.github.com/airawat/6666608#file-09-groupingcomparator) hosted with ❤ by [**GitHub**](https://github.com/)

### 3.0.8. The reducer

In the setup method of the reducer (called only once for the task)-  
We are checking if the side data, a map file with department data is in the distributed cache and if found, initializing the map file reader  
  
In the reduce method, -  
While iterating through the value list -  
1. If the data is employee data (sourceIndex=1), we are looking up the department name in the map file with the deptNo, which is the last attribute in the employee data, and appending the department name to the employee data.  
2. If the data is historical salary data, we are only emitting salary where the last attribute is '9999-01-01'.  
  
Key point-  
We have set the sort comparator to sort on empNo and sourceIndex.  
The sourceIndex of employee data is lesser than salary data - as set in the driver.  
Therefore, we are assured that the employee data is always first followed by salary data.  
So for each distinct empNo, we are iterating through the values, and appending the same and emitting as output.

|  |  |
| --- | --- |
|  | package khanolkar.mapreduce.join.samples.reducesidejoin; |
|  |  |
|  | import java.io.File; |
|  | import java.io.IOException; |
|  | import java.net.URI; |
|  |  |
|  | import org.apache.hadoop.filecache.DistributedCache; |
|  | import org.apache.hadoop.fs.FileSystem; |
|  | import org.apache.hadoop.fs.Path; |
|  | import org.apache.hadoop.io.MapFile; |
|  | import org.apache.hadoop.io.NullWritable; |
|  | import org.apache.hadoop.io.Text; |
|  | import org.apache.hadoop.mapreduce.Reducer; |
|  |  |
|  | //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
|  | //Class: ReducerRSJ |
|  | //Purpose: Reducer |
|  | //Author: Anagha Khanolkar |
|  | //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
|  |  |
|  | public class ReducerRSJ extends |
|  | Reducer<CompositeKeyWritableRSJ, Text, NullWritable, Text> { |
|  |  |
|  | StringBuilder reduceValueBuilder = new StringBuilder(""); |
|  | NullWritable nullWritableKey = NullWritable.get(); |
|  | Text reduceOutputValue = new Text(""); |
|  | String strSeparator = ","; |
|  | private MapFile.Reader deptMapReader = null; |
|  | Text txtMapFileLookupKey = new Text(""); |
|  | Text txtMapFileLookupValue = new Text(""); |
|  |  |
|  | @Override |
|  | protected void setup(Context context) throws IOException, |
|  | InterruptedException { |
|  |  |
|  | // {{ |
|  | // Get side data from the distributed cache |
|  | Path[] cacheFilesLocal = DistributedCache.getLocalCacheArchives(context |
|  | .getConfiguration()); |
|  |  |
|  | for (Path eachPath : cacheFilesLocal) { |
|  |  |
|  | if (eachPath.getName().toString().trim() |
|  | .equals("departments\_map.tar.gz")) { |
|  | URI uriUncompressedFile = new File(eachPath.toString() |
|  | + "/departments\_map").toURI(); |
|  | initializeDepartmentsMap(uriUncompressedFile, context); |
|  | } |
|  | } |
|  | // }} |
|  | } |
|  |  |
|  | @SuppressWarnings("deprecation") |
|  | private void initializeDepartmentsMap(URI uriUncompressedFile, Context context) |
|  | throws IOException { |
|  | // {{ |
|  | // Initialize the reader of the map file (side data) |
|  | FileSystem dfs = FileSystem.get(context.getConfiguration()); |
|  | try { |
|  | deptMapReader = new MapFile.Reader(dfs, |
|  | uriUncompressedFile.toString(), context.getConfiguration()); |
|  | } catch (Exception e) { |
|  | e.printStackTrace(); |
|  | } |
|  | // }} |
|  | } |
|  |  |
|  | private StringBuilder buildOutputValue(CompositeKeyWritableRSJ key, |
|  | StringBuilder reduceValueBuilder, Text value) { |
|  |  |
|  | if (key.getsourceIndex() == 1) { |
|  | // Employee data |
|  | // {{ |
|  | // Get the department name from the MapFile in distributedCache |
|  |  |
|  | // Insert the joinKey (empNo) to beginning of the stringBuilder |
|  | reduceValueBuilder.append(key.getjoinKey()).append(strSeparator); |
|  |  |
|  | String arrEmpAttributes[] = value.toString().split(","); |
|  | txtMapFileLookupKey.set(arrEmpAttributes[3].toString()); |
|  | try { |
|  | deptMapReader.get(txtMapFileLookupKey, txtMapFileLookupValue); |
|  | } catch (Exception e) { |
|  | txtMapFileLookupValue.set(""); |
|  |  |
|  | } finally { |
|  | txtMapFileLookupValue |
|  | .set((txtMapFileLookupValue.equals(null) || txtMapFileLookupValue |
|  | .equals("")) ? "NOT-FOUND" |
|  | : txtMapFileLookupValue.toString()); |
|  | } |
|  | // }} |
|  |  |
|  | // {{ |
|  | // Append the department name to the map values to form a complete |
|  | // CSV of employee attributes |
|  | reduceValueBuilder.append(value.toString()).append(strSeparator) |
|  | .append(txtMapFileLookupValue.toString()) |
|  | .append(strSeparator); |
|  | // }} |
|  |  |
|  | } else if (key.getsourceIndex() == 2) { |
|  | // Current recent salary data (1..1 on join key) |
|  | // Salary data; Just append the salary, drop the effective-to-date |
|  | String arrSalAttributes[] = value.toString().split(","); |
|  | reduceValueBuilder.append(arrSalAttributes[0].toString()).append( |
|  | strSeparator); |
|  | } else // key.getsourceIndex() == 3; Historical salary data |
|  | { |
|  | // {{ |
|  | // Get the salary data but extract only current salary |
|  | // (to\_date='9999-01-01') |
|  | String arrSalAttributes[] = value.toString().split(","); |
|  | if (arrSalAttributes[1].toString().equals("9999-01-01")) { |
|  | // Salary data; Just append |
|  | reduceValueBuilder.append(arrSalAttributes[0].toString()) |
|  | .append(strSeparator); |
|  | } |
|  | // }} |
|  |  |
|  | } |
|  |  |
|  | // {{ |
|  | // Reset |
|  | txtMapFileLookupKey.set(""); |
|  | txtMapFileLookupValue.set(""); |
|  | // }} |
|  |  |
|  | return reduceValueBuilder; |
|  | } |
|  |  |
|  | @Override |
|  | public void reduce(CompositeKeyWritableRSJ key, Iterable<Text> values, |
|  | Context context) throws IOException, InterruptedException { |
|  |  |
|  | // Iterate through values; First set is csv of employee data |
|  | // second set is salary data; The data is already ordered |
|  | // by virtue of secondary sort; Append each value; |
|  | for (Text value : values) { |
|  | buildOutputValue(key, reduceValueBuilder, value); |
|  | } |
|  |  |
|  | // Drop last comma, set value, and emit output |
|  | if (reduceValueBuilder.length() > 1) { |
|  |  |
|  | reduceValueBuilder.setLength(reduceValueBuilder.length() - 1); |
|  | // Emit output |
|  | reduceOutputValue.set(reduceValueBuilder.toString()); |
|  | context.write(nullWritableKey, reduceOutputValue); |
|  | } else { |
|  | System.out.println("Key=" + key.getjoinKey() + "src=" |
|  | + key.getsourceIndex()); |
|  |  |
|  | } |
|  |  |
|  | // Reset variables |
|  | reduceValueBuilder.setLength(0); |
|  | reduceOutputValue.set(""); |
|  |  |
|  | } |
|  | @Override |
|  | protected void cleanup(Context context) throws IOException, |
|  | InterruptedException { |
|  | deptMapReader.close(); |
|  | } |
|  | } |

[**view raw**](https://gist.github.com/airawat/6666608/raw/e0e3d7d47ec32195632d66957a3c9e26316d0091/10-Reducer)[**10-Reducer**](https://gist.github.com/airawat/6666608#file-10-reducer) hosted with ❤ by [**GitHub**](https://github.com/)

### 3.0.9. The driver

Besides the usual driver code, we are-  
1. Adding side data (department lookup data in map file format - in HDFS) to the distributed cache  
2. Adding key-value pairs to the configuration, each key value pair being filename, source index.  
This is used by the mapper, to tag data with sourceIndex.  
3. And lastly, we are associating all the various classes we created to the job.

|  |  |
| --- | --- |
|  | package khanolkar.mapreduce.join.samples.reducesidejoin; |
|  |  |
|  | import java.net.URI; |
|  |  |
|  | import org.apache.hadoop.conf.Configuration; |
|  | import org.apache.hadoop.conf.Configured; |
|  | import org.apache.hadoop.filecache.DistributedCache; |
|  | 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 org.apache.hadoop.util.Tool; |
|  | import org.apache.hadoop.util.ToolRunner; |
|  |  |
|  | //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
|  | //Class: DriverRSJ |
|  | //Purpose: Driver for Reduce Side Join of two datasets |
|  | // with a 1..1 or 1..many cardinality on join key |
|  | //Author: Anagha Khanolkar |
|  | //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
|  |  |
|  | public class DriverRSJ extends Configured implements Tool { |
|  |  |
|  | @Override |
|  | public int run(String[] args) throws Exception { |
|  |  |
|  | // {{ |
|  | // Exit job if required arguments have not been provided |
|  | if (args.length != 3) { |
|  | System.out |
|  | .printf("Three parameters are required for DriverRSJ- <input dir1> <input dir2> <output dir>\n"); |
|  | return -1; |
|  | } |
|  | // }{ |
|  |  |
|  | // {{ |
|  | // Job instantiation |
|  | Job job = new Job(getConf()); |
|  | Configuration conf = job.getConfiguration(); |
|  | job.setJarByClass(DriverRSJ.class); |
|  | job.setJobName("ReduceSideJoin"); |
|  | // }} |
|  |  |
|  | // {{ |
|  | // Add side data to distributed cache |
|  | DistributedCache |
|  | .addCacheArchive( |
|  | new URI( |
|  | "/user/akhanolk/joinProject/data/departments\_map.tar.gz"), |
|  | conf); |
|  |  |
|  | // }} |
|  |  |
|  | // { |
|  | // Set sourceIndex for input files; |
|  | // sourceIndex is an attribute of the compositeKey, |
|  | // to drive order, and reference source |
|  | // Can be done dynamically; Hard-coded file names for simplicity |
|  | conf.setInt("part-e", 1);// Set Employee file to 1 |
|  | conf.setInt("part-sc", 2);// Set Current salary file to 2 |
|  | conf.setInt("part-sh", 3);// Set Historical salary file to 3 |
|  |  |
|  | // } |
|  |  |
|  | // { |
|  | // Build csv list of input files |
|  | StringBuilder inputPaths = new StringBuilder(); |
|  | inputPaths.append(args[0].toString()).append(",") |
|  | .append(args[1].toString()); |
|  | // } |
|  |  |
|  | // {{ |
|  | // Configure remaining aspects of the job |
|  | FileInputFormat.setInputPaths(job, inputPaths.toString()); |
|  | FileOutputFormat.setOutputPath(job, new Path(args[2])); |
|  |  |
|  | job.setMapperClass(MapperRSJ.class); |
|  | job.setMapOutputKeyClass(CompositeKeyWritableRSJ.class); |
|  | job.setMapOutputValueClass(Text.class); |
|  |  |
|  | job.setPartitionerClass(PartitionerRSJ.class); |
|  | job.setSortComparatorClass(SortingComparatorRSJ.class); |
|  | job.setGroupingComparatorClass(GroupingComparatorRSJ.class); |
|  |  |
|  | job.setNumReduceTasks(4); |
|  | job.setReducerClass(ReducerRSJ.class); |
|  | job.setOutputKeyClass(NullWritable.class); |
|  | job.setOutputValueClass(Text.class); |
|  | // }} |
|  |  |
|  | boolean success = job.waitForCompletion(true); |
|  | return success ? 0 : 1; |
|  | } |
|  |  |
|  | public static void main(String[] args) throws Exception { |
|  | int exitCode = ToolRunner.run(new Configuration(), new DriverRSJ(), |
|  | args); |
|  | System.exit(exitCode); |
|  | } |
|  | } |

[**view raw**](https://gist.github.com/airawat/6666608/raw/e0e3d7d47ec32195632d66957a3c9e26316d0091/11-Driver)[**11-Driver**](https://gist.github.com/airawat/6666608#file-11-driver) hosted with ❤ by [**GitHub**](https://github.com/)

28) Recordreader,inputsplit,recordwriter

Answer)

# Hadoop: RecordReader and FileInputFormat

[27 MAY 2013](https://hadoopi.wordpress.com/2013/05/27/understand-recordreader-inputsplit/) / [ANTOINE AMEND](https://hadoopi.wordpress.com/author/aamend/)

Today’s new challenge…  
I want to create a custom MapReduce job that can handle more than 1 single line at a time. Actually, it took me some time to understand the implementation of default **LineRecordReader** class, not because of its implementation Vs. my Java skill set, but rather that I was not familiar with its concept. I am describing in this article my understanding on this implementation.

As **InputSplit** is nothing more than a chunk of 1 or several blocks, it should be pretty rare to get a block boundary ending up at the exact location of a end of line (EOL). Some of my records located around block boundaries should be therefore split in 2 different blocks. This triggers the following issues:

1. How Hadoop can guarantee lines read are 100% complete ?
2. How Hadoop can consolidate a line that is starting on block B and that ends up on B+1 ?
3. How Hadoop can guarantee we do not miss any line ?
4. Is there a limitation in term of line’s size ? Can a line be greater than a block (i.e. spanned over more than 2 blocks) ? If so, is there any consequence in term of MapReduce performance ?

## Definitions

### InputFormat

Definition taken from

Hadoop relies on the input format of the job to do three things:  
1. Validate the input configuration for the job (i.e., checking that the data is there).  
2. Split the input blocks and files into logical chunks of type InputSplit, each of which is assigned to a map task for processing.  
3. Create the RecordReader implementation to be used to create key/value pairs from the raw InputSplit. These pairs are sent one by one to their mapper.

### RecordReader

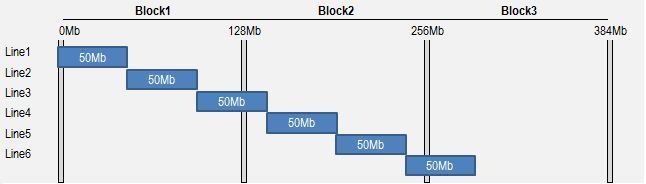
Definition taken from

A RecordReader uses the data within the boundaries created by the input split to generate key/value pairs. In the context of file-based input, the “start” is the byte position in the file where the RecordReader should start generating key/value pairs. The “end” is where it should stop reading records. These are not hard boundaries as far as the API is concerned—there is nothing stopping a developer from reading the entire file for each map task. While reading the entire file is not advised, reading outside of the boundaries it often necessary to ensure that a complete record is generated

## Example

I jumped right into the code of **LineRecordReader** and found it not that obvious to understand. Let’s get an example first that will hopefully make the code slightly more readable.  
Suppose my data set is composed on a single 300Mb file, spanned over 3 different blocks (blocks of 128Mb), and suppose that I have been able to get 1 InputSplit for each block. Let’s imagine now 3 different scenarios.

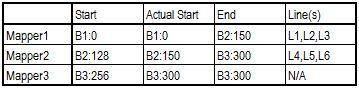
### File is composed on 6 lines of 50Mb each

[](https://hadoopi.files.wordpress.com/2013/05/inputsplit1.jpg)

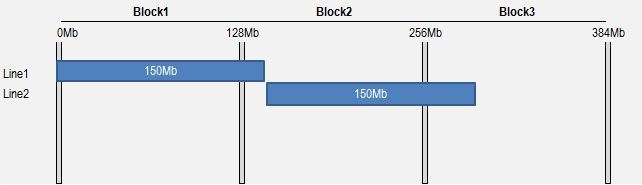
* The first Reader will start reading bytes from Block B1, position 0. The first two EOL will be met at respectively 50Mb and 100Mb. 2 lines (L1 & L2) will be read and sent as key / value pairs to Mapper 1 instance. Then, starting from byte 100Mb, we will reach end of our Split (128Mb) before having found the third EOL. This incomplete line will be completed by reading the bytes in Block B2 until position 150Mb. First part of Line L3 will be read locallyfrom Block B1, second part will be read remotelyfrom Block B2 (by the mean of **FSDataInputStream**), and a complete record will be finally sent as key / value to Mapper 1.
* The second Reader starts on Block B2, at position 128Mb. Because 128Mb is not the start of a file, there are strong chance our pointer is located somewhere in an existing record that has been already processed by previous Reader. We need to skip this record by jumping out to the next available EOL, found at position 150Mb. Actual start of RecordReader 2 will be at 150Mb instead of 128Mb.

We can wonder what happens in case a block starts exactly **on** a EOL. By jumping out until the next available record (through readLine method), we might miss 1 record. Before jumping to next EOL, we actually need to decrement initial “start” value to “start – 1”. Being located at at least 1 offset before EOL, we ensure no record is skipped !

Remaining process is following same logic, and everything is summarized in below table.

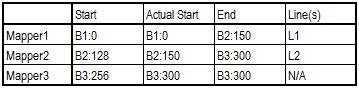
[](https://hadoopi.files.wordpress.com/2013/05/inputsplit_meta1.jpg)

### File composed on 2 lines of 150Mb each

[](https://hadoopi.files.wordpress.com/2013/05/inputsplit2.jpg)

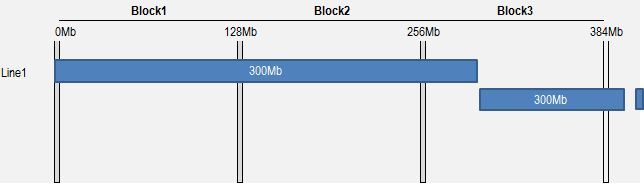
Same process as before:

* Reader 1 will start reading from block B1, position 0. It will read line L1 locallyuntil end of its split (128Mb), and will then continue reading remotelyon B2 until EOL (150Mb)
* Reader 2 will not start reading from 128Mb, but from 150Mb, and until B3:300

[](https://hadoopi.files.wordpress.com/2013/05/inputsplit_meta2.jpg)

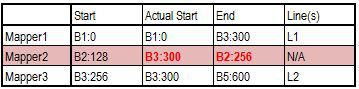
### File composed on 2 lines of 300Mb each

OK, this one is a tricky and perhaps unrealistic example, but I was wondering what happens in case a record is larger than 2 blocks (spanned over at least 3 blocks).

[](https://hadoopi.files.wordpress.com/2013/05/inputsplit5.jpg)

* Reader 1 will start reading locallyfrom B1:0 until B1:128, then remotelyall bytes available on B2, and finally remotely on B3 until EOL is reached (300Mb). There is here some overhead as we’re trying to read a lot of data that is not locally available
* Reader 2 will start reading from B2:128 and will jump out to next available record located at B3:300. Its new start position (B3:300) is actually greater than its maximum position (B2:256). This reader will therefore not provide Mapper 2 with any key / value. I understand it somehow as a kind of security feature ensuring data locality (that makes Hadoop so efficient in data processing) is preserved (i.e. Do not process a line that is not starting in the chunk I’m responsible for).
* Reader 3 will start reading from B3:300 to B5:600

This is summarized in below table

[](https://hadoopi.files.wordpress.com/2013/05/inputsplit_meta51.jpg)

#### Maximum size for a single record

There is a maximum size allowed for a single record to be processed. This value can be set using below parameter.

|  |  |
| --- | --- |
| 1 | conf.setInt("mapred.linerecordreader.maxlength", Integer.MAX\_VALUE); |

A line with a size greater than this maximum value (default is 2,147,483,647) will be ignored.

I hope these 3 examples gives you a high level understanding on **RecordReader** and **InputFormat**. If so, let’s jump to the code, else, let me know.

I doubt a single record is hundreds of Mb large (300Mb in my example) in a real environment… With hundreds of Kb for a single record, the overhead due to a line spanning over different blocks should not be that significant, and overall performance should not be really affected

## Implementation

### RecordReader

I added some (a tons of) comments in the code in order to point out what has been previously said in the example section. Hopefully this makes it slightly clearer. A new Reader must extends class **RecordReader** and override several methods.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188 | public class CustomLineRecordReader      extends RecordReader<LongWritable, Text> {        private long start;      private long pos;      private long end;      private LineReader in;      private int maxLineLength;      private LongWritable key = new LongWritable();      private Text value = new Text();        private static final Log LOG = LogFactory.getLog(              CustomLineRecordReader.class);        /\*\*       \* From Design Pattern, O'Reilly...       \* This method takes as arguments the map task’s assigned InputSplit and       \* TaskAttemptContext, and prepares the record reader. For file-based input       \* formats, this is a good place to seek to the byte position in the file to       \* begin reading.       \*/      @Override      public void initialize(              InputSplit genericSplit,              TaskAttemptContext context)              throws IOException {            // This InputSplit is a FileInputSplit          FileSplit split = (FileSplit) genericSplit;            // Retrieve configuration, and Max allowed          // bytes for a single record          Configuration job = context.getConfiguration();          this.maxLineLength = job.getInt(                  "mapred.linerecordreader.maxlength",                  Integer.MAX\_VALUE);            // Split "S" is responsible for all records          // starting from "start" and "end" positions          start = split.getStart();          end = start + split.getLength();            // Retrieve file containing Split "S"          final Path file = split.getPath();          FileSystem fs = file.getFileSystem(job);          FSDataInputStream fileIn = fs.open(split.getPath());            // If Split "S" starts at byte 0, first line will be processed          // If Split "S" does not start at byte 0, first line has been already          // processed by "S-1" and therefore needs to be silently ignored          boolean skipFirstLine = false;          if (start != 0) {              skipFirstLine = true;              // Set the file pointer at "start - 1" position.              // This is to make sure we won't miss any line              // It could happen if "start" is located on a EOL              --start;              fileIn.seek(start);          }            in = new LineReader(fileIn, job);            // If first line needs to be skipped, read first line          // and stores its content to a dummy Text          if (skipFirstLine) {              Text dummy = new Text();              // Reset "start" to "start + line offset"              start += in.readLine(dummy, 0,                      (int) Math.min(                              (long) Integer.MAX\_VALUE,                              end - start));          }            // Position is the actual start          this.pos = start;        }        /\*\*       \* From Design Pattern, O'Reilly...       \* Like the corresponding method of the InputFormat class, this reads a       \* single key/ value pair and returns true until the data is consumed.       \*/      @Override      public boolean nextKeyValue() throws IOException {            // Current offset is the key          key.set(pos);            int newSize = 0;            // Make sure we get at least one record that starts in this Split          while (pos < end) {                // Read first line and store its content to "value"              newSize = in.readLine(value, maxLineLength,                      Math.max((int) Math.min(                              Integer.MAX\_VALUE, end - pos),                              maxLineLength));                // No byte read, seems that we reached end of Split              // Break and return false (no key / value)              if (newSize == 0) {                  break;              }                // Line is read, new position is set              pos += newSize;                // Line is lower than Maximum record line size              // break and return true (found key / value)              if (newSize < maxLineLength) {                  break;              }                // Line is too long              // Try again with position = position + line offset,              // i.e. ignore line and go to next one              // TODO: Shouldn't it be LOG.error instead ??              LOG.info("Skipped line of size " +                      newSize + " at pos "                      + (pos - newSize));          }              if (newSize == 0) {              // We've reached end of Split              key = null;              value = null;              return false;          } else {              // Tell Hadoop a new line has been found              // key / value will be retrieved by              // getCurrentKey getCurrentValue methods              return true;          }      }        /\*\*       \* From Design Pattern, O'Reilly...       \* This methods are used by the framework to give generated key/value pairs       \* to an implementation of Mapper. Be sure to reuse the objects returned by       \* these methods if at all possible!       \*/      @Override      public LongWritable getCurrentKey() throws IOException,              InterruptedException {          return key;      }        /\*\*       \* From Design Pattern, O'Reilly...       \* This methods are used by the framework to give generated key/value pairs       \* to an implementation of Mapper. Be sure to reuse the objects returned by       \* these methods if at all possible!       \*/      @Override      public Text getCurrentValue() throws IOException, InterruptedException {          return value;      }        /\*\*       \* From Design Pattern, O'Reilly...       \* Like the corresponding method of the InputFormat class, this is an       \* optional method used by the framework for metrics gathering.       \*/      @Override      public float getProgress() throws IOException, InterruptedException {          if (start == end) {              return 0.0f;          } else {              return Math.min(1.0f, (pos - start) / (float) (end - start));          }      }        /\*\*       \* From Design Pattern, O'Reilly...       \* This method is used by the framework for cleanup after there are no more       \* key/value pairs to process.       \*/      @Override      public void close() throws IOException {          if (in != null) {              in.close();          }      }    } |

### FileInputFormat

Now that you have created a custom Reader, you need to use it from a class extending **FileInputFormat**, as reported below …

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class CustomFileInputFormat extends FileInputFormat<LongWritable,Text>{        @Override      public RecordReader<LongWritable, Text> createRecordReader(              InputSplit split, TaskAttemptContext context) throws IOException,              InterruptedException {          return new CustomLineRecordReader();      }  } |

### MapReduce

… and to use this new **CustomFileInputFormat** in your MapReduce driver code when specifying Input format.

|  |  |
| --- | --- |
| 1  2  3  4 | .../...  FileInputFormat.addInputPath(job, inputPath);  job.setInputFormatClass(CustomFileInputFormat.class);  .../... |

Congratulations, if you followed this article you have just re-invented the wheel. We did not do anything more that re-implementing **LineRecordReader**and **FileInputFormat**, default implementations for Text file. However, I hope you now understand a bit better how these 2 classes works, allowing you to create your custom Reader and therefore being able to handle specific file format.

I hope you liked this article, that it was not too high-level and therefore not a waste of time..  
Should you have any question / remarks / suggestions, feel free to comment. Feel also free to share it !

Cheers !

Advertisements

# **Hadoop Tutorial : Custom Record Reader with TextInputFormat**

In this hadoop tutorial we will have a look at the modification to our previous program wordcount with our own custom mapper and reducer by implementing a concept called as custom record reader. Before we attack the problem let us look at some theory required to understand the topic.

So far in the series of articles we have seen [how to create a mapreduce program without writing explicit mapper or reducer](http://analyticspro.org/2012/06/15/hadoop-a-wordcount-without-explicit-mapperreducer/) also in the second part we wrote the [wordcount with our own custom mapper and reducer](http://analyticspro.org/2012/07/24/word-count-with-custom-mapper-and-reducer/)

(Input format theory information **reference** from yahoo tutorial )

The InputFormat defines how to read data from a file into the Mapper instances. Hadoop comes with several implementations of InputFormat; some work with text files and describe different ways in which the text files can be interpreted. Others, like SequenceFileInputFormat, are purpose-built for reading particular binary file formats.

More powerfully, you can define your own InputFormat implementations to format the input to your programs however you want. For example, the default TextInputFormat reads lines of text files. The key it emits for each record is the byte offset of the line read (as a LongWritable), and the value is the contents of the line up to the terminating '\n' character (as a Text object). If you have multi-line records each separated by a $ character, you could write your own InputFormat that parses files into records split on this character instead.

Another important job of the InputFormat is to divide the input data sources (e.g., input files) into fragments that make up the inputs to individual map tasks. These fragments are called “splits” and are encapsulated in instances of the InputSplit interface. Most files, for example, are split up on the boundaries of the underlying blocks in HDFS, and are represented by instances of the FileInputSplitclass. Other files may be unsplittable, depending on application-specific data. Dividing up other data sources (e.g., tables from a database) into splits would be performed in a different, application-specific fashion. When dividing the data into input splits, it is important that this process be quick and cheap. The data itself should not need to be accessed to perform this process (as it is all done by a single machine at the start of the MapReduce job).

So in nutshell InputFormat does 2 tasks :

1. Divide the data source ( the data files ) into fragments or blocks which are sent to a mapper. These are called splits.
2. These splits are further divided into records and these records are provided one at a time to the mapper for processing. This is achieved through a class called as Record Reader

We will concentrate on customizing #2 above customizing #1 will be left for one of the next articles. By customizing record reader as in #2 above we get immense power of sending any kind of records / xml sections / JSON objects to the mapper after reading it from the source text files

Okey. Now that we understand how mapper is fed data from source files lets look at what we will try to achieve in the example program in this article.

**Problem :** We want our mapper to receive 3 records ( 3 lines ) from the source file at a time instead on 1 line as provided by default by the TextInputFormat.

**Approach :**

1. We will extend from  **TextInputFormat** class to create our own **NLinesInputFormat** .
2. We will also create our own **RecordReader** class called **NLinesRecordReader** where we will implement the logic of feeding 3 lines/records at a time.
3. We will make a change in our driver program to use our new **NLinesInputFormat** class.
4. To prove that we are really getting 3 lines at a time, instead of actually counting words ( which we already know now how to do ) , we will emit out number of lines we get in the input at a time as a key and 1 as a value , which after going through reducer will give us frequency of  each unique number of lines to the mappers.

**Example :**

**Step 1 : Creating NLinesInputFormat class as  a custom inputformat class.**

This is really straightforward, we will inherit our class from **TextInputFormat** and override **createInputFormat( )** function to use our custom record reader class **NLinesRecordReader**which we will soon write.   The sourcelisting for this follows :

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class NLinesInputFormat extends TextInputFormat{      @Override      public RecordReader<LongWritable, Text> createRecordReader(InputSplit split, TaskAttemptContext context) {          return new NLinesRecordReader();      }  } |

Now that we have our new inputformat ready lets look at creating custom record reader. This is little complicated and the source code is a modified version of hadoop’s own LineInputFormat.

**Step 2:  Creating NLinesRecordReader class as a custom RecordReader class.**

We will inherit from RecordReader class. RecordReader has 6 abstract methods which we will have to implement.

* close ( )
* getCurrentKey ( )
* getCurrentValue ( )
* getProgress ( )
* initialize ( )
* nextKeyValue ( )

The most important ones for our discussion are the **initialize** and **nextKeyvalue** functions which we will override. The initialize function will be called only once for each split so we will do setup in this function and the nextKeyValue function is called for providing records, here we will write logic so that we send 3 records in the **value**instead of default 1. Here is the source listing for the class :

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97 | public class NLinesRecordReader extends RecordReader<LongWritable, Text>{      private final int NLINESTOPROCESS = 3;      private LineReader in;      private LongWritable key;      private Text value = new Text();      private long start =0;      private long end =0;      private long pos =0;      private int maxLineLength;    @Override      public void close() throws IOException {          if (in != null) {              in.close();          }      }    @Override      public LongWritable getCurrentKey() throws IOException,InterruptedException {          return key;      }    @Override      public Text getCurrentValue() throws IOException, InterruptedException {          return value;      }    @Override      public float getProgress() throws IOException, InterruptedException {          if (start == end) {              return 0.0f;          }          else {              return Math.min(1.0f, (pos - start) / (float)(end - start));          }      }    @Override      public void initialize(InputSplit genericSplit, TaskAttemptContext context)throws IOException, InterruptedException {          FileSplit split = (FileSplit) genericSplit;          final Path file = split.getPath();          Configuration conf = context.getConfiguration();          this.maxLineLength = conf.getInt(&quot;mapred.linerecordreader.maxlength&quot;,Integer.MAX\_VALUE);          FileSystem fs = file.getFileSystem(conf);          start = split.getStart();          end= start + split.getLength();          boolean skipFirstLine = false;          FSDataInputStream filein = fs.open(split.getPath());            if (start != 0){              skipFirstLine = true;              --start;              filein.seek(start);          }          in = new LineReader(filein,conf);          if(skipFirstLine){              start += in.readLine(new Text(),0,(int)Math.min((long)Integer.MAX\_VALUE, end - start));          }          this.pos = start;      }    @Override      public boolean nextKeyValue() throws IOException, InterruptedException {          if (key == null) {              key = new LongWritable();          }          key.set(pos);          if (value == null) {              value = new Text();          }          value.clear();          final Text endline = new Text(&quot;\n&quot;);          int newSize = 0;          for(int i=0;i<NLINESTOPROCESS;i++){              Text v = new Text();              while (pos < end) {                  newSize = in.readLine(v, maxLineLength,Math.max((int)Math.min(Integer.MAX\_VALUE, end-pos),maxLineLength));                  value.append(v.getBytes(),0, v.getLength());                  value.append(endline.getBytes(),0, endline.getLength());                  if (newSize == 0) {                      break;                  }                  pos += newSize;                  if (newSize < maxLineLength) {                      break;                  }              }          }          if (newSize == 0) {              key = null;              value = null;              return false;          } else {              return true;          }      }  } |

**Step 3 : Change in driver to use new Inputformat**

Now that we have the custom record reader ready lets modify our driver to use the new input format by adding following line of code

|  |  |
| --- | --- |
| 1 | job.setInputFormatClass(NLinesInputFormat.class); |

**Step 4 : Change the mapper to emit number of lines it gets each time**  
Here is the listing; its pretty self explanatory. I am only putting listing of map function here for the listing here.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public void map(LongWritable key, Text value,Context context) throws java.io.IOException ,InterruptedException  {      String lines = value.toString();      String []lineArr = lines.split(&quot;\n&quot;);      int lcount = lineArr.length;      context.write(new Text(new Integer(lcount).toString()),new IntWritable(1));   } |

**Sample Data Input :**

I have used sample data input files of 10000 lines of following format

*Shantanu , Deo*

*Suruchi, Bhide*

*Shamika, Deo*

*…*

*Mujtaba, Ahmed*

**Sample Output from Reducer:**

*1        1*

*3        3333*

This is because our mapper got 3333 records of 3 lines each and 1 last record of 1 line.

I hope you understood the article. If you liked it please feel free to share this . Also comment.

Thanks and have great day!

## **1. Introduction**

To better understand RecordReader, we have to understand the InputFormat first.

InputFormat defines how the data is read from the input file and passed into the Mapper instance from processing.

Hadoop performs the following things with the input format:

* Validate the input for the job to make sure the data is present for processing.
* Split the input blocks into chunks of the format InputSplit.
* Assign each of the InputSplits to a map task for processing.
* Create the RecordReader implementation to be used to create key-value pairs from the raw InputSplit and pass these key-value pairs to mappers one at a time.

Apache Hadoop provides several implementations of InputFormat by default. For example, TextInputFormat reads lines of text files one at a time and SequenceFileInputFormat is used to read binary file formats. We can always build out own InputFormat implementation with a separate RecordReader based on the input data being used in Apache Hadoop.

So in this article we will concentrate on the RecordReader part and see how we can implement a custom RecordReader.

## **2. Record Reader**

RecordReader uses the data in the InputSplit and createS key-value pairs for the mapper. Now when we customize this, we can implement any kind of record reader. We can send JSON objects, XML objects or any other format to the mapper for processing.

A RepordReader usually stays in between the boundaries created by the input split to generate key-value pairs but this is not a hard restriction. A custom implementation can even read more data outside of the input split, it is not encouraged a lot but if it is needed to be done for a specific case then it is also fine.

As we can implement a custom reader with the custom length of the line but there is an option to set the limit on the size a single record can be allowed to have otherwise the whole record will not be processed and simply ignored. This parameter can be set using the following code:

|  |  |
| --- | --- |
| 1 | configuration.setInt("mapred.linerecordreader.maxlength", Integer.MAX\_VALUE); |

So here in the above code we have set the maximum length to be the max value an integer can have which is 2,147,483,647. Any records with the size greater then this will be ignored.

## **3. Custom RecordReader**

In this section, we will see how we can write our own Custom RecordReader. We have a lot of comments in the code itself to make it self-explanatory but we will still go through the important parts of the code after looking at the code below:

|  |  |
| --- | --- |
| 001 | package com.javacodegeeks.examples.CustomRecordReder; |
| 002 |  |

|  |  |
| --- | --- |
| 003 | import java.io.IOException; |
| 004 | import org.apache.commons.logging.Log; |

|  |  |
| --- | --- |
| 005 | import org.apache.commons.logging.LogFactory; |
| 006 | import org.apache.hadoop.conf.Configuration; |

|  |  |
| --- | --- |
| 007 | import org.apache.hadoop.fs.FSDataInputStream; |
| 008 | import org.apache.hadoop.fs.FileSystem; |

|  |  |
| --- | --- |
| 009 | import org.apache.hadoop.fs.Path; |
| 010 | import org.apache.hadoop.io.LongWritable; |

|  |  |
| --- | --- |
| 011 | import org.apache.hadoop.io.Text; |
| 012 | import org.apache.hadoop.mapreduce.InputSplit; |

|  |  |
| --- | --- |
| 013 | import org.apache.hadoop.mapreduce.RecordReader; |
| 014 | import org.apache.hadoop.mapreduce.TaskAttemptContext; |

|  |  |
| --- | --- |
| 015 | import org.apache.hadoop.mapreduce.lib.input.FileSplit; |
| 016 | import org.apache.hadoop.util.LineReader; |

|  |  |
| --- | --- |
| 017 |  |
| 018 | public class CustomLineRecordReader extends RecordReader { |

|  |  |
| --- | --- |
| 019 |  |
| 020 | private long start; |

|  |  |
| --- | --- |
| 021 | private long pos; |
| 022 | private long end; |

|  |  |
| --- | --- |
| 023 | private LineReader in; |
| 024 | private int maxLineLength; |

|  |  |
| --- | --- |
| 025 | private LongWritable key = new LongWritable(); |
| 026 | private Text value = new Text(); |

|  |  |
| --- | --- |
| 027 |  |
| 028 | private static final Log LOG = LogFactory.getLog(CustomLineRecordReader.class); |

|  |  |
| --- | --- |
| 029 |  |
| 030 | /\*\* |

|  |  |
| --- | --- |
| 031 | \* This method takes as arguments the map task’s assigned InputSplit and |
| 032 | \* TaskAttemptContext, and prepares the record reader. For file-based input |

|  |  |
| --- | --- |
| 033 | \* formats, this is a good place to seek to the byte position in the file to |
| 034 | \* begin reading. |

|  |  |
| --- | --- |
| 035 | \*/ |
| 036 | @Override |

|  |  |
| --- | --- |
| 037 | public void initialize(InputSplit genericSplit, TaskAttemptContext context) throwsIOException { |
| 038 |  |

|  |  |
| --- | --- |
| 039 | // This InputSplit is a FileInputSplit |
| 040 | FileSplit split = (FileSplit) genericSplit; |

|  |  |
| --- | --- |
| 041 |  |
| 042 | // Retrieve configuration, and Max allowed |

|  |  |
| --- | --- |
| 043 | // bytes for a single record |
| 044 | Configuration job = context.getConfiguration(); |

|  |  |
| --- | --- |
| 045 | this.maxLineLength = job.getInt("mapred.linerecordreader.maxlength", Integer.MAX\_VALUE); |
| 046 |  |

|  |  |
| --- | --- |
| 047 | // Split "S" is responsible for all records |
| 048 | // starting from "start" and "end" positions |

|  |  |
| --- | --- |
| 049 | start = split.getStart(); |
| 050 | end = start + split.getLength(); |

|  |  |
| --- | --- |
| 051 |  |
| 052 | // Retrieve file containing Split "S" |

|  |  |
| --- | --- |
| 053 | final Path file = split.getPath(); |
| 054 | FileSystem fs = file.getFileSystem(job); |

|  |  |
| --- | --- |
| 055 | FSDataInputStream fileIn = fs.open(split.getPath()); |
| 056 |  |

|  |  |
| --- | --- |
| 057 | // If Split "S" starts at byte 0, first line will be processed |
| 058 | // If Split "S" does not start at byte 0, first line has been already |

|  |  |
| --- | --- |
| 059 | // processed by "S-1" and therefore needs to be silently ignored |
| 060 | boolean skipFirstLine = false; |

|  |  |
| --- | --- |
| 061 | if (start != 0) { |
| 062 | skipFirstLine = true; |

|  |  |
| --- | --- |
| 063 | // Set the file pointer at "start - 1" position. |
| 064 | // This is to make sure we won't miss any line |

|  |  |
| --- | --- |
| 065 | // It could happen if "start" is located on a EOL |
| 066 | --start; |

|  |  |
| --- | --- |
| 067 | fileIn.seek(start); |
| 068 | } |

|  |  |
| --- | --- |
| 069 |  |
| 070 | in = new LineReader(fileIn, job); |

|  |  |
| --- | --- |
| 071 |  |
| 072 | // If first line needs to be skipped, read first line |

|  |  |
| --- | --- |
| 073 | // and stores its content to a dummy Text |
| 074 | if (skipFirstLine) { |

|  |  |
| --- | --- |
| 075 | Text dummy = new Text(); |
| 076 | // Reset "start" to "start + line offset" |

|  |  |
| --- | --- |
| 077 | start += in.readLine(dummy, 0, (int) Math.min((long) Integer.MAX\_VALUE, end - start)); |
| 078 | } |

|  |  |
| --- | --- |
| 079 |  |
| 080 | // Position is the actual start |

|  |  |
| --- | --- |
| 081 | this.pos = start; |
| 082 | } |

|  |  |
| --- | --- |
| 083 |  |
| 084 | /\*\* |

|  |  |
| --- | --- |
| 085 | \* Like the corresponding method of the InputFormat class, this reads a |
| 086 | \* single key/ value pair and returns true until the data is consumed. |

|  |  |
| --- | --- |
| 087 | \*/ |
| 088 | @Override |

|  |  |
| --- | --- |
| 089 | public boolean nextKeyValue() throws IOException { |
| 090 | // Current offset is the key |

|  |  |
| --- | --- |
| 091 | key.set(pos); |
| 092 |  |

|  |  |
| --- | --- |
| 093 | int newSize = 0; |
| 094 |  |

|  |  |
| --- | --- |
| 095 | // Make sure we get at least one record that starts in this Split |
| 096 | while (pos < end) { |

|  |  |
| --- | --- |
| 097 |  |
| 098 | // Read first line and store its content to "value" |

|  |  |
| --- | --- |
| 099 | newSize = in.readLine(value, maxLineLength, Math.max((int) Math.min(Integer.MAX\_VALUE, end - pos), maxLineLength)); |
| 100 |  |

|  |  |
| --- | --- |
| 101 | // No byte read, seems that we reached end of Split |
| 102 | // Break and return false (no key / value) |

|  |  |
| --- | --- |
| 103 | if (newSize == 0) { |
| 104 | break; |

|  |  |
| --- | --- |
| 105 | } |
| 106 |  |

|  |  |
| --- | --- |
| 107 | // Line is read, new position is set |
| 108 | pos += newSize; |

|  |  |
| --- | --- |
| 109 |  |
| 110 | // Line is lower than Maximum record line size |

|  |  |
| --- | --- |
| 111 | // break and return true (found key / value) |
| 112 | if (newSize < maxLineLength) { |

|  |  |
| --- | --- |
| 113 | break; |
| 114 | } |

|  |  |
| --- | --- |
| 115 |  |
| 116 | // Line is too long |

|  |  |
| --- | --- |
| 117 | // Try again with position = position + line offset, |
| 118 | // i.e. ignore line and go to next one |

|  |  |
| --- | --- |
| 119 | // TODO: Shouldn't it be LOG.error instead ?? |
| 120 | LOG.info("Skipped line of size " + newSize + " at pos " + (pos - newSize)); |

|  |  |
| --- | --- |
| 121 | } |
| 122 |  |

|  |  |
| --- | --- |
| 123 | if (newSize == 0) { |
| 124 | // We've reached end of Split |

|  |  |
| --- | --- |
| 125 | key = null; |
| 126 | value = null; |

|  |  |
| --- | --- |
| 127 | return false; |
| 128 | } else { |

|  |  |
| --- | --- |
| 129 | // Tell Hadoop a new line has been found |
| 130 | // key / value will be retrieved by |

|  |  |
| --- | --- |
| 131 | // getCurrentKey getCurrentValue methods |
| 132 | return true; |

|  |  |
| --- | --- |
| 133 | } |
| 134 | } |

|  |  |
| --- | --- |
| 135 |  |
| 136 | /\*\* |

|  |  |
| --- | --- |
| 137 | \* This methods are used by the framework to give generated key/value pairs |
| 138 | \* to an implementation of Mapper. Be sure to reuse the objects returned by |

|  |  |
| --- | --- |
| 139 | \* these methods if at all possible! |
| 140 | \*/ |

|  |  |
| --- | --- |
| 141 | @Override |
| 142 | public LongWritable getCurrentKey() throws IOException, |

|  |  |
| --- | --- |
| 143 | InterruptedException { |
| 144 | return key; |

|  |  |
| --- | --- |
| 145 | } |
| 146 |  |

|  |  |
| --- | --- |
| 147 | /\*\* |
| 148 | \* This methods are used by the framework to give generated key/value pairs |

|  |  |
| --- | --- |
| 149 | \* to an implementation of Mapper. Be sure to reuse the objects returned by |
| 150 | \* these methods if at all possible! |

|  |  |
| --- | --- |
| 151 | \*/ |
| 152 | @Override |

|  |  |
| --- | --- |
| 153 | public Text getCurrentValue() throws IOException, InterruptedException { |
| 154 | return value; |

|  |  |
| --- | --- |
| 155 | } |
| 156 |  |

|  |  |
| --- | --- |
| 157 | /\*\* |
| 158 | \* Like the corresponding method of the InputFormat class, this is an |

|  |  |
| --- | --- |
| 159 | \* optional method used by the framework for metrics gathering. |
| 160 | \*/ |

|  |  |
| --- | --- |
| 161 | @Override |
| 162 | public float getProgress() throws IOException, InterruptedException { |

|  |  |
| --- | --- |
| 163 | if (start == end) { |
| 164 | return 0.0f; |

|  |  |
| --- | --- |
| 165 | } else { |
| 166 | return Math.min(1.0f, (pos - start) / (float) (end - start)); |

|  |  |
| --- | --- |
| 167 | } |
| 168 | } |

|  |  |
| --- | --- |
| 169 |  |
| 170 | /\*\* |

|  |  |
| --- | --- |
| 171 | \* This method is used by the framework for cleanup after there are no more |
| 172 | \* key/value pairs to process. |

|  |  |
| --- | --- |
| 173 | \*/ |
| 174 | @Override |

|  |  |
| --- | --- |
| 175 | public void close() throws IOException { |
| 176 | if (in != null) { |

|  |  |
| --- | --- |
| 177 | in.close(); |
| 178 | } |

|  |  |
| --- | --- |
| 179 | } |
| 180 | } |

Following are the code snippets which we will highlight in the above class:

* **Lines: 49-50:** We fetch the start and the end of the input split we have.
* **Lines: 61-68:** Contains the code where we check where the RecordReader should start
* **Lines: 88-134:** This is the function overwritten to implement the functionality to check if the next key-value pair exists or not.

Besides these, all other methods and the code snippets in the class are self-explanatory.

## **4. Custom File Input Format**

Once we have our custom line record reader finished, we then need to extend the FileInputFormat class and overwrite the method to use out CustomLineRecordReder class.

|  |  |
| --- | --- |
| 01 | package com.javacodegeeks.examples.CustomRecordReder; |
| 02 |  |

|  |  |
| --- | --- |
| 03 | import java.io.IOException; |
| 04 |  |

|  |  |
| --- | --- |
| 05 | import org.apache.hadoop.io.LongWritable; |
| 06 | import org.apache.hadoop.io.Text; |

|  |  |
| --- | --- |
| 07 | import org.apache.hadoop.mapreduce.InputSplit; |
| 08 | import org.apache.hadoop.mapreduce.RecordReader; |

|  |  |
| --- | --- |
| 09 | import org.apache.hadoop.mapreduce.TaskAttemptContext; |
| 10 | import org.apache.hadoop.mapreduce.lib.input.FileInputFormat; |

|  |  |
| --- | --- |
| 11 |  |
| 12 | public class CustomFileInputFormat extends FileInputFormat{ |

|  |  |
| --- | --- |
| 13 |  |
| 14 | @Override |

|  |  |
| --- | --- |
| 15 | public RecordReader createRecordReader( |
| 16 | InputSplit split, TaskAttemptContext context) throws IOException, |

|  |  |
| --- | --- |
| 17 | InterruptedException { |
| 18 |  |

|  |  |
| --- | --- |
| 19 | return new CustomLineRecordReader(); |
| 20 | } |

|  |  |
| --- | --- |
| 21 | } |

Code in the CustomFileInputFormat is quite straightput. It uses the CustomLineRecordReader and returns the same object instance when needed.

## **5. Word Count Driver Class**

Now it is time to use the CustomFileInputFormat in out Hadoop Application, we will use the same old WordCount example but instead of the default FileInputFormat we will use out CustomFileInputFormat which in fact uses CustomLineRecordReaderfor reading the lines of input format.

|  |  |
| --- | --- |
| 01 | package com.javacodegeeks.examples.CustomRecordReder; |
| 02 |  |

|  |  |
| --- | --- |
| 03 | import org.apache.hadoop.conf.Configured; |
| 04 | import org.apache.hadoop.fs.Path; |

|  |  |
| --- | --- |
| 05 | import org.apache.hadoop.io.IntWritable; |
| 06 | import org.apache.hadoop.io.Text; |

|  |  |
| --- | --- |
| 07 | import org.apache.hadoop.mapreduce.Job; |
| 08 | import org.apache.hadoop.mapreduce.lib.input.FileInputFormat; |

|  |  |
| --- | --- |
| 09 | import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat; |
| 10 | import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat; |

|  |  |
| --- | --- |
| 11 | import org.apache.hadoop.util.Tool; |
| 12 | import org.apache.hadoop.util.ToolRunner; |

|  |  |
| --- | --- |
| 13 |  |
| 14 | /\*\* |

|  |  |
| --- | --- |
| 15 | \* The entry point for the WordCount example, |
| 16 | \* which setup the Hadoop job with Map and Reduce Class |

|  |  |
| --- | --- |
| 17 | \* |
| 18 | \* @author Raman |

|  |  |
| --- | --- |
| 19 | \*/ |
| 20 | public class WordCount extends Configured implements Tool{ |

|  |  |
| --- | --- |
| 21 |  |
| 22 | /\*\* |

|  |  |
| --- | --- |
| 23 | \* Main function which calls the run method and passes the args using ToolRunner |
| 24 | \* @param args Two arguments input and output file paths |

|  |  |
| --- | --- |
| 25 | \* @throws Exception |
| 26 | \*/ |

|  |  |
| --- | --- |
| 27 | public static void main(String[] args) throws Exception{ |
| 28 | int exitCode = ToolRunner.run(new WordCount(), args); |

|  |  |
| --- | --- |
| 29 | System.exit(exitCode); |
| 30 | } |

|  |  |
| --- | --- |
| 31 |  |
| 32 | /\*\* |

|  |  |
| --- | --- |
| 33 | \* Run method which schedules the Hadoop Job |
| 34 | \* @param args Arguments passed in main function |

|  |  |
| --- | --- |
| 35 | \*/ |
| 36 | public int run(String[] args) throws Exception { |

|  |  |
| --- | --- |
| 37 | if (args.length != 2) { |
| 38 | System.err.printf("Usage: %s needs two arguments   files\n", |

|  |  |
| --- | --- |
| 39 | getClass().getSimpleName()); |
| 40 | return -1; |

|  |  |
| --- | --- |
| 41 | } |
| 42 |  |

|  |  |
| --- | --- |
| 43 | //Initialize the Hadoop job and set the jar as well as the name of the Job |
| 44 | Job job = new Job(); |

|  |  |
| --- | --- |
| 45 | job.setJarByClass(WordCount.class); |
| 46 | job.setJobName("WordCounter"); |

|  |  |
| --- | --- |
| 47 |  |
| 48 | //Add input and output file paths to job based on the arguments passed |

|  |  |
| --- | --- |
| 49 | CustomFileInputFormat.addInputPath(job, new Path(args[0])); |
| 50 | job.setInputFormatClass(CustomFileInputFormat.class); |

|  |  |
| --- | --- |
| 51 |  |
| 52 | FileOutputFormat.setOutputPath(job, new Path(args[1])); |

|  |  |
| --- | --- |
| 53 |  |
| 54 | job.setOutputKeyClass(Text.class); |

|  |  |
| --- | --- |
| 55 | job.setOutputValueClass(IntWritable.class); |
| 56 | job.setOutputFormatClass(TextOutputFormat.class); |

|  |  |
| --- | --- |
| 57 |  |
| 58 | //Set the MapClass and ReduceClass in the job |

|  |  |
| --- | --- |
| 59 | job.setMapperClass(MapClass.class); |
| 60 | job.setReducerClass(ReduceClass.class); |

|  |  |
| --- | --- |
| 61 |  |
| 62 | //Wait for the job to complete and print if the job was successful or not |

|  |  |
| --- | --- |
| 63 | int returnValue = job.waitForCompletion(true) ? 0:1; |
| 64 |  |

|  |  |
| --- | --- |
| 65 | if(job.isSuccessful()) { |
| 66 | System.out.println("Job was successful"); |

|  |  |
| --- | --- |
| 67 | } else if(!job.isSuccessful()) { |
| 68 | System.out.println("Job was not successful"); |

|  |  |
| --- | --- |
| 69 | } |
| 70 |  |

|  |  |
| --- | --- |
| 71 | return returnValue; |
| 72 | } |

|  |  |
| --- | --- |
| 73 | } |

This is the driver class for out MapReduce job. The most important snippet of code for this example are lines 49 and 50. We set the path of the input file which we set in out CustomFileInputFormat and we set the job input format class to CustomFileInputFormat.

**Note:** For this example we will skip the Map and the Reduce class used in the MapReduce Driver class above. Map and Reduce used in this example are the same present in the article [Apache Hadoop Wordcount Example](http://examples.javacodegeeks.com/enterprise-java/apache-hadoop/apache-hadoop-wordcount-example/) and are also available in the code available at the bottom of the article.

## **6. Conclusion**

This brings us to the end of the article. So let us conclude what we understood in the article. We started with understanding what exactly is the RecordReader and InputSplit. How and what it is used. Followed by digging into the code to understand how to write custom RecordReader and Input Split. You can find the complete example in the download section below.

/\*\*

\* Licensed to the Apache Software Foundation (ASF) under one

\* or more contributor license agreements. See the NOTICE file

\* distributed with this work for additional information

\* regarding copyright ownership. The ASF licenses this file

\* to you under the Apache License, Version 2.0 (the

\* "License"); you may not use this file except in compliance

\* with the License. You may obtain a copy of the License at

\*

\* http://www.apache.org/licenses/LICENSE-2.0

\*

\* Unless required by applicable law or agreed to in writing, software

\* distributed under the License is distributed on an "AS IS" BASIS,

\* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

\* See the License for the specific language governing permissions and

\* limitations under the License.

\*/

package org.apache.hadoop.examples;

import java.io.DataInput;

import java.io.DataOutput;

import java.io.IOException;

import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configured;

import org.apache.hadoop.fs.FSDataInputStream;

import org.apache.hadoop.fs.FileSystem;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.io.WritableComparable;

import org.apache.hadoop.mapreduce.InputSplit;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.RecordReader;

import org.apache.hadoop.mapreduce.TaskAttemptContext;

import org.apache.hadoop.mapreduce.lib.input.CombineFileInputFormat;

import org.apache.hadoop.mapreduce.lib.input.CombineFileRecordReader;

import org.apache.hadoop.mapreduce.lib.input.CombineFileSplit;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.mapreduce.lib.reduce.IntSumReducer;

import org.apache.hadoop.util.LineReader;

import org.apache.hadoop.util.Tool;

import org.apache.hadoop.util.ToolRunner;

/\*\*

\* MultiFileWordCount is an example to demonstrate the usage of

\* MultiFileInputFormat. This examples counts the occurrences of

\* words in the text files under the given input directory.

\*/

public class MultiFileWordCount extends Configured implements Tool {

/\*\*

\* This record keeps &lt;filename,offset&gt; pairs.

\*/

public static class WordOffset implements WritableComparable {

private long offset;

private String fileName;

public void readFields(DataInput in) throws IOException {

this.offset = in.readLong();

this.fileName = Text.readString(in);

}

public void write(DataOutput out) throws IOException {

out.writeLong(offset);

Text.writeString(out, fileName);

}

public int compareTo(Object o) {

WordOffset that = (WordOffset)o;

int f = this.fileName.compareTo(that.fileName);

if(f == 0) {

return (int)Math.signum((double)(this.offset - that.offset));

}

return f;

}

@Override

public boolean equals(Object obj) {

if(obj instanceof WordOffset)

return this.compareTo(obj) == 0;

return false;

}

@Override

public int hashCode() {

assert false : "hashCode not designed";

return 42; //an arbitrary constant

}

}

/\*\*

\* To use {@link CombineFileInputFormat}, one should extend it, to return a

\* (custom) {@link RecordReader}. CombineFileInputFormat uses

\* {@link CombineFileSplit}s.

\*/

public static class MyInputFormat

extends CombineFileInputFormat<WordOffset, Text> {

public RecordReader<WordOffset,Text> createRecordReader(InputSplit split,

TaskAttemptContext context) throws IOException {

return new CombineFileRecordReader<WordOffset, Text>(

(CombineFileSplit)split, context, CombineFileLineRecordReader.class);

}

}

/\*\*

\* RecordReader is responsible from extracting records from a chunk

\* of the CombineFileSplit.

\*/

public static class CombineFileLineRecordReader

extends RecordReader<WordOffset, Text> {

private long startOffset; //offset of the chunk;

private long end; //end of the chunk;

private long pos; // current pos

private FileSystem fs;

private Path path;

private WordOffset key;

private Text value;

private FSDataInputStream fileIn;

private LineReader reader;

public CombineFileLineRecordReader(CombineFileSplit split,

TaskAttemptContext context, Integer index) throws IOException {

this.path = split.getPath(index);

fs = this.path.getFileSystem(context.getConfiguration());

this.startOffset = split.getOffset(index);

this.end = startOffset + split.getLength(index);

boolean skipFirstLine = false;

//open the file

fileIn = fs.open(path);

if (startOffset != 0) {

skipFirstLine = true;

--startOffset;

fileIn.seek(startOffset);

}

reader = new LineReader(fileIn);

if (skipFirstLine) { // skip first line and re-establish "startOffset".

startOffset += reader.readLine(new Text(), 0,

(int)Math.min((long)Integer.MAX\_VALUE, end - startOffset));

}

this.pos = startOffset;

}

public void initialize(InputSplit split, TaskAttemptContext context)

throws IOException, InterruptedException {

}

public void close() throws IOException { }

public float getProgress() throws IOException {

if (startOffset == end) {

return 0.0f;

} else {

return Math.min(1.0f, (pos - startOffset) / (float)(end - startOffset));

}

}

public boolean nextKeyValue() throws IOException {

if (key == null) {

key = new WordOffset();

key.fileName = path.getName();

}

key.offset = pos;

if (value == null) {

value = new Text();

}

int newSize = 0;

if (pos < end) {

newSize = reader.readLine(value);

pos += newSize;

}

if (newSize == 0) {

key = null;

value = null;

return false;

} else {

return true;

}

}

public WordOffset getCurrentKey()

throws IOException, InterruptedException {

return key;

}

public Text getCurrentValue() throws IOException, InterruptedException {

return value;

}

}

/\*\*

\* This Mapper is similar to the one in {@link WordCount.MapClass}.

\*/

public static class MapClass extends

Mapper<WordOffset, Text, Text, IntWritable> {

private final static IntWritable one = new IntWritable(1);

private Text word = new Text();

public void map(WordOffset key, Text value, Context context)

throws IOException, InterruptedException {

String line = value.toString();

StringTokenizer itr = new StringTokenizer(line);

while (itr.hasMoreTokens()) {

word.set(itr.nextToken());

context.write(word, one);

}

}

}

private void printUsage() {

System.out.println("Usage : multifilewc <input\_dir> <output>" );

}

public int run(String[] args) throws Exception {

if(args.length < 2) {

printUsage();

return 2;

}

Job job = new Job(getConf());

job.setJobName("MultiFileWordCount");

job.setJarByClass(MultiFileWordCount.class);

//set the InputFormat of the job to our InputFormat

job.setInputFormatClass(MyInputFormat.class);

// the keys are words (strings)

job.setOutputKeyClass(Text.class);

// the values are counts (ints)

job.setOutputValueClass(IntWritable.class);

//use the defined mapper

job.setMapperClass(MapClass.class);

//use the WordCount Reducer

job.setCombinerClass(IntSumReducer.class);

job.setReducerClass(IntSumReducer.class);

FileInputFormat.addInputPaths(job, args[0]);

FileOutputFormat.setOutputPath(job, new Path(args[1]));

return job.waitForCompletion(true) ? 0 : 1;

}

public static void main(String[] args) throws Exception {

int ret = ToolRunner.run(new MultiFileWordCount(), args);

System.exit(ret);

}

}

**MapReduce Inputs And Splitting**

Uptill now we have used only Text file as input files. Can we use any other file format? Can we use binary format or XML format? Let us find out. First, we need to get familiar with few terms . **Input split**: It is part of input processed by a single map. Each split is processed by a single map. In other words InputSplit represents the data to be processed by an individual Mapper. Each split is divided into records , and the map processes each record, which is a key value pair. Split is basically a number of rows and record is that number.  
  
The length of the InputSplit is measured in bytes. Every InputSplit has a storage locations (hostname strings). The storage locations are used by the MapReduce system to place map tasks as close to split's data as possible. The tasks are processed in the order of the size of the splits, largest one get processed first(**greedy approximation algorithm**). This is done in order to minimize the job runtime. One important thing to remember is that InputSplit doesn't contain input data but a reference to the data.

public abstract class InputSplit

{

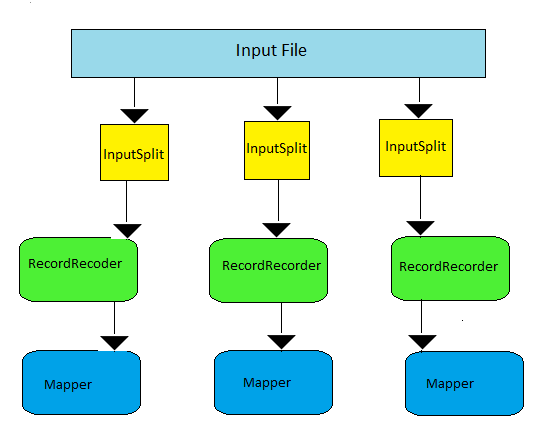
public abstract long getLength() throws IOException, InterruptedException;

public abstract String[] getLocations() throws IOException,InterruptedException;

}

As a user, we don't have to use InputSplits directly, InputFormat does that job. An InputFormat is a class that provides the following functionality:

* Selects the files or other objects that should be used for input.
* Defines the InputSplits that break a file into tasks.
* Provides a factory for RecordReader objects that read the file.

[](http://2.bp.blogspot.com/-47l79_b_owE/UvBgU3PKebI/AAAAAAAAALE/s1RQdtW0LhM/s1600/inputfile.png)

The overall process can be explained in following points:

* The client which runs the job calculates the splits for the job by calling getSplits().
* Client then sends the splits to the jobtracker, which uses their storage locations to schedule map tasks that will process them on the tasktrackers.
* On a tasktracker, the map task passes the split to the createRecordReader() method on InputFormat to obtain a RecordReader for that split.
* Map task uses RecordReader to generate record key-value pairs, which it passes to the map function. We can see this by looking at the Mapper’s run() method:

public void run(Context context) throws IOException, InterruptedException

{

setup(context);

while (context.nextKeyValue())

{

map(context.getCurrentKey(), context.getCurrentValue(), context);

}

cleanup(context);

}

First, the setup() method is called, then the nextKeyValue() is called repeatedly on the Context to populate the key and value objects for the mapper. Each key-value pair is retreived from the RecordReader and are passed to the map() method. The nextKeyValue() method returns false, when there is no more key-value pair left to get read. Then the map task runs its cleanup() method at the end.

**FileInputFormat**

It is the base class for all implementations of InputFormat. It provides two things: a place to define which files are included as the input to a job and an implementation for generating splits for the input files.

**How to split input files?**

Altough, **FileInputFormat** splits only those files which are larger than HDFS block. The split size can be controlled by various Hadoop properties. Input path and filter properties are given in the below table.

Property name Type Default value Description

mapred.min.split.size int 1 Smallest valid size in

bytes for a file split

mapred.max.split.size long Long.MAX\_VALUE, that is, Largest valid size in

9223372036854775807 bytes for a file split

dfs.block.size long 64 MB, The size block

in HDFS

The minimum split size is usually 1 byte, although some formats have a lower bound on the split size. We may impose a minimum split size. By setting this to a value larger than the block size, they can force splits to be larger than a block. But this is not good while using HDFS, because doing so will increase the number of blocks that are not local to a map task. The maximum split size defaults to the maximum value that can be represented by a Java long type. It has an effect only when it is less than the block size, forcing splits to be smaller than a block. The split size is calculated by the formula (see the computeSplitSize() method in FileInputFormat):  
  
max(minimumSize, min(maximumSize, blockSize))  
and by default:  
minimumSize < blockSize < maximumSize  
so the split size is blockSize.

**How to prevent splitting?**

Some applications don’t want files to be split, as this allows a single mapper to process each input file in its entirety. For example, a simple way to check if all the records in a file are sorted is to go through the records in order, checking whether each record is not less than the preceding one. Implemented as a map task, this algorithm will work only if one map processes the whole file. There are a couple of ways to ensure that an existing file is not split. The first (quick and dirty) way is to increase the minimum split size to be larger than the largest file in your system. Setting it to its maximum value, Long.MAX\_VALUE, has this effect. The second is to subclass the concrete subclass of FileInputFormat that you want to use, to override the isSplitable() method4 to return false. For example, here’s a nonsplittable TextInputFormat:

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.mapreduce.JobContext;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

public class NonSplittableTextInputFormat extends TextInputFormat {

@Override

protected boolean isSplitable(JobContext context, Path file) {

return false;

}

}

**Text Input**

**TextInputFormat:** As yoy may know, text files are usually taken as input files.Similarly, **TextInputFormat** is the default **InputFormat**. Now each record is a line of input and key (a LongWritable) is the byte offset within the file of the beginning of the line. Finally, value is the contents of the line( excuding any line terminators).For instance, file having folowing content:

A king should hunt regularly

A queen should shop daily,

Other people should just try.

The records are interpreted as the following key-value pairs.

0, A king should hunt regularly

29,A queen should shop daily,

55,Other people should just try.

Rememeber guys, the keys are not line numbers but offset from begining of the file. Offset are sufficient to serve as a unique identifier for each line. And if we combine it with the file name, it would be unique with in the filesystem.   
  
**NLineInputFormat**: The number of lines that each mapper receives depends on the size of split and the length of the lines. If we want to set this number then **NLineInputFormat** is the **InputFormat** to use. In this type too, the keys are the byte offsets within the file and values are the lines themselves. Here N refers to the number of lines of input that each mapper receives. The default value of N is 1, so each mapper receives excatly one line of input.

mapreduce.input.lineinputformat.linespermap

Above writen property controls the value of N.   
  
**XML**: Hadoop provides class for called **StreamXmlRecordReader**. We can use it by setting our input format to StreamInputFormat and setting the **stream.recordreader.class property to org.apache.hadoop.streaming.StreamXmlRecordReader**. The reader is configured by setting job configuration properties to tell it the patterns for the start and end tags.

**Binary Input**

Here is the answer to the question asked at the beginning of this post. Yes we can use data other than textual data. Hadoop MapReduce also support binary formats.  
  
**SequenceFileInputFormat :** It stores sequence of binary key-value pairs. To use data from sequence files as the input to MapReduce, you use SequenceFileIn putFormat.To use data from sequence files as the input to MapReduce, you use [SequenceFileIn putFormat.](http://dailyhadoopsoup.blogspot.in/2014/01/sequence-file.html)SequenceFileIn putFormat. SequenceFileInputFormat can read MapFiles as well as sequence files. SequenceFileInputFormat assumes that it is reading a MapFile and uses its datafile. This is why there is no MapFileInputFormat class.  
  
**SequenceFileAsTextInputFormat**: SequenceFileAsTextInputFormat is a variant of **SequenceFileInputFormat**which converts the sequence file’s keys and values to Text objects. The conversion is performed by  calling **toString()** method on the keys and values. This format makes sequence files suitable input for Streaming.

**Database Input and Output**

DBInputFormat is an input format for reading data from a relational database, using JDBC. We need to be careful not to overwhelm the database from which you are reading by running too many mappers, as it doesn’t have any sharding capabilities. For this reason, it is best used for loading relatively small datasets, perhaps for joining with larger datasets from HDFS using MultipleInputs. The corresponding output format is DBOutputFormat, which is useful for dumping job outputs (of modest size) into a database.

# InputSplit vs Block

The central idea behind MapReduce is distributed processing and hence the most important thing is to divide the dataset in to chunks and you have separate process working on the dataset on every chunk of data.

Lets assign some technical jargons now. The chunks are called input splits and the process working on the chunks (InputSplits) are called Mappers.

## Are InputSplits Same As Blocks?

InputSplit is not the same as the block.

A block is a hard division of data at the block size. So if the block size in the cluster is 128 MB, each block for the dataset will be 128 MB except for the last block which could be less than the block size if the file size is not entirely divisible by the block size. So a block is a hard cut at the block size and blocks can end even before a logical record ends.

Consider the block size in your cluster is 128 MB and each logical record is your file is about 100 Mb. (yes.. huge records)

So the first record will perfectly fit in the block no problem since the record size 100 MB is well with in the block size which is 128 MB. However the 2nd record can not fit in the block, so the record number 2 will start in block 1 and will end in block 2.

If you assign a mapper to a block 1, in this case, the Mapper can not process Record 2 because block 1 does not have the complete record 2. That is exactly the problem InputSplit solves. In this case InputSplit 1 will have both record 1 and record 2. InputSplit 2 does not start with Record 2 since Record 2 is already included in the Input Split 1. So InputSplit 2 will have only record 3. As you can see record 3 is divided between Block 2 and 3 but still InputSplit 2 will have the whole of record 3.

Blocks are physical chunks of data store in disks where as InputSplit is not physical chunks of data.  It is a Java class with pointers to start and end locations in blocks. So when Mapper tries to read the data it clearly knows where to start reading and where to stop reading. The start location of an InputSplit can start in a block and end in another block.

InputSplit respect logical record boundary and that is why it becomes very important.   During MapReduce execution Hadoop scans through the blocks and create InputSplits and each InputSplit will be assigned to individual mappers for processing.

[SEP](http://johnjianfang.blogspot.com/2014/09/outputformat-and-recordwriter-in-hadoop.html" \o "11th September 2014)

[11](http://johnjianfang.blogspot.com/2014/09/outputformat-and-recordwriter-in-hadoop.html" \o "11th September 2014)

# [**OutputFormat and RecordWriter in Hadoop Two**](http://johnjianfang.blogspot.com/2014/09/outputformat-and-recordwriter-in-hadoop.html)

OutputFormat is used to control how to write the mapreduce task output files. The OutputFormat only defines tree methods in this class.

public abstract class OutputFormat {

/\*\*

\* Get the {@link RecordWriter} for the given task.

\*

\* @param context the information about the current task.

\* @return a {@link RecordWriter} to write the output for the job.

\*/

public abstract RecordWriter

getRecordWriter(TaskAttemptContext context

) throws IOException, InterruptedException;

/\*\*

\* Check for validity of the output-specification for the job.

\*

\*

This is to validate the output specification for the job when it is

\* a job is submitted. Typically checks that it does not already exist,

\* throwing an exception when it already exists, so that output is not

\* overwritten.

\*

\* @param context information about the job

\* @throws IOException when output should not be attempted

\*/

public abstract void checkOutputSpecs(JobContext context

) throws IOException,

InterruptedException;

/\*\*

\* Get the output committer for this output format. This is responsible

\* for ensuring the output is committed correctly.

\* @param context the task context

\* @return an output committer

\*/

public abstract

OutputCommitter getOutputCommitter(TaskAttemptContext context

) throws IOException, InterruptedException;

}

getRecordWriter() is to provide a RecordWriter, checkOutputSpecs() is to check for validity of the output-specification for the job, and getOutputCommitter() is to return an OutputCommitter. As I described in previous posts, the OutputCommitter is used to move the data from a temporary folder to the final destination.  
  
The RecordWriter writes key/value pairs to an output file.

public abstract class RecordWriter {

/\*\*

\* Writes a key/value pair.

\*

\* @param key the key to write.

\* @param value the value to write.

\*/

public abstract void write(K key, V value

) throws IOException, InterruptedException;

/\*\*

\* Close this RecordWriter to future operations.

\*

\* @param context the context of the task

\*/

public abstract void close(TaskAttemptContext context

) throws IOException, InterruptedException;

}

A good example of the RecordWriter and the OutputFormat is illustrated in the class TextOutputFormat, where the LineRecordWriter is defined as follows.

public class TextOutputFormat extends FileOutputFormat {

public static String SEPERATOR = "mapreduce.output.textoutputformat.separator";

protected static class LineRecordWriter

extends RecordWriter {

private static final String utf8 = "UTF-8";

private static final byte[] newline;

static {

try {

// use "\n" as the newline delimiter

newline = "\n".getBytes(utf8);

} catch (UnsupportedEncodingException uee) {

throw new IllegalArgumentException("can't find " + utf8 + " encoding");

}

}

protected DataOutputStream out;

private final byte[] keyValueSeparator;

public LineRecordWriter(DataOutputStream out, String keyValueSeparator) {

this.out = out;

try {

// set key value separator

this.keyValueSeparator = keyValueSeparator.getBytes(utf8);

} catch (UnsupportedEncodingException uee) {

throw new IllegalArgumentException("can't find " + utf8 + " encoding");

}

}

// if not key value separator is defined, use tab as the separator

public LineRecordWriter(DataOutputStream out) {

this(out, "\t");

}

/\*\*

\* Write the object to the byte stream, handling Text as a special

\* case.

\* @param o the object to print

\* @throws IOException if the write throws, we pass it on

\*/

private void writeObject(Object o) throws IOException {

if (o instanceof Text) {

Text to = (Text) o;

out.write(to.getBytes(), 0, to.getLength());

} else {

out.write(o.toString().getBytes(utf8));

}

}

public synchronized void write(K key, V value)

throws IOException {

boolean nullKey = key == null || key instanceof NullWritable;

boolean nullValue = value == null || value instanceof NullWritable;

// return if null key or null value

if (nullKey && nullValue) {

return;

}

// write key first

if (!nullKey) {

writeObject(key);

}

// write separator

if (!(nullKey || nullValue)) {

out.write(keyValueSeparator);

}

// write value

if (!nullValue) {

writeObject(value);

}

// write newline

out.write(newline);

}

// close the output stream

public synchronized

void close(TaskAttemptContext context) throws IOException {

out.close();

}

}

}

The getRecordWriter() method not only creates a LineRecordWriter, but also setup the input stream properly in case that the output is required to be compressed.

public class TextOutputFormat extends FileOutputFormat {

public RecordWriter

getRecordWriter(TaskAttemptContext job

) throws IOException, InterruptedException {

Configuration conf = job.getConfiguration();

// check if the output should be compressed by checking the parameter "mapreduce.output.fileoutputformat.compress"

boolean isCompressed = getCompressOutput(job);

// use the key value separator from ""mapreduce.output.textoutputformat.separator" and tab if not specified

String keyValueSeparator= conf.get(SEPERATOR, "\t");

CompressionCodec codec = null;

String extension = "";

if (isCompressed) {

// get the codec class from "mapreduce.output.fileoutputformat.compress.codec"

Class codecClass =

getOutputCompressorClass(job, GzipCodec.class);

codec = (CompressionCodec) ReflectionUtils.newInstance(codecClass, conf);

// set the output extension

extension = codec.getDefaultExtension();

}

// get output file path

Path file = getDefaultWorkFile(job, extension);

// get the file system from the output file path

FileSystem fs = file.getFileSystem(conf);

if (!isCompressed) {

// create output stream as FSDataOutputStream if no compression

FSDataOutputStream fileOut = fs.create(file, false);

return new LineRecordWriter(fileOut, keyValueSeparator);

} else {

// create DataOutputStream wrapped with compression codec

FSDataOutputStream fileOut = fs.create(file, false);

return new LineRecordWriter(new DataOutputStream

(codec.createOutputStream(fileOut)),

keyValueSeparator);

}

}

}

All the other methods are defined in its base class FileOutputFormat.

public abstract class FileOutputFormat extends OutputFormat {

/\*\* Construct output file names so that, when an output directory listing is

\* sorted lexicographically, positions correspond to output partitions.\*/

private static final NumberFormat NUMBER\_FORMAT = NumberFormat.getInstance();

protected static final String BASE\_OUTPUT\_NAME = "mapreduce.output.basename";

protected static final String PART = "part";

static {

NUMBER\_FORMAT.setMinimumIntegerDigits(5);

NUMBER\_FORMAT.setGroupingUsed(false);

}

private FileOutputCommitter committer = null;

public static final String COMPRESS ="mapreduce.output.fileoutputformat.compress";

public static final String COMPRESS\_CODEC = "mapreduce.output.fileoutputformat.compress.codec";

public static final String COMPRESS\_TYPE = "mapreduce.output.fileoutputformat.compress.type";

public static final String OUTDIR = "mapreduce.output.fileoutputformat.outputdir";

public void checkOutputSpecs(JobContext job

) throws FileAlreadyExistsException, IOException{

// Ensure that the output directory is set and not already there

Path outDir = getOutputPath(job);

if (outDir == null) {

throw new InvalidJobConfException("Output directory not set.");

}

// get delegation token for outDir's file system

TokenCache.obtainTokensForNamenodes(job.getCredentials(),

new Path[] { outDir }, job.getConfiguration());

// check if the output directory already exists

if (outDir.getFileSystem(job.getConfiguration()).exists(outDir)) {

throw new FileAlreadyExistsException("Output directory " + outDir +

" already exists");

}

}

public synchronized

OutputCommitter getOutputCommitter(TaskAttemptContext context

) throws IOException {

if (committer == null) {

Path output = getOutputPath(context);

// return FileOutputCommitter if no committer is specified

committer = new FileOutputCommitter(output, context);

}

return committer;

}

}

29) Inputformat and outputformat

Answer)

# Hadoop :Custom Input Format

Posted on [**October 22, 2013**](https://shrikantbang.wordpress.com/2013/10/22/hadoop-custom-input-format/)

## Preface:

[Hadoop](http://hadoop.apache.org/)is popular open source distributed computing framework. The data to be processed on top of Hadoop is usually stored on Distributed File System. e.g. [HDFS (Hadoop Distributed File System)](http://hadoop.apache.org/docs/stable/hdfs_design.html).  
  
To read the data to be processed, Hadoop comes up with InputFormat, which has following responsibilities:

* Compute the input splits of data
* Provide a logic to read the input split

## From implementation point of view:

[InputFormat](http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/InputFormat.html)defines how to read the input data. Its responsibilities are :

1. Compute the input splits of data :  
   [InputSplit](http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/InputSplit.html)represents the part of data to be processed by Mapper instance .Each Mapper will get unique input split to process.  
   Before MR job starts, InputFormat splits the data into multiple parts based on their logical boundaries and HDFS block size.  
   Following method computes input splits.

|  |
| --- |
| abstract List getSplits(JobContext context) |

1. Provide a logic to read the input split :  
   Each Mapper gets unique input split to process. Input format provides a logic to read the split, which is an implementation of [RecordReader](http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/RecordReader.html). The record reader will read input split and emit <Key,Value> as input for each map function call.  
   Following method creates record reader for given split.

|  |
| --- |
| abstract RecordReader <K,V> createRecordReader(InputSplit is,  TaskAttemptContext context) |

## Example : Writing custom input format to read email dataset

Lets write the custom input format to read email data set. Usually emails are stored under the user-directory in sub-folders like inbox, outbox, spam, sent etc.[Email header](http://en.wikipedia.org/wiki/Email#Header_fields) contains sender,receivers, subject, date, message-ID and other metadata fields. We can parse the email header using Java APIs.

In following example we will read each email file and parse sender and receivers. Input key for map function will be email participants (sender and receiver) and input value will be [NullWritable](http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/io/NullWritable.html).

Following class contains email participants(sender and receivers) and will be input key for map function.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57 | // it contains email participants (sender and receivers)  public class EmailParticipants implements WritableComparable {        private Set receivers = null;      private String sender;      private int receiverCnt;        public EmailParticipants(String sender, Set receivers) {          this.sender = sender;          this.receivers = receivers;          receiverCnt = receivers.size();      }        public String getSender() {          return sender;      }        public void setSender(String from) {          this.sender = from;      }        public Set getReceiver() {          return receivers;      }        public void setReceivers(Set receivers) {          this.receivers = receivers;      }        public EmailParticipants() {      }        @Override      public void readFields(DataInput dataIp) throws IOException {          sender = dataIp.readUTF();          receiverCnt = dataIp.readInt();          receivers = new HashSet(receiverCnt);          for (int i = 0; i &lt; receiverCnt; i++) {              receivers.add(dataIp.readUTF());          }      }        @Override      public void write(DataOutput dataOp) throws IOException {          dataOp.writeUTF(receivers.toString());          dataOp.writeInt(receivers.size());          Iterator rcvr = receivers.iterator();          while (rcvr.hasNext()) {              dataOp.writeUTF(rcvr.next());          }      }        @Override      public int compareTo(EmailParticipants arg0) {          return sender.compareTo(arg0.getSender());      }  } |

Following implementation of input format will recursively read each file present under the input data directory. Each input split will contain a single unique email file. So total number of splits will be total number of emails.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | // Input format for reading email dataset  public class EmailInputFormat extends FileInputFormat<EmailParticipants, NullWritable> {      @Override    public RecordReader<EmailParticipants, NullWritable>      createRecordReader(InputSplit split,                         TaskAttemptContext context) {     return new EmailRecordReader();    }       @Override      protected List<FileStatus> listStatus(JobContext job) throws IOException {          return MapRedUtil.getAllFileRecursively(super.listStatus(job),                  job.getConfiguration());      }    @Override    protected boolean isSplitable(JobContext context, Path file) {      CompressionCodec codec =        new CompressionCodecFactory(context.getConfiguration()).getCodec(file);      return codec == null;    }  } |

Record reader will read entire split (email file) in nextKeyValue()in one go and emit <Key,Value> as <EmailParticipants, NullWritable> as input arguments to map function.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72 | // Record reader to parse email participants  public class EmailRecordReader extends          RecordReader<EmailParticipants, NullWritable> {      private boolean toggle = false;        private NullWritable value = NullWritable.get();      private Path file = null;      private Configuration jc;      private EmailParticipants emailData = null;        public EmailRecordReader() {      }        public void initialize(InputSplit genericSplit, TaskAttemptContext context)              throws IOException {          FileSplit split = (FileSplit) genericSplit;          file = split.getPath();          jc = context.getConfiguration();      }          public boolean nextKeyValue() throws IOException {          InputStream mailFileInputStream=null;          try {              mailFileInputStream= FileSystem.get(jc).open(file);              Properties props = new Properties();              Session session = Session.getDefaultInstance(props, null);              MimeMessage message = new MimeMessage(session, mailFileInputStream);              MimeMessageParser mimeParser = new MimeMessageParser(message);              Set<String> receivers = new HashSet<String>();              populateMailParticipients(mimeParser.getTo(), receivers);              populateMailParticipients(mimeParser.getCc(), receivers);              populateMailParticipients(mimeParser.getBcc(), receivers);              String sender = mimeParser.getFrom().replaceAll("\\.+", ".");              emailData = new EmailParticipants(sender, receivers);            } catch (MessagingException e) {              e.printStackTrace();              throw new IOException(e);            } catch (Exception e) {              e.printStackTrace();              throw new IOException(e);            }finally{              if (mailFileInputStream != null) {                  mailFileInputStream.close();              }            }            toggle = !toggle;            return toggle;         }        private void populateMailParticipients(List<Address> participants,              Set<String> list) {          for (Address addr : participants) {              String str = addr.toString().replaceAll("\\.+", ".");              list.add(str);          }      }        @Override      public EmailParticipants getCurrentKey() throws IOException, InterruptedException {          return emailData;      }        @Override      public NullWritable getCurrentValue() {          return value;      }        @Override      public float getProgress() throws IOException, InterruptedException {          return 0; // TODO      }  } |

## A Quick Test:

To test custom input format class we have to configure Hadoop [Job](http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/Job.html)as:

|  |  |
| --- | --- |
| 1  2 | Job job = new Job(conf, &quot;custom-input-format-test&quot;);  job.setInputFormatClass(EmailInputFormat.class); |

# **Map Reduce Input formats**

[August 20, 2015](http://www.teckstory.com/hadoop-ecosystem/map-reduce-input-format-mr-concepts-part-2/) by [Prashant Pandey](http://www.teckstory.com/author/prashant/)

We have already discussed some basic concepts in Map Reduce in earlier [post](http://www.teckstory.com/hadoop-ecosystem/map-reduce-concepts-part-1/) . This post is to cover one of the most important, complex  and least understood area of map reduce programming called map reduce input format. We will uncover internals of map reduce input format. We will briefly discuss about splits, record reader, multiple inputs and multiple outputs in map reduce programs. Essentially they all relate to Map Reduce Input Format.

# **Map Reduce Input Format ( FileInputFormat )**

In Hadoop Map Reduce, most of the time, you will be processing data stored as a file in HDFS. How do you specify your data files to your map reduce program. In any map reduce example, you will find reference to FileInputFormat class. It is used to specify input directory where your data files are located. FileInputFormat is the base class to specify map reduce input format which defines following things.

1. Input Path – Which files are included as the input to a job
2. Input Split – How to generate splits for the input files

## **Input Path**

First task is simple. You can specify list of input files using one of four static methods.

**FileInputFormat.addInputPath (Job job, Path   path)  
         FileInputFormat.addInputPaths(Job job, String commaSeparatedPaths)**

You can call above methods repeatedly to build the list of paths.  
There are two other methods. Below two methods set the entire list of paths in one go replacing any paths set on the Job in previous calls.

**FileInputFormat.setInputPaths(Job job, Path… inputPaths)          
         FileInputFormat.setInputPaths(Job job, String commaSeparatedPaths)**

A path may represent a file or a directory. A path representing a directory includes all the files in the directory except hidden files (begin with a dot or underscore). If the directory contains a sub directory, it will be interpreted as a file, which will cause an error.

You can set**mapreduce.input.fileinputformat.input.dir.recursive** configuration property to true to force the input directory to be read recursively. However such situations are rare because in a properly designed directory organization only leaf nodes will contain files.

In certain situations, it is a required to use custom filters on files based on their names. Which means you have many files in a directory but you want to exclude some of them from your map reduce processing. To achieve this, you can use below static method to set filters.

**FileInputFormat.setInputPathFilter(Job job, RegexExcludePathFilter.class);**

A sample implementation of RegexExcludePathFilter class is given below.

**public** **class** RegexExcludePathFilter **implements** PathFilter {  
**private** **final** String regex;  
**public** RegexExcludePathFilter(String regex) {  
**this**.regex = regex;  
}  
**public** **boolean** accept(Path path) {  
**return** !path.toString().matches(regex);  
}  
}

## **Input Split**

Second task of FileInputFormat class is to generate splits. Split is second essential part of map reduce input format. You may never want to modify split generation. However, you may need some degree of control over tweaking split size. Split size is calculated by **FileInputFormat.computeSplitSize()** method. Default implementation uses following formula.

**max(minimumSize, min(maximumSize, blockSize))**

Above values can be controlled by setting following hadoop properties.

**mapreduce.input.fileinputformat.split.minsize**               default value is 1

**mapreduce.input.fileinputformat.split.maxsize**default value is Long.MAX\_VALUE (un usually very large value)

**dfs.blocksize**                                                                                     Assume 128 MB

Using these default values in split size formula, split size comes equal to block size. If you want to make split size smaller than block size, you can set maximum size to lower than block size. If you want to make split size larger than block size, you can set minimum size to larger than block size.

# **Records in Map Reduce**

We already understand that each split is processed by one mapper and framework will call our map function passing one records from the split at a time. How are these records identified in a split? A split is a chunk of data which mappers will process.

There is another component involved at this stage called record reader. **Record reader** is responsible to divide splits into records. Record reader is probably most important part of any solution as it is the component which understands your data format. Record reader is at the core of map reduce input format.

Record readers are specified by subclasses of FileInputFormat. You may want to specify your own record reader. You can do this by extending FileInputFormat class, we will also cover this further. However there are several general purpose subclasses already available for use. Some of them are listed below.

* TextInputFormat
* KeyValueTextInputFormat
* NLineInputFormat
* CombineFileInputFormat
* SequenceFileInputFormat

It is not sufficient to specify file location but you will also have to specify your map reduce input format. You can specify your map reduce input format class using below method to your job.

**job.setInputFormatClass(TextInputFormat.class);**

You map reduce input format class will also determine Key and Value data types for your mapper. It is very common to write your own map reduce input format if your data doesn’t comply with any of the general purpose map reduce input formats bundled with hadoop ecosystem. Most of the venders willing to allow processing their data using hadoop ecosystem will publish their map reduce input format class. For example Cassandra provides ColumnFamilyInputFormat to read data from Cassandra database in your map reduce jobs.

**TextInputFormat** is the default map reduce Input format. Each record is a line of input. The key is a LongWritable object which is the byte offset of the beginning of the line within the file (whole file not just one split). So it will be unique if combined with the file name. The value is a Text object which is contents of the line, excluding line terminators (newline or carriage return).

**It is important to note that Splits will honor logical record boundaries, in these case lines. So you will not get broken lines to your mapper even though your blocks may store broken records.**

**KeyValueTextInputFormat** is similar to TextInputFormat except your line represents two parts, key and value separated by a tab. In this case key passed to your mapper will be the key from the line instead of byte offset. Default separator is tab but you can change it by setting following property.

**mapreduce.input.keyvaluelinerecordreader.key.value.separator**

If your data file stores key and value separated by a delimiter, you may choose to use this map reduce input format.

**NLineInputFormat** is another variant of TextInputFormat where key is again byte offset and values are content of the line. The only difference is that each mapper gets a fixed number of lines irrespective of line size. This map reduce input format actually controls splits and each split is created with N number of lines. The value of N can be specified setting following parameter.

**mapreduce.input.lineinputformat.linespermap**

**CombineFileInputFormat** was designed to work well with small files. It packs many files into each split so that each mapper has more to process. It takes node and rack locality into account when deciding which blocks to place in the same split, so it does not compromise the speed. This map reduce input form is used when you dealing with small files.

**SequenceFileInputFormat** is designed to deal with sequence files. They are binary files and stores sequences of binary key-value pairs. To read data from sequence files as the input to Map Reduce, you should use this map reduce input format class. The keys and values are determined by the sequence file (as metadata is stored inside sequence file), and you need to make sure that your map input types correspond. For example, if your sequence file has IntWritable keys and Text values, then the map signature would be Mapper<IntWritable, Text, K, V>, where K and V are the types of the map output keys and values.

# **Implementing your own Record Reader**

There are two steps to implement your own record reader. First step is to extend FileInputFormat class and override createRecordReader method. An example is shown below from the most reputed book “Hadoop Definitive Guide” by Tom White.

**public** **class** RegexExcludePathFilter **implements** PathFilter {  
**private** **final** String regex;  
**public** RegexExcludePathFilter(String regex) {  
**this**.regex = regex;  
}  
**public** **boolean** accept(Path path) {  
**return** !path.toString().matches(regex);  
}  
}

Example implemented above does two things.

1. Extend FileInputFormat class and override createRecordReader. You need to specify your own class for your custom record reader. This example is creating a new instance of WholeFileRecordReader which is a custom record reader. Sample code for this class is given below.
2. Second thing we did is to makes sure that we get one split per file by overriding isSplitable and returning false. If you have multiple files, you still get multiple splits, one split per file. This is not a general step which you will be doing while implementing your own record reader but it is necessary for this example.

Second part is to extend RecordReader base class and implement your own record reader.

**class** WholeFileRecordReader **extends** RecordReader<NullWritable, BytesWritable> {  
**private** FileSplit fileSplit;  
**private** Configuration conf;  
**private** BytesWritable value = **new** BytesWritable();  
**private** **boolean** processed = **false**;

    @Override  
**public** **void** initialize(InputSplit split, TaskAttemptContext context)  
**throws** IOException, InterruptedException {  
**this**.fileSplit = (FileSplit) split;  
**this**.conf = context.getConfiguration();  
}

    @Override  
**public** **boolean** nextKeyValue() **throws** IOException, InterruptedException {  
**if** (!processed) {  
**byte**[] contents = **new** **byte**[(**int**) fileSplit.getLength()];  
Path file = fileSplit.getPath();  
FileSystem fs = file.getFileSystem(conf);  
FSDataInputStream in = **null**;  
**try** {  
in = fs.open(file);  
IOUtils.readFully(in, contents, 0, contents.length);  
value.set(contents, 0, contents.length);  
} **finally** {  
IOUtils.closeStream(in);  
}  
processed = **true**;  
**return** **true**;  
}  
**return** **false**;  
}

@Override  
**public** NullWritable getCurrentKey() **throws** IOException, InterruptedException {  
**return** NullWritable.get();  
}

    @Override  
**public** BytesWritable getCurrentValue() **throws** IOException, InterruptedException {  
**return** value;  
}

    @Override  
**public** **float** getProgress() **throws** IOException {  
**return** processed ? 1.0f : 0.0f;  
}

    @Override  
**public** **void** close() **throws** IOException {  
// do nothing  
}  
}

To understand this, we will have to understand “How mapper is called over a split”. The run method of mapper is shown below.

**public** **void** run(Context context) **throws** IOException, InterruptedException {  
setup(context);  
**while** (context.nextKeyValue()) {  
      map(context.getCurrentKey(), context.getCurrentValue(), context);  
}  
cleanup(context);  
}

Ok, so it’s a loop which will ask for nextKeyValue() from the context each iteration. Context will internally call nextKeyValue() method from the record reader. That’s why we override nextKeyValue() method in our record reader. Now explanation is simple, when nextKeyValue() is called first time, it reads the entire split at once and set it into value ( a BytesWritable object) and returns true. When nextKeyValue() is called second time, it returns false indicating end of split.

One more important observation is that a split doesn’t contain the input data. It is just a reference to the data which contains length of split and location. That’s why we call fileSplit.getPath() and open/read that file from file system.

getCurrentKey() and getCurrentValue() are simple. getCurrentKey() returns null because we don’t have any key in our example. getCurrentValue() returns last read value.

You should investigate few more map reduce input format classes to see how they implement record reader for example TextInputFormat record reader map reduce input format class.

But this all is complex when it comes to write a new record reader for your file and only an experienced java programmer can handle this complexity of creating new map reduce input format.

# **File information in the mapper**

Sometimes, you may need to know which file your mapper is processing. You can get file name to which your current split belongs to. You can get this information using below line of code.

**InputSplit split = context.getInputSplit();  
Path path = ((FileSplit) split).getPath();**

# **Preventing splitting**

Sometime, you don’t want your data file to split. This may be required if you want a single mapper which will process entire file. You can do this by extending your map reduce input format class (for example TextInputFormat) and override the isSplitable() method.

**public** **class** NonSplittableTextInputFormat **extends** TextInputFormat {  
    @Override  
**protected** **boolean** isSplitable(JobContext context, Path file) {  
**return** **false**;  
}  
}

Note that by doing this, you have single mapper processing entire file but record by record. And how records are identified will depend upon your map reduce input format class which you have extended. In my example TextInputFormat map reduce input format class is extended which reads each line as a record.

# **Multiple mappers for Multiple Inputs**

We have already learned that you can add multiple input directories to you Map Reduce job using FileInputFormat.addInputPath or FileInputFormat.addInputPaths method. But you expect all of the data files though they are placed in different directories are of same map reduce input format hence your mapper will be able to process them. What do you do if your files are in different map reduce input format? Files in directory “A” are different than files in directory “B” however they represent same data. This can happen if your data sources make modification in their schema. You may have to specify different mapper for each directory. The situation can be worse if record delimiter or file format is also changed, you will need to specify different input format classes.

**MultipleInputs** class allows you to specify which InputFormat and which Mapper to use on a per-path basis.

**MultipleInputs.addInputPath(job, Path1, TextInputFormat.class, Mapper1.class);  
MultipleInputs.addInputPath(job, Path2, TextInputFormat.class, Mapper2.class);**

You will have to replace your call to FileInputFormat.addInputPath with above call. Note that reducers in this case are not aware of the different mappers used. So you will have to handle your schema change in mappers.

# **Multiple Outputs**

Usually, there is one file per reducer named as part-r-nnnnn created in a single directory. If you have got 10 such files in your output directories, it is most likely that you had 10 reducers and each part file is created by one reducer. There are requirements to control naming convention of the output file. Sometimes, you may not want all of your output files dumped into a single directory. You want to control where each file goes. This all can be done by MultipleOutputs class. This class allows following things.

1. Control the naming of the output files
2. Make a reducer to create more than a single file.
3. Placing the file in different directories.

If you just have a mapper and no reducer, you can use this class in your mapper also.

An example of such a reducer is shown below.

**class** MultipleOutputsReducer **extends** Reducer<Text, Text, NullWritable, Text> {  
**private** MultipleOutputs<NullWritable, Text> multipleOutputs;  
    @Override  
**protected** **void** setup(Context context) **throws** IOException, InterruptedException {  
multipleOutputs = **new** MultipleOutputs<NullWritable, Text>(context);  
}  
    @Override  
**protected** **void** reduce(Text key, Iterable values, Context context)  
**throws** IOException, InterruptedException {  
**for** (Text value : values) {  
multipleOutputs.write(NullWritable.get(), value, key.toString());  
}  
}  
    @Override  
**protected** **void** cleanup(Context context)  
**throws** IOException, InterruptedException {  
multipleOutputs.close();  
}  
}

We construct an instance of MultipleOutputs in the setup() method and assign it to an instance variable. We then use the MultipleOutputs instance in the reduce() method to write to the output instead of the context.

Note that The write() method takes the key and value, it also takes a Base Name. So Overall effect of this example is to produce output files with the naming scheme key-r-nnnnn instead of part-r-nnnnn. The BaseName specified in the write() method of MultipleOutputs is interpreted relative to the output directory. It can contain file path separator characters (/) allowing us to create sub directories of arbitrary depth.

More complex setups are possible for example you can create named outputs, each with its own OutputFormat and key and value types. Consult the Java documentation for more information.

In this post we discussed map reduce input format and tried to cover most important aspects of map reduce input format. If you are interested in learning more about map reduce input format or looking for some examples on map reduce input format, 4th edition of Tom Whit’s[book](http://shop.oreilly.com/product/0636920033448.do) is your best reference.

# **Custom Input Format in Hadoop**



In this post, we will be looking at ways to implement custom input format in Hadoop. For doing this, we have taken the Titanic [Bigdata](https://acadgild.com/blog/ambari-installation-guide-part/)set as an example and have implemented the following problem statement.

## Problem Statement:

Find out the number of people who died and survived, along with their genders.

### ****Data Set Description:****

The dataset description is as follows:

**Column 1**: PassengerId

**Column 2**: Survived (survived=0 & died=1)

**Column 3:** Pclass



## **Fascinated by latest technology?**

Subscribe to our blog and we'll keep you updated with latest technologies like AngularJS, Machine Learning, Big Data, Spark, Android, Analytics, NodeJS, Cloud Computing and many more!

Top of Form



Subscribe, It’s Free!

Bottom of Form

**Column 4**: Name

**Column 5**: Sex

**Column 6**: Age

**Column 7**: SibSp

**Column 8**: Parch

**Column 9**: Ticket

**Column 10**: Fare

**Column 11**: Cabin

**Column 12**: Embarked

You can download the data set from the below link:

[Titanic Data set](https://drive.google.com/open?id=0ByJLBTmJojjzNmV0dk1EMmwwQ1U)

Here, we need to implement a custom key which is a combination of two columns i.e., 2ndcolumn, which consists of the dead or the survivors and the 5th column, which contains the gender of the person. So, let’s prepare a custom key by combining both these columns and sort them using the gender column.

To begin with, we need to prepare our custom key. To prepare a custom, we need to implement the WritableComparable interface. Below is the source code, which contains the implementation of custom key.

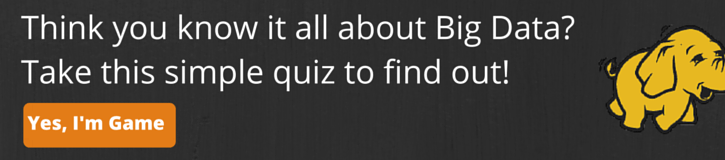
To know more about writableComparable,[click here](https://acadgild.com/blog/writable-and-writablecomparable-in-hadoop/).

***Custom Key***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101 | import java.io.DataInput;    import java.io.DataOutput;    import java.io.IOException;    import org.apache.hadoop.io.\*;    import com.google.common.collect.ComparisonChain;    public class Key\_value implements WritableComparable<Key\_value> {    private String x;    private String y;    public String getX() {    return x;    }    public void setX(String x) {    this.x = x;    }    public String getY() {    return y;    }    public void setY(String y) {    this.y = y;    }    public Key\_value(String x, String y) {    this.x = x;    this.y = y;    }    public void write(DataOutput out) throws IOException {    out.writeUTF(x);    out.writeUTF(y);    }    public void readFields(DataInput in) throws IOException {    x = in.readUTF();    y = in.readUTF();    }    public Key\_value(){    }    @Override    public int compareTo(Key\_value o) {    // TODO Auto-generated method stub    return ComparisonChain.start().compare(this.y,o.y).compare(this.x,o.x).result();    }    public boolean equals(Object o1) {    if (!(o1 instanceof Key\_value)) {    return false;    }    Key\_value other = (Key\_value)o1;    return this.x == other.x && this.y == other.y;    }    @Override    public String toString() {    return x.toString()+","+y.toString();    }    } |

In the compareTo method, we have written our logic to sort the keys by the gender column. We have taken the ComparisionChain class and first compared the gender column and then compared the 1st column. Therefore, this logic will print the keys sorted by Gender column.

***Note:****If you compare only one column, then the second will be considered as a single value by the WritableComparable interface.*

[](https://acadgild.com/blog/take-quiz-test-hadoop-knowledge-intermediate?utm_source=Blog(organic)&utm_medium=Blog%20article&utm_campaign=Big%20Data%20Landing%20Page)

Now, we have written a custom key. Next, we need to write one inputFormat class which extends the default FileInputFormat.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | import java.io.IOException;    import org.apache.hadoop.io.IntWritable;    import org.apache.hadoop.mapreduce.InputSplit;    import org.apache.hadoop.mapreduce.RecordReader;    import org.apache.hadoop.mapreduce.TaskAttemptContext;    import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;    public class Titanic\_input extends FileInputFormat<Key\_value,IntWritable> {    @Override    public RecordReader<Key\_value,IntWritable> createRecordReader(InputSplit arg0,    TaskAttemptContext arg1) throws IOException, InterruptedException {    return new MyRecordReader();    }    } |

Here, we are implementing the custom input format by extending the default FileInputFormat, which accepts the parameters key and value as our custom \_key and the value as IntWritable.

Now, these values are passed to the Record reader, which does the actual formatting of the inputs. The custom RecordReader class is as follows.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119 | import java.io.IOException;    import org.apache.hadoop.io.IntWritable;    import org.apache.hadoop.io.Text;    import org.apache.hadoop.mapreduce.InputSplit;    import org.apache.hadoop.mapreduce.RecordReader;    import org.apache.hadoop.mapreduce.TaskAttemptContext;    import org.apache.hadoop.mapreduce.lib.input.LineRecordReader;    public class MyRecordReader extends RecordReader<Key\_value,IntWritable> {    private Key\_value key;    private IntWritable value;    private LineRecordReader reader = new LineRecordReader();    @Override    public void close() throws IOException {    // TODO Auto-generated method stub    reader.close();    }    @Override    public Key\_value getCurrentKey() throws IOException, InterruptedException {    // TODO Auto-generated method stub    return key;    }    @Override    public IntWritable getCurrentValue() throws IOException, InterruptedException {    // TODO Auto-generated method stub    return value;    }    @Override    public float getProgress() throws IOException, InterruptedException {    // TODO Auto-generated method stub    return reader.getProgress();    }    @Override    public void initialize(InputSplit is, TaskAttemptContext tac)    throws IOException, InterruptedException {    reader.initialize(is, tac);    }    @Override    public boolean nextKeyValue() throws IOException, InterruptedException {    // TODO Auto-generated method stub    boolean gotNextKeyValue = reader.nextKeyValue();    if(gotNextKeyValue){    if(key==null){    key = new Key\_value();    }    if(value == null){    value = new IntWritable();    }    Text line = reader.getCurrentValue();    String[] tokens = line.toString().split(",");    key.setX(new String(tokens[1]));    key.setY(new String(tokens[4]));    value.set(new Integer(1));    }    else {    key = null;    value = null;    }    return gotNextKeyValue;    }    } |

In the RecordReader, the nextKeyVlaue() is the method passed to our inputs. From the dataset, this RecordReader will take each line as input and sets the columns into our custom key as follows:

|  |  |
| --- | --- |
| 1  2  3 | key.setX(new String(tokens[1]));    key.setY(new String(tokens[4])); |

As discussed earlier, we need 2nd and 5th columns passed to our custom key.

The value is set as ‘**1**‘ since we need to count the number of people.

|  |  |
| --- | --- |
| 1 | value.set(new Integer(1)); |

Now, the Mapper class is as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public static class Map extends Mapper<Key\_value, Text, Key\_value, IntWritable> {    private final static IntWritable one = new IntWritable(1);    public void map(Key\_value key, IntWritable value, Context context ) throws IOException, InterruptedException {    context.write(key1, one);    }    } |

The Mapper class will just emit the keys and values, as it is sent by the RecordReader.

The Reducer class is as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public static class Reduce extends Reducer<Key\_value, IntWritable, Key\_value, IntWritable> {    public void reduce(Key\_value key, Iterable<IntWritable> values, Context context)    throws IOException, InterruptedException {    int sum = 0;    for (IntWritable val : values) {    sum += val.get();    }    context.write(key, new IntWritable(sum));    }    } |

The Reducer will count all the values for each unique Reducer.

The Driver class is as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | public static void main(String[] args) throws IOException, ClassNotFoundException, InterruptedException {    Configuration conf=new Configuration();    Job job=new Job();    job.setJarByClass(Female\_survived.class);    job.setMapperClass(Map.class);    job.setReducerClass(Reduce.class);    job.setOutputKeyClass(Key\_value.class);    job.setMapOutputKeyClass(Key\_value.class);    job.setMapOutputValueClass(IntWritable.class);    job.setOutputValueClass(IntWritable.class);    job.setInputFormatClass(Titanic\_input.class);    job.setOutputFormatClass(TextOutputFormat.class);    Path out=new Path(args[1]);    out.getFileSystem(conf).delete(out);    FileInputFormat.addInputPath(job,new Path( args[0]));    FileOutputFormat.setOutputPath(job, new Path(args[1]));    job.waitForCompletion(true);    } |

We need to set the **InputFormatClass** as our **CustomInput** class. The **OutputKey** and **OutputValue**classes also needs to be set appropriately, otherwise it will throw an error

The final output after running this program is as shown in the below screen shot.

The Keys have been sorted by the gender column. We have successfully implemented custom input format in Hadoop.

Hope this post has been helpful in understanding how to implement custom input format in Hadoop. In case of any queries, feel free to comment below and we will get back to you at the earliest.

31) Skiping badrecord using mapreduce

Answer)

# **op 3 Methods of Skipping Big Data’s Bad Data in Hadoop !**

Posted on [August 28, 2014](https://kumarchinnakali.wordpress.com/2014/08/28/top-3-methods-of-skipping-big-datas-bad-data-in-hadoop/) by [kumarchinnakali](https://kumarchinnakali.wordpress.com/author/kumarchinnakali/) • Posted in [Analytics](https://kumarchinnakali.wordpress.com/category/analytics/), [Big Data](https://kumarchinnakali.wordpress.com/category/big-data/), [Hadoop](https://kumarchinnakali.wordpress.com/category/hadoop/) • [Leave a comment](https://kumarchinnakali.wordpress.com/2014/08/28/top-3-methods-of-skipping-big-datas-bad-data-in-hadoop/#respond)

Team, this time i go with the title called “**Top 3 Methods of Skipping Big Data’s Bad Records in Hadoop !**“ which describes about how to get corrupt records out from the large data sets which has different format of data.

While doing our analysis if the corrupt records are in small percentage we can ignore or skip, else we need to retry the task by assuming hardware or network failures. After four times of retrying, it will be marked failed if it’s not succeed.

**Method One:**TextInputFormat. We can use the Hadoop’s default input format called TextInputFormat to set the maximum expected length of records or instance.  This can be set by **mapred.linerecordreader.maxlength** parameter in bytes. The best to give is to give greater than the length of the records. This helps us to skip the record with out task failing.

**Method Two:**Writing the Map Reduce code to handle the corrupt record is one of the best method in practice. Either in our Mapper side or Reducer side we can detect the bad record and ignore it, or we can terminate the job by throwing an exception. With the help of MapReduce program we can use in built Hadoop’s counters to see the total number of bad records or skipped records. You can also count the total number of bad records in the job using counters to see  
how widespread the problem is impacting our outcome. And few subjective counters are Hadoop built in counters are MAP\_SKIPPED\_RECORDS, REDUCE\_SKIPPED\_GROUPS, REDUCE\_SKIPPED\_RECORDS, FAILED\_SHUFFLE.

**Method Three:**Hadoop’sSkipping mode. While working with vast amounts of data, if the failure is not handled properly bad data can easily cause to error to the applications. In built Hadoop’s skipping mechanism can be help us to pinpoint the bad data and log it for review and validations. To enable the skipping of ‘n’ bad records in a map and reduce job, we need to add to the run() where the job config is setup.  **SkipBadRecords.setMapperMaxSkipRecords(conf, n)**and **SkipBadRecords.setReduceMaxSkipGroups(conf, n)** respectively. And please be informed that skipping mode is off  by default.

Map/Reduce tasks report the records being processed back to the tasktracker if the skipping mode is enabled. When the task fails, the tasktracker retries the task, skipping the records that caused the failure. The skipping mode is turned on for a task only after it has failed twice. Thus, for a task consistently failing on a bad record, the tasktracker runs the following  
task attempts with these outcomes:  
1. Task fails.  
2. Task fails.  
3. Skipping mode is enabled. Task fails, but failed record is stored by the tasktracker.  
4. Skipping mode is still enabled. Task succeeds by skipping the bad record that failed in the previous attempt.

Please feel free to comment or suggest and if you find any other method to skip the bad data in Hadoop ecosystem please do share that.

Thanks for your valuable time !

See you in next post, Kumar Chinnakali

## Class SkipBadRecords

[java.lang.Object](http://download.oracle.com/javase/6/docs/api/java/lang/Object.html?is-external=true)

extended by **org.apache.hadoop.mapred.SkipBadRecords**

@InterfaceAudience.Public

@InterfaceStability.Stable

public class [**SkipBadRecords**](http://pleiades.ucsc.edu/doc/hadoop/2.5.2/api/src-html/org/apache/hadoop/mapred/SkipBadRecords.html#line.55)

extends [Object](http://download.oracle.com/javase/6/docs/api/java/lang/Object.html?is-external=true)

Utility class for skip bad records functionality. It contains various settings related to skipping of bad records.

Hadoop provides an optional mode of execution in which the bad records are detected and skipped in further attempts.

This feature can be used when map/reduce tasks crashes deterministically on certain input. This happens due to bugs in the map/reduce function. The usual course would be to fix these bugs. But sometimes this is not possible; perhaps the bug is in third party libraries for which the source code is not available. Due to this, the task never reaches to completion even with multiple attempts and complete data for that task is lost.

With this feature, only a small portion of data is lost surrounding the bad record, which may be acceptable for some user applications. see [setMapperMaxSkipRecords(Configuration, long)](http://pleiades.ucsc.edu/doc/hadoop/2.5.2/api/org/apache/hadoop/mapred/SkipBadRecords.html#setMapperMaxSkipRecords(org.apache.hadoop.conf.Configuration, long))

The skipping mode gets kicked off after certain no of failures see [setAttemptsToStartSkipping(Configuration, int)](http://pleiades.ucsc.edu/doc/hadoop/2.5.2/api/org/apache/hadoop/mapred/SkipBadRecords.html#setAttemptsToStartSkipping(org.apache.hadoop.conf.Configuration, int))

In the skipping mode, the map/reduce task maintains the record range which is getting processed at all times. Before giving the input to the map/reduce function, it sends this record range to the Task tracker. If task crashes, the Task tracker knows which one was the last reported range. On further attempts that range get skipped.

33) Identity mapper and reducer

Answer)

# [**Identity Mapper & Reducer**](http://bigdataconsultants.blogspot.com/2013/12/identity-mapper-reducer.html)

**What is the difference between 0 reducer and Identity reducer?**  
0 reducer means that reducer won't run. The output of the mapper will be the output of the mapreduce program. With identity reducer, you enforce that sorting/shuffling will happen but there wouldn't be any aggregation.   
So.. you will have the sorted data from teh different mapper, but they would not be aggregated. For example, in word count example,  
<http://stackoverflow.com/questions/19539683/when-is-an-identity-mapper-reducer-used>  
<http://stackoverflow.com/questions/10630447/hadoop-difference-between-0-reducer-and-identity-reducer>  
" An **identity mapper** is used can be used (among others!) if you would only want to **sort** your input. An **identity reducer** can be used for example to implement **embarrasingly parallel algorithms** where you just use the mappers to perform the parallel tasks but you want the output key value pairs to be sorted."  
  
For Zero Reducer  
  
File 1  
<Hello, 1>  
<Life, 1>  
  
File 2  
<Hello, 1>  
<World, 1>  
<Chick, 1>  
  
  
In case of Identity reducer, you will have one file output as follows (No aggregating is done). Only Sorting and shuffling happen when you implement the identiy reducer.  
<File 1>  
Hello  <1,1>  
<Life, 1>  
<World, 1>  
<Chick, 1>

# **WHAT IS A IDENTITYMAPPER AND IDENTITYREDUCER IN MAPREDUCE ?**

* org.apache.hadoop.mapred.lib.IdentityMapper Implements the identity function, mapping inputs directly to outputs. If MapReduce programmer do not set the Mapper Class using JobConf.setMapperClass then IdentityMapper.class is used as a default value.
* org.apache.hadoop.mapred.lib.IdentityReducer Performs no reduction, writing all input values directly to the output. If MapReduce programmer do not set the Reducer Class using JobConf.setReducerClass then IdentityReducer.class is used as a default value.

### What is Identity Mapper and Identity Reducer and where we use it?

**Identity Mapper**  
  
There is a default map function supplied by Hadoop Framework,which is known as **Identity Mapper**. If this mapper is used, nothing is specially executed and the data is transfer to reduce phase without any mapping work.  
  
**Purpose of Identity Mapper**  
  
  
An identity mapper can be used if you would only want to sort your input.

**Identity Reducer**

 There is a default reducer method, called **Identity Reducer**.It doesn’t do any thing other than sending the output of map function to its output**.**

**Purpose of Identity Reducer**

An identity reducer can be used for example to implement embarrassingly parallel algorithms where you just use the mappers to perform the parallel tasks but you want the output key value pairs to be sorted.

34) WRITE mapreduce program to find duplicate records

Answer)

# [MapReduce: Remove duplicate records from input file on HDFS](http://www.devinline.com/2015/12/mapreduce-remove-duplicate-records-from-file.html)

[Nikhil Ranjan](https://plus.google.com/100186727343917396064)     [December 12, 2015](http://www.devinline.com/2015/12/mapreduce-remove-duplicate-records-from-file.html)     [Hadoop](http://www.devinline.com/search/label/Hadoop), [MapReduce](http://www.devinline.com/search/label/MapReduce)     [1 comment](http://www.devinline.com/2015/12/mapreduce-remove-duplicate-records-from-file.html#comment-form)

In previous post we learned about [how to analyse time-temperature statistics and generate report with max/min temperature for each city(MultipleOutputs<Text, Text>)](http://www.devinline.com/2015/12/weather-report-poc-mapreduce-sample-program.html). In this post we will write sample MapReduce program to understand how to remove duplicate records from a file. [Download sample input file](https://drive.google.com/file/d/0B-ur4R5mlgGLMVBVLWNWd3RfcTg/view?usp=sharing).  
**Mapper class**:- In map method, we read input file line by line and make whole line as key of mapper output and NullWritable.get() as value which are written to context object.

con.write(row, NullWritable.get());

After grouping and mapper output will appear something like

|  |  |
| --- | --- |
| **key** | **value list** |
| <unique\_employee\_record\_1> | <NW, NW, NW> |
| <unique\_employee\_record\_2> | <NW, NW> |
| <unique\_employee\_record\_3> | <NW, NW,NW ,NW,NW> |
| <unique\_employee\_record\_4> | <NW, NW,NW> |

**Reducer class**:-  In reduce method each key **unique\_employee\_record** is written to context as key and NullWritable.get() as value.

con.write(<unique\_employee\_record\_1>, NullWritable.get());

#### Sample code of mapper,reducer and driver class

**Mapper class**

**import** **java.io.IOException**;

**import** **org.apache.hadoop.io.NullWritable**;

**import** **org.apache.hadoop.io.Text**;

**import** **org.apache.hadoop.mapreduce.Mapper**;

**public** **class** **RemoveDuplicateMapper** **extends**

Mapper<Object, Text, Text, NullWritable> {

**@Override**

**public** **void** **map**(Object key, Text row, Context con) {

**try** {

con.write(row, NullWritable.get());

} **catch** (IOException e) {

e.printStackTrace();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**Reducer class** :-

**import** **java.io.IOException**;

**import** **org.apache.hadoop.io.NullWritable**;

**import** **org.apache.hadoop.io.Text**;

**import** **org.apache.hadoop.mapreduce.Reducer**;

**public** **class** **RemoveDuplicateReducer** **extends**

Reducer<Text, NullWritable, Text, NullWritable> {

**@Override**

**public** **void** **reduce**(Text key, Iterable<NullWritable> Value, Context con) {

**try** {

con.write(key, NullWritable.get());

} **catch** (IOException e) {

e.printStackTrace();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**Driver class** :-

**import** **java.io.IOException**;

**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** **RemoveDuplicateRecordsDriver** {

**public** **static** **void** **main**(String[] str) {

Configuration conf = **new** Configuration();

**try** {

Job job = Job.getInstance(conf, "Duplicate removal");

job.setMapperClass(RemoveDuplicateMapper.class);

job.setReducerClass(RemoveDuplicateReducer.class);

job.setJarByClass(RemoveDuplicateRecordsDriver.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(NullWritable.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(NullWritable.class);

FileInputFormat.addInputPath(job,**new** Path(

"hdfs://localhost:54310/user/hduser1/employee\_records\_duplicates"));

FileOutputFormat.setOutputPath(job,**new** Path(

"hdfs://localhost:54310/user/hduser1/testfs/output\_employee1"));

System.exit(job.waitForCompletion(**true**) ? **1** : **0**);

} **catch** (IOException e) {

e.printStackTrace();

} **catch** (ClassNotFoundException e) {

e.printStackTrace();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

Execute driver class(hadoop services are running and input file location modified as per your inconvenience). It will create an output directory with file  part-r-00000(no duplicate records).  
To upload input file from local file system to HDFS

**hduser1@ubuntu:/usr/local/hadoop2.6.1/bin$** ./hadoop fs -put /home/zytham/workspaceJuno/MapreduceSampleProject/employee\_records\_duplicates /user/hduser1/

Check the size of both input file and output file so that it can be verified duplicate record has been removed.Input file size is 13 K and output file size is 7.9 K.

**hduser1@ubuntu:/usr/local/hadoop2.6.1/bin$** ./hadoop fs -ls -h /user/hduser1/employee\_records\_duplicates

-rw-r--r-- 1 hduser1 supergroup **13.1 K** 2015-12-12 22:50 /user/hduser1/employee\_records\_duplicates

**hduser1@ubuntu:/usr/local/hadoop2.6.1/bin$** ./hadoop fs -ls -h /user/hduser1/testfs/output\_employee1/part-r-00000

-rw-r--r-- 3 zytham supergroup **7.9 K** 2015-12-12 22:56 /user/hduser1/testfs/output\_employee1/part-r-00000

Lets modify this problem, **how to remove duplicates row based on employee\_id** as key(Just keep first record wit the given employee\_id).   
In **map method**, make employee\_id as key and whole row as value and write the same in context.

con.write(<emp\_id>, row);

After **grouping** mapper output looks like

|  |  |
| --- | --- |
| **key** | **value list** |
| <unique\_employee\_id\_1> | <emp\_record, emp\_record, emp\_record> |
| <unique\_employee\_id\_2> | <emp\_record, emp\_record> |
| <unique\_employee\_id\_3> | <emp\_record, emp\_record,emp\_record,emp\_record> |
| <unique\_employee\_id\_4> | <emp\_record, emp\_record,emp\_record> |

In **reduce method**, iterate over list for each input row and take first emp\_record and write into context.

**for** (TextWritable val : values) {

context.write(val, NullWritable.get()); break;

}

If you’ve read my [**beginners guide to Hadoop**](https://blog.matthewrathbone.com/2013/04/17/what-is-hadoop.html) you should remember that an important part of the Hadoop ecosystem is HDFS, Hadoop’s distributed file system. Like other file systems the format of the files you can store on HDFS is entirely up to you. For example, you can use HDFS to store cat memes in GIF format, text data in plain-text CSV format, or spreadsheets in XLS format. This is not specific to Hadoop, you can store these same files on your computer file system.

However unlike a regular file system, HDFS is best used in conjunction with a data processing toolchain like MapReduce or Spark. These processing systems typically (although not always) operate on some form of textual data like webpage content, server logs, or location data.

In this article I will talk about what file formats actually are, go through some common Hadoop file format features, and give a little advice on which format you should be using.

But before all that, a word of warning.

## **Hadoop Noob? Stop Here**

If you’re just getting started with Hadoop, HDFS, Hive and wondering what file format you should be using to begin with, let me give you some advice.

Just use tab delimited files for your prototyping (and first production jobs). They’re easy to debug (because you can read them), they are the default format of Hive, and they’re easy to create and reason about.

Once you have a production MapReduce or Spark job regularly generating data come back and pick something better.

Tab delimited data looks like this:

user\_1 Matthew Rathbone Dallas, TX

user\_2 Joe Bloggs London, England

Ok, time to get to the guts of this.

## **A Quick Overview of Storage Formats**

A storage format is just a way to define how information is stored in a file. This is usually indicated by the extension of the file (informally at least).

For example images have several common storage formats, PNG, JPG, and GIF are commonly used. All three of those formats can store the same image, but [**each has specific characteristics**](http://stackoverflow.com/questions/2336522/png-vs-gif-vs-jpeg-vs-svg-when-best-to-use). For example JPG files tend to be smaller, but store a compressed version of the image that is of lower quality.

When dealing with Hadoop’s filesystem not only do you have all of these traditional storage formats available to you (like you can store PNG and JPG images on HDFS if you like), but you also have some Hadoop-focused file formats to use for structured and unstructured data.

Some common storage formats for Hadoop include:

* Plain text storage (eg, CSV, TSV files)
* Sequence Files
* Avro
* Parquet

## **Why Storage Formats are Important**

A huge bottleneck for HDFS-enabled applications like MapReduce and Spark is the time it takes to find relevant data in a particular location and the time it takes to write the data back to another location. These issues are exacerbated with the difficulties managing large datasets, such as evolving schemas, or storage constraints. The various Hadoop file formats have evolved as a way to ease these issues across a number of use cases.

Choosing an appropriate file format can have some significant benefits:

1. Faster read times
2. Faster write times
3. Splittable files (so you don’t need to read the whole file, just a part of it)
4. Schema evolution support (allowing you to change the fields in a dataset)
5. Advanced compression support (compress the files with a compression codec without sacrificing these features)

Some file formats are designed for general use (like MapReduce or Spark), others are designed for more specific use cases (like powering a database), and some are designed with specific data characteristics in mind. So there really is quite a lot of choice.

## **Storage Formats and HDFS-enabled Apps**

MapReduce, Spark, and Hive are three primary ways that you will interact with files stored on Hadoop. Each of these frameworks comes bundled with libraries that enable you to read and process files stored in many different formats.

In MapReduce file format support is provided by the InputFormat and OutputFormat classes. Here is an example configuration for a simple MapReduce job that reads and writes to text files:

Job job = new Job(getConf());

...

job.setInputFormatClass(TextInputFormat.class);

job.setOutputFormatClass(TextOutputFormat.class);

You can implement your own InputFormat classes if you want to store data in your own custom format. For example at Foursquare we stored some data in [**MongoDB’s BSON**](http://bsonspec.org/) format wrapped in a custom [**Thrift model**](https://thrift.apache.org/), sometimes written to a sequence file, sometimes stored in BSON format. To do this we needed [**our own input format and record reader**](https://github.com/rathboma/mongo-hdfs-export/tree/master/src/main/scala/com/foursquare/hadoop/io), which are now open source. This allowed us to directly access data generated from a MongoDB database dump, saving a lot of needless additional ETL.

Both Hive and Spark have similar mechanisms for reading and writing custom file formats which wrap the Hadoop InputFormat described above, so the InputFormat is truly the gateway to file formats on Hadoop.

### WORKFLOW CHANGES REQUIRED

While you can easily swap the storage formats used in Hadoop it is not usually as simple as switching a couple of lines of code. Different storage formats are set up to provide different types of data to their consumers.

For example the TextInputFormat gives you a string that represents a single line of the file it reads, whereas the AVRO file format is designed to provide you with structured data that can be deserialized to a java object.

To illustrate, take a look at this example of a MapReduce map class from the AVRO documentation ( I simplified it a little ). Notice how the Map task transparently receives an instance of the User class?

public static class ColorCountMapper extends Mapper<AvroKey<User>, NullWritable, Text, IntWritable> {

@Override

public void map(AvroKey<User> key, NullWritable value, Context context) {

CharSequence color = key.datum().getFavoriteColor();

if (color == null) {

color = "none";

}

context.write(new Text(color.toString()), new IntWritable(1));

}

}

Contrast that with how you’d accomplish it with a vanilla text input workflow:

public static class ColorCountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {

@Override

public void map(LongWritable key, Text value, Context context) {

String[] user = value.getString().split("\\n");

String color = "none";

if(user.length >= 2) {

color = user[1];

}

context.write(new Text(color), new IntWritable(1));

}

}

For advanced InputFormat usage and if you want to learn more about how to standardize MapReduce inputs check out the [**RecordReader**](https://hadoop.apache.org/docs/r2.4.1/api/org/apache/hadoop/mapreduce/RecordReader.html), which you can use in conjunction with a InputFormat to standardize inputs to Map and Reduce jobs.

## **Common Storage Formats**

There are many storage formats available but I’ll just go through the major ones and talk about their various pros and cons.

### TEXT FILES (E.G. CSV, TSV)

Simple text-based files are common in the non-Hadoop world, and they’re super common in the Hadoop world too. Data is laid out in lines, with each line being a record. Lines are terminated by a newline character \n in the typical unix fashion.

Text-files are inherently splittable (just split on \n characters!), but if you want to compress them you’ll have to use a file-level compression codec that support splitting, such as BZIP2

Because these files are just text files you can encode anything you like in a line of the file. One common example is to make each line a JSON document to add some structure. While this can waste space with needless column headers, it is a simple way to start using structured data in HDFS.

I’ve used this approach many times and it’s a great stepping stone to more structured data formats.

### SEQUENCE FILES

[**Website**](http://wiki.apache.org/hadoop/SequenceFile)

Sequence files were originally designed for MapReduce, so the integration is smooth. They encode a key and a value for each record and nothing more. Records are stored in a binary format that is smaller than a text-based format would be. Like text files, the format does not encode the structure of the keys and values, so if you make schema migrations they must be additive.

Sequence files by default use Hadoop’s Writable interface in order to figure out how to serialize and deserialize classes to the file.

Typically if you need to store complex data in a sequence file you do so in the value part while encoding the id in the key. The problem with this is that if you add or change fields in your Writable class it will not be backwards compatible with the data stored in the sequence file.

One benefit of sequence files is that they support block-level compression, so you can compress the contents of the file while also maintaining the ability to split the file into segments for multiple map tasks.

Sequence files are well supported across Hadoop and many other HDFS enabled projects, and I think represent the easiest next step away from text files.

### AVRO FILES

[**Website**](https://avro.apache.org/)

Avro is an opinionated format which understands that data stored in HDFS is usually not a simple key/value combo like Int/String. The format encodes the schema of its contents directly in the file which allows you to store complex objects natively.

Honestly, Avro is not really a file format, it’s a file format plus a serialization and deserialization framework. With regular old sequence files you **can** store complex objects but you have to manage the process. Avro handles this complexity whilst providing other tools to help manage data over time.

Avro is a well thought out format which defines file data schemas in JSON (for interoperability), allows for schema evolutions (remove a column, add a column), and multiple serialization/deserialization use cases. It also supports block-level compression. For most Hadoop-based use cases Avro is a really good choice.

### COLUMNAR FILE FORMATS (PARQUET, RCFILE)

[**Parquet Website**](https://parquet.apache.org/)  
[**RCFile Website**](https://en.wikipedia.org/wiki/RCFile)

The latest hotness in file formats for Hadoop is columnar file storage. Basically this means that instead of just storing rows of data adjacent to one another you also store column values adjacent to each other. So datasets are partitioned both horizontally and vertically. This is particularly useful if your data processing framework just needs access to a subset of data that is stored on disk as it can access all values of a single column very quickly without reading whole records.

One huge benefit of columnar oriented file formats is that data in the same column tends to be compressed together which can yield some massive storage optimizations (as data in the same column tends to be similar).

If you’re chopping and cutting up datasets regularly then these formats can be very beneficial to the speed of your application, but frankly if you have an application that usually needs entire rows of data then the columnar formats may actually be a detriment to performance due to the increased network activity required.

Overall these formats can drastically optimize workloads, especially for Hive and Spark which tend to just read segments of records rather than the whole thing (which is more common in MapReduce).

Of the two file formats I mention, Parquet seems to have the most community support and is the format I would use.

## **Bonus: Compression Codecs**

I’ll touch on this in a later post but there are two ways you can compress data in Hadoop.

1. File-Level Compression
2. Block-Level Compression

File-level compression means you compress entire files regardless of the file format, the same way you would compress a file in Linux. Some of these formats are splittable (e.g. bzip2, or LZO if indexed).

So you’d end up with a file called user-data.csv.gzip for example.

Block-level compression is internal to the file format, so individual blocks of data within the file are compressed. This means that the file remains splittable even if you use a non-splittable compression codec like Snappy. However, this is only an option if the specific file format supports it.

So you’d still have a file called user-data.sequence, which would include a header noting the compression codec needed to read the remaining file contents.

If you’re seriously thinking about file formats then you should use compression. Snappy is a great balance of speed and compression ratio, and I’ve used it with great success in the past.

As I mentioned earlier, to start with just keep things simple – use text files with GZIP compression (GZIP is natively supported by Hadoop out of the box).

## **How do the most common Hadoop file formats stack up?**

### 1. Formats to Avoid

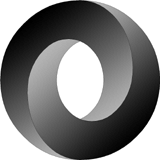
Any format that is not splittable should generally be avoided. The use of XML File and JSON File formats is a common mistake. Each of these formats contain a single document per file with an opening tag at the beginning and a closing tag at the end. Note that files containing JSON records are ok and will be discussed below.

### 2. Text/CSV Files

CSV files are still quite common and often used for exchanging data between Hadoop and external systems. They are readable and ubiquitously parsable. They come in handy when doing a dump from a database or bulk loading data from Hadoop into an analytic database. However, CSV files do not support block compression, thus compressing a CSV file in Hadoop often comes at a significant read performance cost.

When working with Text/CSV files in Hadoop, never include header or footer lines. Each line of the file should contain a record. This, of course, means that there is no metadata stored with the CSV file. You must know how the file was written in order to make use of it. Also, since the file structure is dependent on field order, new fields can only be appended at the end of records while existing fields can never be deleted. As such, CSV files have limited support for schema evolution.

### 3. JSON Records



JSON records are different from JSON Files in that each line is its own JSON datum -- making the files splittable. Unlike CSV files, JSON stores metadata with the data, fully enabling schema evolution. However, like CSV files, JSON files do not support block compression. Additionally, JSON support was a relative late comer to the Hadoop toolset and many of the native serdes contain significant bugs. Fortunately, third party serdes are frequently available and often solve these challenges. You may have to do a little experimentation and research for your use cases.

### 4. Avro Files

Avro files are quickly becoming the best multi-purpose storage format within Hadoop. Avro files store metadata with the data but also allow specification of an independent schema for reading the file. This makes Avro the epitome of schema evolution support since you can rename, add, delete and change the data types of fields by defining new independent schema. Additionally, Avro files are splittable, support block compression and enjoy broad, relatively mature, tool support within the Hadoop ecosystem.

### 5. Sequence Files

Sequence files store data in a binary format with a similar structure to CSV. Like CSV, sequence files do not store metadata with the data so the only schema evolution option is appending new fields. However, unlike CSV, sequence files do support block compression. Due to the complexity of reading sequence files, they are often only used for “in flight” data such as intermediate data storage used within a sequence of MapReduce jobs.

### 6. RC Files

RC Files or Record Columnar Files were the first columnar file format adopted in Hadoop. Like columnar databases, the RC file enjoys significant compression and query performance benefits. However, the current serdes for RC files in Hive and other tools do not support schema evolution. In order to add a column to your data you must rewrite every pre-existing RC file. Also, although RC files are good for query, writing an RC file requires more memory and computation than non-columnar file formats. They are generally slower to write.

### 7. ORC Files

ORC Files or Optimized RC Files were invented to optimize performance in Hive and are primarily backed by HortonWorks. ORC files enjoy the same benefits and limitations as RC files just done better for Hadoop. This means ORC files compress better than RC files, enabling faster queries. However, they still don’t support schema evolution. Some benchmarks indicate that ORC files compress to be the smallest of all file formats in Hadoop. It is worthwhile to note that, at the time of this writing, Cloudera Impala does not support ORC files.

### 8. Parquet Files

Parquet Files are yet another columnar file format that originated from Hadoop creator Doug Cutting’s Trevni project. Like RC and ORC, Parquet enjoys compression and query performance benefits, and is generally slower to write than non-columnar file formats. However, unlike RC and ORC files Parquet serdes support limited schema evolution. In Parquet, new columns can be added at the end of the structure. At present, Hive and Impala are able to query newly added columns, but other tools in the ecosystem such as Hadoop Pig may face challenges. Parquet is supported by Cloudera and optimized for Cloudera Impala. Native Parquet support is rapidly being added for the rest of the Hadoop ecosystem.

One note on Parquet file support with Hive... It is very important that Parquet column names are lowercase. If your Parquet file contains mixed case column names, Hive will not be able to read the column and will return queries on the column with null values *and not log any errors*. Unlike Hive, Impala handles mixed case column names. A truly perplexing problem when you encounter it!

### How to choose a file format?

As discussed, each file format is optimized by purpose. Your choice of format is driven by your use case and environment. Here are the key factors to consider:

* **Hadoop Distribution**- Cloudera and Hortonworks support/favor different formats
* **Schema Evolution**- Will the structure of your data evolve?  In what way?
* **Processing Requirements** - Will you be crunching the data and with what tools?
* **Read/Query Requirements**- Will you be using SQL on Hadoop?  Which engine?
* **Extract Requirements**- Will you be extracting the data from Hadoop for import into an external database engine or other platform?
* **Storage Requirements**- Is data volume a significant factor?  Will you get significantly more bang for your storage buck through compression?

So, with all the options and considerations are there any obvious choices?  If you are storing intermediate data between MapReduce jobs, then Sequence files are preferred. If query performance against the data is most important, ORC (HortonWorks/Hive) or Parquet (Cloudera/Impala) are optimal --- but these files will take longer to write. (We’ve also seen order of magnitude query performance improvements when using Parquet with Spark SQL.) Avro is great if your schema is going to change over time, but query performance will be slower than ORC or Parquet. CSV files are excellent if you are going to extract data from Hadoop to bulk load into a database.

Because you will likely have multiple use cases for your Hadoop data, you will use multiple file formats. It is quite common to store the same (or very similar) data in multiple file formats to support varied processing, query and extract requirements. Embrace diversity!

# **OPULAR FILE FORMATS FOR HADOOP／HDFS**

1 Votes

Hadoop is an ecosystem including many tools to store and process big data. HDFS is one that is used for storage, and it’s a special file system different than the one used on our desktop machines. We are not going to explain why it’s special, instead, we will introduce several special file formats supported by HDFS.

**Text/CSV Files**

CSV file is the most commonly used data file format. It’s the most readable and also ubiquitously easy to parse. It’s the choice of format to use when export data from an RDBMS table. However, human readable does not mean it’s machine readable. It has three major drawbacks when used for HDFS. First of all, all lines in a CSV file is a record, therefore, we should not include any headers or footers. In other word, CSV file cannot be stored in HDFS with any meta data. Second of all, CSV file has very limited support for schema evolution. Because the fields for each record are ordered, we are not able to change the orders. We can only append new fields to the end of each line. Last, CSV file does not support block compression which many other file formats support. The whole file has to be compressed and decompressed for reading, adding a significant read performance cost to the files.

**JSON Files**

JSON is in text format that stores meta data with the data, so it fully supports schema evolution. You can easily add or remove attributes for each datum. However, because it’s text file, it doesn’t support block compression.

**Avro Files**

Avro File is serialized data in binary format. It uses JSON to define data types, therefore it is row based. It is the most popular storage format for Hadoop. Avro stores meta data with the data, and it also allows specification of independent schema used for reading the files. Therefore, you can easily add, delete, update data fields by just creating a new independent schema. Also, Avro files are splittable, support block compression and enjoys a wide arrange of tool support within Hadoop ecosystem.

**Sequence Files**

Sequence files are binary files with a CSV-like structure. It does not store meta data, nor does it support schema evolution, but it does support block compression. Due to its unreadability, they are mostly used for intermediate data storage within a sequence of MapReduce jobs.

**ORC Files**

RC files or Record Columnar files are columnar file format. It’s great for compression and best for query performance, with the sacrifice of cost of more memory and poor write performance. ORC are optimized RC files that works better with Hive. It compresses better, but still does not support schema evolution. It is worthwhile to note that OCR is a format primarily backed by Hortonworks, and it’s not supported by Cloudera Impala.

**Parquet Files**

Paquet file format is also a columnar format. Just like ORC file, it’s great for compression with great query performance. It’s especially efficient when querying data from specific columns. Parquet format is computationally intensive on the write side, but it reduces a lot of I/O cost to make great read performance. It enjoys more freedom than ORC file in schema evolution, that it can add new columns to the end of the structure. It is also backed by Cloudera and optimized with Impala.

Since Avro and Parquet have so much in common, let’s review a little bit more of both. When choosing a file format to use with HDFS, we need to consider read performance and write performance. Because the nature of HDFS is to store data that is write once, read multiple times, we want to emphasize on the read performance. The fundamental difference in terms of how to use either format is this: Avro is a Row based format. If you want to retrieve the data as a whole, you can use Avro. Parquet is a Column based format. If your data consists of lot of columns but you are interested in a subset of columns, you can use Parquet.