**PROJECT 1: REPORT**

**[Maaz Syed (0003544185) & Venkat Sambandhan (0003544188)]**

**ABSTRACT**

Our implementation of the project is as follows:

* The Map function is given multiple lines at a time.
* The Map function loops through the 100 values, computes the partial values (partialSum, partialSumOfSquare, partialMin, partialMax and partialCount) and sends out 1 key value pair (i.e. 1 key value pair for every 100 values)
* The format of the key value pair is as shown below (**partial values are separated by tabs**)

**<”mapperKey”, (partialSum partialSumOfSquares partialMin partialMax partialCount)>**

Example: **<mapperKey, (47.97 30.78 0.02 0.96 100)>**

* The Reduce function iterates through the key value pairs it receives and computes the full and complete values from the partial values.
* The Reduce function then outputs the min, max, mean and standard deviation.

**NOTE**

* We have [customized the **FileInputFormat** and the **RecordReader** to send multiple lines](http://stackoverflow.com/questions/21866728/how-to-customize-fileinputformat-to-read-multiple-lines-of-a-file-in-java-when-u) at a time to the Map function.
* The number of lines that are sent to the Map function have **not been hardcoded** and can be changed through command line arguments. However, the default number of lines is set to a **100** if the 4th argument is not provided in the execution command.

Command example: **hadoop jar statistics.jar statistics input output 50**

* + - 1. **Transformation of data during the computations, i.e. data type of key, value**

**MAPPER**

**Input:**

The Record Reader sends 100 lines (by default) at a time in this format: **<offset, value>**

Example: **<0, 0.48> ; <1, 0.32> ; <2, 0.54> ; . . . . .**

**Data type of offset:** Long Writable

The offset is assigned by the record reader and it is of type Long Writable.

Hadoop needs to be able to serialize data in and out of Java types via DataInput and Data Output Objects (IO Streams usually). Specifically LongWritable is a Writable class that wraps a java long data type.

**Data type of value:** Text

This data is taken from the input file and is of type text. Here the input file is stored in text format.

Text is a class which stores text using standard UTF8 encoding. It provides methods to serialize, de-serialize and compare texts at byte level.

**Output:**

We have configured the mapper to read **100** lines (by default) at a time. In each read, the lines are processed and the 5 partial values (partialSum, partialSumOfSquares, partialMin, partialMax and partialCount) are computed/re-computed. After the mapper is done reading all the lines, it sends the partially computed values in a text format (separated by tabs). The mapper produces (key, value) pairs as the output and is given as input to the reducer.

Example: **<mapperKey, (47.97 30.78 0.02 0.96 100)>**

**Data type of mapperKey:** Text

Here the key of the mapper is set by the user to “mapperkey” and is of type text. We focus on getting all the key value pairs to a single reducer as all the computed partial values are sent as a string object.

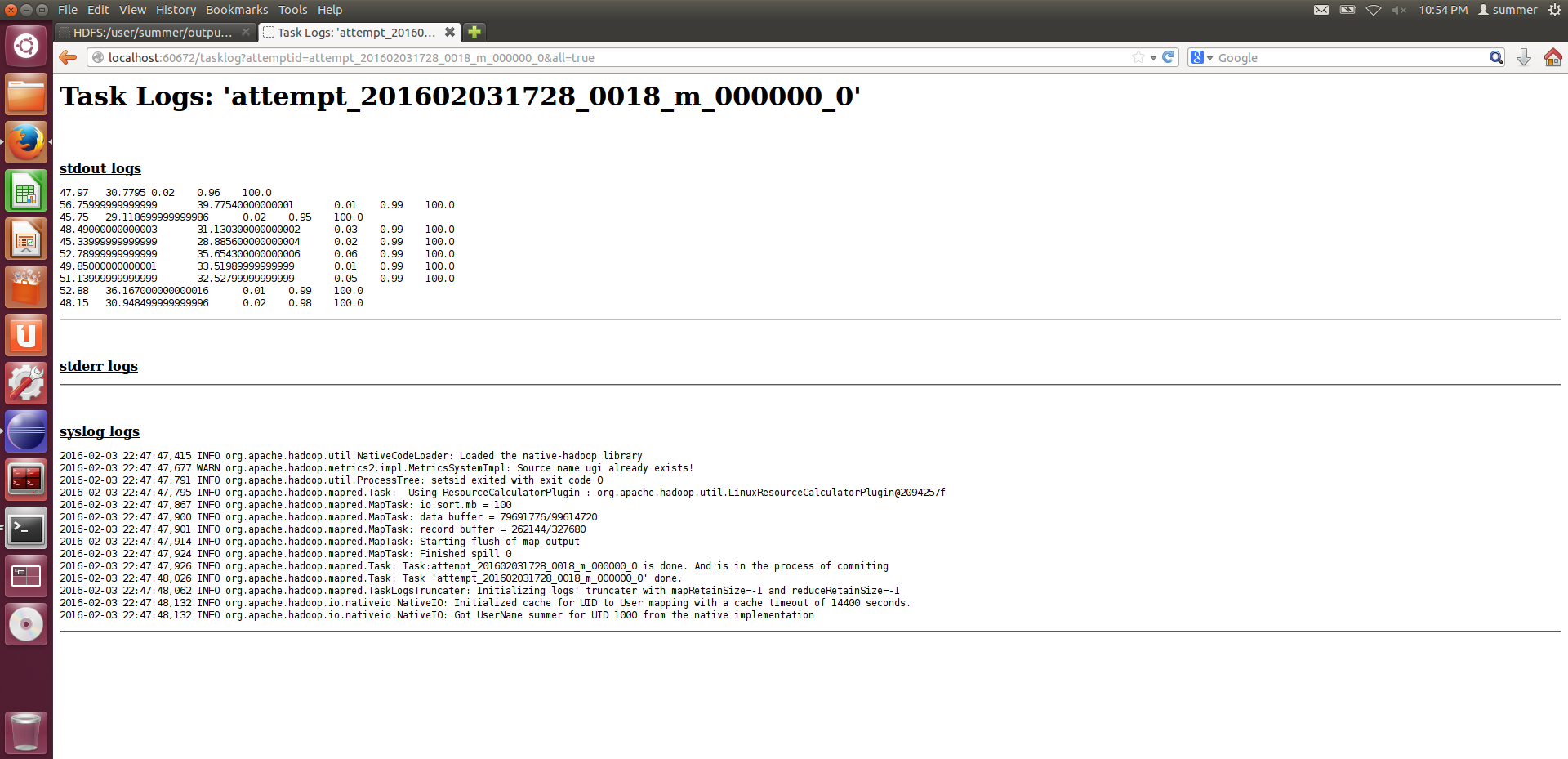
**Data type of Value:** Text

A set of **100** (default) numbers are taken and 5 partial values (partialSum, partialSumOfSquares, partialMin, partialMax and partialCount) are computed and sent as a string object

(MapperKey, (47.97, 30.78, 0.02, 0.96, 100))

Key Value

**MAPPER LOG**



**REDUCER**

**Input:**

The output of the mapper is given as input to the reducer. It contains several <key, value> pairs and the ones with same keys are aggregated. Since all the <key, value> pairs have same key in our implementation, everything is aggregated together and computed to produce the finalized sum, min, max and count.

**Data Type of Key:** Text

This is the output sent from the mapper. Here we have one unique key of type text used for all the values.

**Value:** Text

The values are of type text and contains a list of 5 partial values stored in a text iterator. We iterate through these values and tokenize them to extract the partial values

Example: **<mapperKey, [(47.97, 30.78, 0.02, 0.96, 100), (56.76, 39.76, 0.01, 0.99, 100), …. ,(48.15, 30.95, 0.02, 0.98, 100))] >**

**Output:**

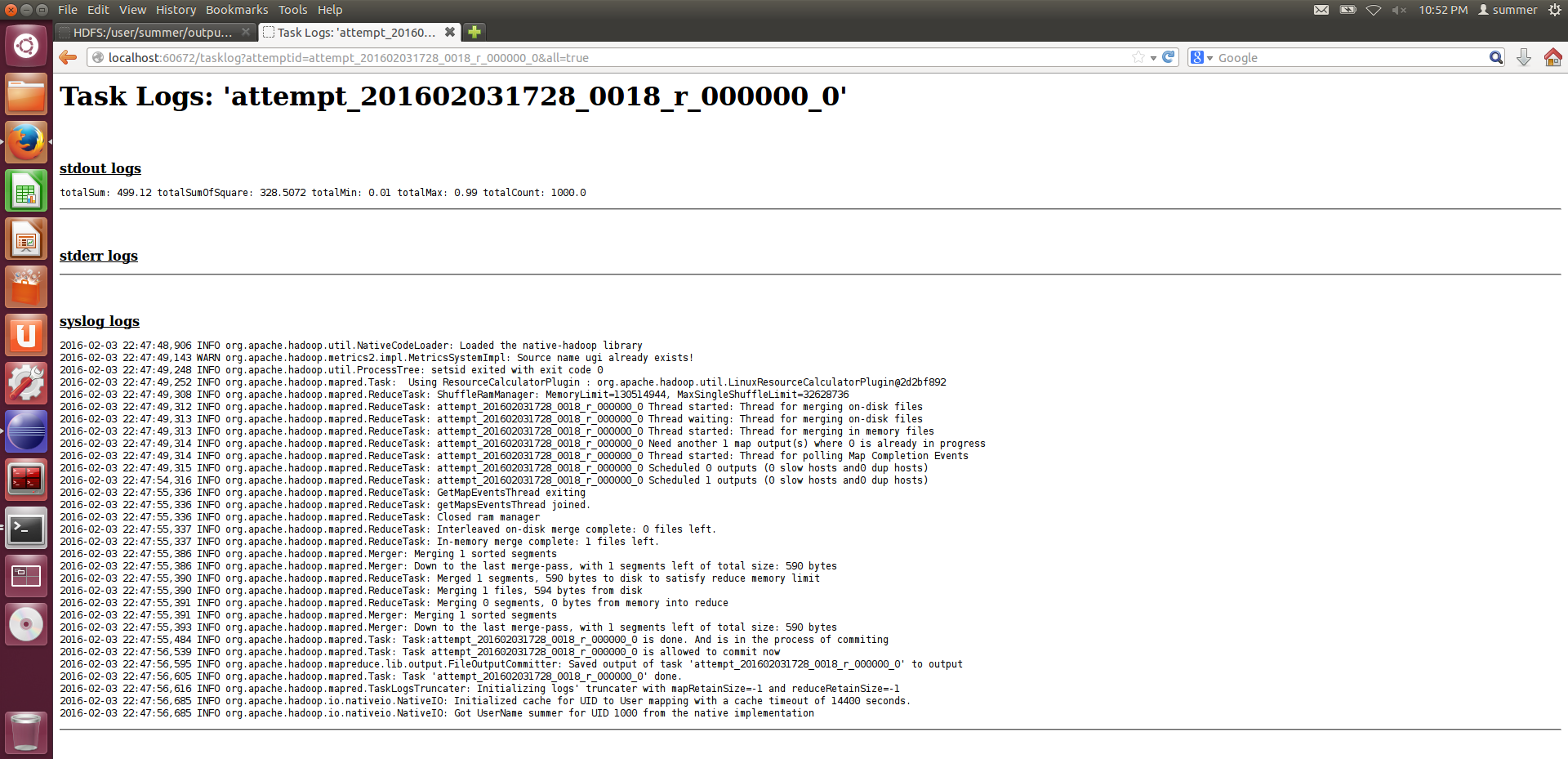
**Data Type of Key:** Text

**Value:** DoubleWritable

It contains the final result computed from the overall partial parameters calculated at the mapper. So the final parameters include: min, max, average and standard deviation (using the sum\_of\_squares).

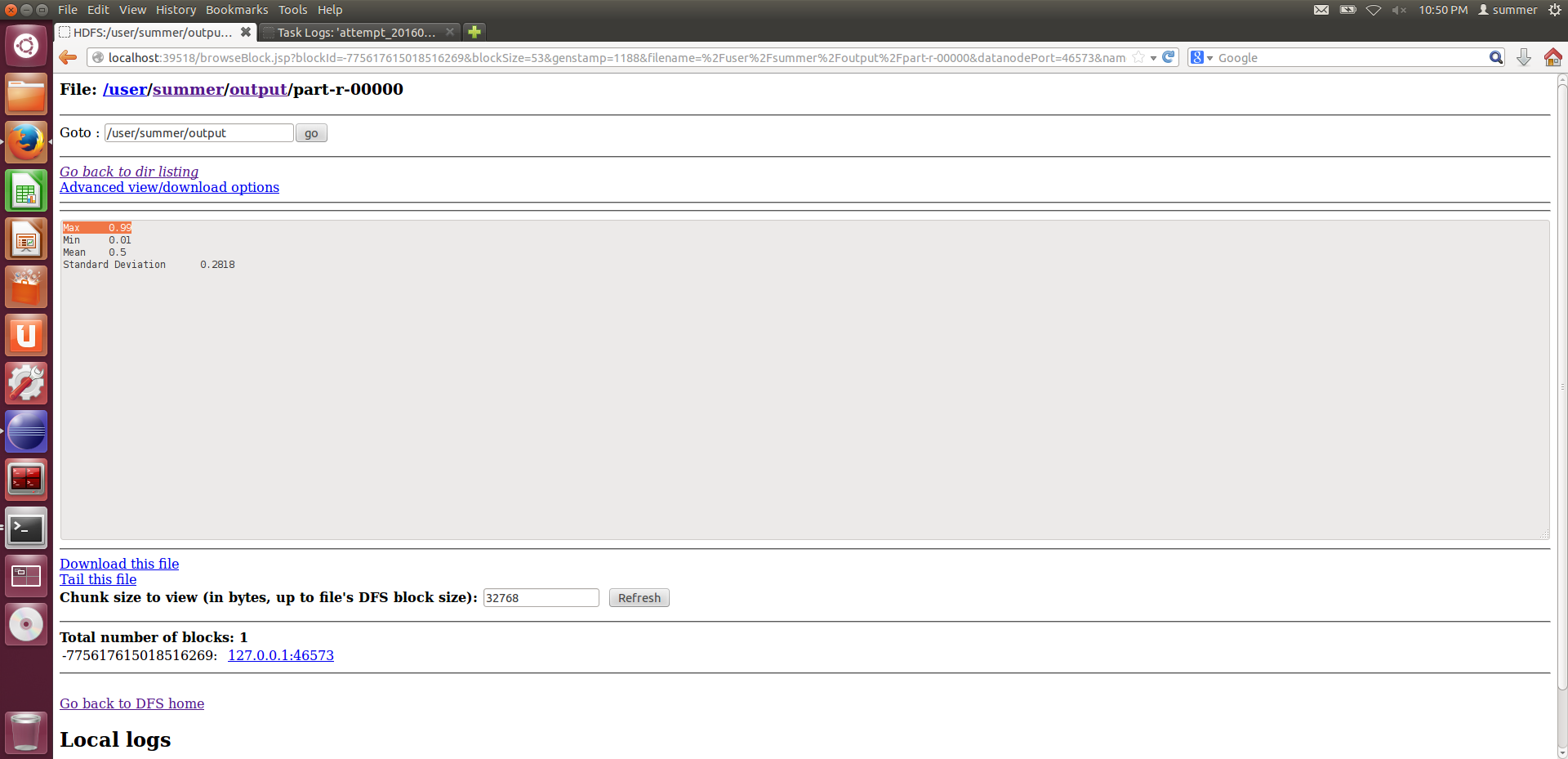
Hadoop needs to be able to serialize data in and out of Java types via DataInput and Data Output Objects (IO Streams usually). The Writable classes do this by implementing two methods: write(DataOuput) and readFields(DataInput). Specifically DoubleWritable is a Writable class that wraps a java double.

**REDUCER LOG**



**Continued…**

**REDUCER OUTPUT**



* + - 1. **The data structure used to transfer between Map and Reduce phases**

Here we use an iterator of type text: **Iterable<Text>**

For every 100 lines that the mapper receives and processes, it produces a single key value **text** pairs (where the value contains partial computed values separated by tabs) of the following format:

**<”mapperKey”, partialSum partialSumOfSquares partialMin partialMax partialCount>**

The values are stored as a list. Example: **[“mapperKey”, ((partialSum1), (partialSumOfSquares1), (partialMin1), (partialMax1), (partialCount1)), ((partialSum2), (partialSumOfSquares2), (partialMin2), (partialMax2), (partialCount2)), ((partialSum3), (partialSumOfSquares3), (partialMin3), (partialMax3), (partialCount3))…]**

At the reducer, we iterate through the list of values. In our implementation, the value is a **string object**.

Then tokenize it to access the individual elements of that text for computation of the final values.

**Continued…**

* + - 1. **How the data flow happens through disk and memory during the computation**

The below block diagram shows the map-reduce flow of the statistics program.

RESULTS

REDUCE

INPUT

MAP

KEY VALUE KEY VALUE

Min = 0.01

0.48 (mapperKey, (47.97, 30.78, 0.02, 0.96, 100)) mapperKey, **[**(47.97, 30.78, 0.02, 0.96, 100),

0.32 (mapperKey, (56.76, 39.76, 0.01, 0.99, 100)) (56.76, 39.76, 0.01, 0.99, 100),

0.54 . .

Max =0.99

0.04 . .

0.66 . .

. (mapperKey, 48.15, 30.95, 0.02, 0.98, 100)) (48.15, 30.95, 0.02, 0.98, 100))]

Avg = 0.5

.

.

SD = 0.2818

.

0.32

The required input text file is taken in the specified format from the Hadoop Distributed File System. The input file is then split and fed to the mapper with an offset as key. The map function is executed and its output (<key, value> pairs) is stored in the local file system. (This is a temporary directory which can set in configuration). This is followed by the reduce function.

Hadoop needs to be able to serialize data in and out of Java types via DataInput and Data Output Objects (IO Streams usually). The Writable classes do this by implementing two methods: write(DataOuput) and readFields(DataInput).

The mapper and reducer run as separate processes, so communication between them happens through files in Hadoop. Hadoop Daemons are Java processes, so they use a custom RPC (remote procedure call) communication where the Java Objects pass through the network in serialized object form.

**REFERENCES**

1. <http://stackoverflow.com/questions/11002000/nlineinputformat-has-no-effect>
2. <http://stackoverflow.com/questions/26821652/mapreduce-how-to-get-mapper-to-process-multiple-lines>
3. <https://gist.github.com/airawat/6639753>
4. <https://gist.github.com/airawat/6647007>https://ssl.gstatic.com/ui/v1/icons/mail/images/cleardot.gif