JASMINE - Batch Processing

Simone Mancini
University of Rome - "Tor Vergata"
0259568
2simonemancini5@gmail.com

Francesco Ottaviano
University of Rome - "Tor Vergata"
0259193
fr.ottaviano@gmail.com

Andrea Silvi
University of Rome - "Tor Vergata"
0258596
andrea.silvi94@gmail.com

***Abstract*— Big Data is all around us! Nowadays, thanks to the spread of the new technologies and exponential growing of computational power, Big Data is commonness in everyday life. So it is essential to find the most efficient way to gather relevant information from it. Weather is one of the Big Data application fields. The aim of this project is to create a batch processing distributed big data system which can easily handle weather information through some of the most widespread frameworks.**

Keywords: Apache Spark, Batch Processing, Big Data, Apache NiFi, Cloud Computing, Distributed Systems.

I. Introduction

The aim is to provide some relevant information on weather conditions in a well-known period.

In particular it is required to handle dataset[1] containing hourly temperature[2], humidity[3] and pressure[4] measurements concerning some cities of USA and Israel between 2012 and 2017. The dataset must be stored on a distributed file system: *HDFS*.

For this reason it has been necessary to develop a data injector in order to put data into file system and to get them from it.

Despite the size of the provided dataset is not large enough to consider it a "real" Big Data system, the purpose is to develop a distributed system for Big Data analysis and management so that it would keep its robustness even when the dataset will be replaced with a larger one.

Below, the project is described from different points of view: significant design choices are firstly analyzed; afterwards, queries processing with related DAGs[5] and data ingestion phases are illustrated; later, the architecture is completely described with reference to the deployment choices.

Furthermore, at the end of the report the section *"Performances"* provides significant measurements of batch processing performances for each query.

II. Design Choices

The main relevant design choice concerns the processing framework: the project is based on *Apache Spark*[6]. It has been decided to implement the project core in *Scala Programming Language*[7] because of its strong compatibility with the chosen batch processing framework. Moreover, a formatted documentation can be provided by *ScalaDoc* which simplifies the understanding of the code. As high-level framework, it has been established to use *Spark-SQL*.

To ingest data into and from the system it has been chosen to use *Apache Nifi*[8] framework. Some of its components are required to be custom in order to provide better data cleaning. For this reason it has been necessary to implement them. They are written in *Java* and a *JavaDoc* documentation is reported.

Another relevant choice concerns how to balance the load among the frameworks and the components. The team designed the system so to lighten the load on Spark in order to guarantee the batch processing as fast, simple and efficient as possible.

[1] Reduced dataset (about 50MB)
[2] Unit of measurement: Kelvin (°K)
[3] Unit of measurement: %
[4] Unit of measurement: hPa

[5] Directed Acyclic Graph
[6] Version 2.4.1
[7] Version 2.11
[8] Version 1.9.2

Store frameworks are also involved in the system development: in particular, *Redis* is used to store information concerning cities' countries and time offsets so that it is possible to avoid third services queries whenever a city-country relationship has been already computed and it can be fast to retrieve it because of in-memory storing. In this system cities' names are assumed to be unique because of the files structure[9].

MongoDB is also part of the system: it stores cities provided by dataset and queries final results because it is read-intensive data store.

Moreover, a simple web user interface, written in *Angular*, allows to show results of the queries for *Spark-Core*[10].

Other relevant design choices concern countries and timezones information retrieving which is indispensable to execute the queries. This kind of information is retrieved from *GeoNames* Services, or alternatively Google API Services[11], in order to guarantee more flexibility. In fact in such a way even when a city belonging to other countries[12] is added to dataset it is still possible to retrieve correct information on its country and time-zone. Different techniques, such as clustering ones, could be more efficient but do not guarantees correctness[13].

III. Batch Processing

Batch Processing is the most important phase of the system. For this reason the team has decided to make it as fast as possible.

The queries are now described.

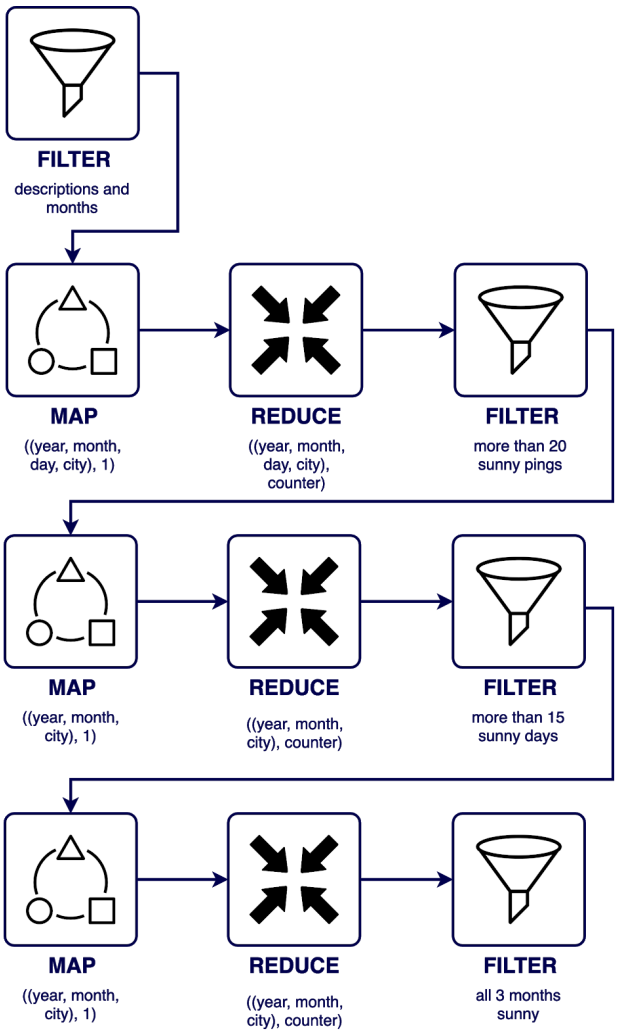
A. Query 1

The first query asks to identify the cities with clean sky, all day long, for at least 15 days a month in March, April and May.

The "*clear sky*" condition is given by a string description provided by one of the csv files.

Besides, information on sky conditions is provided hourly so it is crucial to find a criterion to establish when a day can be considered "*clear*". Evidently, this choice impacts on the results of the first query. It could be possible, for example, to have a single sample (a single hour measurement) for a day. Considering it as a "clear day" sounds to be unrealistic. So it has been established a day to be clear if it is "*clear*" for at least 20 hours a day.

The following DAG describes the first query flow with its transformations.



As it is possible to see, three "*Map-Reduce-Filter*" steps are involved. A *filter* transformation is initially performed. It guarantees that only tuples referred to March, April and May with "*sky is clear*" as description will be processed later.

The first thing to do is to retrieve days with at least 20 hours of clear sky:

- *MAP* ((*Year, Month, Day, City*), 1)
- *REDUCE* ((*Year, Month, City*), Counter)
- *FILTER* (more than 20 hours/day)

Then it is necessary to filter only those cities with more than 15 days of clear sky:

- *MAP* ((*Year, Month, City*), 1)
- *REDUCE* ((*Year, Month, City*), Counter)

[9] It's impossible to discriminate cities with the same name because coordinates are not reported in measurements files.
[10] *Spark-SQL* results are the same.
[11] Google Places API.
[12] Different from USA and Israel provided by original files.
[13] Example: Milan is closer to Switzerland centroid than Italy centroid.

- *FILTER* (more than 15 clear days/month)

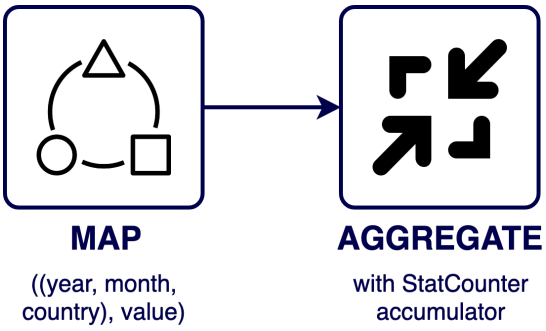
At the end, only cities with each month are returned:

- *MAP* ((*Year, Month, City*), 1)
- *REDUCE* ((*Year, Month, City*), Counter)
- *FILTER* (at least 3 months)

B. Query 2

The second query involves statistical calculations on the three different weather measurements. In particular for each single country it is required to compute mean, standard deviation, minimum and maximum values of temperature, humidity and pressure in each month of each year.

No information on city-country relationships is provided by the dataset so it is necessary to find an efficient and flexible way to retrieve countries the cities belong to. According to what it has been said in the previous paragraph, the data ingestion framework handles the original files and provides information on countries so that the processing can be fast. The countries retrieving process is detailed in the *"Data Ingestion"* paragraph. The following picture describes the DAG for the second query[14].



It is a very simple DAG, with only 2 components:

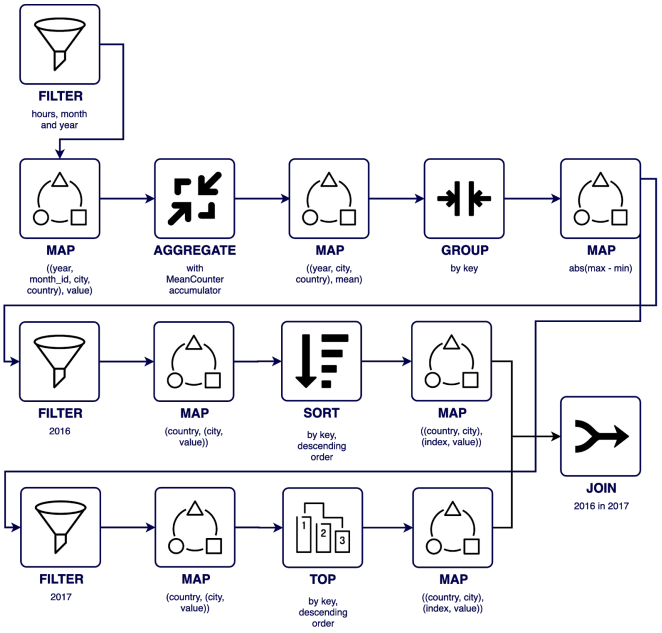
- MAP ((*Year, Month, Country*), value)
- AGGREGATE (StatCounter accumulator)

The StatCounter has being used to aggregate values in a struct that can be merged with other same structs in order to efficiently compute all required statistics.

C. Query 3

The last query involves, as well as the second one, countries and temperatures. In particular it is required to show the three different cities that have experienced the higher temperature difference between 12:00 and 15:00[15] in June, July, August and September with reference to January, February, March and April for each country in 2017. It is also required to compare cities ranking position with 2016 ranking.

The following DAG describes how this query has been developed.



The initial filter allows to get only essential tuples, filtering in 12:00-15:00 hours period, in the requested months for 2016 and 2017, then it finds the mean with aggregate operator using and custom version of StatCounter called MeanCounter focused on mean computation for both month groups, it splits the the flow for 2016 and 2017 and pass them to custom made topByKey and sortByKey aggregation functions. At the end it joins both rank to be able to create a comparison result.

D. Spark-SQL

The high-level framework *Spark-SQL* has been used to implement the same queries described before so that it can be possible to point out the metrics differences with *Spark-Core*, evaluated later. Processed files format is *Parquet*.

IV. Data Ingestion

Data ingestion phases are managed by *Apache Nifi*. It handles files and provides them to batch processing framework as clean as possible.

[14] The incoming file has the structure: *city, lat, lon, country, timezone*

[15] Local hour

In particular, it is useful to provide missing information such as countries and local datetime[16].

Data ingestion acts also as a dataset checker and filter. In fact, regex controls are performed in order to evaluate csv fields. Temperature accepted values are between 220 K and 330 K while humidity field takes values between 0.0 and 100.0%. Moreover, pression values are considered to be valid between 770 hPa and 1300 hPa[17].

Custom *NiFi* processors developed are the following:

- *ReverseGeocodingProcessor*: it scraps *city_attributes* file and it uses GeoNames services, or Google Places API Services, to get country and timezone, given latitude and longitude of the city;
- *FileFilterProcessor*: it filters csv rows according to the given regex property. It also can execute an action on line that doesn't match regex and then parse it again. In order to achieve desired results, there is the need of pass to processor a list of strings like: *'filename<-regex<-action-value(-type)<-[position, position,...]'*;
- *ReformatCSV*: it converts the original csv files into a csv containing cities in rows instead of columns;

Another custom component which merges files has been implemented and it can be added to avoid *Spark* to perform expensive *join* transformation upon files retrieved from *HDFS*.

V. Architecture and Deployment

Architecture design is one of the most relevant choice. The team has decided to deploy the system on the Cloud[18] in order to improve computational power and performances as more as possible.

Dataset is firstly stored in *AWS S3* bucket and data ingestion framework retrieves files from it.

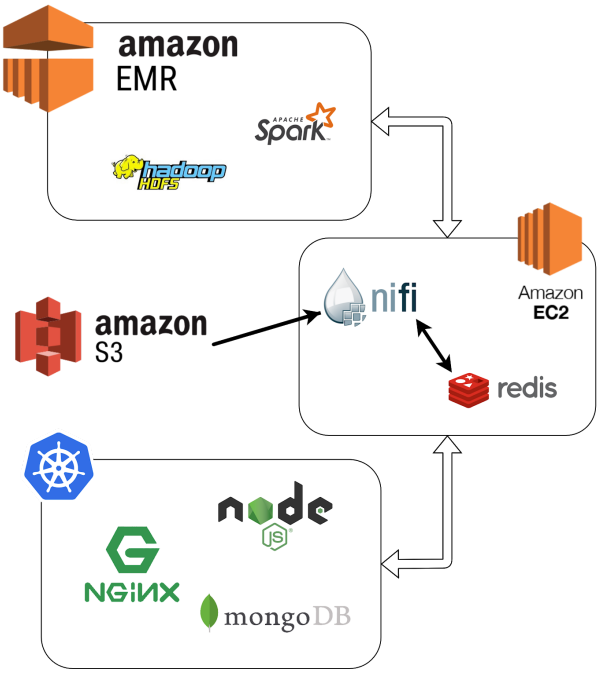
Apache NiFi is deployed on *m4-large AWS EC2* single instance together with *Redis*, used to cache relevant information.

Apache Spark and *HDFS* instead, run on *EMR Amazon* cluster.

Furthermore, *Kubernetes* has been used as cluster manager to deploy *Nginx* for the web user interface, *NodeJS* for the Rest API and *MongoDB*.

Communication between components of the system is allowed by *AWS Security Group* management.

Complete architecture is now shown.



VI. Evaluation

Queries processing time is only requested but data ingestion performances can be useful to show how the load is distributed along the system. For this reason, *NiFi* has been tested through custom components, previously described, which calculate[19] execution time for the entire flow and for each file format transformation.

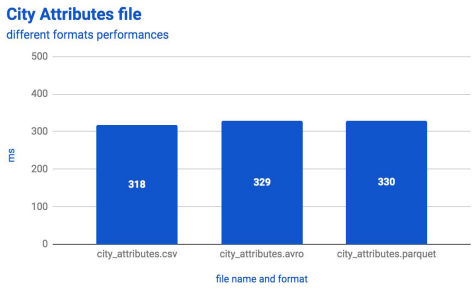
Performances are measured on a *NiFi* instance (*AWS EC2*[20]) and they are evaluated in two different cases: in the first one merging files is a *Spark* task; in the second case, data ingestion framework merges files so that *Spark* can experience only "pure" processing time.

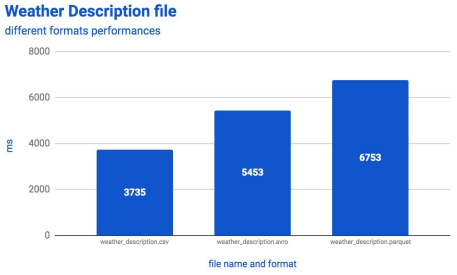
A. Case 1 - Spark Join

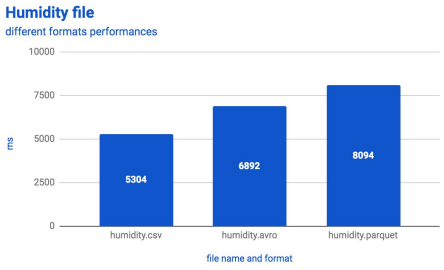
Results shown in the following pictures are related to the entire input flow for each file so that it is possible to see how each file is affected by data ingestion phase.

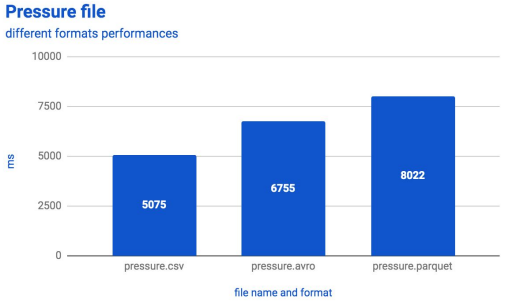
[16] Daylight-saving is not considered
[17] These values are chosen close to historical best and worst values ever recorded.
[18] Using AWS educate grant

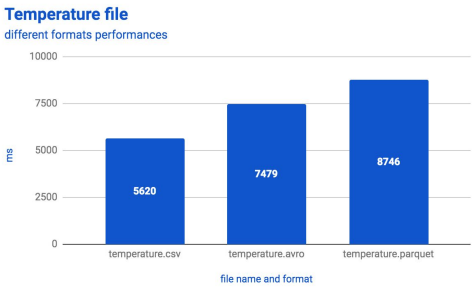
[19] In milliseconds
[20] *M4-large*, 2 CPUs, Mem: 8 GiB, 2,4 GHz Intel Xeon E5-2676 v3.





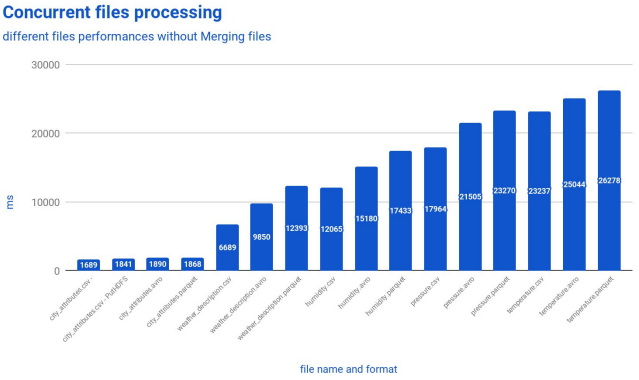






Charts show that parquet transformation requires more time than other transformations, like it is expected from flow composition. Moreover, it is natural that *city_attributes* files requires a few milliseconds, much less than other files, because of its smaller length and size.

In the following picture, concurrent files processing is shown. Entire flow, concerning putting information related to cities on *MongoDB,* as described before, and putting files on *HDFS* requires about 26 seconds because of files queuing.



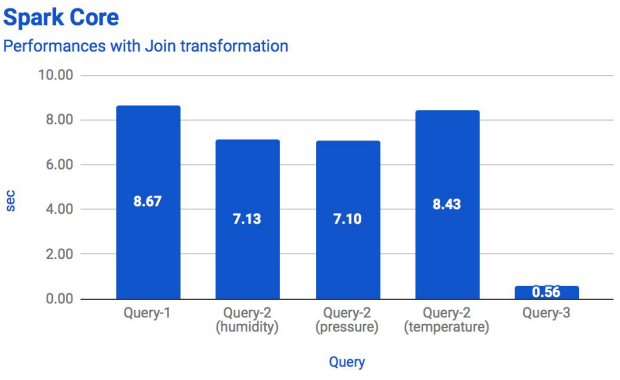
Queries' response times are now evaluated in order to show the differences between *Spark-Core* and *Spark-SQL*. Metrics are taken from the *Spark History Server* provided by Amazon EMR. Cluster has the following configurations:

- Master (1 instance) :
  - m3.xlarge, 8 vCore, 15 GiB memory, 80 SSD GB storage
- Nodes (2 instances) :
  - m3.xlarge, 8 vCore, 15 GiB memory, 80 SSD GB storage

Every run has the following configurations:

- Executors: 2
- Cores/executor: 3
- Executor memory: 6GB

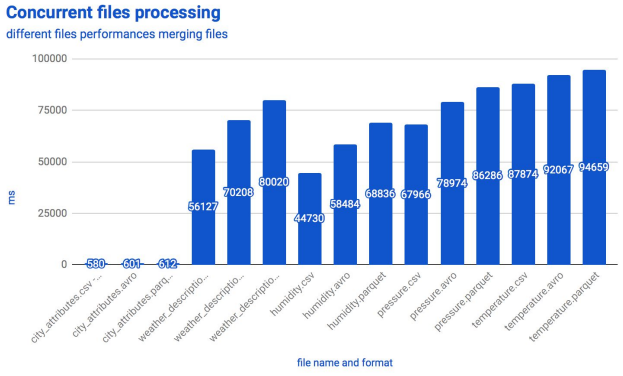
Results for *Spark-Core* and *Spark-SQL* are the following.



It is noticeable the difference of the third query performance in Spark Core due to the caching operation of the temperature input used in the query 2.

B. Case 2 - NiFi Merge

In the second case, a custom *NiFi* component has been created with the purpose of merging transformed original files. So, it is foreseeable that *NiFi* will experience more ingestion time. On the contrary, *Spark* could process files drastically faster than the first case.



As it is possible to see by the chart, *NiFi* experiences important time increment. Spark's metrics are similar to metrics computed before, contrary to what it could be expected, is not affected in relevant way by this design choice. For this reason, it is advisable to perform *join* transformation in *Spark*.

VII. Configuration

To configure, deploy and run the application, follow the instructions reported on *README.txt* file at https://github.com/sashacodes/JASMINE-Batch/tree/master/Deploy.
Source code is available here: https://github.com/sashacodes/JASMINE-Batch.
Results are available to this link: www.simonemancini.eu:32080.

VIII. Conclusions

Some of the possible improvements are now listed.

Because of its relevant usage, it would be better for system performances to deploy *NiFi* with a cluster manager such as *Kubernetes* or *Mesos* instead of a single instance.

Another relevant improvement could be *MongoDB* usage as a second level cache because of its read-intensive property.

Finally, it should be implement an advanced check on weather descriptions to guarantee that clear days are only identified by "sky is clear" string.

Reference

[1] *Apache Spark™ website:* https://spark.apache.org
[2] *Scala Programming Language documentation:* https://www.scala-lang.org
[3] *ScalaDoc documentation:* https://docs.scala-lang.org/style/scaladoc.html
[4] *Apache NiFi website: https://nifi.apache.org*
[5] *Amazon Web Services EMR: https://aws.amazon.com/it/emr/*
[6] *Redis documentation: https://redis.io/documentation*
[7] *MongoDB website: https://www.mongodb.com/it*
[8] *GeoNames website: https://www.geonames.org*
[9] *Angular website: https://angular.io*
[10] *Google API Services: https://developers.google.com/maps/documentation/timezone/intro*