

UNIVERSITÀ DEGLI STUDI DEL SANNIO

DIPARTIMENTO DI INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Data Science

Homework 3

Prof.:
Antonio Pecchia

Studenti:
Stocchetti Federico, matr. 399000543
Fioretti Gabriele, matr. 399000522
Razzano Federica, matr. 399000542

Indice

1	Introduzione	2
2	Parte 1 - MapReduce	2
2.1	Operazioni preliminari	2
2.2	Esercizio 1.1	2
2.2.1	RecordMapper.java	2
2.2.2	RecordReducer.java	3
2.2.3	ApplicationDriver.java	4
2.2.4	Esecuzione	5
2.3	Esercizio 1.2	8
2.3.1	RecordMapper.java	8
2.3.2	RecordReducer.java	9
2.3.3	Esecuzione	9
2.4	Esercizio 1.3	10
2.4.1	RecordMapper.java	10
2.4.2	RecordReducer.java	11
2.4.3	Esecuzione	12
3	Parte 2 - Spark	12
3.1	Implementazione della soluzione	13
3.1.1	SocketSpout	13
3.1.2	SamplingBolt	14
3.1.3	AlertBolt	16
3.1.4	AlertLogBolt	17
3.1.5	MyTopology	19
3.2	Esecuzione	20
3.2.1	Esecuzione con LocalCluster	20
3.2.2	Esecuzione con SparkSubmitter	23

1 Introduzione

In questo report è descritto lo svolgimento dei compiti assegnati nell'**Homework 3** relativi alla realizzazione di diversi applicativi adottando il pattern **MapReduce** e ad un applicativo per la piattaforma **Storm**. Questo report è suddiviso in base alle 2 parti dell'homework in oggetto.

2 Parte 1 - MapReduce

La prima parte dell'homework richiede di sviluppare degli applicativi adottando il pattern **MapReduce** in Java per svolgere determinate operazioni sul dataset di partenza *websites.txt*. La scelta di quali esercizi svolgere è ricaduta sull'insieme degli esercizi **1**, ovvero la scrittura di applicativi MapReduce per svolgere le seguenti operazioni:

- Calcolare la somma di visite mensili dei siti per ogni mese
- Calcolare il numero di siti distinti presenti nel file
- Calcolare il minimo della terza colonna, ovvero il minimo del numero di visite sui siti

2.1 Operazioni preliminari

Prima di poter eseguire gli esercizi di MapReduce, è stato necessario inizializzare il DFS effettuando le seguenti operazioni:

- Formattazione del namenode hadoop
- Avvio del DFS
- Creazione della cartella in cui inserire l'input degli esercizi successivamente descritti

Per effettuare queste operazioni, sono stati eseguiti i seguenti comandi:

```
cd /home/studente/hadoop-3.2.1
hdfs namenode {format --> Per formattare namenode hadoop
sbin/start-dfs.sh --> Avvio del DFS (per stopparlo sbin/stop-dfs.sh)
hdfs dfs -mkdir /user
hdfs dfs -mkdir /user/studente
hdfs dfs -mkdir /user/studente/input
```

2.2 Esercizio 1.1

Il primo esercizio sul pattern MapReduce richiede di sviluppare un applicativo per poter calcolare la somma di visite mensili dei siti per ogni mese.

Per tale esercizio sono state scritte le seguenti classi Java:

2.2.1 RecordMapper.java

Questa classe rappresenta il **Mapper** all'interno del pattern MapReduce. Questo mapper legge le righe del file in input e per ogni riga, mediante la tokenizzazione, ricava il mese, il sito e il numero di visite effettuate in un determinato mese su un certo sito. Successivamente propaga verso il Reducer una coppia {chiave, valore} in cui la chiave è il nome del mese e il valore è il numero di visite.

Di seguito il codice del file RecordMapper.java:

```
package count;
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 RecordMapper extends Mapper<LongWritable, Text, Text, LongWritable> {

    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWritable,
        Text, Text, LongWritable>.Context context)
        throws IOException, InterruptedException {
        String line = value.toString(); // Stringa in input

        // Per separare la riga del file in parole
        StringTokenizer st = new StringTokenizer(line);

        logMessage(line);
        if (st.countTokens() != 3) {
            logMessage("Invalid line at position " + key + ": " + line + ".
                Expected 3 tokens!");
            return;
        }

        String month = st.nextToken();
        String site = st.nextToken();
        String visitsStr = st.nextToken();

        long visits = -1;
        try {
            visits = Long.parseLong(visitsStr);
        } catch (Exception e) {
            logMessage("Invalid visit count at line number " + key + ". This
                line will be discarded");
            return;
        }

        context.write(new Text(month), new LongWritable(visits));
    }

    private void logMessage(String msg) {
        System.out.println(msg);
    }
}

```

2.2.2 RecordReducer.java

Questa classe rappresenta il **Reduce** all'interno del pattern MapReduce. Nel metodo reduce si hanno in input una chiave e una lista di valori. In questo caso la chiave è il nome del mese e la lista di valori è la lista di visite effettuate nel mese specificato nella chiave sui vari siti presenti nel file. Sommando questi valori si ottiene il numero totale di visite effettuato nel mese in chiave. Di seguito il codice del file RecordReducer.java:

```

package count;
import java.io.IOException;

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

```

```

public class RecordReducer extends Reducer<Text, LongWritable, Text, LongWritable> {

    @Override
    protected void reduce(Text month, Iterable<LongWritable> values,
                          Reducer<Text, LongWritable, Text, LongWritable>.Context context)
    {
        long sum = 0L;
        for (LongWritable value : values) {
            sum += value.get();
        }

        context.write(month, new LongWritable(sum));
    }
}

```

2.2.3 ApplicationDriver.java

Questa classe contiene il main dell'applicazione, il quale si occupa di configurare l'applicazione MapReduce e di avviarla. Come è possibile notare nel codice sottostante, si è scelto di usare i program arguments passati all'atto di esecuzione dell'applicazione per specificare:

- Il path del DFS da cui leggere l'input (ovvero il file websites.txt)
- Il path di output in cui salvare i risultati dell'applicazione
- Il numero di reducers da utilizzare nell'applicazione

```

package count;
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.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class ApplicationDriver {
    public static void main(String[] args) throws IOException, ClassNotFoundException, InterruptedException {
        // Pattern di lancio - istanziazione
        Job job = Job.getInstance();

        // Hadoop deve sapere da dove partire.
        // Specificiamo quindi l'entrypoint del job
        job.setJarByClass(count.ApplicationDriver.class);

        // Impostiamo un nome simbolico al Job
        job.setJobName("CountViewsByMonthApplication");

        // Si possono settare qui il numero di reduce da istanziare, mentre
        // i mapper sono gestiti dal framework. Se non si setta il numero,
        // di default si usa un solo reducer
        job.setNumReduceTasks(Integer.parseInt(args[2]));

        // Specificiamo il path da usare come input dell'applicazione a partire dal DFS
        // Noi glielo facciamo prendere da program arguments
        FileInputFormat.addInputPath(job, new Path(args[0]));
    }
}

```

```
// Path di uscita
FileOutputFormat.setOutputPath(job, new Path(args[1]));

job.setMapperClass(count.RecordMapper.class);
job.setReducerClass(count.RecordReducer.class);
// Qui si poteva usare anche un combiner, ma non lo facciamo

// Specifichiamo il formato dell'uscita
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(LongWritable.class);

// Facciamo attendere il main finché il Job non completa
// Con questa riga, inoltre, lanciamo il job
System.exit(job.waitForCompletion(true) ? 0 : 1);

}
}
```

2.2.4 Esecuzione

Per eseguire l'applicazione è stato inserito il file di input *websites.txt* all'interno del DFS utilizzando il seguente comando dalla directory contenente il file *websites.txt*:

```
hdfs dfs -put websites.txt /home/studente/input/websites.txt
```

Successivamente si è cancellata la cartella contenente l'output:

```
hdfs dfs -rm -r /home/studente/output1
```

Per eseguire l'applicazione è stato lanciato il comando dalla cartella contenente il file jar dell'applicazione in due modi, ovvero specificando prima un solo reducers e poi 3 reducers:

```
hadoop jar MapReduce1.jar count.ApplicationDriver /home/studente/input \
/home/studente/output1 1
hadoop jar MapReduce1.jar count.ApplicationDriver /home/studente/input \
/home/studente/output1-1 3
```

Di seguito l'output ottenuto utilizzando un solo reducer:

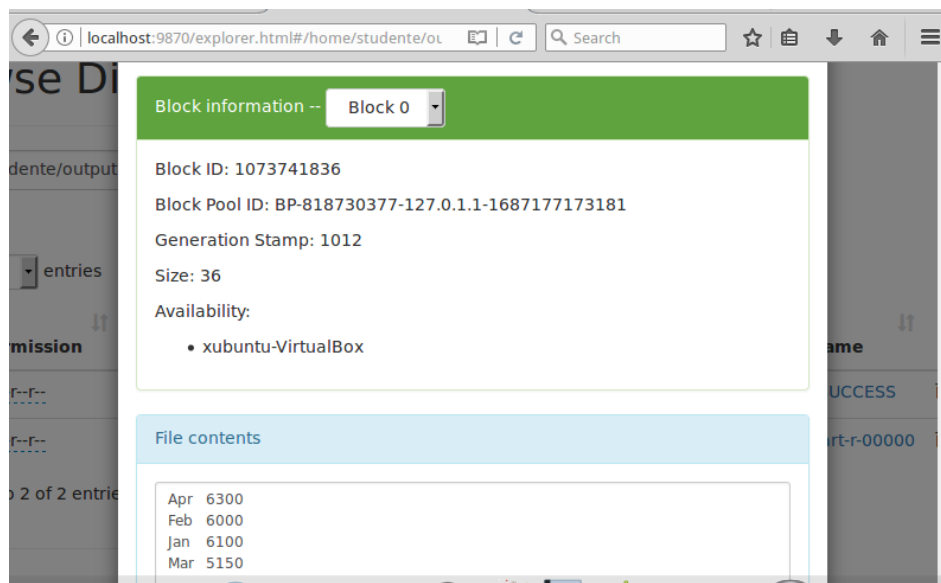
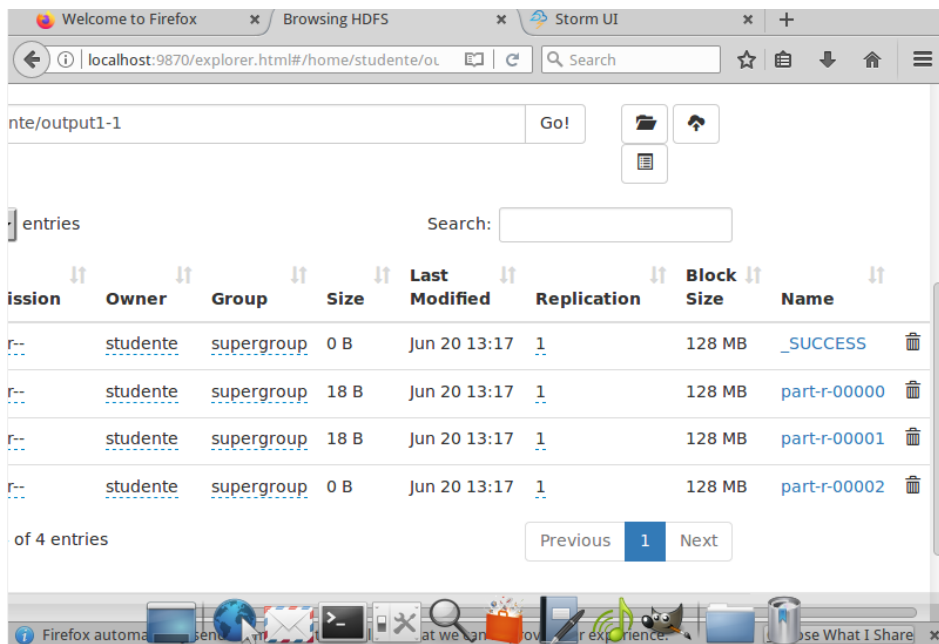
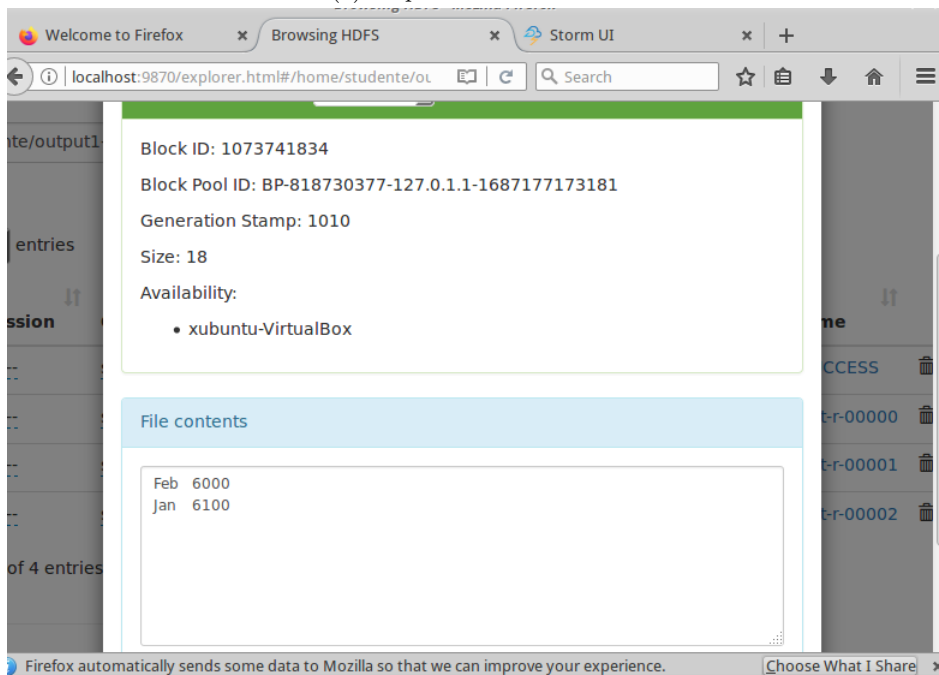


Figura 1: MapReduce result

Di seguito l'output ottenuto specificando 3 come numero di reducers:



(a) MapReduce1 3 reducers



(b) MapReduce1 3 reducers



Come si nota dalle immagini, eseguendo l'applicazione utilizzando più di un reducer fa sì che l'output venga diviso tra i vari reducers utilizzati. In questo caso, vediamo come due di questi reducers abbiano elaborato dati mentre uno non ha elaborato niente. Questo è dovuto al modo in cui il framework gestisce la distribuzione del carico di lavoro, considerando anche il raggruppamento per chiave fatto in fase di Mapping.

2.3 Esercizio 1.2

L'obiettivo di questo esercizio è contare il numero di siti distinti presente nel file. Per fare ciò sono state implementate le classi **RecordMapper.java**, **RecordReducer.java** e **RecordDriver.java**. Questo esercizio è abbastanza simile al precedente per quanto riguarda la classe Driver sviluppata, si pone dunque l'attenzione sulle classi che implementano le funzioni Map e Reduce.

2.3.1 RecordMapper.java

Nel metodo **map**, a differenza dell'esercizio precedente, viene usata come key una stringa costante, ovvero "distSites", poiché l'obiettivo è di propagare verso il reducer un solo valore con chiave "distSites" e con valore il mese letto dalla riga del file. Il codice della classe è il seguente:

```
package distinct;
import java.io.IOException;
import java.util.StringTokenizer;

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

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

    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text, Text, Text>.Context context
        throws IOException, InterruptedException {
        String line = value.toString(); // Stringa in input
        StringTokenizer st = new StringTokenizer(line); // Per separare la riga del file in parole

        String distWebsiteKey = "distSites";

        logMessage(line);
        if (st.countTokens() != 3) {
            logMessage("Invalid line at position " + key + ": " + line + ". Expected 3 tokens!");
            return;
        }

        String month = st.nextToken();
        String site = st.nextToken();
        String visitsStr = st.nextToken();

        long visits = -1;
        try {
            visits = Long.parseLong(visitsStr);
        } catch (Exception e) {
            logMessage("Invalid visit count at line number " + key + ". This line will be discarded");
            return;
        }

        context.write(new Text(distWebsiteKey), new Text(site));
    }
}
```

```
private void logMessage(String msg) {
    System.out.println(msg);
}
}
```

2.3.2 RecordReducer.java

Nel metodo **reduce** la key attesa è la stringa "distSites" e la lista di valori attesa è la lista di mesi contenuti nel file. Per completare l'esercizio, il metodo inserisce i mesi letti dal file in un set con lo scopo di eliminare i duplicati e, contando la dimensione del set, si ottiene il numero di siti distinti nel file. Il codice della classe è il seguente:

```
package distinct;
import java.io.IOException;
import java.util.HashSet;
import java.util.Set;

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class RecordReducer extends Reducer<Text, Text, Text, LongWritable> {

    @Override
    protected void reduce(Text key, Iterable<Text> values,
        Reducer<Text, Text, Text, LongWritable>.Context context) throws IOException, InterruptedException {
        Set<String> distinctSites = new HashSet<String>();
        for (Text value : values) {
            distinctSites.add(value.toString());
        }

        context.write(key, new LongWritable(distinctSites.size()));
    }
}
```

2.3.3 Esecuzione

Per eseguire l'applicazione è stato inserito il file di input *websites.txt* all'interno del DFS utilizzando il seguente comando dalla directory contenente il file *websites.txt*:

```
hdfs dfs -put websites.txt /home/studente/input/websites.txt
```

Successivamente si è cancellata la cartella contenente l'output:

```
hdfs dfs -rm -r /home/studente/output2
```

Per eseguire l'applicazione è stato lanciato il comando dalla cartella contenente il file jar dell'applicazione:

```
hadoop jar MapReduce2.jar count.ApplicationDriver /home/studente/input \
/home/studente/output2 3
```

In questo comando sono stati specificati rispettivamente i path di input e di output come program arguments. Il numero finale imposta il numero di reducers da utilizzare nell'applicazione. Sono stati effettuati test con 1 reducer, 2 reducer e 3 reducer. Si è osservato che, a prescindere dal numero di reducer impostati, solo un reducer conteneva l'output non vuoto, con una sola riga contenente l'output atteso, ovvero il numero di mesi distinti nel file.

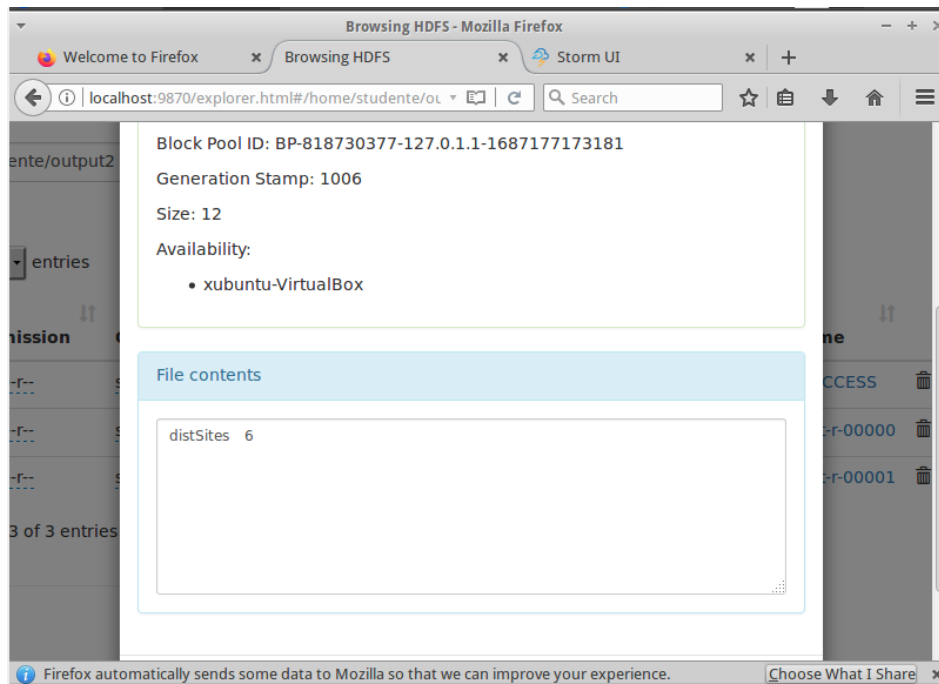


Figura 3: MapReduce2 result

2.4 Esercizio 1.3

L'obiettivo di questo esercizio è contare il numero minimo di visite presente nel file. Per fare ciò sono state implementate le classi **RecordMapper.java**, **RecordReducer.java** e **RecordDriver.java**. Questo esercizio è abbastanza simile al precedente per quanto riguarda la classe Driver sviluppata, si pone dunque l'attenzione sulle classi che implementano le funzioni Map e Reduce.

2.4.1 RecordMapper.java

Nel metodo **map**, come nell'esercizio precedente, viene usata come key una stringa costante, ovvero "minvisits", poiché l'obiettivo è di propagare verso il reducer un solo valore con chiave "minvisits" e con valore il numero di visite letto dalla riga del file. Il codice della classe è il seguente:

```
package minvisits;
import java.io.IOException;
import java.util.StringTokenizer;

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

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

    @Override
    protected void map(LongWritable key, Text value,
        Mapper<LongWritable, Text, Text, LongWritable>.Context context)
        throws IOException, InterruptedException {
        String line = value.toString(); // Stringa in input
        StringTokenizer st = new StringTokenizer(line); // Per separare la riga del file in parole

        String minvisits = "minvisits";

        logMessage(line);
    }
}
```

```

if (st.countTokens() != 3) {
    logMessage("Invalid line at position " + key + ": " + line + ". Expected 3 tokens!");
    return;
}

String month = st.nextToken();
String site = st.nextToken();
String visitsStr = st.nextToken();

long visits = -1;
try {
    visits = Long.parseLong(visitsStr);
} catch (Exception e) {
    logMessage("Invalid visit count at line number " + key + ". This line will be discarded");
    return;
}

context.write(new Text(minvisits), new LongWritable(visits));
}

private void logMessage(String msg) {
    System.out.println(msg);
}
}

```

2.4.2 RecordReducer.java

Nel metodo **reduce** la key attesa è la stringa "minvisits" e la lista di valori attesa è la lista di visite contenuti nel file. Per completare l'esercizio, il metodo reduce calcola qual'è il valore minimo tra la lista di visite letta in output.

Il codice del reducer è il seguente:

```

package minvisits;
import java.io.IOException;
import java.util.HashSet;
import java.util.Set;

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class RecordReducer extends Reducer<Text, LongWritable, Text, LongWritable> {

    @Override
    protected void reduce(Text key, Iterable<LongWritable> values,
        Reducer<Text, LongWritable, Text, LongWritable>.Context context) throws IOException, InterruptedException {
        Long minValue = null;
        for (LongWritable value : values) {
            long parsedValue = value.get();
            if (minValue == null) minValue = parsedValue;
            if (minValue > parsedValue) minValue = parsedValue;
        }
        context.write(key, new LongWritable(minValue));
    }
}

```

2.4.3 Esecuzione

Per eseguire l'applicazione è stato inserito il file di input *websites.txt* all'interno del DFS utilizzando il seguente comando dalla directory contenente il file *websites.txt*:

```
hdfs dfs -put websites.txt /home/studente/input/websites.txt
```

Successivamente si è cancellata la cartella contenente l'output:

```
hdfs dfs -rm -r /home/studente/output3
```

Per eseguire l'applicazione è stato lanciato il comando dalla cartella contenente il file jar dell'applicazione:

```
hadoop jar MapReduce3.jar count.ApplicationDriver /home/studente/input \
/home/studente/output3 2
```

In questo comando sono stati specificati rispettivamente i path di input e di output come program arguments. Il numero finale imposta il numero di reducers da utilizzare nell'applicazione. Sono stati effettuati test con 1 reducer, 2 reducer e 3 reducer. Si è osservato che, a prescindere dal numero di reducer impostati, solo un reducer conteneva l'output non vuoto, con una sola riga contenente l'output atteso, ovvero il numero di mesi distinti nel file.

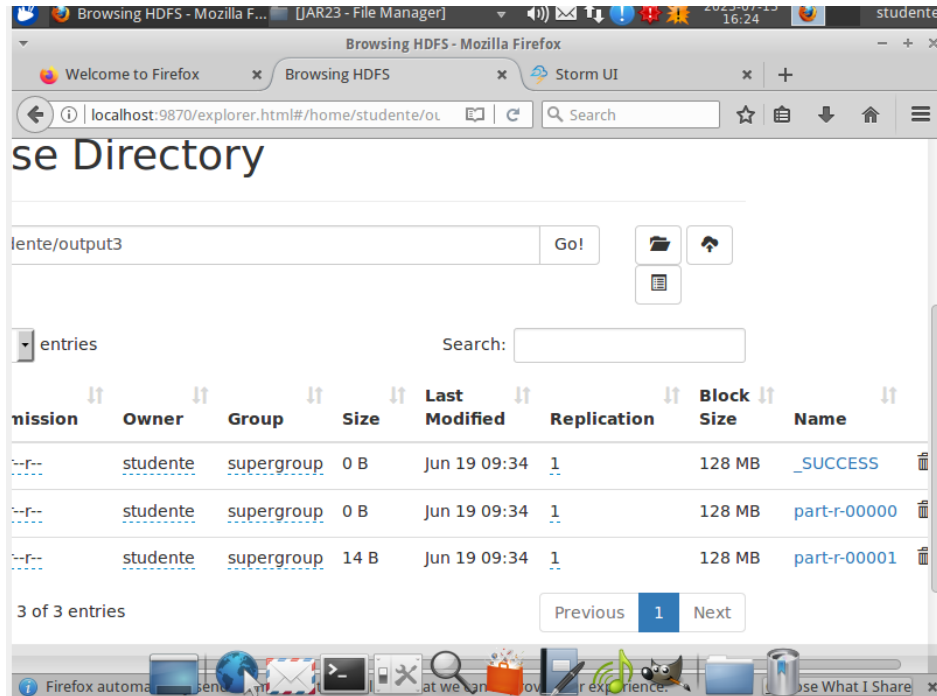


Figura 4: MapReduce3 result

3 Parte 2 - Spark

L'obiettivo di questa parte di esercitazione è sviluppare una topologia di Apache Storm consistente dei seguenti elementi:

- **SocketSpout**: Si connette ad un server, il cui codice è contenuto nello script **temperature.sh**, in esecuzione su localhost:7777 con lo scopo di accettare informazioni relative a sensori che misurano la temperatura, organizzati in tuple ID, **location**, **temperature**.
- **SamplingBolt**: Scrive il 20% delle tuple lette dal SocketSpout su un file (logfile.txt) effettuando il campionamento sull'attributo **ID**.

- **AlertBolt**: Legge le tuple dal SocketSpout e le manda in output se la temperature è \geq di 35. Il formato dell'uscita è una tupla **timestamp, location, temperature**.
- **AlertLogBolt**: Accetta le tuple emesse dall'AlertBolt e le scrive su un file (alertfile.txt)

3.1 Implementazione della soluzione

Per poter realizzare la topologia descritta dalla traccia sono state realizzate le classi Java per gli Spout e Bolt precedentemente elencati, oltre alla classe contenente il main dell'applicazione e la definizione della topologia.

3.1.1 SocketSpout

Lo spout **SocketSpout**, come detto precedentemente, ha il compito di leggere le righe emesse dal server mediante socket e, per ognuna di esse, emettere verso i bolt in ascolto i dati sottoforma di tuple (id,location,temperature).

Per agevolare i calcoli effettuati dai bolt successivi, viene effettuata la conversione della temperatura e dell'id da stringa a Long.

Di seguito il codice del SocketSpout:

```
package temperature;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Map;

import org.apache.storm.spout.SpoutOutputCollector;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.base.BaseRichSpout;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Values;

public class SocketSpout extends BaseRichSpout{

    private SpoutOutputCollector collector;
    private BufferedReader reader;

    @Override
    public void nextTuple() {
        // TODO Auto-generated method stub
        String line = null;
        try {
            line = reader.readLine();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return;
    }

    if (line == null) return;
    String[] tokens = line.split(" ");
    String id = tokens[5];
    String location = tokens[6];
    String temperatureStr = tokens[8];
```

```

    long temperature = -1;
    long idLong = -1;
    try {
        temperature = Long.parseLong(temperatureStr);
        idLong = Long.parseLong(id);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return;
}

collector.emit(new Values(idLong, location, temperature));

}

@Override
public void open(Map arg0, TopologyContext arg1, SpoutOutputCollector arg2) {
    this.collector = arg2;

    Socket socket;
    try {
        socket = new Socket("localhost", 7777);
        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        this.reader = in;
    } catch (IOException e) {
        e.printStackTrace();
    }

}

@Override
public void declareOutputFields(OutputFieldsDeclarer arg0) {
    arg0.declare(new Fields("id", "location", "temperature"));

}

}

```

3.1.2 SamplingBolt

Il bolt **SamplingBolt** ha il compito di scrivere il 20% delle tuple lette su un file (logfile.txt) effettuando il sampling per decidere quali tuple scrivere filtrando per il campo **id**.

La soluzione scelta per applicare tale filtro consiste nel verificare se l'id è minore di 2, in questo modo vengono scelti solo i sensori con id 1, ovvero solo 1 su 5 dei possibili id.

Di seguito il codice del SocketSpout:

```

package temperature;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Map;

```

```

import org.apache.storm.task.OutputCollector;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.base.BaseRichBolt;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Tuple;
import org.apache.storm.tuple.Values;

public class SamplingBolt extends BaseRichBolt {

    private OutputCollector collector;
    private FileOutputStream fileOutputStream;

    @Override
    public void execute(Tuple arg0) {
        Long id = arg0.getLong(0);
        String location = arg0.getString(1);
        Long temperature = arg0.getLong(2);

        if (id%10 < 2) {
            // Get only 20% of event by filtering IDs
            collector.emit(new Values(id, location, temperature));
            String fileStr = "" + id + "," + location + "," + temperature + "\n";
            System.out.println(fileStr);
            try {
                fileOutputStream.write(fileStr.getBytes());
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        collector.ack(arg0);
    }

    @Override
    public void prepare(Map arg0, TopologyContext arg1, OutputCollector arg2) {
        this.collector = arg2;
        try {
            this.fileOutputStream = new FileOutputStream("logfile.txt");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer arg0) {
        arg0.declare(new Fields("id", "location", "temperature"));
    }

    @Override
    public void cleanup() {
        // TODO Auto-generated method stub
        super.cleanup();
        if (fileOutputStream != null)

```



```

try {
fileOutputStream.close();
} catch (IOException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
}
}

```

In alternativa alla soluzione usata per il sampling, è possibile adottare una soluzione che non prevede la conoscenza del valore dei possibili id che il server può generare. Questa soluzione alternativa prevedrebbe di determinare il massimo valore che può assumere il campo ID durante la ricezione degli eventi e ricalibrare il filtro di conseguenza.

Di seguito una versione del metodo execute() rivisitato applicando quest'ultima soluzione:

```

[...]
public class SamplingBolt extends BaseRichBolt {

private OutputCollector collector;
private FileOutputStream fileOutputStream;

private Long maxId = null;

@Override
public void execute(Tuple arg0) {
Long id = arg0.getLong(0);
String location = arg0.getString(1);
Long temperature = arg0.getLong(2);

if (maxId == null) maxId = id;
long threshold = maxId * 20 / 100;

if (id % (maxId+1) <= threshold) {
// Get only 20% of event by filtering IDs
collector.emit(new Values(id, location, temperature));
String fileStr = "" + id + "," + location + "," + temperature + "\n";
System.out.println(fileStr);
try {
fileOutputStream.write(fileStr.getBytes());
} catch (IOException e) {
e.printStackTrace();
}
}
collector.ack(arg0);
}

[...]
}

```

3.1.3 AlertBolt

Il bolt **AlertBolt** ha il compito di leggere le tuple provenienti dal SocketSpout, filtrarle scegliendo solo quelle con un valore di temperature ≥ 35 e propagare verso l'AlertLogBolt queste tuple in cui al posto dell'id viene inserito il timestamp attuale **id**.

Di seguito il codice di AlertBolt:

```

package temperature;

```

```

import java.io.FileOutputStream;

import java.util.Map;

import org.apache.storm.task.OutputCollector;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.base.BaseRichBolt;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Tuple;
import org.apache.storm.tuple.Values;

public class AlertBolt extends BaseRichBolt {

    private OutputCollector collector;

    @Override
    public void execute(Tuple arg0) {
        Long id = arg0.getLong(0);
        String location = arg0.getString(1);
        Long temperature = arg0.getLong(2);

        if (temperature >= 35L) {
            long timestamp = System.currentTimeMillis();
            collector.emit(new Values(timestamp, location, temperature));
        }
        collector.ack(arg0);
    }

    @Override
    public void prepare(Map arg0, TopologyContext arg1, OutputCollector arg2) {
        this.collector = arg2;
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer arg0) {
        arg0.declare(new Fields("timestamp", "location", "temperature"));
    }
}

```

3.1.4 AlertLogBolt

Il bolt **AlertLogBolt** ha il compito di leggere le tuple provenienti da AlertBolt e scriverle su un file (alertfile.txt). **id**.

Di seguito il codice di AlerLogtBolt:

```

package temperature;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.FileWriter;

```

```

import java.io.IOException;
import java.util.Map;

import org.apache.storm.task.OutputCollector;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.base.BaseRichBolt;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Tuple;
import org.apache.storm.tuple.Values;

public class AlertLogBolt extends BaseRichBolt {

    private OutputCollector collector;
    private FileOutputStream fileOutputStream;

    @Override
    public void execute(Tuple arg0) {
        Long timestamp = arg0.getLong(0);
        String location = arg0.getString(1);
        Long temperature = arg0.getLong(2);

        collector.emit(new Values(timestamp, location, temperature));
        String fileStr = "" + timestamp + "," + location + "," + temperature + "\n";
        try {
            fileOutputStream.write(fileStr.getBytes());
        } catch (IOException e) {
            e.printStackTrace();
        }
        collector.ack(arg0);
    }

    @Override
    public void prepare(Map arg0, TopologyContext arg1, OutputCollector arg2) {
        this.collector = arg2;
        try {
            this.fileOutputStream = new FileOutputStream("alertfile.txt");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer arg0) {
        arg0.declare(new Fields("timestamp", "location", "temperature"));
    }

    @Override
    public void cleanup() {
        // TODO Auto-generated method stub
        super.cleanup();
        if (fileOutputStream != null)
            try {

```

```

fileOutputStream.close();
} catch (IOException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
}
}
}

```

3.1.5 MyTopology

La classe **MyTopology** contiene la definizione della topologia Spark da sottomettere al framework. Nella topologia sono definiti gli Spout e i Bolt utilizzati dall'applicazione e la modalità in cui questi sono interconnessi. Poiché la traccia non specifica la necessità di effettuare partizionamenti per chiave, si è scelto di interconnettere le componenti con **ShuffleGrouping** anziché FieldGrouping. Le modalità di esecuzione della topologia implementate consistono nell'uso del LocalCluster e dello StormSubmitter, entrambe presenti nel codice sorgente.

Di seguito il codice di MyTopology:

```

package temperature;

import org.apache.storm.Config;
import org.apache.storm.LocalCluster;
import org.apache.storm.topology.TopologyBuilder;

public class MyTopology {
public static void main(String[] args) throws Exception {
Config config = new Config();
config.setDebug(false);

// Topology
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("SocketSpout", new SocketSpout());
builder.setBolt("SamplingBolt", new SamplingBolt())
.shuffleGrouping("SocketSpout");
builder.setBolt("AlertBolt", new AlertBolt())
.shuffleGrouping("SocketSpout");
builder.setBolt("AlertLogBolt", new AlertLogBolt())
.shuffleGrouping("AlertBolt");

// Execute using LocalCluster
LocalCluster cluster = new LocalCluster();
cluster.submitTopology("myTopology", config, builder.createTopology());

// Wait 100 seconds before stopping
Thread.sleep(100000);

// Shutdown local cluster
cluster.shutdown();

/*
// Execute using StormSubmitter
StormSubmitter.submitTopology("myTopology", config, builder.createTopology());

*/
}
}

```

```
}
```

3.2 Esecuzione

L'applicazione precedentemente descritta è stata eseguita in 2 modalità:

- LocalCluster
- StormSubmitter

3.2.1 Esecuzione con LocalCluster

L'esecuzione con **LocalCluster** prevede di eseguire l'applicazione Storm lasciando all'applicazione stessa il compito di creare il cluster sul quale operare. Per questo motivo nella classe della topologia è stato implementato il seguente codice:

```
[...]
// Execute using LocalCluster
LocalCluster cluster = new LocalCluster();
cluster.submitTopology("myTopology", config, builder.createTopology());
// Wait 100 seconds before stopping
Thread.sleep(100000);
// Shutdown local cluster
cluster.shutdown();
[...]
```

Questa modalità è chiaramente la più semplice e come risultato gli output, ovvero i files logfile.txt e alertfile.txt, vengono salvati nella stessa directory da cui si esegue l'applicazione.

Per eseguire l'applicazione è stato lanciato il comando dalla cartella contenente il file jar dell'applicazione (SparkTemperature.jar):

```
storm jar ./SparkTemperature.jar temperature.MyTopology
```

La durata dell'esecuzione è di 100 secondi, così da dare il tempo al server da cui l'applicazione legge i dati di generare tuple il più variegata possibili.

Durante l'esecuzione vengono creati e popolati i files alertfile.txt e logfile.txt

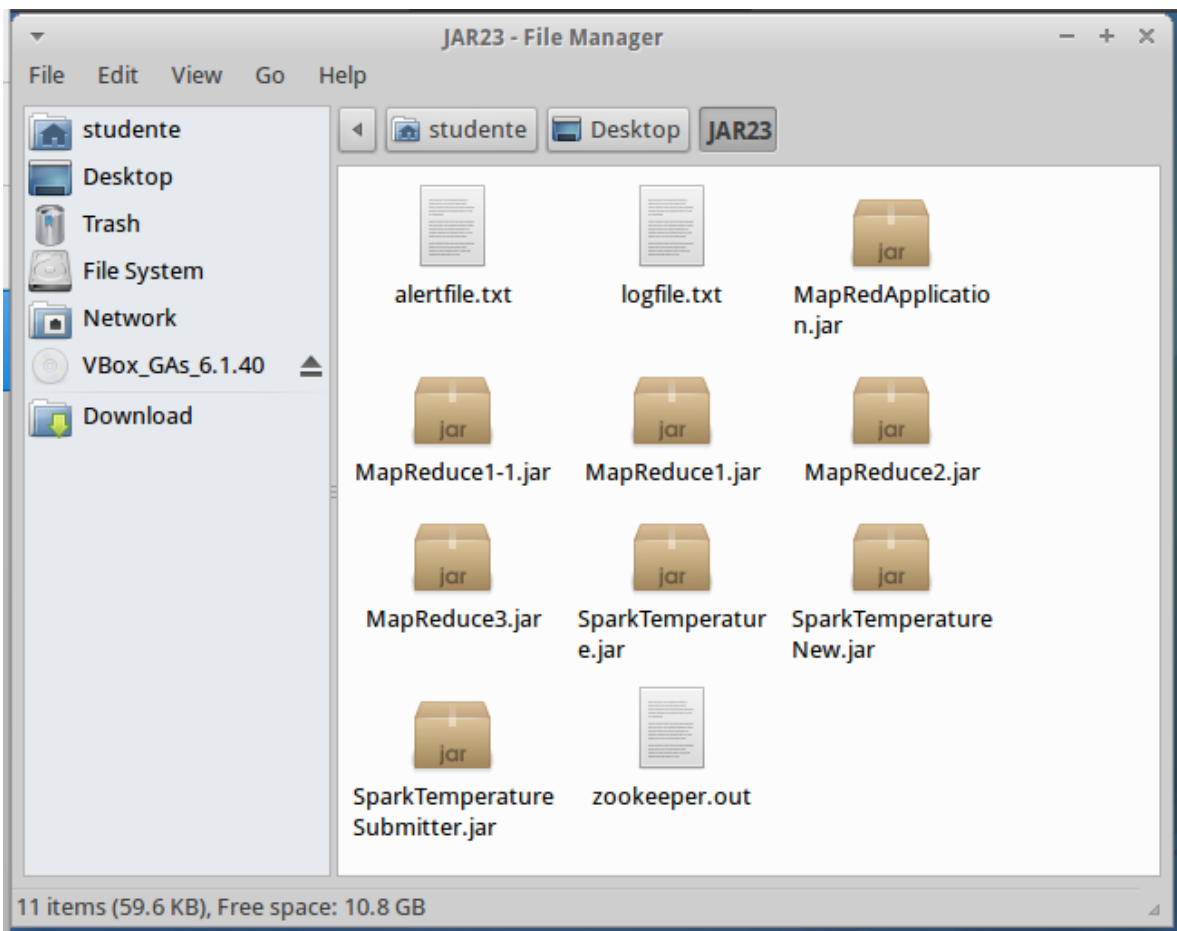


Figura 5: Spark Local Cluster folder results

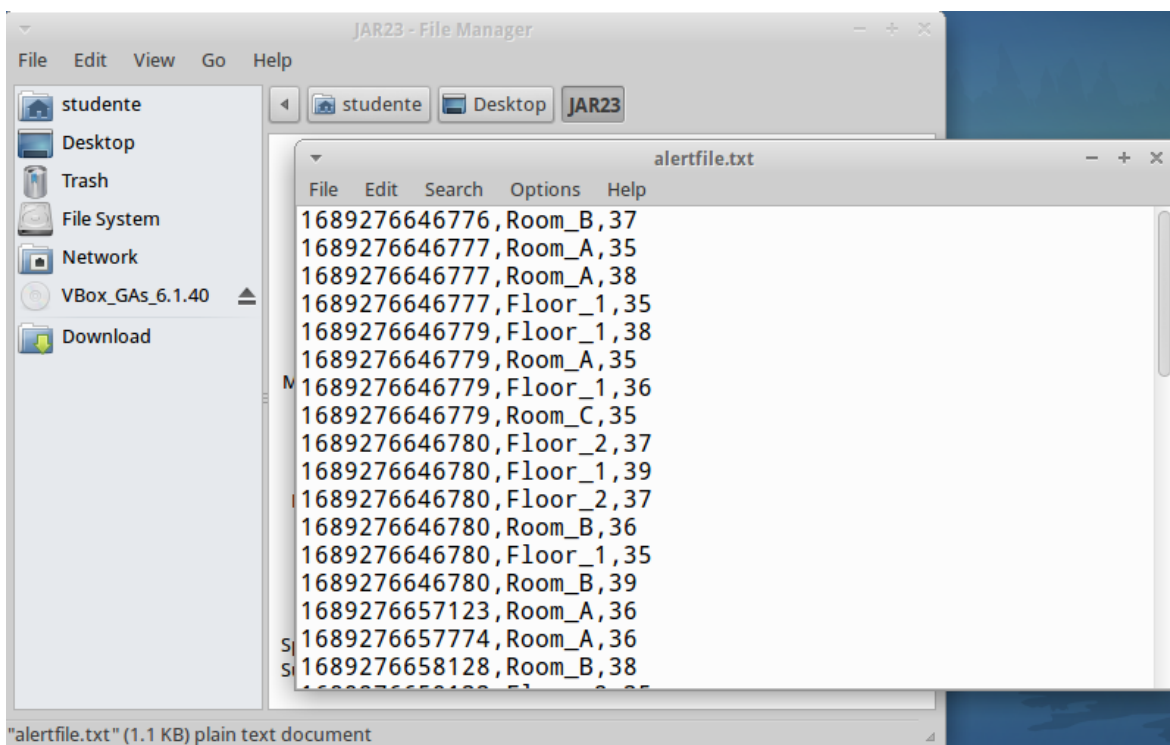


Figura 6: Alertfile.txt

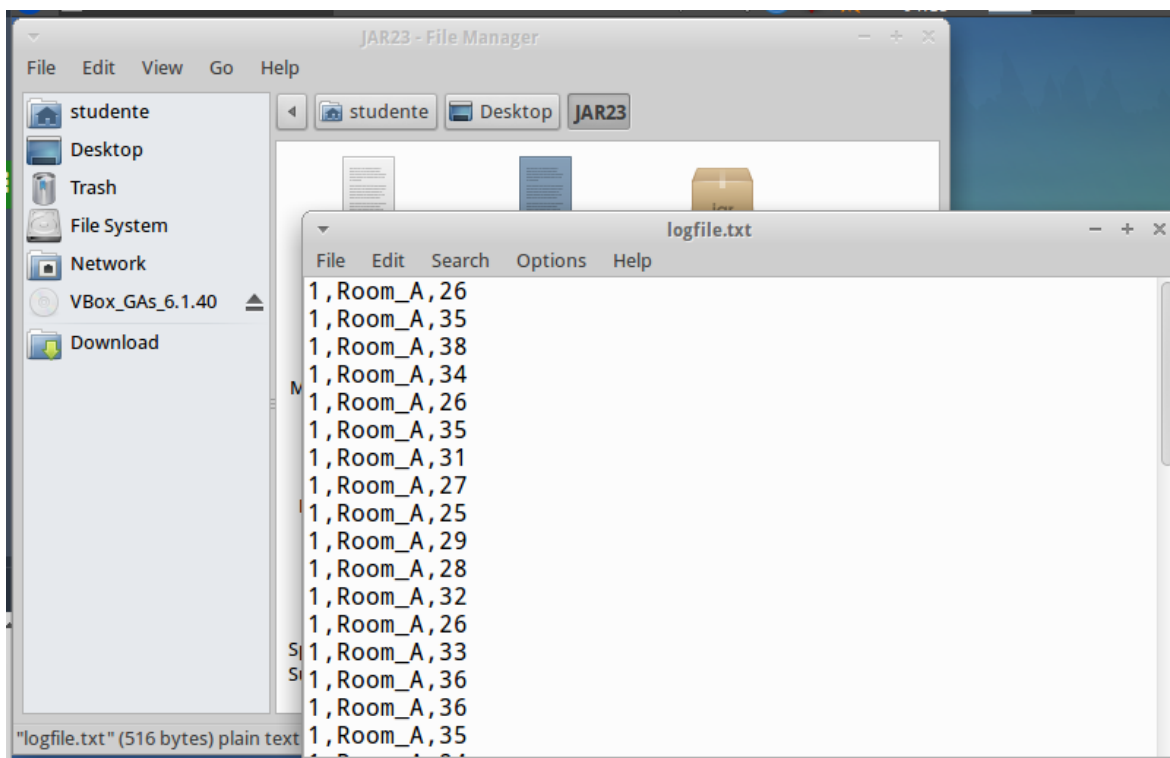


Figura 7: Logfile.txt

Come si può notare dalle immagini precedenti, il file **alertfile.txt** contiene solo tuple il cui valore di temperatura è $t = 35$ e il file **logfile.txt** contiene solo tuple il cui ID è 1, ovvero l'id corrispondente al 20% dei possibili ID generabili dal server.

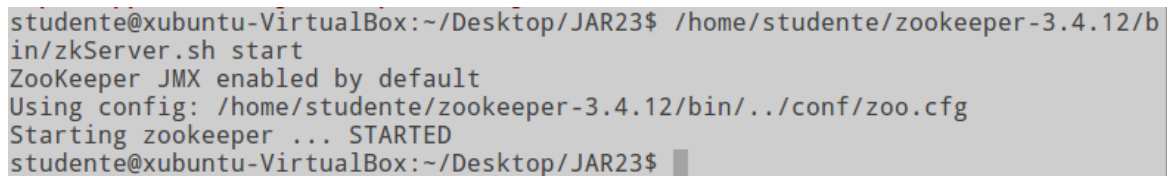
3.2.2 Esecuzione con SparkSubmitter

L'esecuzione con **SparkSubmitter** prevede di eseguire l'applicazione Storm dopo aver manualmente svolto il compito di creare il cluster sul quale operare. Per questo motivo nella classe della topologia è stato implementato il seguente codice:

```
[...]
// Execute using StormSubmitter
StormSubmitter.submitTopology("myTopology", config, builder.createTopology());
[...]
```

Per poter adottare questo metodo di esecuzione, è stato innanzitutto necessario avviare il server di **Zookeeper**. Per fare ciò, dalla cartella di installazione di Zookeeper (/home/studente/zookeeper-3.4.12/bin) è stato eseguito lo script

```
./zkServer start
```

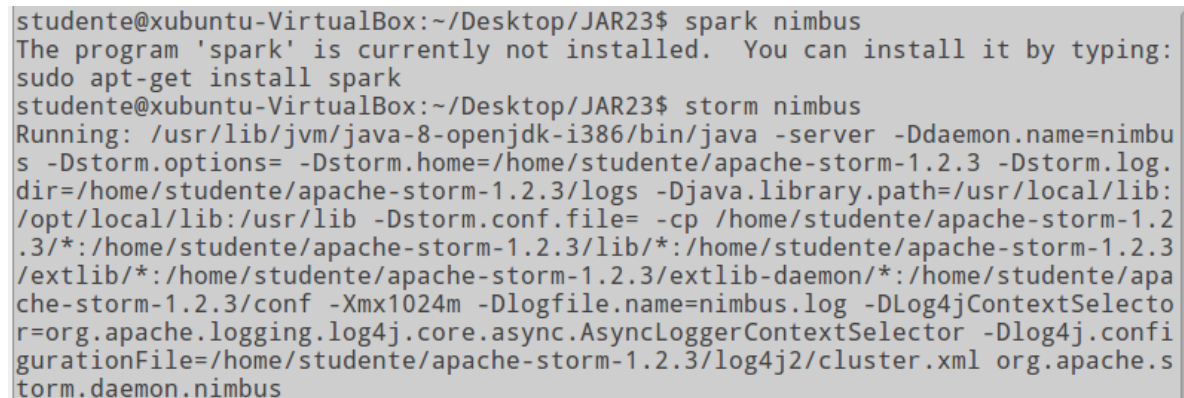


```
studente@xubuntu-VirtualBox:~/Desktop/JAR23$ /home/studente/zookeeper-3.4.12/bin/zkServer.sh start
ZooKeeper JMX enabled by default
Using config: /home/studente/zookeeper-3.4.12/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
studente@xubuntu-VirtualBox:~/Desktop/JAR23$
```

Figura 8: Start Zookeeper

Successivamente, sono stati avviati il **Nimbus** di Storm, un **Supervisor** di Storm e la **UI** di Storm, eseguendo i seguenti comandi ognuno in un terminale a sé stante:

```
storm nimbus
```



```
studente@xubuntu-VirtualBox:~/Desktop/JAR23$ spark nimbus
The program 'spark' is currently not installed. You can install it by typing:
sudo apt-get install spark
studente@xubuntu-VirtualBox:~/Desktop/JAR23$ storm nimbus
Running: /usr/lib/jvm/java-8-openjdk-i386/bin/java -server -Ddaemon.name=nimbus
-Dstorm.options= -Dstorm.home=/home/studente/apache-storm-1.2.3 -Dstorm.log.dir=/home/studente/apache-storm-1.2.3/logs -Djava.library.path=/usr/local/lib:/opt/local/lib:/usr/lib -Dstorm.conf.file= -cp /home/studente/apache-storm-1.2.3/*:/home/studente/apache-storm-1.2.3/lib/*:/home/studente/apache-storm-1.2.3/extlib/*:/home/studente/apache-storm-1.2.3/extlib-daemon/*:/home/studente/apache-storm-1.2.3/conf -Xmx1024m -Dlogfile.name=nimbus.log -DLog4jContextSelector=org.apache.logging.log4j.core.async.AsyncLoggerContextSelector -Dlog4j.configurationFile=/home/studente/apache-storm-1.2.3/log4j2/cluster.xml org.apache.storm.daemon.nimbus
```

Figura 9: Start Nimbus

```
storm supervisor
```



```

studente@xubuntu-VirtualBox:~/Desktop/JAR23$ storm supervisor
Running: /usr/lib/jvm/java-8-openjdk-i386/bin/java -server -Ddaemon.name=super
visor -Dstorm.options= -Dstorm.home=/home/studente/apache-storm-1.2.3 -Dstorm.
log.dir=/home/studente/apache-storm-1.2.3/logs -Djava.library.path=/usr/local/
lib:/opt/local/lib:/usr/lib -Dstorm.conf.file= -cp /home/studente/apache-storm
-1.2.3/*:/home/studente/apache-storm-1.2.3/lib/*:/home/studente/apache-storm-1
.2.3/extlib/*:/home/studente/apache-storm-1.2.3/extlib-daemon/*:/home/studente
/apache-storm-1.2.3/conf -Xmx256m -Dlogfile.name=supervisor.log -Dlog4j.config
urationFile=/home/studente/apache-storm-1.2.3/log4j2/cluster.xml org.apache.st
orm.daemon.supervisor.Supervisor

```

Figura 10: Start Supervisor

```
storm ui
```

```

studente@xubuntu-VirtualBox:~/Desktop/JAR23$ storm ui
Running: /usr/lib/jvm/java-8-openjdk-i386/bin/java -server -Ddaemon.name=ui -D
storm.options= -Dstorm.home=/home/studente/apache-storm-1.2.3 -Dstorm.log.dir=
/home/studente/apache-storm-1.2.3/logs -Djava.library.path=/usr/local/lib:/opt
/local/lib:/usr/lib -Dstorm.conf.file= -cp /home/studente/apache-storm-1.2.3/*
:/home/studente/apache-storm-1.2.3/lib/*:/home/studente/apache-storm-1.2.3/ext
lib/*:/home/studente/apache-storm-1.2.3/extlib-daemon/*:/home/studente/apache-
storm-1.2.3:/home/studente/apache-storm-1.2.3/conf -Xmx768m -Dlogfile.name=ui.
log -Dlog4jContextSelector=org.apache.logging.log4j.core.async.AsyncLoggerCont
extSelector -Dlog4j.configurationFile=/home/studente/apache-storm-1.2.3/log4j2
/cluster.xml org.apache.storm.ui.core

```

Figura 11: Start UI

Per eseguire l'applicazione è stato successivamente lanciato il comando dalla cartella contenente il file jar dell'applicazione (SparkTemperature.jar):

```
storm jar ./SparkTemperature.jar temperature.MyTopology
```

```

2964 [main] INFO o.a.s.s.a.AuthUtils - Got AutoCreds []
2965 [main] WARN o.a.s.u.NimbusClient - Using deprecated config nimbus.host f
or backward compatibility. Please update your storm.yaml so it only has config
nimbus.seeds
2977 [main] INFO o.a.s.u.NimbusClient - Found leader nimbus : xubuntu-Virtual
Box:6627
3037 [main] INFO o.a.s.StormSubmitter - Uploading dependencies - jars...
3040 [main] INFO o.a.s.StormSubmitter - Uploading dependencies - artifacts...
3040 [main] INFO o.a.s.StormSubmitter - Dependency Blob keys - jars : [] / ar
tifacts : []
3085 [main] INFO o.a.s.StormSubmitter - Uploading topology jar ./SparkTempera
tureSubmitter.jar to assigned location: /home/studente/apache-storm-1.2.3/data
/nimbus/inbox/stormjar-6f2bd7cf-a87d-4997-bb2a-cdebb0046a58.jar
3154 [main] INFO o.a.s.StormSubmitter - Successfully uploaded topology jar to
assigned location: /home/studente/apache-storm-1.2.3/data/nimbus/inbox/stormj
ar-6f2bd7cf-a87d-4997-bb2a-cdebb0046a58.jar
3154 [main] INFO o.a.s.StormSubmitter - Submitting topology myTopology in dis
tributed mode with conf {"storm.zookeeper.topology.auth.scheme":"digest","stor
m.zookeeper.topology.auth.payload":"-9148905215259725703:-7284819678737433878"
,"topology.debug":false}
3154 [main] WARN o.a.s.u.Utils - STORM-VERSION new 1.2.3 old 1.2.3
3457 [main] INFO o.a.s.StormSubmitter - Finished submitting topology: myTopol
ogy
studente@xubuntu-VirtualBox:~/Desktop/JAR23$

```

Figura 12: Start Spark with SparkSubmitter

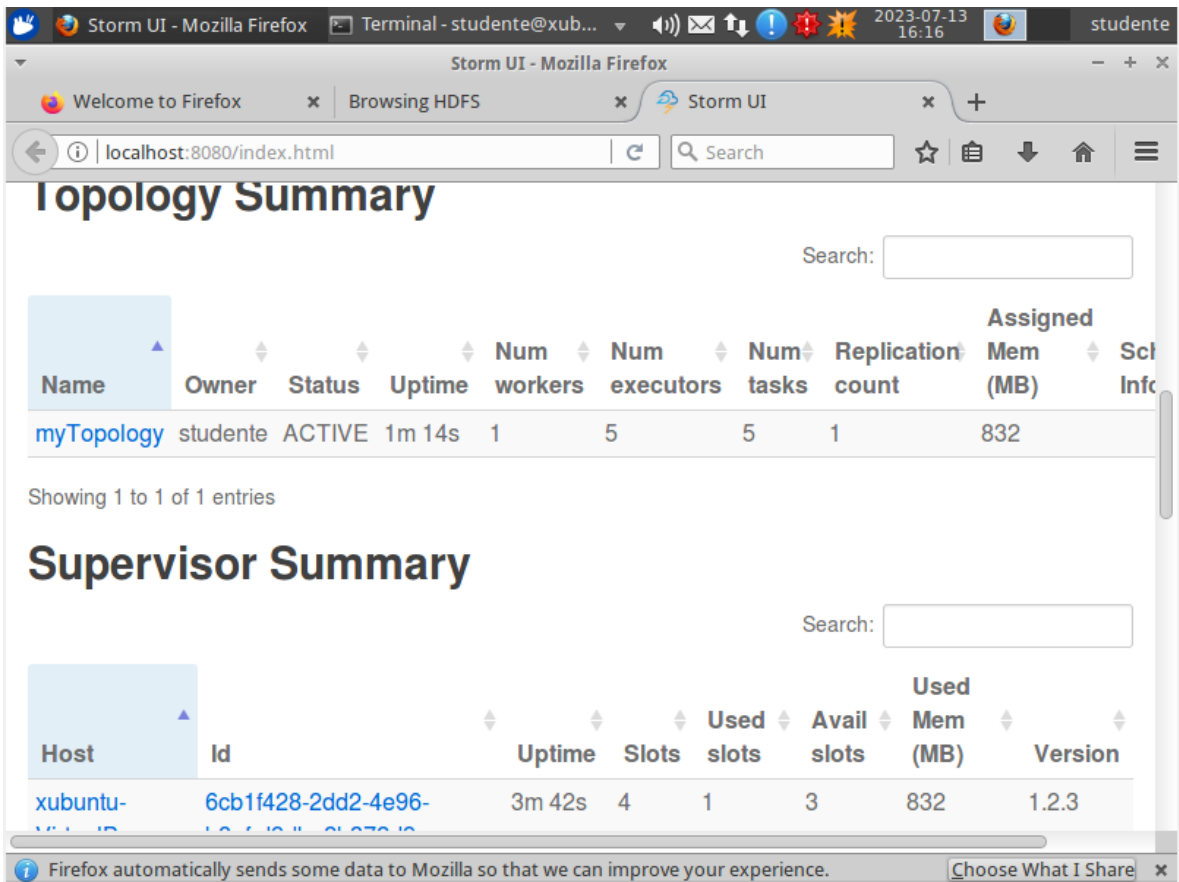


Figura 13: Spark UI with topology

Con questo metodo di esecuzione, la topologia viene eseguita e rimane in esecuzione finché non viene manualmente terminata o va incontro ad un errore.

I files scritti sono gli stessi ottenuti dall'esecuzione in modalità LocalCluster, ma anziché essere localizzati nella directory da cui si sottomette la topologia, questi sono salvati nella directory del processo Worker che è stato istanziato:

```
./home/studente/apache-storm-1.2.3/data/workers/6d88b8eb-2e01-4b3f-bcc0-4886aec75044/alertfile.txt
```

Figura 14: File position in worker process

```
./home/studente/.local/share/Trash/files/logfile.txt
studente@xubuntu-VirtualBox:/$ ^C
studente@xubuntu-VirtualBox:/$ cd ./home/studente/apache-storm-1.2.3/data/workers/6d88b8eb-2e01-4b3f-bcc0-4886aec75044/
studente@xubuntu-VirtualBox:~/apache-storm-1.2.3/data/workers/6d88b8eb-2e01-4b3f-bcc0-4886aec75044$ ls
alertfile.txt  artifacts  heartbeats  logfile.txt  pids  tmp
studente@xubuntu-VirtualBox:~/apache-storm-1.2.3/data/workers/6d88b8eb-2e01-4b3f-bcc0-4886aec75044$
```

Figura 15: Worker folder

Per trovare la posizione dei files sono stati usati i seguenti comandi:

```
cd /
find -name alertfile.txt
```

Il contenuto dei files non è esattamente lo stesso dei files precedenti, dato che i dati creati dal server sono randomizzati, però è mantenuto il criterio per il quale nell'alertfile ci sono solo tuple relative a rilevamenti con temperature ≥ 35 e nel logfile ci sono solo tuple relative al sensore con id=1:

```
studente@xubuntu-VirtualBox:/$ cat ./home/studente/apache-storm-1.2.3/data/workers/6237ce68-8659-4c1a-aeae-972d99e7901f/alertfile.txt
1689324481574,Room_C,36
1689324481585,Room_A,35
1689324481585,Room_C,37
1689324481585,Room_A,35
1689324481589,Floor_2,39
1689324481589,Room_C,35
1689324481590,Room_C,36
1689324481591,Room_B,35
1689324481592,Room_C,35
1689324481592,Floor_1,36
1689324481592,Floor_2,35
1689324481592,Room_A,35
1689324481593,Room_B,37
1689324481593,Room_A,35
1689324481598,Floor_1,38
1689324481598,Room_C,38
1689324481598,Room_B,35
1689324481598,Floor_2,37
1689324481598,Room_A,36
1689324481598,Floor_2,35
```

Figura 16: Alertfile with SparkSubmitter

```
studente@xubuntu-VirtualBox:/$ cat ./home/studente/apache-storm-1.2.3/data/workers/117efb52-703d-4ade-bec1-8f19bf9e8219/logfile.txt
1,Room_A,20
1,Room_A,27
1,Room_A,30
1,Room_A,34
1,Room_A,32
1,Room_A,38
1,Room_A,21
1,Room_A,25
1,Room_A,24
1,Room_A,39
1,Room_A,26
1,Room_A,29
1,Room_A,24
1,Room_A,31
1,Room_A,33
1,Room_A,36
1,Room_A,21
1,Room_A,22
1,Room_A,38
```

Figura 17: Logfile with SparkSubmitter