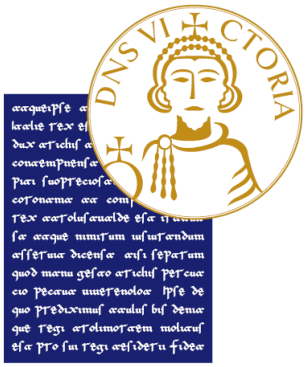


Università degli Studi del Sannio

Dipartimento di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

Data science

Homework 3

Prof:
Pecchia Antonio

Studenti:
Cinelli Jessica, 399000529
Mazzitelli Francesco C., 399000532

Anno Accademico 2022-2023

Indice

Elenco delle figure

1 Introduzione

Il seguente studio[1] ha avuto come obiettivo lo studio e l'approfondimento di alcuni dei tool pricipali per l'analisi dei dati:

- **Hadoop**: piattaforma che agevola lo sviluppo di applicazioni Map-Reduce su file system distribuiti. Mette a disposizione un framework di sviluppo e l'Hadoop Distributed File System (HDFS), un file system distribuito scalabile e portabile.
- **Storm**: framework di calcolo per l'elaborazione di flussi distribuiti. Questo consente quindi di effettuare elaborazioni "real-time" tramite la costruzione di una topologia i cui nodi prendono di "spout" (Sorgente) e "bolt" (Componenti di elaborazione).

Entrambe le applicazioni sono state realizzate in Java utilizzando come riferimento quanto svolto durante le esercitazioni.

((a)) Hadoop

((b)) Storm

Figura 1: Tools

[1] È possibile visionare l'intero progetto al link https://github.com/jessicacinelli/Homework3.git.

2 Hadoop e paradigma Map-Reduce

Il framework MapReduce è composto da diverse funzioni per ogni step:

- **Input Reader**: componente adibita alla lettura dei dati dal file system distribuito e divide l'input in diversi split
- **Map Function**: componente adibita alla generazione di zero o più coppie (chiave, valore) in uscita.
- **Partition Function**: componente adibita alla memorizzazione sul disco e al partizionamento delle coppie bufferizzate in R sezioni .
- **Reduce Function**: componente adibita alla riduzione dei valori mappati in coppie chiave valore, generando di fatto un risultato globale.
- **Output Writer**: componente adibita alla scrittura dei risultati in memoria di massa e una volta finiti tutti i task di Map e Reduce.

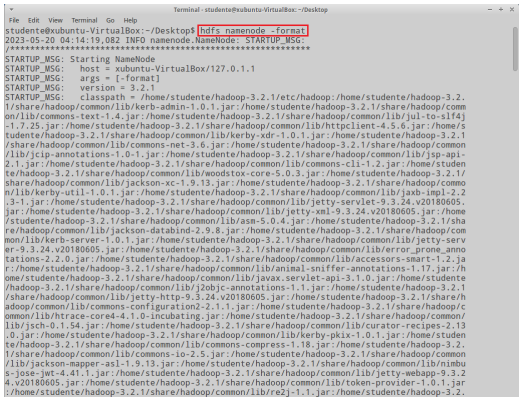
Hadoop nasconde la maggior parte delle funzioni adibite alla gestione dei dati esponendo delle interfacce che consentono la sola implementazione delle funzioni di Map e Reduce, consentendo allo sviluppatore di concentrarsi solo sull'implementazione di tali funzioni.

2.1 Inizializzazione del file system distribuito

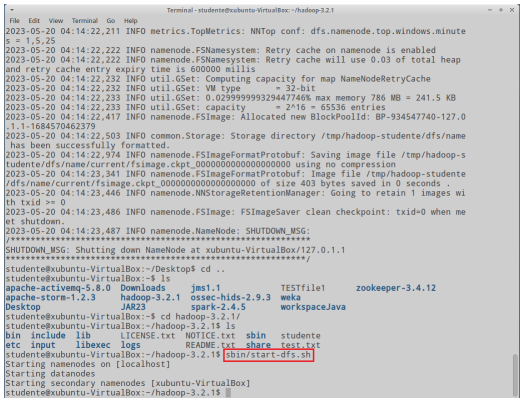
L'intero file system ha un'architettura gerarchica basata sull'interconnessione di due diverse tipologie di nodi:

- **Namenode**: Chiamato anche nodo master, può tenere traccia dei file, gestire il file system e accedere ai metadati di tutti i dati memorizzati al suo interno. In particolare, il namenode contiene dettagli sul numero di blocchi, le posizioni dei datanode in cui sono archiviati i dati, dove sono archiviate le repliche e altri dettagli.
- **Datanode**: Memorizza i dati in esso sotto forma di blocchi. Questo è anche noto come nodo slave e memorizza i dati effettivamente presenti nell'HDFS ed è responsabile di operazioni in lettura e scrittura da parte del client.

La prima operazione svolta è stata l'inizializzazione del Namenode e l'avvio del file system distribuito.



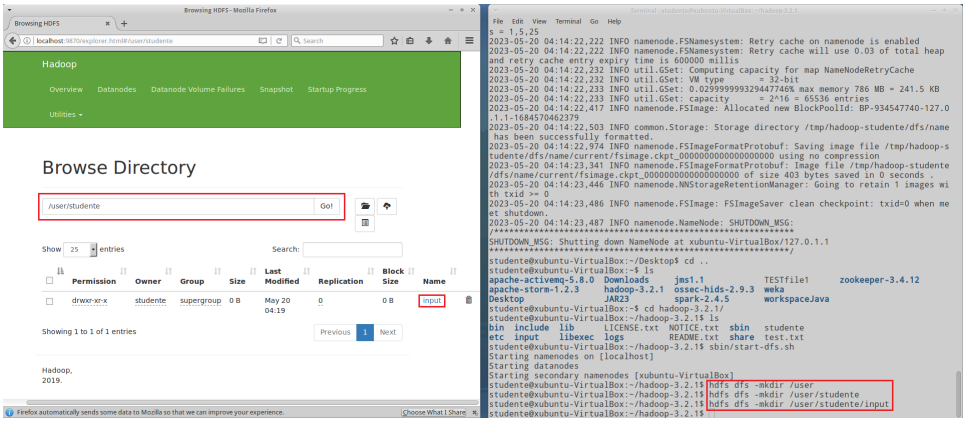
((a)) Inizializzazione Namenode



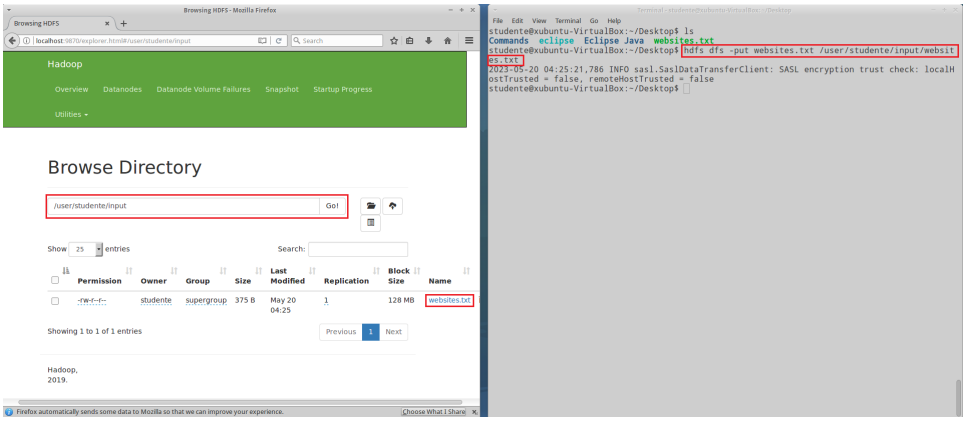
((b)) Start DFS

Figura 2: Inizializzazione File System

Dopo aver inizializzato il file system distribuito sono state create le cartelle di output e di input e quest'ultima è stata popolata con i dati su cui effettuare le elaborazioni.



((a)) Creazione cartelle



((b)) Caricamento dataset

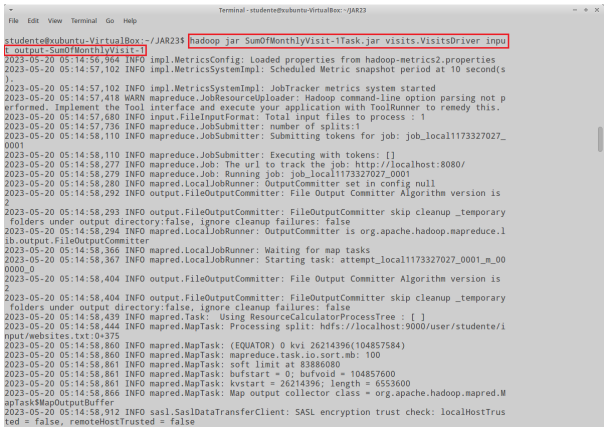
Figura 3: Creazione cartelle e push dei dati

2.2 Somma delle visite per mese

L'obiettivo della prima applicazione è calcolare il numero di visite mensili per ogni mese. Sono state implementate le seguenti classi Java:

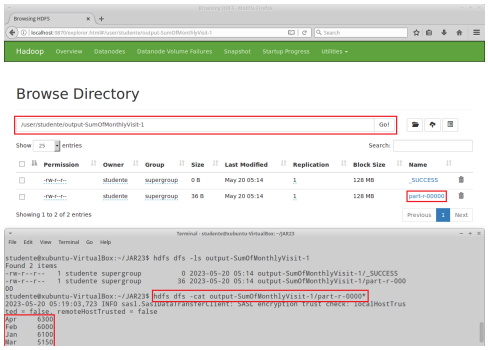
- Map: riceve in input il contentuto del file e produce in output una tupla *(chiave, valore)*: la chiave è il mese, il valore è il numero di visite.
- Reduce: riceve in input la tupla prodotta dal Mapper e calcola il numero di visite mensili. Produce in output la coppia *(mese, numero di visite mensili)*.
- Driver: ha il compito di abilitare l'esecuzione dell'applicazione e di gestirne le configurazioni.

L'implementazione dell'applicazione è consultabile nell'appendice A.

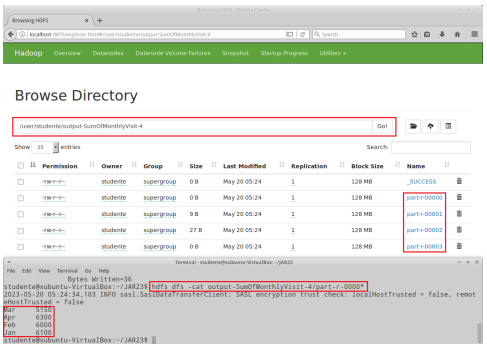


((a)) Esecuzione del JAR

Figura 4: Esecuzione del JAR



((a)) Risultato somma delle visite con un reducer



((b)) Risultato somma delle visite con più reducer

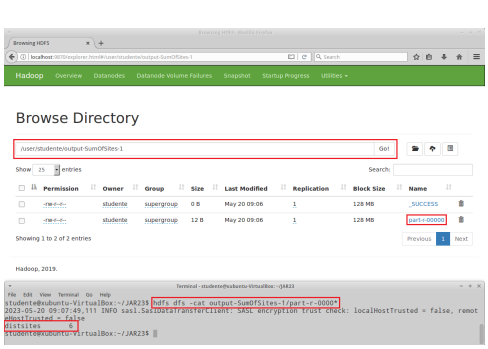
Figura 5: Risultato somma delle visite

2.3 Conteggio del numero di siti unici

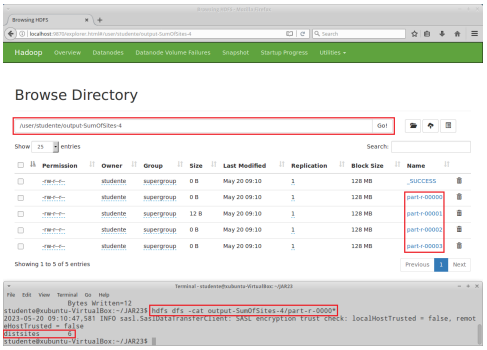
L'obiettivo della seconda applicazione è calcolare il numero di siti che compaiono all'interno del dataset. Sono state implementate le seguenti classi Java:

- Map: riceve in input il contentuto del file e produce in output una tupla *(chiave, valore)*: la chiave è una stringa "risultato", il valore è una stringa rappresentante il nome del sito.
- Reduce: riceve in input la tupla prodotta dal Mapper e calcola il numero di siti unici. Si è utilizzato un HashSet che garantisce, tramite il salvataggio all'interno dello stesso, l'unicità della stringa *sito*. Il valore relativo al conteggio è uttenuto tramite il metodo .size() associato alla struttura dati. Produce in output la coppia *(Stringa, numero di siti unici)*.
- Driver: ha il compito di abilitare l'esecuzione dell'applicazione e di gestirne le configurazioni.

L'implementazione dell'applicazione è consultabile nell'appendice B.



((a)) Risultato conteggio del numero di siti unici con un reducer



((b)) Risultato conteggio del numero di siti unici con più reducer

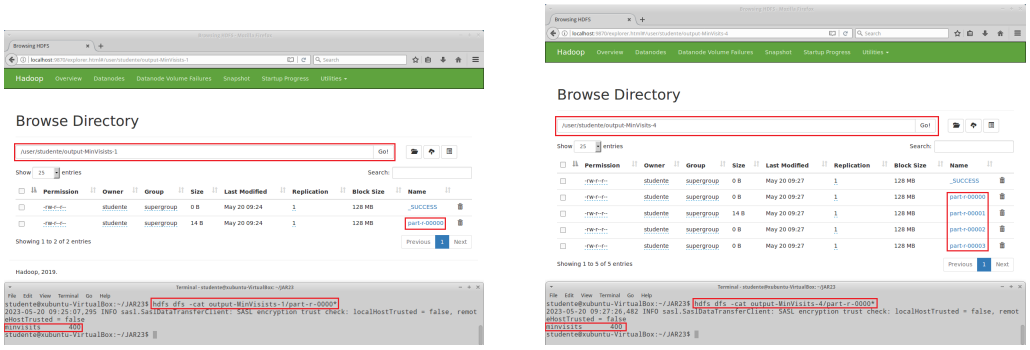
Figura 6: Risultato conteggio del numero di siti unici

2.4 Conteggio del numero minimo di visite

L'obiettivo della terza applicazione è calcolare il numero minimo di visite dell'intero dataset. Sono state implementate le seguenti classi Java:

- Map: riceve in input il contentuto del file e produce in output una tupla *(chiave, valore)*: la chiave è una stringa "risultato", il valore è una l'intero rappresentante il numero di visite per sito.
- Reduce: riceve in input la tupla prodotta dal Mapper e calcola il numero di siti unici. Si è utilizzata una lista che, tramite la funzione Collections.min(lista) ha agevolato le operazioni di calcolo del *minimo*. Produce in output la coppia *(Stringa, valore minimo di visite)*.
- Driver: ha il compito di abilitare l'esecuzione dell'applicazione e di gestirne le configurazioni.

L'implementazione dell'applicazione è consultabile nell'appendice C.



((a)) Risultato conteggio del numero minimo di visite con un reducer

((b)) Risultato conteggio del numero minimo di visite con più reducer

Figura 7: Risultato conteggio del numero minimo di visite

3 Storm e analisi di flussi di dati

Storm è un framework che agevola la realizzazione di applicazioni orientate all'elaborazione dei flussi di dati. Il tutto si basa sulla realizzazione di una pipeline con una topologia ben definita caratterizzata dalle seguenti componenti:

- **Spout**: Elemento della pipeline che è in grado di agganciarsi alla sorgente dati, realizzare un'operazione preliminare di filtraggio dei dati ed emettere uno stream di dati filtrati verso i nodi con competenze elaborative.
- **Bolt**: Elemento della pipeline che consuma lo stream di dati per svolgervi operazioni di:
  - Filtraggio
  - Summarization
  - Aggregazioni
  - Join
  - Salvataggio su un DB
  - Creazione di un nuovo stream

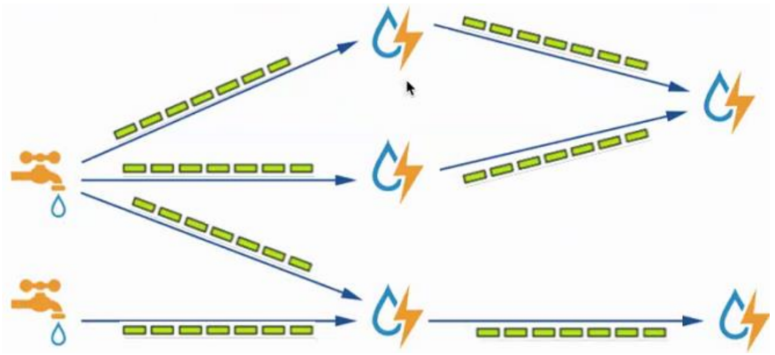
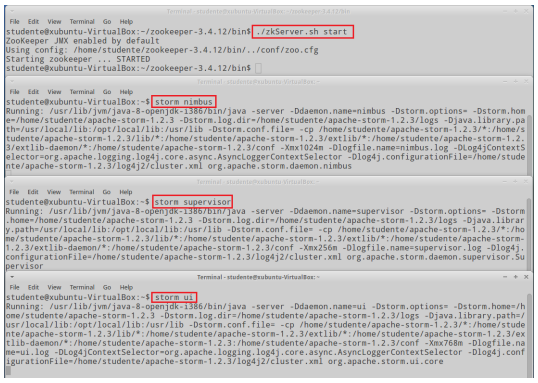


Figura 8: Esempio topologia di un'applicazione Storm

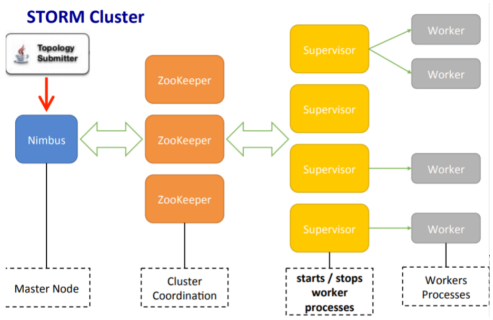
3.1 Inizializzazione Storm

Nel caso si utilizzasse un TopologySubmitter è necessario avviare esplicitamente i processi come segue.
La prima operazione effettuata è stata l'avvio onfigurazione, naming, fornitura del Server **ZooKeeper**, un servizio centralizzato per il mantenimento delle informazioni di cdi sincronizzazione distribuita e fornitura di servizi di gruppo. Sostanzialmente questo si comporta come il core principale del cluster che consente la comunicazione tra Nimbus e Supervisor.

- **Nimbus** fa le veci di un Master Node in quanto deteniene informazioni sulla topologia della rete, è il responsabile della distribuzione del codice nel cluster, si occupa dell'assegnazione di task alle macchine ed effettua monitoraggio per fallimenti.
- Il **Supervisor** ascolta il lavoro assegnato alla sua macchina; avvia/arresta i processi del worker secondo le necessità
- Il **Worker** è la componente terminale dell'infrastruttura e può essere uno Spout o un Bolt.
- Infine viene avviato il processo **UI** che consente la visualizzazione di informazioni sul cluster direttamente da browser, in quanto web applicaton.



((a)) Inizializzazione Storm



((b)) Architettura Storm

Figura 9: Risultato conteggio del numero minimo di visite

3.2 Applicazione Storm: Temperature Topology

L'applicazione Temperature Topology implementa una topologia Storm formata da uno Spout e tre Bolt, come in figura 10.

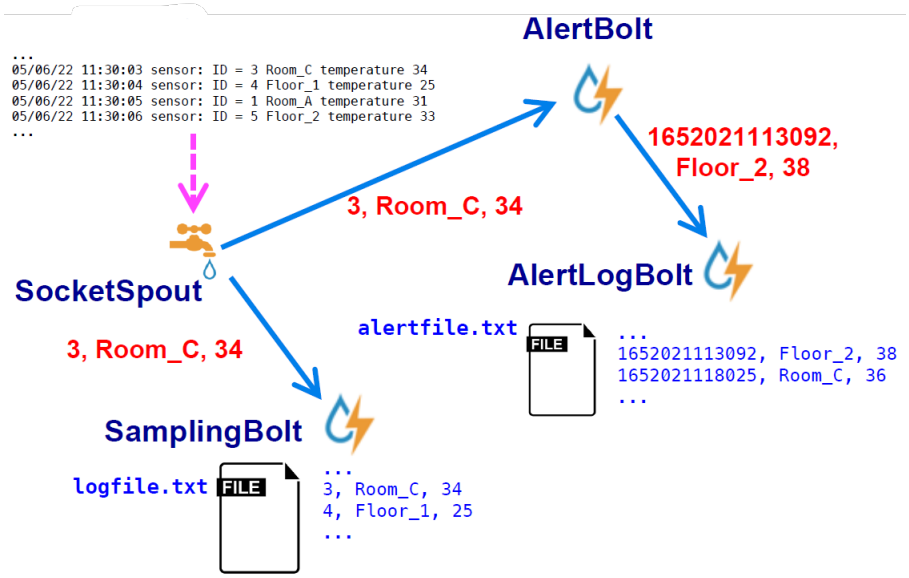


Figura 10: Topologia dell'applicazione Storm realizzata

SocketSpout rappresenta la sorgente dello stream di dati: legge da un server locale (temperatureStream.sh) le misure di temperatura ed emette delle tuple nel formato *ID, location, temperature* su un unico stream di uscita.

SamplingBolt è il bolt sottoscritto a SocketSpout che deve campionare il 20% dello stream letto. Quindi, applica una funzione hash sull'id, che può restituire un numero tra 0 e 5 (il server locale produce 5 id diversi) e campiona solo l'id la cui funzione di hash restituisce 0. I campioni vengono salvati su un file locale *logfile.txt*.

AlertBolt è il bolt sottoscritto a SocketSpout che per ogni tupla ricevuta deve effettuare un check sulla temperatura: se la temperatura è maggiore o uguale di 35, allora deve emettere sullo stream di output una tupla nel formato *timestamp, location, temperature*.

AlertLogBolt è il bolt sottoscritto a AlertBolt che memorizza le tuple ricevute su un file locale denominato *alertlog.txt*.

TemperatureTopology ha il compito di definire la configurazione della topologia di nostro interesse. In particolare, la classe *TopologyBuilder* permette di definire e istanziare la topologia. I metodi *setSpout*() e *setBolt*() consentono la registrazione degli Spout e dei Bolt definiti. La politica

di grouping è *shuffle grouping*, che prevede che le tuple dello stream vengano suddivise in maniera casuale tra gli executor associati ad un particolare Bolt.
La topologia è stata sottomessa utilizzando due alternative:

- *LocalCluster* che si occupa della simulazione del cluster Storm e, tramite una sua istanza è possibile sottomettere e avviare la topologia creata dal metodo createTopology() dell'oggetto builder;
- *TopologySubmitter* che emula un cluster reale in cui sono presenti Nimbus e Supervisor, i quali comunicano mediante il server Zookeeper.

L'implementazione dell'applicazione è consultabile nell'Appendice D.

3.3 Risultati delle elaborazioni con Storm

A seguito delle operazioni di dispiegamento ed esecuzione dei Jar risultanti sono stati prodotti in output due file:

- **logfile.txt** contenente i valori campionati al 20% dello stream totale basandosi sull'ID.
- **alertfile.txt** contenente tutti i valori generati dal server con valore superiore a 35.

Osservando la figura 11 è possibile notare come l'esecuzione dei due Bolt sia avvenuta in parallelo in quanto i valori riportati in entrambi file coincidono.

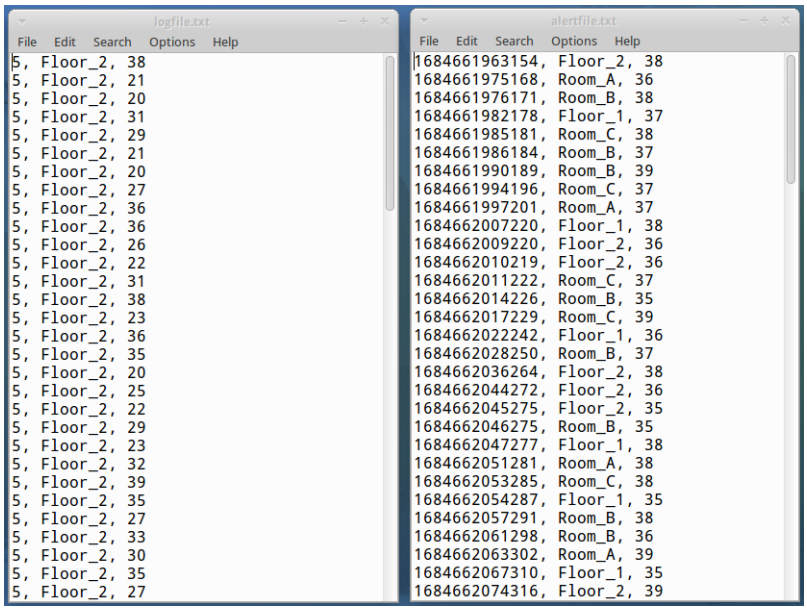


Figura 11: Risultati

Tramite la GUI, raggiungibile all'indirizzo localhost:8080, è stato possibile racogliere informazioni sulla topologia dispiegata e accedere ai valori relativi ai singoli Bolt e Spout.

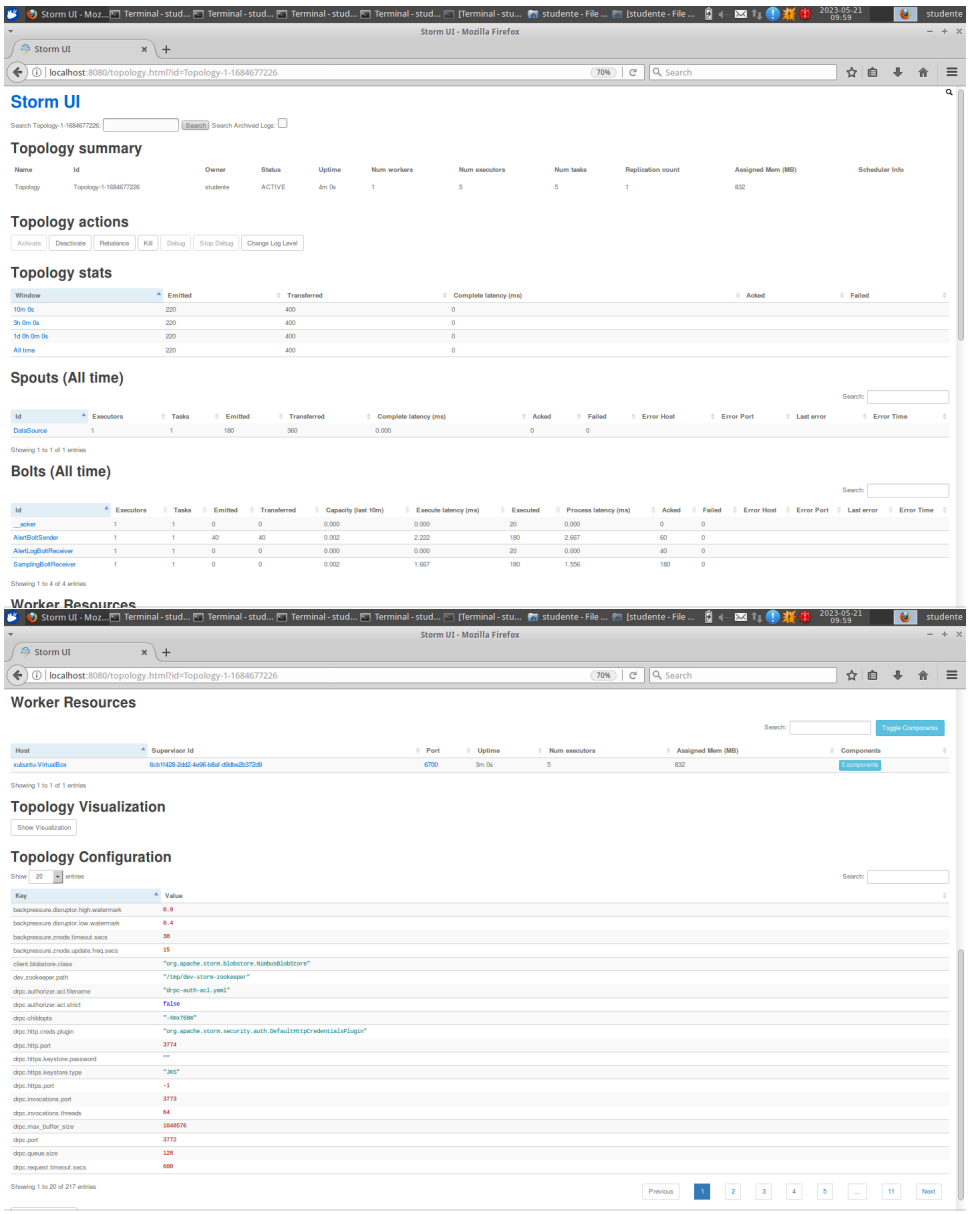


Figura 12: GUI

Nella sezione Topology Actions è possibile arrestare la topologia e rimuoverla da Storm. La topologia non viene più visualizzata e per poterla riattivare è necessario che venga nuovamente dispiegata.

Inoltre, è possibile consultare il grafo della topologia nella sezione Topology Visualization (figura 13).

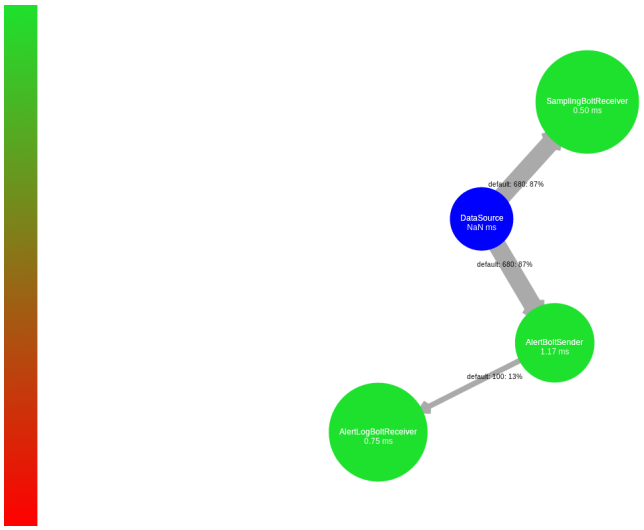


Figura 13: Grafo della topologia dispiegata

La componente blu rappresenta la prima componente della topologia. Il colore degli altri componenti della topologia indica se il componente supera la capacità del cluster: i componenti rossi rappresentano un collo di bottiglia e i componenti verdi sono componenti che operano all'interno capacità. Infine, lo spessore delle linee è un indice del flusso di dati: linee più spesse tra i componenti indicano flussi di dati più grandi.

Appendice A Somma delle visite per mese

A.1 Mapper

```
public class VisitsMap extends Mapper<LongWritable, Text, Text, IntWritable>{
    public void map ( LongWritable key ,Text value ,Context ctx) throws
         IOException,InterruptedException{
        String[] st = value.toString().split(" ");
        if( st != null){
            String month = new String(st[0]);
            int  visits = new Integer(st[2]);
            ctx.write(new Text(month), new IntWritable(visits));
        }
    }
}
```

A.2 Reducer

```
public class VisitsReduce extends Reducer<Text, IntWritable,Text, IntWritable>{
    public void reduce(Text month, Iterable <IntWritable> list , Context ctx) throws IOException,
         InterruptedException {
        int sum = 0;
        for(IntWritable value: list){
            sum += value.get();
        }
        ctx.write(month, new IntWritable(sum));
    }
}
```

A.3 Driver

```
public class VisitsDriver {
    public static void main(String[] args) throws Exception{
        Job job = Job.getInstance();
        job.setJarByClass(visits.VisitsDriver.class);
        job.setJobName("SumVisits");
        job.setNumReduceTasks(4);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        job.setMapperClass(visits.VisitsMap.class);
        job.setReducerClass(visits.VisitsReduce.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        System.exit( job.waitForCompletion(true) ? 0:1);
    }
}
```

Appendice B Conteggio del numero di siti unici

B.1 Mapper

```
public class SitesMap extends Mapper<LongWritable, Text, Text, Text>{
    public void map ( LongWritable key ,Text value ,Context ctx) throws
         IOException,InterruptedException{
        String[] st = value.toString().split(" ");
        if( st != null){
            String sites = new String(st[1]);
            ctx.write(new Text("distsites"), new Text(sites));
        }
    }
}
```

B.2 Reducer

```
public class SitesReduce extends Reducer<Text, Text,Text, IntWritable>{
    public HashSet<String> set = new HashSet<String>();
    public void reduce(Text site, Iterable <Text> list , Context ctx) throws IOException,
         InterruptedException {
        for(Text value: list){
            set.add(value.toString());
        }
        ctx.write(site, new IntWritable(set.size()));
    }
}
```

B.3 Driver

```
public class SitesDriver {
    public static void main(String[] args) throws Exception{
        Job job = Job.getInstance();
        job.setJarByClass(sites.SitesDriver.class);
        job.setJobName("SumVisits");
        job.setNumReduceTasks(4);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        job.setMapperClass(sites.SitesMap.class);
        job.setReducerClass(sites.SitesReduce.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);
        System.exit( job.waitForCompletion(true) ? 0:1);
    }
}
```

Appendice C Conteggio del numero minimo di visite

C.1 Mapper

```
public class MinVisitsMap extends Mapper<LongWritable, Text, Text, IntWritable>{
    public void map ( LongWritable key ,Text value ,Context ctx) throws
         IOException,InterruptedException{
        String[] st = value.toString().split(" ");
        if( st != null){
            int  visits = new Integer(st[2]);
            ctx.write(new Text("minvisits"), new IntWritable(visits));
        }
    }
}
```

C.2 Reducer

```
public class MinVisitsReduce extends Reducer<Text, IntWritable,Text, IntWritable>{
    public List<Integer> listA = new ArrayList<Integer>();
    public void reduce(Text string, Iterable <IntWritable> list , Context ctx) throws IOException,
         InterruptedException {
        for(IntWritable value: list){
            listA.add(value.get());
        }
        int min = Collections.min(listA);
        ctx.write(string, new IntWritable(min));
    }
}
```

C.3 Driver

```
public class MinVisitsDriver {
    public static void main(String[] args) throws Exception{
        Job job = Job.getInstance();
        job.setJarByClass(min.MinVisitsDriver.class);
        job.setJobName("SumVisits");
        job.setNumReduceTasks(4);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        job.setMapperClass(min.MinVisitsMap.class);
        job.setReducerClass(min.MinVisitsReduce.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        System.exit( job.waitForCompletion(true) ? 0:1);
    }
}
```

Appendice D Applicazione Storm: TemperatureTopology

D.1 SocketSpout.java

```
public class SocketSpout extends BaseRichSpout {

 private SpoutOutputCollector collector;
 private BufferedReader buf;

 @Override
 public void nextTuple() {
  String s= null;
  //impacchettiamo la stringa da inviare
  try {
   s=buf.readLine();
  } catch (IOException e) {
   e.printStackTrace();
  }

  //Selezione dei campi dalla stringa letta: 05/06/22 11:30:04 sensor: ID = 4 Floor_1 temperature 25
  String [] st =s.split(" ");
  int id= new Integer(st[5]);
  String location= st[6];
  int temperature = new Integer(st[8]);

  //Emit della tupla
  collector.emit(new Values (id, location, temperature));
  System.out.println("spout: emesso: " + id+ ", " + location + ", " + temperature + " eseguito da "
       + Thread.currentThread().getName());
  Utils.sleep(1000);
 }

 @Override
 public void open(Map arg0, TopologyContext arg1, SpoutOutputCollector arg2) {
  collector = arg2;
  try {
   Socket s = new Socket ("127.0.0.1", 7777);
   buf= new BufferedReader(new InputStreamReader(s.getInputStream()));

  } catch (IOException e) {
   e.printStackTrace();
  }
 }

 @Override
 public void declareOutputFields(OutputFieldsDeclarer arg0) {
  //definzione del formato della tupla da emettere
  arg0.declare(new Fields("ID", "location", "temperature"));
 }
}
```

D.2 SamplingBolt.java

```
public class SamplingBolt extends BaseRichBolt {

 private OutputCollector collector;
 private PrintWriter pw;

       @Override
 public void execute(Tuple arg0) {
  // 1) lettura tupla e processing: "ID", "Location", "Temperature"
  int i = arg0.getIntegerByField("ID");
  String location = arg0.getString(1);
  int temperature = arg0.getIntegerByField("temperature");

  //stampiamo il valore della tupla
  System.out.println("    Bolt ricevuto "+ i + ", " + location + ", " + temperature + " da "
       +arg0.getSourceComponent() + " - " + arg0.getSourceStreamId());

  //Salvataggio delle tuple sul file logfile.txt
  //funzione di hash per campionare solo il 20% delle tuple ricevute
  int hash= Math.abs(i) % 5;
  if(hash == 0){
   pw.println(i +", " + location + ", " + temperature );
   pw.flush();
  }

  //ack della tupla
  collector.ack(arg0);
 }

 @Override
 public void prepare(Map arg0, TopologyContext arg1, OutputCollector arg2) {
  // TODO Auto-generated method stub
  collector=arg2;

  try {
   pw= new PrintWriter("/home/studente/logfile.txt");
  } catch (FileNotFoundException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
  }
 }

 @Override
 public void declareOutputFields(OutputFieldsDeclarer arg0) {
 }

}
```

D.3 AlertBolt.java

```
public class AlertBolt extends BaseRichBolt {

 private OutputCollector collector;
 private PrintWriter pw;

 @Override
 public void execute(Tuple arg0) {
  // TODO Auto-generated method stub
  // 1) lettura tupla
  int i = arg0.getIntegerByField("ID");
  String location = arg0.getString(1);
  int temperature = arg0.getIntegerByField("temperature");

  // stampiamo il valore della tupla
  System.out.println("    Bolt ricevuto "+ i + ", " + location + ", " + temperature + " da "
       +arg0.getSourceComponent() + " - " + arg0.getSourceStreamId());

  //2) emit della tupla con temperatura maggiore di 35
  if(temperature >= 35){
   long timestamp=System.currentTimeMillis();
   collector.emit(new Values (timestamp, location, temperature));
   System.out.println( "            ALERT: "+ timestamp + ", " + location + ", " + temperature );

   //ack della tupla
   collector.ack(arg0);
  }
 }

 @Override
 public void prepare(Map arg0, TopologyContext arg1, OutputCollector arg2) {
  // TODO Auto-generated method stub
  collector=arg2;
 }

 @Override
 public void declareOutputFields(OutputFieldsDeclarer arg0) {
  // TODO Auto-generated method stub
  //definzione del formato della tupla emessa
  arg0.declare(new Fields("timestamp", "location", "temperature"));
 }
}
```

D.4 AlertLogBolt.java

```
public class AlertLogBolt extends BaseRichBolt {

 private OutputCollector collector;
 private PrintWriter pw;

 @Override
 public void execute(Tuple arg0) {
  // TODO Auto-generated method stub
  // 1) lettura tupla e processing: "ID", "Location", "Temperature"
  long timestamp = arg0.getLongByField("timestamp");
  String location = arg0.getString(1);
  int temperature = arg0.getIntegerByField("temperature"); //se non ricordiamo il mapping tra la
       posizione e il count.


  //stampiamo il valore della tupla
  System.out.println("            ALERT LOG "+ timestamp + ", " + location + ", " + temperature + "
       da " +arg0.getSourceComponent() + " - " + arg0.getSourceStreamId());

  //Salvataggio delle tuple sul file alertfile.txt
  pw.println(timestamp +", " + location + ", " + temperature );
  pw.flush();

  //ack della tupla
  collector.ack(arg0);
 }

 @Override
 public void prepare(Map arg0, TopologyContext arg1, OutputCollector arg2) {
  // TODO Auto-generated method stub
  collector=arg2;

  try {
   pw= new PrintWriter("/home/studente/alertfile.txt");
  } catch (FileNotFoundException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
  }
 }

 @Override
 public void declareOutputFields(OutputFieldsDeclarer arg0) {

 }
}
```

D.5 TemperatureTopology

```
public class TemperatureTopology {

 public static void main(String[] args) throws InterruptedException, AlreadyAliveException,
      InvalidTopologyException, AuthorizationException {
  // TODO Auto-generated method stub
  Config cfg= new Config();
  cfg.setDebug(false);

  // Topologia DAG.
  TopologyBuilder builder =new TopologyBuilder();

  builder.setSpout("DataSource", new SocketSpout());

  builder.setBolt("SamplingBoltReceiver", new SamplingBolt()).shuffleGrouping("DataSource");
  builder.setBolt("AlertBoltSender", new AlertBolt()).shuffleGrouping("DataSource");
  builder.setBolt("AlertLogBoltReceiver", new AlertLogBolt()).shuffleGrouping("AlertBoltSender");

  //invio topologia:
  // 1) creazione local cluster


  LocalCluster cluster = new LocalCluster();
  cluster.submitTopology("TemperatureTopology",cfg, builder.createTopology());
  Thread.sleep(120000);
  cluster.shutdown();


  // 2) Storm Submitter
  //StormSubmitter.submitTopology("Topology",cfg, builder.createTopology());
 }

}
```