LOMBARDIA AIR POLLUTION

processing streaming data with Spark Structured Streaming

postgraduate project by Tomasz Kubat

Poznań University of Technology

2019

INFO

1 Info

1 Scope&Out of scope

Scope

This document is an written documentation for project prepared during “Data Warehouses and Data Analyses” postgraduate studies. The documentations contains description of:

- main issue and potential solution,

- software installation and configuration,

- running the Spark cluster and applications,

- potential upgrades for production deployment.

Out of scope

Out of project scope was:

- configuration of data broker (e.g. Apache Kafka Streams),

- creating final application for end user.

2 Project archives

Code examples (sources and jar files) can be found on github:

https://github.com/tomaszkubat/SparkStreaming

3 Literature

1. ARPA Lombardia – data source, https://dati.lombardia.it/stories/s/auv9-c2sj, 2019
2. Spark 2.4.0 Documentation, https://spark.apache.org/, 2019
3. Spark Configuration, https://medium.com/ymedialabs-innovation/apache-spark-on-a-multi-node-cluster-b75967c8cb2b, 2019

PURPOSE

1 Purpose

Issue

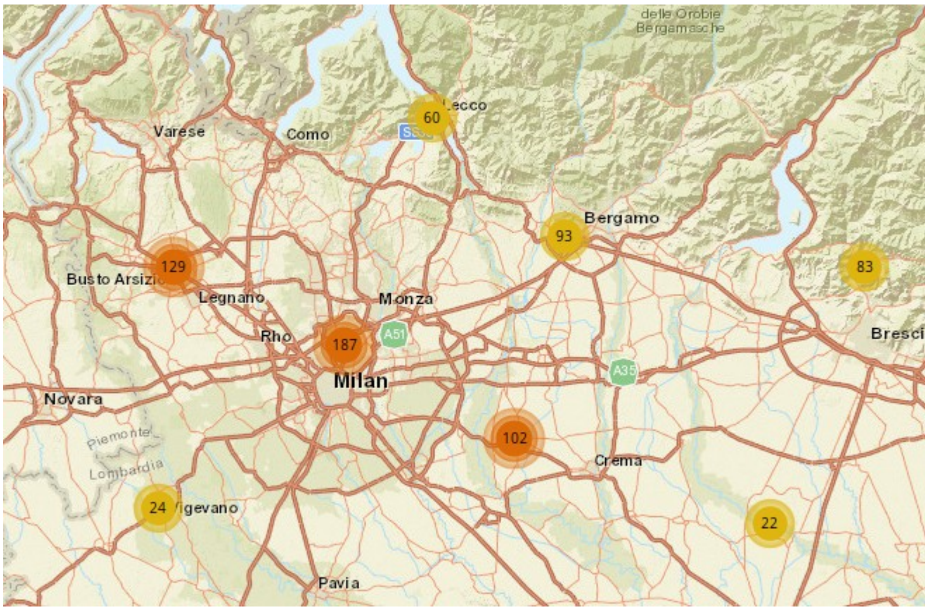
Our goal is to analyze air pollution data from Lombardia (Italy) publicized by ARPA Agency (Regional Agency for Environmental Protection).

We face few potential issues, because data:

- are generated hourly by ~950 sensors simultaneously (**frequency**),
- concern various (17) sensor types (**diversity**),
- have broad history (data back to 1968) and to prepare reasonable solution presentation it’s required to load a data from a few years (**amount**).

This is a considerable amount of data, since it concerns the history of surveys of several tens of years - to date - of all the survey probes in the territory, as well as some interpolations made directly by ARPA.

Img: Lomardia – sensors main localization



https://www.dati.lombardia.it/Ambiente/Mappa-stazioni-qualit-dell-aria/npva-smv6

More information about ARPA and datasets could be found here: https://dati.lombardia.it/stories/s/auv9-c2sj

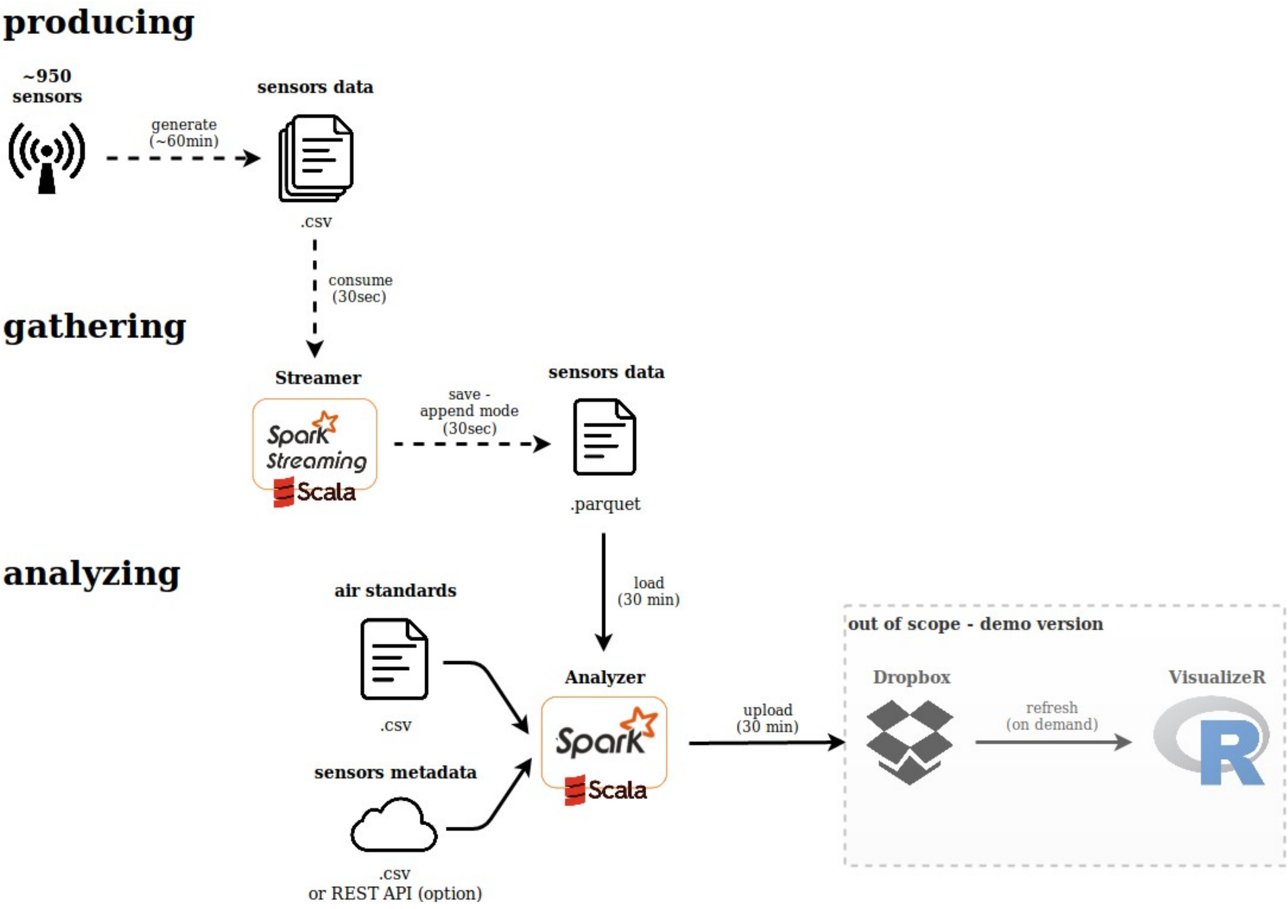
Architecture

In general – our goal is to:

- handle streaming data,
- manipulate them,
- analyze and return an output in human readable/friendly form.

To achieve this purpose we have to operate on set of different technologies – suggested architecture is presented below. Whole concept was divide into three main, logical parts: producing, gathering and analyzing data. As we see three main applications occurs: **Streamer**, **Analyzer**, **VisualizeR.**

Graph: Architecture



** Getting sensors metadata from REST API is an option implemented in the Scala code; by default flat file source is being used.*
*** Final steps (uploading/downloading from Dropbox and visualizing data) are out of scope. Only in demo visualization version was prepared.*

Producing

Because we are operating on historical data, we have to simulate streaming behavior – we use simple bash scripts[1] to:

- split data into smaller batches (‘**splitTestData.sh**’)

- continuously move the data to to input localization with given refresh interval and save as csv files **(‘simnulateDataStream.sh’).**

Gathering

Streamer is a Scala application which is responsible for manipulating streaming data (e.g. reading, removing duplicates, handling late data, writing). It uses user friendly Spark Structured Streaming API, which allows operating on data frames instead of RDDs[2]. Unfortunately, we are unable to implement a few interesting solutions because:

- we are operating on crafted stream and generated timestamp strongly differs than current timestamp,
- current version of Spark Structured Stream has some restrictions, especially when aggregating the data.

Analyzing

Analyzer operates on batch data instead of data stream. It is responsible for:

- merging sensors data and metadata with air standards,
- running pre-filters operations,
- aggregating data,
- calculating simple metrics,
- classifying sensor state.

VisualizeR is a simple app (a demo) which only point the need of visualization.

[1] Apache Kafka Streams technology was out of cope.

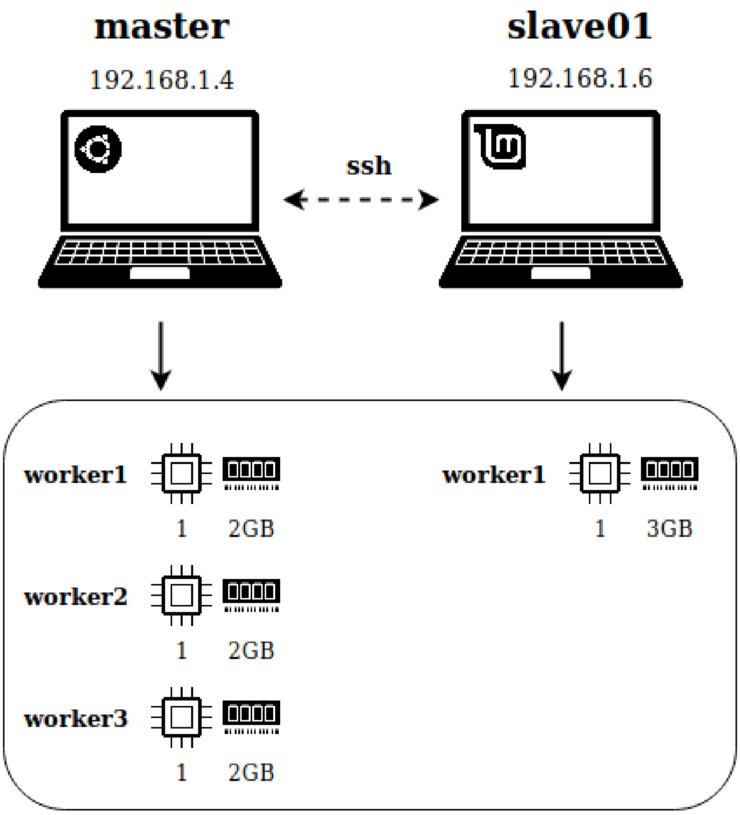
[2] Every time I see RDD programming concept, I vomit on my keyboard.

Apache Spark Cluster

Let’s see only our simply Spark Cluster master/slave schema – we use the smallest possible set of machines – only two. Machines are physically separated computers are set on Linux OS distributions (Ubuntu 18.04 and LinuxMint) and are communicating by ssh connection.

Totally, we gathered resources: 4 cores and 9GB memory.

Graph: Spark Cluster schema

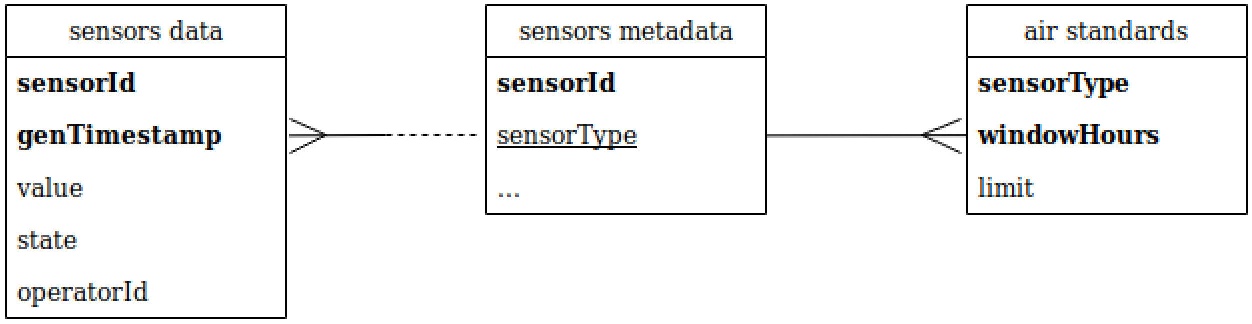


Info about data set

Full information about data sets could be found here: https://dati.lombardia.it/stories/s/auv9-c2sj But generally, we will be operating on three main input data sets:

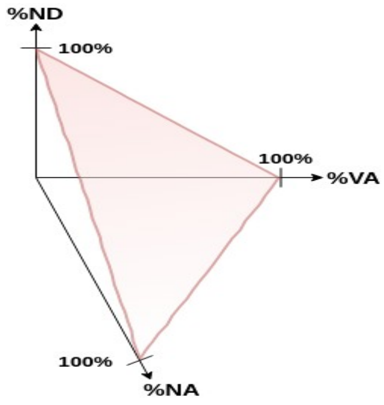
- sensors data, produced constantly,
- sensors metadata (sensor type, localization, e.t.c.),
- air standards for each sensor type described by time window in hours and pollution limit; one sensor type have one one more related air standards.

Graph: ERD diagram



Worth to know – data are described by “State” variable, which can appear in two possible states: VA (valid) and NA (not valid). Naturally, the third state occurs: ND, which represents lack of data, when sensor is not working.

Graph: Set of possible data states



CONFIGURATION

2 Configuration

1 Overview

This section describes the process of installing software and configuring the Spark Cluster and all related components. The milestones are described below:

- software installation:
    - Java (JDK 10.0.2)
    - Scala (2.11.12)
    - Spark (2.4.0 with Hadoop 2.7)
- Spark configuration:
    - add exports in .bashrc
    - create user
    - add entries in *hosts* file
    - configure ssh connection
- final touches (.bashrc configuration)

The installation/configuration process is rather straightforward, but bear in mind, that simple mistakes can stop you progress. Be careful and **keep installing order,** e. g. Java should be installed before Scala, and installation of Scala should be followed by installation of Spark. **Some steps should be reproduced on both master and all the slaves.** On the other hand a few configuration actions should be performed on the master side only, so **read the instructions carefully** and follow the descriptions shown in the brackets – good luck!

2 Software installation

Install Java (master and slaves)

Install Java and check installed Java version.

```
$ sudo apt-get install default-jre
$ java -version
openjdk version "10.0.2" 2018-07-17
```

Install Scala (master and slaves)

Install Scala and installed Scala version[3].

```
$ sudo apt-get installa scala
$ scala -version
Scala code runner version 2.11.12 -- Copyright 2002-2017, LAMP/EPFL
```

Install Spark (master and slaves)

Download the Spark 2.4.0 with Hadoop 2.7.

```
$ wget http://www-us.apache.org/dist/spark/spark-2.4.0/spark-2.4.0-bin-hadoop2.7.tgz
```

Untar package

```
$ tar xvf spark-2.4.0-bin-hadoop2.7.tgz
```

Move files to */usr/local/spark*

```
$ sudo mv spark-2.4.0-bin-hadoop2.7 /usr/local/spark
```

[3] Some guides suggests, that Scala installation is only optional, because scala-shell is installed in Spark by default. Installing Scala is a good idea, if you also would like use Scala without Spark.

3 Spark configuration

Add exports in .bashrc (master and slaves)

Add a few exports to the *~/.bashrc* file:

```
$ sudo vim ~/.bashrc
# add lines
export SPARK_HOME=/usr/local/spark
export PATH=$PATH:/$SPARK_HOME/bin
```

Reload *~/.bashrc* to enable added exports.

```
$ source ~/.bashrc
```

Add entries in hosts file (master and slaves)

Edit /*etc/*hosts file:

```
$ sudo vim /etc/hosts
```

Add entries of master and slaves in hosts file:
<MASTER-IP> master
<SLAVE01-IP> slave01
<SLAVE02-IP> slave02

Example:

```
127.0.0.1     localhost
127.0.1.1     mycomputer
192.168.1.4   master
192.168.1.3   slave01
192.168.1.6   slave02
```

To check your IP use command shown below:

```
$ ifconfig
# or
$ hostname -I
```

Create user (master and slaves)

To simplify configuration of the ssh connection on the cluster, on each node create an user with the same username and password. This solution allows you not do specify username and password to establish proper connection between the nodes.

Create new user.

```
$ su – root
$ adduser usr_spark
```

Add new user to the sudo group

```
$ usermod -aG sudo usr_spark
$ exit
```

Login as usr_spark

```
$ su – usr_spark
```

Configure SSH (master only)

To enable SSH communication you have to generate private ssh key and distribute a file between all the machines – computers have to encrypt/decrypt information.

Install open SSH server-client:

```
$ sudo apt-get install openssh-server openssh-client
```

Generate key pairs:

```
$ ssh-keygen -t rsa -P “”
```

Copy the content of .ssh/id_rsa.pub from master and redistribute it to .ssh/authorized_key on master and slaves:

```
$ cd ~/.ssh
$ cp id_rsa.pub authorized_keys
$ scp id_rsa.pub usr_spark@slave01:/home/usr_spark/.ssh/authorized_keys
```

Check the ssh connection:

```
$ ssh slave01
$ ssh slave02
```

Edit spark-env.sh file (master only)

Navigate to $SPARK_HOME and create spark-env.sh file which contains main Spark parameters:

```
$ cd $SPARK_HOME/conf
$ cp spark-env.sh.template spark-env.sh
$ sudo vim spark-env.sh
```

Add lines:

export SPARK_MASTER_HOST='<MASTER-IP>'

export JAVA_HOME=<Path_of_JAVA_installation>

Example:

export SPARK_MASTER_HOST='192.168.1.3'

export JAVA_HOME=/opt

Edit slaves file (master only)

Add workers in the slaves file:

```
$ cd $SPARK_HOME/conf
$ sudo vim slaves
```

Add lines:

master

slave01

slave02

RUNNING APPLICATIONS

3 Running applications

1 Prerequisites

To enable running the application add a few exports in ~/.bashrc file:

```
export MASTER_IP=`hostname -I | awk '{print $1}'`
export SPARK_APP=/home/usr_spark/Projects/SparkStreaming
```

Retyping the same commands million times during the development/tests isn’t the most exciting activity ever. Save your time and make your work easier[4]! Add following lines to the ~/*.bashrc* file.

Aliases to fast traversing the directories:

```
alias    ..='cd ./../'
alias   ...='cd ./../../'
alias  ....='cd ./../../../'
alias .....='cd ./../../../../'
```

For fast testing your application:

```
export runapp=’cd $SPARK_HOME && ./bin/spark-submit --class
tk.streaming.Runner --master local[*]
/home/usr_spark/Projects/Spark/StreamingApp/target/scala-2.11/streaminga
pp_2.11-0.1.jar’
```

2 Assumptions

- Air standards do not change in time (but it’s possible to manipulate them in csv file)

[4] “Good programmer is lazy one” - Tadeusz’s Moorzy first golden rule.

3 Running cluster

Running cluster

Spark Standalone mode allows you to use predefined bash scripts to operate on cluster. To run cluster type:

```
$ cd $SPARK_HOME
$ ./sbin/start-all.sh
```

How to check if my cluster is working?

You can check running processes on Java Virtual Machine:

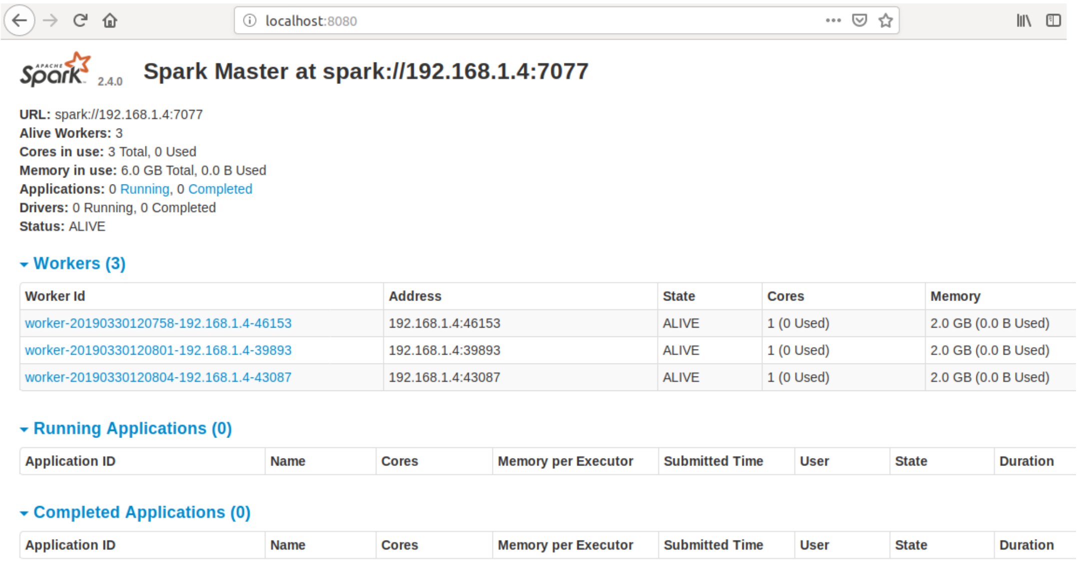
```
$ jps
```

Example result (one master and one worker processes are running on JVM):

```
$ jps
7220 Worker
6937 Master
7467 Jps
```

Another way is to use web-gui, which by default is deployed in localhost on 8080 port. Type 'http://localhost:8080' in your web browser.

Img. Spark web-gui



Spark Master at spark://192.168.1.4:7077

URL: spark://192.168.1.4:7077
Alive Workers: 3
Cores in use: 3 Total, 0 Used
Memory in use: 6.0 GB Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (3)

Worker Id	Address	State	Cores	Memory
worker-20190330120758-192.168.1.4-46153	192.168.1.4:46153	ALIVE	1 (0 Used)	2.0 GB (0.0 B Used)
worker-20190330120801-192.168.1.4-39893	192.168.1.4:39893	ALIVE	1 (0 Used)	2.0 GB (0.0 B Used)
worker-20190330120804-192.168.1.4-43087	192.168.1.4:43087	ALIVE	1 (0 Used)	2.0 GB (0.0 B Used)

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration

Completed Applications (0)

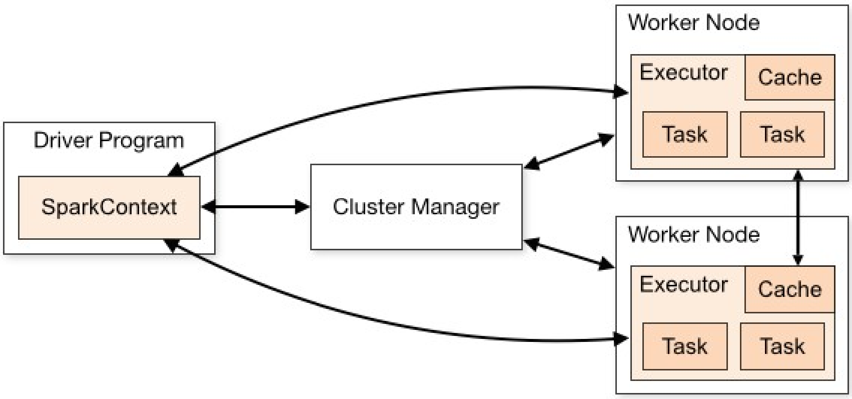
Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration

4 Running applications

Terminology

“Why the author gives the information about terminology at the end of the documentation?”. When I started learning Spark I used to read Spark documentation and watch a lot of instructional movies. They focus on theoretical approach and serve a great deal of unintuitive information. I started to fully understand the terminology, when I was manipulating on Spark web-gui. **I recommend you experimenting with Spark web-gui – it contains a lot of information**!

Graph. Spark cluster architecture



https://spark.apache.org/docs/latest/cluster-overview.html

Spark applications run as independent sets of processes on a cluster, coordinated by the **SparkContext**[5] object in your main program (called the *driver program*). When you run the application on the Spark Cluster, the **Driver Program**[6] lives begin. Driver contacts with Cluster Manager to identify free resources on the cluster. Next, if possible, SparkContext send a tasks to the **Executors** localized on worker nodes.

Each application gets its own executor processes, which stay up for the duration of the whole application and run tasks in multiple threads. **One application is isolated from another**.

[5] SparkSontext is an object which enables to conduct main operations, like running SQL queries.
[6] Driver should be run close to the worker nodes, preferably on the same local area network.

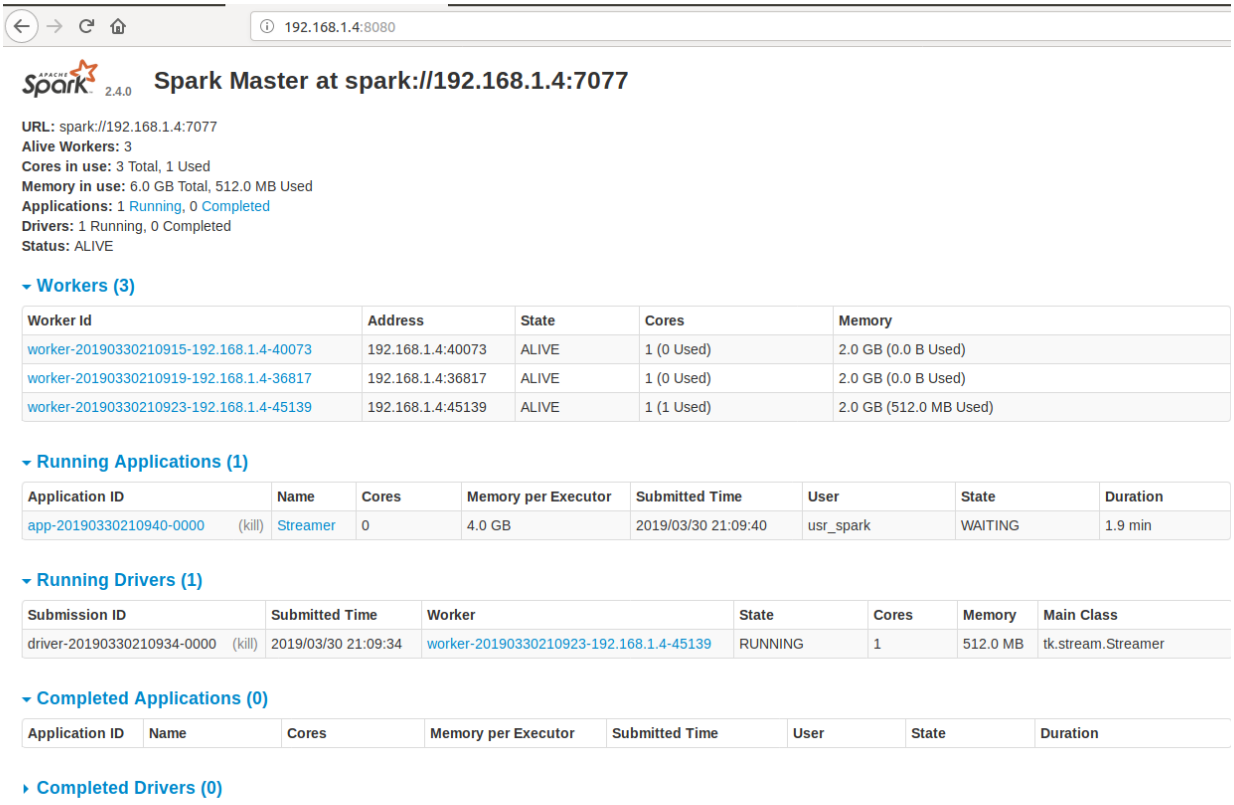
Running application

running application - Loading Configuration from a File. The `spark-submit` script can load default Spark configuration values from a properties file and pass them on to your application. By default, it will read options from `conf/spark-defaults.conf` in the Spark directory.

```
${SPARK_HOME}/bin/spark-submit \
    --class tk.stream.Streamer \
    --master ${MASTER} \
    --deploy-mode cluster \
    --driver-memory 512M \
    --conf spark.executor.memory=4g \
    --conf spark.cores.max=3 \
    $DIR_TARG/scala-2.11/sparkapps_2.11-0.1.jar ${MASTER} $
{STREAMER_REFRESH_INTERVAL} ${DIR_DATA}
```

Arguments passed before the .jar file will be arguments to the JVM, where as arguments passed after the jar file will be passed on to the user's program.

Graph. Application gui



Spark Master at spark://192.168.1.4:7077

URL: spark://192.168.1.4:7077
Alive Workers: 3
Cores in use: 3 Total, 1 Used
Memory in use: 6.0 GB Total, 512.0 MB Used
Applications: 1 Running, 0 Completed
Drivers: 1 Running, 0 Completed
Status: ALIVE

Workers (3)

Worker Id	Address	State	Cores	Memory
worker-20190330210915-192.168.1.4-40073	192.168.1.4:40073	ALIVE	1 (0 Used)	2.0 GB (0.0 B Used)
worker-20190330210919-192.168.1.4-36817	192.168.1.4:36817	ALIVE	1 (0 Used)	2.0 GB (0.0 B Used)
worker-20190330210923-192.168.1.4-45139	192.168.1.4:45139	ALIVE	1 (1 Used)	2.0 GB (512.0 MB Used)

Running Applications (1)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20190330210940-0000 (kill)	Streamer	0	4.0 GB	2019/03/30 21:09:40	usr_spark	WAITING	1.9 min

Running Drivers (1)

Submission ID	Submitted Time	Worker	State	Cores	Memory	Main Class
driver-20190330210934-0000 (kill)	2019/03/30 21:09:34	worker-20190330210923-192.168.1.4-45139	RUNNING	1	512.0 MB	tk.stream.Streamer

Completed Applications (0)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration

Completed Drivers (0)

Monitoring application

Each driver program has a web UI, by default set on port 4040, that displays information about running tasks, executors, and storage usage. If you run multiple applications on one cluster, other applications get ports numbers 4041, 4042, and so on, respectively.

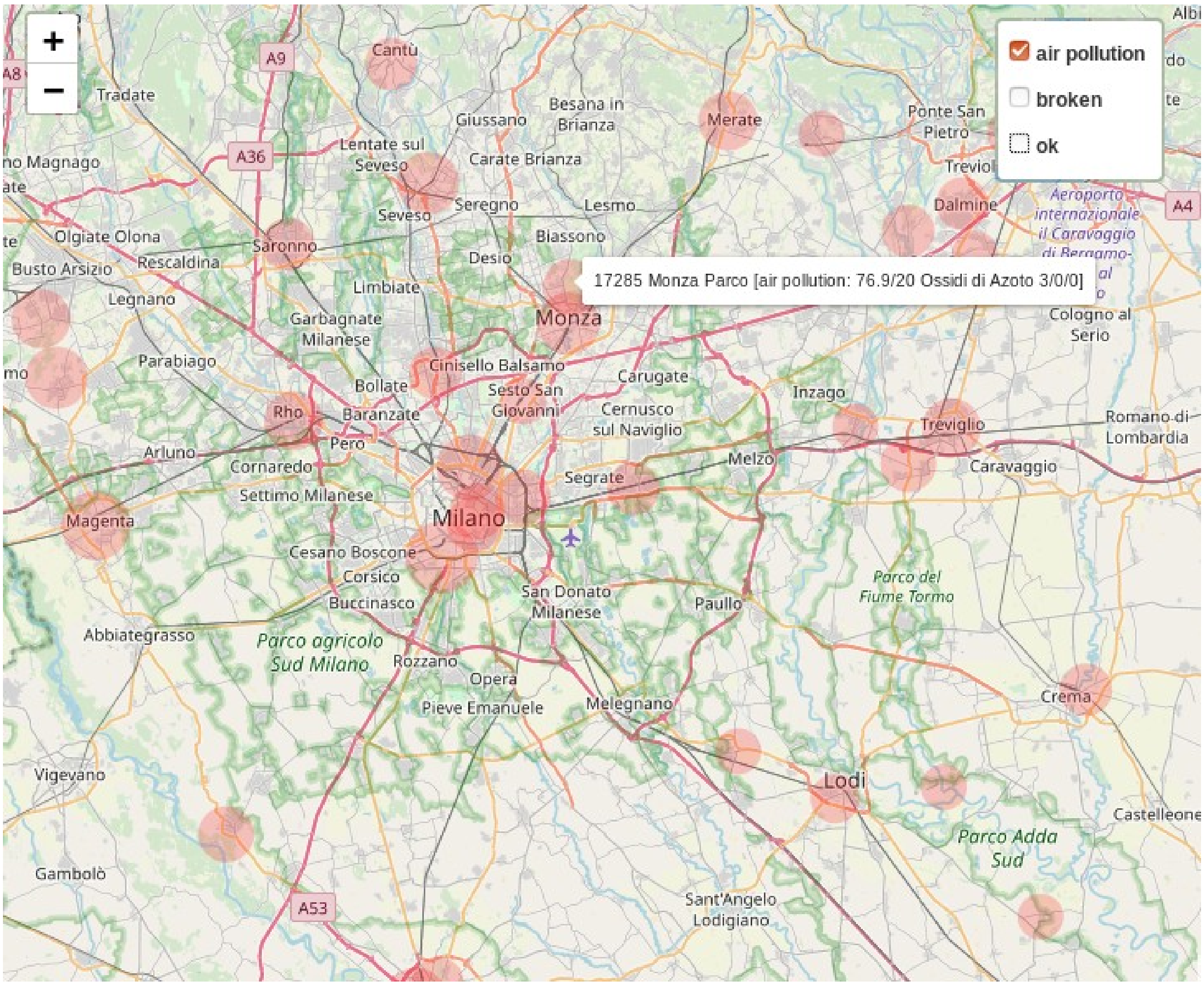
More details about running the application on Spark Cluster can be found in official documentation:

https://spark.apache.org/docs/latest/spark-standalone.html#connecting-an-application-to-the-cluster

Application results

Visualization:

Img. VisualizeR



☑ air pollution

□ broken

□ ok

17285 Monza Parco [air pollution: 76.9/20 Ossidi di Azoto 3/0/0]

POSSIBLE UPGRADES

4 POSSIBLE UPGRADES

Solution described in this documentations is probably not the most sophisticated, but it never had to be! My goal was to prepare working, well organized and reproducible example of spark application. When starting on the production, you may consider some interesting upgrades mentioned below.

Streaming real data (Kafka Streams)

Data providing solution prepared in this documentation was based on the simple bash scripts. They enables to go-live the historical data to present some of Spark abilities. Of course this solution can fail the ‘final exam’ in the production environment realities.

It’s recommended to use a data broker like Kafka Streams, which is responsible for getting data directly from sensors (producers) and pass them, in real time, to the Spark Streaming application (consumer).

Faster/easier Spark deployment (Docker)

Remember: “typing is better than clicking” and “running script is better than typing every command”. Installing/configuring software on the machines can be fun, but when you make the same activities tenth time, it can be boring and eventually frustrating.

You may consider using spark docker file/image to run Spark instantly on the new machines. Remember, that running Spark from Docker image is a bit different than traditional approach. More information could be found in this movie:

https://www.youtube.com/watch?v=CZ3WArfiTkk

Different Spark Cluster management (Messos, Kubernetes)

Spark Standalone is a simple mode to run Spark cluster. In production you would like, probably, to use your favorites technologies to manage cluster. It’s possible to use Apache Messos or Kubernetes.

Distributed or remote data storage (HDFS, Cloud)

Storing data in files is ok until you don’t need more complex solutions, to: ensure data availability, backups, additional data operations. Consider HDFS or cloud storage.

Better and more fault-tolerant Scala code

Streaming live data, where creation timestamp is close to current timestamp, opens additional Spark possibilities:

- handling late data and watermarking
- deduplicating data

More information can be found here: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

R application deployment

In this project we use demo version of R application. To fully use R code we shall deploy R application – locally on in public service like https://www.shinyapps.io/

Cloud data storage

To ensure easy data accessibility we can use Dropbox or similar cloud data storage.