

# Progetto Finale Big Data

**Manuel Granchelli**

**man.granchelli@stud.uniroma3.it**

*Matricola: 512406*

*Dipartimento di Ingegneria Informatica*

*Università degli Studi Roma Tre*

**Repo GitHub:** <https://github.com/mgranchelli/bigdata-finalproject>

## Introduzione

In questo rapporto verranno illustrate le procedure di analisi, elaborazione, salvataggio e visualizzazione di dati catturati da sensori IoT installati in un campo di peschi. Non avendo accesso diretto ai sensori sul campo, i dati sono stati raccolti in file formato JSON, un file per ogni sensore, e tramite uno script Python è stato simulato il flusso streaming dei dati inviati dai sensori. Lo script che genera il flusso di dati e tramite Kafka invia messaggi a Spark Streaming che si occupa di processare i dati. Una volta processati i dati vengono salvati all'interno di un Time Series Database (TSDB), questo perchè i dati prodotti dai sensori IoT sono tutti associati ad un timestamp e sono dati che devono essere monitorati nel tempo. I Time Series Database (TSDB) si prestano molto bene per essere utilizzati in queste situazioni. Nel seguente progetto si è scelto di utilizzare InfluxDB.

I dati memorizzati vengono a loro volta letti da Grafana che è uno strumento di visualizzazione di serie temporali. Grafana è accessibile via browser ed è possibile generare una dashboard contenente i grafici dei dati memorizzati su InfluxDB.

L'architettura utilizzata in questo progetto è stata testata utilizzando docker compose che permette di realizzare applicazioni composte da più contenitori che comunicano tra di loro. Ogni contenitore è stato creato per svolgere una determinata attività con una determinata tecnologia.

Tutto il materiale utilizzato è disponibile nel repo GitHub:

<https://github.com/mgranchelli/bigdata-finalproject>.

# Indice

<b>1</b>	<b>Dati Paradise</b>	<b>1</b>
<b>2</b>	<b>Architettura e tecnologie usate</b>	<b>3</b>
2.1	Dati da sensori IoT . . . . .	3
2.1.1	Pseudocodice . . . . .	4
2.1.2	Log - Output . . . . .	5
2.2	Messaging Layer . . . . .	5
2.3	Processing Layer . . . . .	5
2.3.1	Pseudocodice . . . . .	6
2.3.2	Log - Output . . . . .	7
2.4	Storage Layer . . . . .	7
2.5	Visualization Layer . . . . .	9
<b>3</b>	<b>Conclusioni e Sviluppi futuri</b>	<b>12</b>
<b>4</b>	<b>Riferimenti</b>	<b>13</b>

# 1 Dati Paradise

I dati a disposizione sono stati catturati da 10 sensori installati in un campo di peschi. Questi dati sono stati raccolti in 10 file separati in formato JSON. I file utilizzati sono stati generati dai seguenti dispositivi:

- anemometer,
- anemometer2,
- humidity\_sensor,
- humidity\_sensor2,
- leaf\_wetness,
- leaf\_wetness2,
- rain\_sensor,
- solar\_radiation,
- thermometer,
- thermometer2.

Nel dettaglio:

- (**anemometer**, **anemometer2**): Anemometri, inviano dati relativi al vento. I dati raccolti sono: velocità minima (`wind_speed_min`), velocità massima (`wind_speed_max`) e direzione (`wind_dir`) del vento,
- (**humidity\_sensor**, **humidity\_sensor2**): Sensori umidità, inviano dati relativi all'umidità dell'aria. I dati raccolti sono: umidità minima (`humidity_min`) e umidità massima (`humidity_max`).
- (**leaf\_wetness**, **leaf\_wetness2**): Bagnatura fogliare, inviano dati relativi alle condizioni di bagnatura delle superfici fogliari (rileva condensazione di particelle d'acqua sulla superficie delle foglie). I dati raccolti sono: bagnatura grezza superiore (`top_page_raw`) e inferiore delle foglie (`bottom_page_raw`) e bagnatura in percentuale superiore (`top_page_perc`) e inferiore delle foglie (`bottom_page_perc`).
- **rain\_sensor**: Pluviometro, invia dati relativi la pioggia (`rain`).
- **solar\_radiation**: Radiometro, invia dati relativi ai raggi solari (`solar`).
- (**thermometer**, **thermometer2**): Termo-igrometro, invia dati relativi alla temperatura. I dati raccolti sono: temperatura minima (`temperature_min`) e massima (`temperature_max`).

Esempio di file JSON:

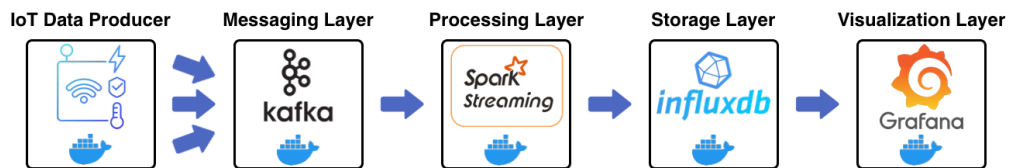
```
1 {
2   "wind_speed_max": [
3     {
4       "ts": 1656369928000,
5       "value": "14.39"
6     }, ...
7   ],
8   "wind_speed_min": [
9     {
10      "ts": 1656369928000,
11      "value": "1.93"
12    }, ...
13  ],
14  "wind_dir": [
15    {
16      "ts": 1656369928000,
17      "value": "79.0"
18    }, ...
19  ]
20 }
```

In ogni file JSON sono presenti i dati raccolti con le chiavi sopra citate. Per ogni chiave ci sono due valori: uno è il timestamp (ts) e l'altro è il valore catturato dal sensore (value). I dati a disposizione sono stati raccolti nel periodo che va dal 01/04/2022, al 27/04/2022.

I sensori inviano i dati ogni 15 minuti, quindi i dati presenti nei file JSON hanno intervalli di 15 minuti. In ogni file i dati sono stati raccolti a intervalli di 15 minuti tranne per i sensori di bagnatura fogliare dove i dati sono stati raccolti ogni minuto.

## 2 Architettura e tecnologie usate

Lo scopo del seguente progetto è stato quello di analizzare i dati provenienti da sensori IoT, processarli, salvarli all'interno di un database e visualizzarli su un browser in real time. Per poter realizzare tutto il sistema è stato utilizzato docker compose che è uno strumento che consente di realizzare applicazioni composte da più contenitori che comunicano tra di loro. Sono stati creati vari contenitori, nei quali sono stati installati i servizi necessari per il funzionamento del sistema e collegati tra di loro tramite una rete virtuale che permette loro di interagire. Nella seguente figura è possibile osservare schematicamente l'architettura del sistema realizzato con i relativi servizi.



Rappresentazione schematica architettura sistema

Come rappresentato nella figura precedente, il sistema è composto principalmente da 5 blocchi:

- **IoT data producer:** sensori installati nel campo di peschi, i quali rilevano dati con una diversa frequenza, come riportato in precedenza nel capitolo 1, che vengono inviati al blocco successivo. Come detto in precedenza, non avendo accesso diretto ai sensori e avendo a disposizione file JSON con i dati raccolti, in questo blocco è stato simulato l'invio dei dati dei sensori.
- **Messaging Layer:** si occupa di memorizzare i dati provenienti dai sensori e li fornisce al blocco successivo.
- **Processing Layer:** processa in real time i dati provenienti dal messaging layer.
- **Storage Layer:** memorizza i dati processati dal blocco precedente.
- **Visualizzazione Layer:** visualizza i dati salvati in real time.

Grazie a docker, i cinque blocchi sono stati fatti partire su 5 container separati. Per il primo blocco, relativo alla simulazione dei sensori IoT, è stato creato un Dockerfile nel quale è stato definito il container e successivamente è stata creata l'immagine contenente il container. Per gli altri blocchi sono state utilizzate delle immagini di container già esistenti nel **Docker Hub Container Image Library**.

Le variabili d'ambiente necessarie per il corretto funzionamento dei container sono state definite all'interno del file `docker-compose.yml`.

Una volta scaricate le immagini dei container e definite le variabili d'ambiente, il tutto è stato fatto partire con lo script `start-compose.sh` presente all'interno della cartella *script*. Nella stessa cartella è presente anche lo script `stop-compose.sh` per interrompere l'esecuzione dei container.

### 2.1 Dati da sensori IoT

Il primo container realizzato è stato quello per i sensori IoT. Come detto in precedenza, non avendo accesso diretto ai sensori l'invio dei dati è stato simulato mediante un container docker. Il container è stato definito all'interno del file **Dockerfile**. Per leggere i

dati e trasmetterli al messaging layer (realizzato usando Kafka), è stato utilizzato uno script Python, quindi nel Dockerfile è stata definita l'ultima versione di Python da installare nel container e il package *kafka-python* che consiste in un client Kafka per il linguaggio Python, il quale permette allo script di interagire con Kafka. Una volta definito il file Dockerfile, l'immagine del container è stata generata mediante lo script *build-iot-container* presente all'interno della cartella script.

Il file Python scritto per leggere ed inviare i dati a Kafka prende in input il nome del file in formato JSON da leggere e inizializza il producer per inviare i dati al container nel quale è in esecuzione Kafka (raggiungibile all'endpoint *kafka:9093*). Successivamente legge in file JSON, inizializza le chiavi del file in una lista e scorre il file al contrario (questo perchè l'ultima riga è il dato memorizzato più recente). Come visto in precedenza il file ha per ogni chiave un valore e un timestamp. La chiave rappresenta i diversi dati raccolti da un unico sensore, quindi nello stesso istante di tempo il sensore invia più dati. Per poter simulare questo si scorre la lista delle chiavi e si effettuano dei controlli sul nome delle chiavi e sul nome del file perchè ci possono essere sensori chiamati uguali, che inviano dati con le stesse chiavi ma che possono essere installati su punti diversi. Quindi, per ogni riga di ogni chiave del file vengono generate due liste nelle quali, in una vengono inserite le chiavi e nell'altra, seguendo l'ordine delle chiavi inserite nella precedente, vengono inseriti i valori. Il timestamp essendo lo stesso viene inserito in una variabile. Dalle due liste e dal valore del timestamp viene generato un messaggio convertito in una stringa JSON. Il producer definito in precedenza si occupa di inviare il messaggio contenente la stringa JSON e il nome del file letto a Kafka. I dati vengono inviati a Kafka ad intervalli di 5 secondi. Il nome del file letto viene utilizzato da Kafka per creare topic differenti sui quali andare a memorizzare i messaggi provenienti dal seguente container. In questo caso sono stati definiti 10 topic avendo a disposizione 10 file differenti.

L'invio in parallelo dei dati da tutti i sensori è stato realizzato mediante lo script *start-producer.sh* presente all'interno della cartella script, che richiama per ogni file JSON lo script Python descritto e al quale viene passato il nome di ogni file.

### 2.1.1 Pseudocodice

---

#### Algorithm 1: IoT data producer

---

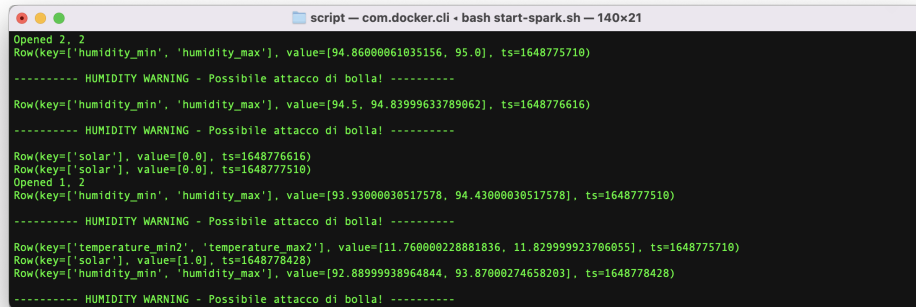
```

1 IOT_NAME ← None
2 function generate_messages(keys, timestamp, values)
3   messages ← {"key": keys, "value": values, "ts": timestamp}
4   return json.dumps(messages).encode('UTF-8')
5 // The main function
6 if __name__ EQUALS __main__ then
7   IOT_NAME ← sys.argv[1]
8   producer ← KafkaProducer(bootstrap_servers = kafka : 9093)
9   with open(jsonFile)
10    data ← load json file
11    keys ← data keys
12    for i in reversed range of data do
13      values ← [ ]
14      timestamp ← timestamp row i
15      new_keys ← [ ]
16      for key in keys do
17        values append data[key][i] value
18      producer.send(IOT_NAME,
19                    generate_messages(new_keys, timestamp, values)
20                    time.sleep(5)

```

---

### 2.1.2 Log - Output



```
script - com.docker.cli - bash start-spark.sh - 140x21
Opened 2. 2
Row(key=['humidity_min', 'humidity_max'], value=[94.86000061035156, 95.0], ts=1648775710)
----- HUMIDITY WARNING - Possibile attacco di bolla! -----
Row(key=['humidity_min', 'humidity_max'], value=[94.5, 94.83999633709062], ts=1648776616)
----- HUMIDITY WARNING - Possibile attacco di bolla! -----
Row(key=['solar'], value=[0.0], ts=1648776616)
Row(key=['solar'], value=[0.0], ts=1648777510)
Opened 1. 2
Row(key=['humidity_min', 'humidity_max'], value=[93.93000030517578, 94.43000030517578], ts=1648777510)
----- HUMIDITY WARNING - Possibile attacco di bolla! -----
Row(key=['temperature_min2', 'temperature_max2'], value=[11.760000228881836, 11.829999923706055], ts=1648777510)
Row(key=['solar'], value=[1.0], ts=1648778428)
Row(key=['humidity_min', 'humidity_max'], value=[92.88999938964844, 93.870000274658203], ts=1648778428)
----- HUMIDITY WARNING - Possibile attacco di bolla! -----
```

Output producer script

## 2.2 Messaging Layer

I messaggi generati e trasmessi attraverso lo script Python descritto in precedenza arrivano al messaging layer, realizzato con Kafka. Il vantaggio di usare un blocco intermedio tra i produttori di dati (sensori IoT) e i consumatori (Spark Streaming) sta nella possibilità di disaccoppiare gli uni dagli altri. Ad esempio, grazie a questo strato intermedio, sarebbe possibile utilizzare una diversa tecnologia per processare gli stream di dati prodotti dai sensori (ad esempio Apache Storm) senza che questo abbia alcun impatto sui produttori di dati. In assenza di Kafka sarebbe stato necessario configurare i sensori affinché inviassero dati al cluster su cui è in esecuzione Spark Streaming oppure Apache Storm.

I messaggi prodotti dallo script Python vengono trasmessi e memorizzati in un container su cui è in esecuzione Kafka. In particolare i messaggi provenienti da diversi sensori, in questo caso da diversi file, vengono memorizzati in *topic* differenti. I topic prendono il nome del file letto e occorre definirli nello script del successivo layer per permettere a Spark di leggere i messaggi. Inoltre, è stata definita la variabile d'ambiente `KAFKA_AUTO_CREATE_TOPICS_ENABLE=true` che permette a Kafka di generare automaticamente i topics dal messaggio che viene inviato.

Occorre ricordare che per poter funzionare Kafka ha bisogno Zookeeper, il quale è un servizio di sincronizzazione per grandi sistemi distribuiti. Nel seguente progetto Zookeeper è stato configurato ed eseguito su un altro container raggiungibile all'endpoint `zookeeper:2181` e per il corretto funzionamento di Kafka è stata aggiunta la variabile d'ambiente `KAFKA_ZOOKEEPER_CONNECT=zookeeper:2181` che permette a Kafka di collegarsi al container nel quale è in esecuzione Zookeeper.

## 2.3 Processing Layer

I dati memorizzati su Kafka vengono inviati ad un cluster di container Spark composto da un container master e da un container worker. Per processare i dati è stato utilizzato Spark Streaming. Lo script è stato scritto in Python. Nello script è stata inizializzata la `SparkSession` e vengono definiti i topic da leggere. Lo script si mette in ascolto del container Kafka (`kafka:9093`) e quando il container dei sensori invia un nuovo dato, Kafka trasmette il messaggio e viene letto lo streaming di dati. I dati vengono salvati su un `DataFrame` e per poter ottenere le chiavi, i valori e il timestamp occorre effettuare una decodifica del `DataFrame`. Per questa operazione è stato definito un `messageSchema` che descrive con i tipi i campi del messaggio ricevuto e ne permette la decodifica.

I dati ricevuti vengono letti in batch di diversa dimensione, in un batch possono essere presenti dati provenienti da più sensori. Per questo motivo viene eseguito un `foreach`

per ogni batch ricevuto e viene richiamata la classe `InfluxDBWriter`. I dati una volta processati vengono salvati su uno storage layer che consiste in un container su cui è installato InfluxDB. Il salvataggio dei dati prodotti avviene utilizzando il package Python `influxdb_client`. La classe `InfluxDBWriter` si occupa di inizializzare la connessione con il database raggiungibile all'endpoint `influx:8086` e di processare le righe del batch passate in input. Viene eseguito un controllo sulle chiavi che riceve e in base ad esse viene calcolato il valore medio. Ad esempio, nel caso delle chiavi: `wind.speed.min`, `wind.speed.max` e `wind.dir`, viene calcolato il valore medio tra i valori `wind.speed.min` e `wind.speed.max`. Inoltre, è stato inserito un controllo sul valore medio dell'umidità. Questo valore, insieme ai valori relativi alla bagnatura fogliare, devono essere tenuti sotto controllo perchè se superiori ad una certa soglia potrebbero provocare l'attacco di bolla sulle piante di pesco. Nel seguente caso è stato inserito un valore di test perchè al momento non ci sono abbastanza dati per stabilire con certezza la soglia. Come si può osservare dalla figura, in console viene stampato un messaggio di warning. Infine, i dati processati vengono salvati sullo storage layer.

Per poter installare il package `influxdb_client` nel container Spark e lanciare il seguente script Python è stato utilizzato lo script `start-spark.sh` presente all'interno della cartella script.

### 2.3.1 Pseudocodice

---

**Algorithm 2:** Spark-app

---

```

1 class InfluxDBWriter
2     function __init__(self)
3         self._token ← token
4         self._org ← 'primary'
5         self.client ←
            InfluxDBClient('influx:8086', self._token, self._org)
6         self.write_api ← self.client.write_api()
7     function open(self, partition_id, epoch_id)
8         Output('Opened:' partition_id, epoch_id)
9         return True
10    function process(self, row)
11        self.write_api.write('primary', self.row_to_line_protocol(row))
12    function close(self, error)
13        self.write_api.__del__()
14        self.client.__del__()
15        Output('Close with error:' error)
16    function row_to_line_protocol(self, row)
17        // Check key and value for all generated data from iot script
18        if row['key'] EQUALS key from iot script then
19            if row['value'] ≥ value then
20                Output(Warning value!)
21            return save data on database
22
23    // The main function
24    if __name__ EQUALS __main__ then
25        spark ← initialize the SparkSession
26        df ← read Kafka stream
27        df1 ← get values from df json
28        df1.writeStream.foreach(InfluxDBWriter())

```

---



### 2.3.2 Log - Output

```
script — com.docker.cli • bash start-spark.sh — 140x21
Opened 2. 2
Row(key=['humidity_min', 'humidity_max'], value=[94.8600061035156, 95.0], ts=1648775710)
----- HUMIDITY WARNING - Possibile attacco di bolla! -----
Row(key=['humidity_min', 'humidity_max'], value=[94.5, 94.83999633789062], ts=1648776616)
----- HUMIDITY WARNING - Possibile attacco di bolla! -----
Row(key=['solar'], value=[0.0], ts=1648776616)
Row(key=['solar'], value=[0.0], ts=1648777510)
Opened 1. 2
Row(key=['humidity_min', 'humidity_max'], value=[93.93000030517578, 94.43000030517578], ts=1648777510)
----- HUMIDITY WARNING - Possibile attacco di bolla! -----
Row(key=['temperature_min2', 'temperature_max2'], value=[11.760000228881836, 11.829999923706055], ts=1648777510)
Row(key=['solar'], value=[1.0], ts=1648778428)
Row(key=['humidity_min', 'humidity_max'], value=[92.88999938964844, 93.87000274658203], ts=1648778428)
----- HUMIDITY WARNING - Possibile attacco di bolla! -----
```

Output spark script

```
script — com.docker.cli • bash start-spark.sh — 140x21
Opened 7. 2
Row(key=['humidity_min2', 'humidity_max2'], value=[89.77999877929688, 90.44000244140625], ts=1648775710)
----- HUMIDITY 2 WARNING - Possibile attacco di bolla! -----
Row(key=['humidity_min2', 'humidity_max2'], value=[88.29000091552734, 89.72000122070312], ts=1648776611)
Row(key=['humidity_min2', 'humidity_max2'], value=[86.68000030517578, 88.11000061035156], ts=1648777510)
Opened 6. 2
Row(key=['top_page_raw', 'bottom_page_raw', 'top_page_perc', 'bottom_page_perc'], value=[1005.0, 298.0, 100.0, 18.40999984741211], ts=1648774810)
Row(key=['humidity_min2', 'humidity_max2'], value=[86.01000213623047, 86.72000122070312], ts=1648778410)
Row(key=['humidity_min2', 'humidity_max2'], value=[85.43000030517578, 85.98999786376953], ts=1648779310)
Row(key=['top_page_raw', 'bottom_page_raw', 'top_page_perc', 'bottom_page_perc'], value=[1005.0, 298.0, 100.0, 18.40999984741211], ts=1648774870)
Row(key=['humidity_min2', 'humidity_max2'], value=[84.5199966430664, 85.77999877929688], ts=1648780210)
Row(key=['humidity_min2', 'humidity_max2'], value=[84.02999877929688, 84.55999755859375], ts=1648781110)
Row(key=['top_page_raw', 'bottom_page_raw', 'top_page_perc', 'bottom_page_perc'], value=[1005.0, 297.0, 100.0, 18.280000686645508], ts=1648774930)
Row(key=['top_page_raw', 'bottom_page_raw', 'top_page_perc', 'bottom_page_perc'], value=[1005.0, 300.0, 100.0, 18.670000076293945], ts=1648774990)
Row(key=['humidity_min2', 'humidity_max2'], value=[83.29000091552734, 83.81999969482422], ts=1648782021)
```

Output spark script

## 2.4 Storage Layer

I dati processati da Spark Streaming vengono salvati all'interno di InfluxDB. InfluxDB è un base di dati che appartiene alla categoria dei *Time Series Database (TSDB)*, ovvero una tipologia di database ottimizzati per *time series data*. I time series data sono dati o eventi che vengono registrati, aggregati e monitorati nel tempo, come ad esempio la quantità di CPU e RAM usata in un server, i dati provenienti da sensori e così via. In un Time Series Database i dati memorizzati sono associati ad un timestamp ed è ottimizzato per misurare i cambiamenti nel tempo di una data metrica (come ad esempio la percentuale di CPU utilizzata nell'esempio precedente).

Nel seguente progetto i dati di partenza provengono da sensori e sono associati tutti ad un timestamp e per questo si è scelto di utilizzare un Time Series Database (TSDB) per memorizzare i dati prodotti da Spark Streaming. Nel panorama dei TSDB la scelta è poi ricaduta su InfluxDB perchè è compatibile con Grafana per la visualizzazione dei dati (descritto nella sezione successiva) e perchè tra i Time Series Database, secondo DB-Engines, si classifica al primo posto.

Rank			DBMS	Database Model	Score		
Aug 2022	Jul 2022	Aug 2021			Aug 2022	Jul 2022	Aug 2021
1.	1.	1.	InfluxDB	Time Series, Multi-model	29.78	+0.56	+0.22
2.	2.	2.	Kdb+	Time Series, Multi-model	9.34	+0.16	+1.35
3.	3.	3.	Prometheus	Time Series	6.62	-0.06	+0.42
4.	4.	4.	Graphite	Time Series	6.03	+0.54	+1.17
5.	5.	5.	TimescaleDB	Time Series, Multi-model	4.79	+0.26	+1.38
6.	6.	6.	Apache Druid	Multi-model	2.73	-0.14	-0.28
7.	7.	7.	RRDtool	Time Series	2.52	+0.12	+0.09
8.	8.	8.	OpenTSDB	Time Series	2.03	+0.07	+0.13
9.	9.	11.	DolphinDB	Time Series, Multi-model	1.60	-0.03	+0.53
10.	10.	9.	Fauna	Multi-model	1.38	+0.05	-0.15

Ranking dei TSDMB secondo DB-Engines

Concetti di base di InfluxDB:

- *Measurement*, che corrisponde al concetto di *tabella* in un RDBMS;
- *Tag*, che può essere visto come una colonna indicizzata in un RDBMS;
- *Field*, che corrisponde al concetto di colonna (non indicizzata) in un RDBMS;
- *Point*, che corrisponde al concetto di riga in un RDBMS.

Si riporta di seguito una sequenza di dati di esempio rappresentati sia in un RDBMS che in InfluxDB:

park_id	planet	time	#_foodships
1	Earth	1429185600000000000	0
1	Earth	1429185601000000000	3
1	Earth	1429185602000000000	15
1	Earth	1429185603000000000	15
2	Saturn	1429185600000000000	5
2	Saturn	1429185601000000000	9
2	Saturn	1429185602000000000	10
2	Saturn	1429185603000000000	14
3	Jupiter	1429185600000000000	20
3	Jupiter	1429185601000000000	21
3	Jupiter	1429185602000000000	21
3	Jupiter	1429185603000000000	20
4	Saturn	1429185600000000000	5
4	Saturn	1429185601000000000	5
4	Saturn	1429185602000000000	6
4	Saturn	1429185603000000000	5

Dati rappresentati in un RDBMS

```

name: foodships
tags: park_id=1, planet=Earth
time                #_foodships
----
2015-04-16T12:00:00Z 0
2015-04-16T12:00:01Z 3
2015-04-16T12:00:02Z 15
2015-04-16T12:00:03Z 15

name: foodships
tags: park_id=2, planet=Saturn
time                #_foodships
----
2015-04-16T12:00:00Z 5
2015-04-16T12:00:01Z 9
2015-04-16T12:00:02Z 10
2015-04-16T12:00:03Z 14

name: foodships
tags: park_id=3, planet=Jupiter
time                #_foodships
----
2015-04-16T12:00:00Z 20
2015-04-16T12:00:01Z 21
2015-04-16T12:00:02Z 21
2015-04-16T12:00:03Z 20

name: foodships
tags: park_id=4, planet=Saturn
time                #_foodships
----
2015-04-16T12:00:00Z 5
2015-04-16T12:00:01Z 5
2015-04-16T12:00:02Z 6
2015-04-16T12:00:03Z 5

```

Dati rappresentati in un InfluxDB

Con riferimento ad InfluxDB in questo esempio:

- **foodship** corrisponde alla measurement;
- **park\_id** e **planet** sono i tag;
- **#\_foodships** corrisponde ad un field.

Per ciascuna riga è presente una colonna aggiuntiva, di nome **time** che rappresenta un timestamp associato alla riga in questione.

I dati processati da Spark Streaming sono stati salvati per sensore ciascuno in una measurement distinto. A ciascun measurement è stato associato un tag identificativo e i field della measurement corrispondono ai dati processati con Spark Streaming. Nel caso della measurement *wind.speed*, i field sono: *Max*, *Mean*, *Min* e *Dir*. Inoltre, prima di salvare i dati all'interno di InfluxDB, sono state inserite alcune variabili d'ambiente necessarie per la scrittura dei dati e una volta salvati, alla lettura degli stessi.

## 2.5 Visualization Layer

L'ultimo layer si occupa della visualizzazione dei dati memorizzati su InfluxDB. In questo layer è stato utilizzato Grafana.

Grafana è uno strumento di visualizzazione di serie temporali, accessibile tramite browser, che supporta una serie di database da cui leggere i dati quali *Graphite*, *MySQL*, *PostgreSQL*, *ElasticSearch* e molti altri oltre al già citato InfluxDB. In particolare, grazie a Grafana è stata realizzata una *dashboard* in cui sono state riportate le serie temporali associate ai sensori con i dati processati tramite Spark Streaming e memorizzati in InfluxDB. Nella dashboard sono stati inseriti 10 grafici, uno per ogni sensore, nei quali è

possibile visualizzare i dati.

Per poter accedere ai dati e visualizzarli sono stati utilizzati file di configurazione presenti all'interno della cartella *grafana/provisioning*. Nel file *datasource.yml* sono stati configurati i parametri per accedere ai dati memorizzati nel container di InfluxDB e nel file *dashboard.json* sono stati configurati i 10 grafici con le relative query per ottenere i dati.

Una volta fatto partire il container è possibile visualizzare i grafici su qualsiasi browser raggiungibili all'indirizzo `localhost:3000`.

Si riporta di seguito istantanee della dashboard realizzata con i grafici:



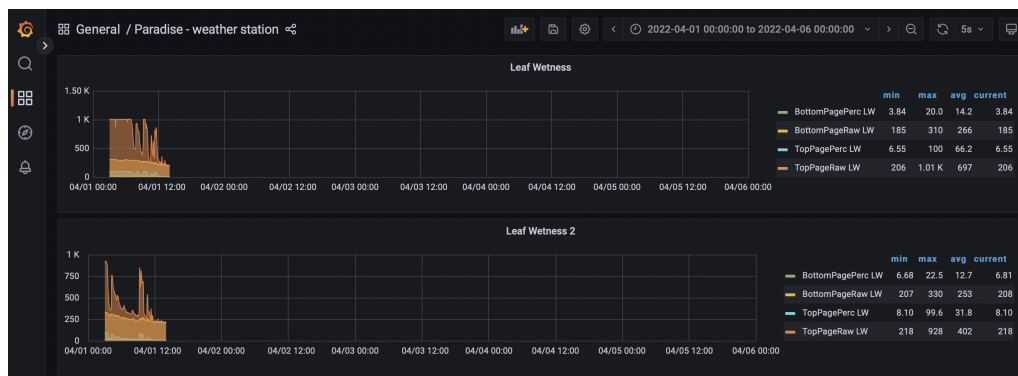
Sensori anemometri



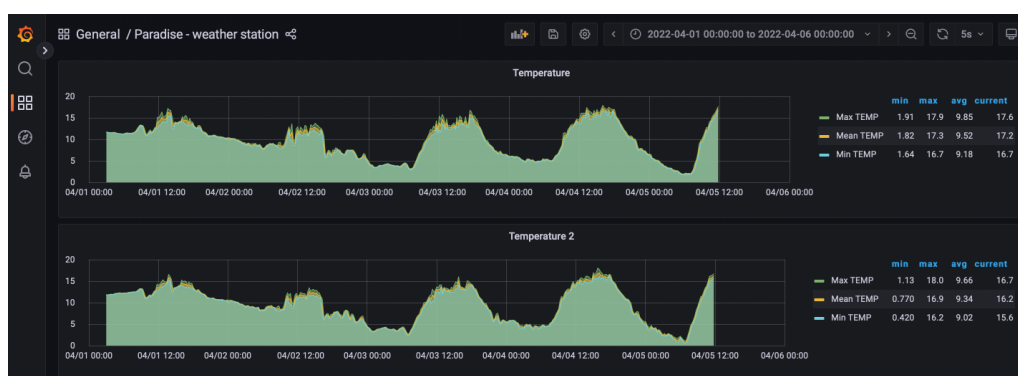
Sensori umidità



Sensori pluviometro e radiometro



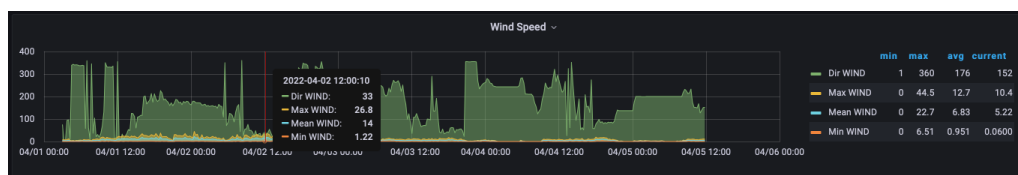
Sensori bagnatura fogliare



Sensori termo-igrometri

Dai grafici si può osservare l'andamento dei dati raccolti, processati e memorizzati. In particolare, in ogni grafico è presente una sezione sulla parte destra del grafico che descrive i diversi dati raccolti in base al colore, ad esempio nell'ultimo grafico il colore verde rappresenta i dati relativi alla temperatura massima, ed è possibile selezionare un singolo dato e visualizzare l'andamento di esso. Sempre sulla sezione, per ogni dato visualizzato è presente il valore minimo, massimo, medio e l'ultimo valore salvato. Dalle istantanee sulla dashboard è possibile anche osservare che i grafici relativi al bagnatura fogliare sono in ritardo rispetto agli altri grafici, questo perchè, come detto in precedenza, i dati relativi alla bagnatura fogliare sono molti di più perchè vengono inviati ogni minuto, al contrario degli altri sensori che inviano dati ogni 15 minuti.

Inoltre, come riportato in figura sotto se viene passato il cursore del mouse sul grafico è possibile osservare nel dettaglio i valori nell'istante di tempo d'interesse.



Dettaglio valori grafico

### 3 Conclusioni e Sviluppi futuri

In conclusione è stato realizzato un sistema di big data per il monitoraggio in tempo reale di dati provenienti da sensori installati in un campo di peschi. Nel seguente caso non avendo a disposizione molti dati, l'attacco di bolla è stato simulato inserendo un valore di test, ma in futuro con molti più dati si potranno effettuare controlli molto più precisi. Oltre all'attacco di bolla si potranno verificare anche altri fattori che possono influenzare lo stato di salute delle piante. Nel codice di Spark Streaming sono stati inseriti controlli per tutti i valori medi dei dati catturati dai sensori con le soglie impostate a  $+\infty$ . Un altro sviluppo futuro potrebbe essere quello di inserire un nuovo layer nell'architettura per eseguire analisi batch dei dati e dai dati raccolti, addestrare un modello di Machine Learning per monitorare lo stato di salute delle piante.

## 4 Riferimenti

- Documentazione Docker: <https://docs.docker.com>;
- Documentazione Docker Compose: <https://docs.docker.com/compose/>;
- Documentazione Kafka:
  - Lezioni ed esercitazioni del corso di Big Data;
  - <https://kafka.apache.org/documentation/>;
- Documentazione Spark Streaming:
  - Lezioni ed esercitazioni del corso di Big Data;
  - Spark Structured Streaming Programming Guide;
- Documentazione InfluxDB: <https://docs.influxdata.com>;
- Documentazione Grafana: <https://grafana.com/docs/>.