Map Reduce program for weather Data set

MapReduce Program: To find average temperature for each year in NCDC data set.

Big data is a framework for storage and processing of data ( structured/unstructured ). Please check out the program below which draw out results out of semi-structured data from a weather sensor. Its a MapReduce program written in java.

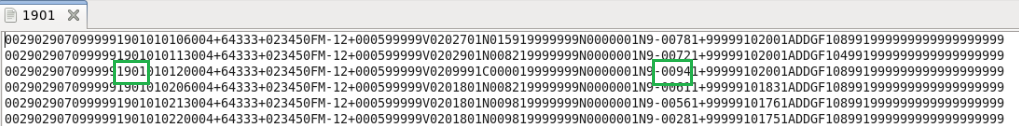
The aim of the program is to find the average temperature in each year of NCDC data.

This program takes a data input of multiple files where each file contains weather data of a particular year. This weather data is shared by NCDC (National Climatic Data Center ) and is collected by weather sensors at many locations across the globe. NCDC input data can be downloaded from

**https://github.com/tomwhite/hadoop-book/tree/master/input/ncdc/all.**

There is a data file for each year. Each data file contains among other things, the year and the temperature information( which is relevant for this program ).

Below is the snapshot of the data with year and temperature field highlighted in green box. This is the snapshot of data taken from year 1901 file:



So, in a MapReduce program there are 2 most important phases - Map Phase and Reduce Phase.

You need to have an understanding of MapReduce concepts so as to understand the intricacies of MapReduce programming. It is one the major component of Hadoop along with HDFS. I will try to include more posts in coming weeks around fundamentals of MapReduce.

Continuing with our current program:

• For writing any MapReduce program, firstly, you need to figure out the data flow, like in this example am taking just the year and temperature information in the map phase and passing it on to the reduce phase. So Map phase in my example is essentially a data preparation phase. Reduce phase on the other hand is more of a data aggregation one.

• Secondly, decide on the types for the key/value pairs—MapReduce program uses lists and (key/value) pairs as its main data primitives. So you need to decide the types for key/value pairs—K1, V1, K2, V2, K3, and V3 for the input, intermediate, and output key/value pairs. In this example, am taking LongWritable and Text as (K1,V1) for input and Text and IntWritable as both for (K2,V2) and (K3,V3)

Map Phase: I will be pulling out the year and temperature data from the log data that is there in the file, as shown in the above snapshot.

Reduce Phase: The data that is generated by the mapper(s) is fed to the reducer, which is another java program. This program takes all the values associated with a particular key and find the average temperature for that key. So, a key in our case is the year and value is a set of IntWritable objects which represent all the captured temperature information for that year.

I will be writing a java class, each for a Map and Reduce phase and one driver class to create a job with configuration information.

So, in this particular example I will be writing 3 java classes:

• AverageMapper.java

• AverageReducer.java

• AverageDriver.java

Let me share the code of all the 3 classes, along with the explanation of working of each class:

**AverageMapper.java**

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

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

import java.io.IOException;

public class AverageMapper extends Mapper <LongWritable, Text, Text, IntWritable>

{

public static final int MISSING = 9999;

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

{

String line = value.toString();

String year = line.substring(15,19);

int temperature;

if (line.charAt(87)=='+')

temperature = Integer.parseInt(line.substring(88, 92));

else

temperature = Integer.parseInt(line.substring(87, 92));

String quality = line.substring(92, 93);

if(temperature != MISSING && quality.matches("[01459]"))

context.write(new Text(year),new IntWritable(temperature));

}

}

Let us get into the details of our AverageMapper class. I need to extend generic class Mapper with four formal data types: input key, input value, output key, output value. The key for the Map phase is the offset of the beginning of the line from the beginning of the file, but as we have no need for it, we can ignore it. The input value would be temperature and output key would be year and output value will be temperature, an integer. The data is fed to the map function one line or record at a time. The map() function converts it into the string and read the year and temperature part from the applicable index value. Also, map() function creates a Context object which is the output object from map(). It contains year value as Text and temperature value as IntWritable.

**AverageReducer.java**

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

import java.io.IOException;

public class AverageReducer extends Reducer <Text, IntWritable,Text, IntWritable >

{

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

{

int max\_temp = 0;

int count = 0;

for (IntWritable value : values)

{

max\_temp += value.get();

count+=1;

}

context.write(key, new IntWritable(max\_temp/count));

}

}

Now coming to Reduce Class. Again, four formal data types: input key, input value, output key, output value is specified for this class. The input type and value of reduce function should match output key and value of the map function: Text and IntWritable objects. The reduce() function iterates through all the values and find the sum and count of the values, and finally the average temperature value from that.

**AverageDriver.java**

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

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

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

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

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

public class AverageDriver

{

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

{

if (args.length != 2)

{

System.err.println("Please Enter the input and output parameters");

System.exit(-1);

}

Job job = new Job();

job.setJarByClass(AverageDriver.class);

job.setJobName("Max temperature");

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

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

job.setMapperClass(AverageMapper.class);

job.setReducerClass(AverageReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

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

}

}

A Job object forms the specification of the job and gives you control over how the job will be run. Hadoop has a special feature of data locality, wherein the code for the program is send to the data instead of other way around. So, Hadoop distributes the jar file of the program across the cluster. we pass the name of the class in setJarByClass() method which hadoop can use to locate the jar file containing this class. We need to specify input and output paths. Input path can specify the file or directory which will be used as an input to the program and output path is a directory which will be created by Reducer. If the directory already exists it leads to an error. Then we specify the map and reduce types to use via setMapperClass() and setReducerClass(). Next we set the output types for the map and reduce functions. waitForCompletion() method submits the job and waits for it to finish. It return 0 or 1, indicating success or failure of the job.