Systems and Architectures for Big Data project 1: Report

Simone D'Aniello
Data Science and Engineering
University of Rome Tor Vergata
daniello.simone.uni@gmail.com

Marius Dragos Ionita
Data Science and Engineering
University of Rome Tor Vergata
ionita.maryus@gmail.com

***Abstract*—The importance Big Data have in 2018 is undeniable. The need of storing and elaborating billions of information gave birth to an increasing number of frameworks able to respond to any demand. Therefore, knowing how to make this tools cooperate each other is a fundamental prerequisite for any data scientist. This report presents the entire life cycle of a dataset produced by real sensors, special attention being paid to the storage and elaboration phases.**

I. Introduction

During the DEBS 2014 Grand Challenge [7] a sample of houses were equipped with a series of smart plugs and monitored for a month. This smart plugs are integrated with sensors which are capable to detect the instant power consumption.

This project analyzes a subset of this dataset in order to execute the next three queries:

1) find the houses whose instant power consumption exceed 350 Watt
2) for each house, compute the average power consumption and its standard deviation during the next four time frames: night, 00:00-17:59, morning, 06:00-11:59, afternoon, 12:00-17:59, evening, 18:00-23:59. The power consumption of each house is calculated by the sum of each smart plug's power consumption in the house.
3) Consider the next time frames differentiated by hour and day of the week: highest rated slot, 06:00-17:59 from Monday to Friday, and lowest rated slot, comprehending night, weekend and holidays. Sort the smart plugs in accordance with the difference between the highest rated slot power consumption and the lowest rated slot one, where in the first positions are the plugs which don't use the lowest slot.

Furthermore, input and output must be saved on the Hadoop Distributed File System [4]. The results have also to be copied on a storage system.

The first section of this report discusses the motivation behind the use of each framework. "Data Analysis" analyzes the phase of pre-processing where the dataset is parsed in order to find errors or inconsistencies. "Data Injection" shows how data are stored. The core of the report is the section "Queries" where each query is explained step by step. The end shows results obtained.

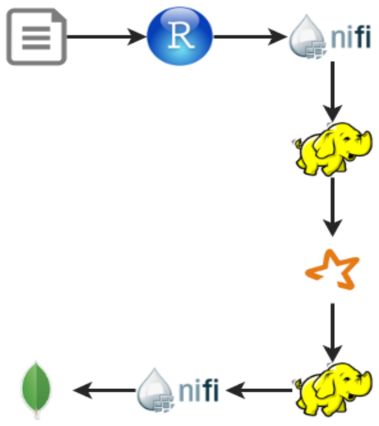


Fig. 1. Data flow

II. Choices

Drawing up and storing data may result in a huge choice of tools to use. This section analyzes the use and the motivation of each framework.

RStudio [1], an open-source IDE for the programming and statistic language R, was used for the pre-processing phase. The execution of this phase is analyzed in details during the next section. The only reason to choose RStudio instead of any other statistic tool is its readiness and the fact that it's an open-source software.

The HDFS Hadoop Distributed File System was the only design constraint for the storing phase.

For the data injection phase was used Apache NiFi [5]. Probably this framework is too powerful to simply put a file from the local storage to the HDFS but its simpleness and potential let this tool be a preliminary knowledge for any project. It's hard discussing about NiFi as an alternative to other frameworks such as Apache Flume or Apache Kafka. This tools show their true potential cooperating with NiFi. It was not considered necessary for this project to use neither Apache Flume, used for collecting, aggregating and moving stream events on HDFS, or Apache Kafka, used for event processing and integration between components of large software systems.

Apache Spark [3] was used for the queries execution instead of Hadoop MapReduce [6]. The reason of this choice can be explained analyzing in which situations this tools are taken. Spark is an in-memory analytics engine for large scale data processing and can be one hundred times faster then MapReduce. It is used for time-critical executions and it's especially useful for parallel processing with iterative algorithms. Hadoop MapReduce is used for processing huge amount of data in non-time critical executions. If data size exceed RAM MapReduce can work better then Spark, but this is not our case.

The final project requirement was to take results from the HDFS and store them in a different storage system. Due to variety of data the document based NoSQL database MongoDB [2] was chosen. The choice of the storage system is strongly linked to the reason why data are stored (for example if the database is accessed frequently it can be useful to use an in-memory storage system such as Redis) and to business requisites. Not having any direction about this requirements, MongoDB, with its flexibility, may result in a suitable choice.

For a general view of frameworks used see fig.1

III. Data Analysis

Each value in the dataset represent a measurement from a real sensor and for this reason can be subject to errors. RStudio was used to obtain data consistency and integrity. Before executing queries, data was pre-processed through a script with whom are analyzed:

- duplicated tuples
- tuples with missing values
- malformed tuples (e.g. String fields instead of integer ones)

This operations are all strictly necessary to eliminate wrong tuples from dataset and define a correct input for the application.

IV. Data Injection

This section analyzes data injection from local storage to the HDFS and from the HDFS to MongoDB.

Apache NiFi was used to manage the data flow by simply configuring the two processors *GetFile* and *PutHDFS*. The first one gets the file from the source directory (removing it from the disk) and pushes it in a queue from which it can be taken and stored in the HDFS.

After the queries computation, results are stored in the HDFS too. A second apache NiFi template can be used to copy data in the local storage. This template comprehend the processors *FetchHDFS* and *PutFile*. The names of this processors are self-explanatory.

After downloading them, results are loaded into MongoDB using a script. It could be possible to load data directly on MongoDB without storing them in local but, due to their small size, keeping a local copy can be useful to explore easily this output.

V. Queries

This section explains the execution of each request. In the first query is asked to compute the houses with instant power consumption greater the 350 Watts. Each house can have multiple smart plugs so the instant power consumption is calculated from the sum of each plug's measurement. Importing data, each tuple is mapped in an object with the next attributes:

- id: unique id of the tuple
- timestamp: timestamp related to the creation of the tuple
- property: boolean representing the meaning of the value. It's 0 if it represents the total power consumption, 1 for the instant power consumption
- value: measurement
- plug_id: unique id of the smart plug
- household_id: unique id of the household
- house_id: unique id of the house
- timezone: time frame of the measurement. It is 0 if the tuple is generated during the time slot 00:00-05:59, 1 if during 06:00-11:59, 2 for the slot 12:00-17:59 and 3 for 18:00-23:59. It's computed using the timestamp
- day: day when the tuple was created. It's computed using the timestamp

This structure is also used for the next two queries. Data are filtered through the field *property*, getting only tuples relative to the instant power consumption. Data are mapped by *house_id* and *timestamp* and during the reduce phase values are summed in order to retrieve the total instant power consumption per house for each moment. The dataset can be filtered getting only values greater then 350 Watts. Storing the results each house is related to the greatest value associated.

For the second query the average and the standard deviation of the total power consumption for each time frame (the same time frames stored in the attribute *timezone*) must be found. To execute this query data are filtered getting only tuples with property 0 and grouped by *house_id*, *plug_id*, *timezone* and *day*. This sets are reduced getting the values with smallest timestamp, i.e. the starting value of each plug. The same is done for the values with greatest timestamp. The total consumption per-house for each day can be calculated by subtracting starter and final values and summing the consumption of plugs in the same house. After knowing this daily value for each house it's simple to compute the average and the standard deviation.

In the third query two different consumption bands, low-end and high-end, must be defined. The requirement is to find the difference between the average consumptions during this two time bands and sorting results in descending order for each plug. Each tuple related to a value obtained during the time frame 18:00-05:59, during the weekend or holidays, is marked with the boolean 0. This value was taken during the low-end time frame. The other tuples are marked with 1 (high-end time frame). The dataset is filtered in order to consider only instances with the attribute *property* set to 0. In a similar way to what was done during the query 2, the starting and the final

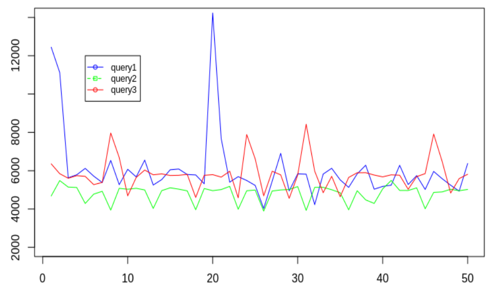


Fig. 2. HP spectre

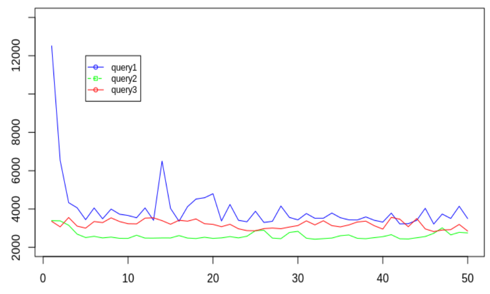
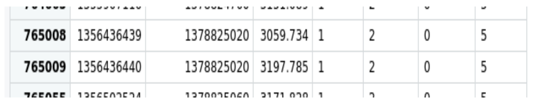


Fig. 3. Macbook Pro 2015



[illegible]	[illegible]	[illegible]	[illegible]	1	2	0	5
765008	1356436439	1378825020	3059.734	1	2	0	5
765009	1356436440	1378825020	3197.785	1	2	0	5
[illegible]	[illegible]	[illegible]	[illegible]	1	2	0	5

Fig. 4. dataset inconsistency

values for each plug and consumption band per-day are found. Now the values are splitted and is computed the average power consumption for each plug during the high-end band and the low-end band independently. This two subsets are joined and the values related to the average consumption are subbed. The Spark function *sort* is used in the end to compute the ranking.

VI. Results

Results shown in this chapter are intended in term of execution time for each query and are strongly influenced by the characteristics of the terminal that runs application. The test was executed repeating fifty times each query. The input file was loaded each time from local storage. The result of each computation generates an output directory that must be deleted on each iteration and that is not stored in the HDFS. Time spent deleting it is not considered in this results. The Spark Context is initialized once at the beginning of this phase and it's closed at the end.

Results shown in fig.2 are obtained from an HP spectre with the following specs:

- Memory: 7.7 Gb
- Processor: Intel Core i7-7500U @2.7 GHz x4
- OS: Ubuntu 18.04 LTS
- OS type: 64 bit

Results shown in fig.3 are obtained from a MacBook Pro 2015:

- Memory: 8.0 Gb
- Processor: Intel Core i5 2.7 GHz
- OS: OSX El Capitan
- OS type: 64 bit

The blue line represent time spent for the computation of the first query (in milliseconds). In green are shown the second query executions and in red the ones related to the third query. Not considering the outliers, the times needed for the execution of each query are mostly similar and constant.

VII. Conclusions

Our choices during planning and implementation phases were strongly influenced by the knowledge acquired during the first part of course classes. This project is intended to show that similar problems on biggest datasets can be solved using a similar approach to the one discussed in this report. The goal of this project was to give a general solution and to show we know how to use frameworks for big data batch processing and storing.

There are several improvements that can be done. The data processing phase doesn't take care that sensors on smart plugs can fail and reset total power consumption value. During the second and third query execution, nodes that had failures are not considered. In a future release, this values can be parsed in order to return the correct total consumption.

Another problem can be found in the execution of the first query and is related to inconsistencies in the dataset. According to the project specifications, each sensor produce a tuple every twenty seconds[1]. As you can see from fig.4 there are sensors which seems to produce multiple tuples with the same timestamp. Summing values with same timestamp for each house as requested by the query 1, the instant power consumption is computed considering this values as if they had been produced by different plugs. Having no direction about this problem, the processing-phase doesn't take care about it.

For the computation of the third query, holidays should be part of low-end time frame. The DEBS 2014 gran challenge dataset covers the period of September 2014 and is related to smart plugs located in houses in Germany. During this month there are no holidays so in the execution this check is avoided.

References

[1] RStudio. 2016. [ONLINE] Available at: https://www.rstudio.com/.
[2] MongoDB. 2009. [ONLINE] Available at: https://www.mongodb.com/.
[3] Apache Spark. 2014. [ONLINE] Available at: https://spark.apache.org/

[1]More specifically the dataset is a subset of that analyzed for DEBS 2014 Grand Challenge. The real sensor data were used to compute stream processing and each smart plug was intended to produce one tuple per second. Data used for this project was reduced getting only one tuple per twenty second and getting results from fewer homes

[4] Apache Hadoop Distributed File System. 2011. [ONLINE] Available at: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[5] Apache NiFi. 2014. 2014. [ONLINE] Available at https://nifi.apache.org/
[6] Hadoop MapReduce. 2011. [ONLINE] Available at: https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html
[7] DEBS 2014 Grand Challenge. http://debs.org/debs-2014-smart-homes/