Big Data batch processing and analytics for smart plugs sensor data with Apache Spark on Google DataProc

Batch processing project for "Systems and Architectures for Big Data"
AY 2017/2018

Ovidiu Daniel Barba
ovi.daniel.b@gmail.com

Laura Trivelloni
laura.trivelloni@gmail.com

Emanuele Vannacci
emanuele.vannacci@gmail.com

ABSTRACT

This paper provides a description of the first project for the *Systems and Architectures for Big Data* course at the University of Rome Tor Vergata: the design and implementation of a scalable distributed system able to perform specific queries on the dataset provided by the *ACM DEBS Grand Challenge 2014*, using a framework for batch analytics like *Apache Hadoop MapReduce* or *Apache Spark*.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous—*batch data analysis*; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

distributed systems, big data, batch analysis, Spark, Google DataProc

1. INTRODUCTION

The aim of the application is the batch processing of data originating from sensors placed on smart plugs in different houses. The measurement data is injected into a distributed file system using a data processing and distribution framework. The so created data lake is later used for answering three specific queries mostly about time windowed descriptive statistics using a fault-tolerant framework capable of processing massive amounts of data. The results are stored on the same distributed storage system and later redirected and saved on a NoSQL database.

2. SYSTEM ARCHITECTURE

The application is structured as a four-layer system, as shown in Figure 1, with their relative purpose:

- Data ingestion layer: filtering and redirecting incoming sensor data into the storage layer
- Data processing layer: batch analytics of stored data using Spark programming model
- Data storage layer: sensor data and results persistence and caching
- High-level interface layer: data elaboration at a higher level programming, with SQL-like capabilities

The overview of the adopted frameworks related to the high level architecture described are shown in Figure 2 and analyzed below.

2.1 Source system

The source system that provides data to the application is represented by a set of sensor devices linked to smart plugs placed in many houses.

Each smart plug has energy consumption sensors that register and emit measurements such as total accumulated consumption (in kWh) and instantaneous load (in Watt) value every 20 seconds.

Each sensor data is identified by:

- `id`: unique measurement identifier
- `timestamp`: when the measurement is taken
- `value`: measurement value (in Watt or kWh) depending on property
- `property`: boolean to indicate if it is a measurement of instantaneous load or total work
- `plug_id`: plug used by a family
- `household_id`: families in a house
- `house_id`: unique identifier

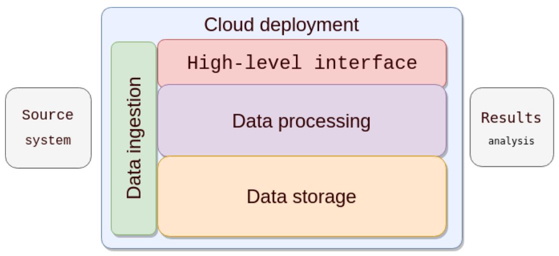

Figure 1: The System structure

2.2 Data ingestion layer

The System architecture includes a data ingestion layer to import input data into the Hadoop Distributed FileSystem, converting data in a wide variety of formats to compare performance: CSV, Parquet and Avro. To implement this layer we adopted the **Apache NiFi** framework. It is a multi-tenant, highly configurable framework that provides a web-based user interface, and offers features to operate within a cluster, expandability and a seamless experience between design, control, feedback and monitoring. NiFi is designed to automate the flow of data between software system and the software design is based on the flow-based programming model. Data operations are modeled as a data-flow that the user can visually control and monitor through features providing data provenance information.

NiFi was chosen against others data acquisition frameworks such as Apache Flume or Apache Kafka for its flexibility and the possibility to extend the framework components. Each type of data is correlated to a *FlowFile* that contains the data itself and metadata. Such metadata include,e.g. a unique identifier of the FlowFile, the data's size and the filename. The *Processors* are the main extension point and they represent the mechanism through which NiFi expose access to the FlowFiles. Several kind of Processors already exists to create, remove and modify FlowFiles. NiFi, as opposed to other tools, natively supports components to manage transformation to Parquet file from another format. This feature is important for our application since,as discussed later, Spark SQL's performance definitely improves when executes queries on Parquet as shown by benchmark [2].

The basic flow includes three main phases:

- **Extract**: The dataset is fetched from the local filesystem and the related FlowFile is created.
- **Transform**: Read the input FlowFile whose data is structured as CSV and then filters records discarding the ones that contain consumption (instantaneous or cumulative one) values of 0, assumed to be redundant and useless.

  An Avro schema is applied to the FlowFile and conversion in the file format are applied.
- **Load**: At the end the NiFi processors write on the underlying HDFS the resulting file in the following formats:
  - Apache Avro: it is simple and largely used by the Hadoop ecosystem and the messages are defined in JSON
  - Apache Parquet: Columnar storage format available to any project in the Hadoop ecosystem
  - CSV: Original file to which is just applied the filtering phase without format conversion.

The following Processors are employed:

- **ListFile** and **FetchFile**: Retrieve a listing of files whose filename matches a regular expression from the local filesystem. For each file that is listed, a FlowFile is creates.
- **QueryRecord**: Evaluates an SQL query on a FlowFile. It is used to select all rows having a non-zero value in the "value" column.
- **PutParquet**: Converts a FlowFile in Parquet format and saves it in HDFS.
- **ConvertCSVToAvro**: Converts CSV files to Avro according to an Avro Schema
- **PutHDFS**:Write FlowFile data to HDFS

After the dataflow design, NiFi offers the ability to export it as a template in XML format. The REST Api provides programmatic access to control the NiFi instance in real-time, to instantiate and deploy templates, to start and stop processors,to monitor data provenance and other things.

2.3 Data storage layer

The main purpose of this layer is to store raw data originating from NiFi. The primary storage system is the *Hadoop Distributed File System* (HDFS), a master-worker scalable file system that handles large data sets designed to run on commodity hardware with high failure rate. Since HDFS only stores and serves files on non-volatile memory without any kind of memory caching, we used *Alluxio* on top of it. Alluxio is a memory-speed distributed virtual file system and acts as a bridge between application (mainly big data oriented) and underlying storage.Its intelligent *cache* buffers reads and writes in memory for high performance on remote data. This way the processing layer needs to interact exclusively with Alluxio, which in turn handles all data read, write and cache operations on HDFS. Operations' performance can be improved up to 10x with Alluxio over HDFS rather than HDFS alone.[1]

Finally, the processing layer's results are stored in *MongoDB* using Apache NiFi as intermediate to read from HDFS and to write them as object in Mongo tables.

2.4 Data processing layer

This layer contains the **Apache Spark** framework used for batch analytics on the data lake. It is considered as an evolution of the *Hadoop MapReduce* framework, because it gains efficiency by avoiding write and read operations from non-volatile storage between each iteration as MapReduce does. So we chose Spark because it delivers fast performance and it is more suitable for iterative processing.

2.5 High-level interface layer

Here we used **Spark SQL**, a component on top of Spark Core that introduced a data abstraction called DataFrame, which provides support for structured and semi-structured data. It also provides uniform data access to a variety of sources and a SQL-like query language.

3. QUERIES

In the next subsections, we describe our implementation of the requested three queries. Each one of them takes as input the plug measurements (in the formats described in the previous section) from the sensors placed on them. Note that the filter function was not used explicitly but embedded in other functions since from our benchmarks, it impacted performance. The time period of the measurements is considered less than a year. With this assumption, a specific date can be identified by the (day, month) tuple.

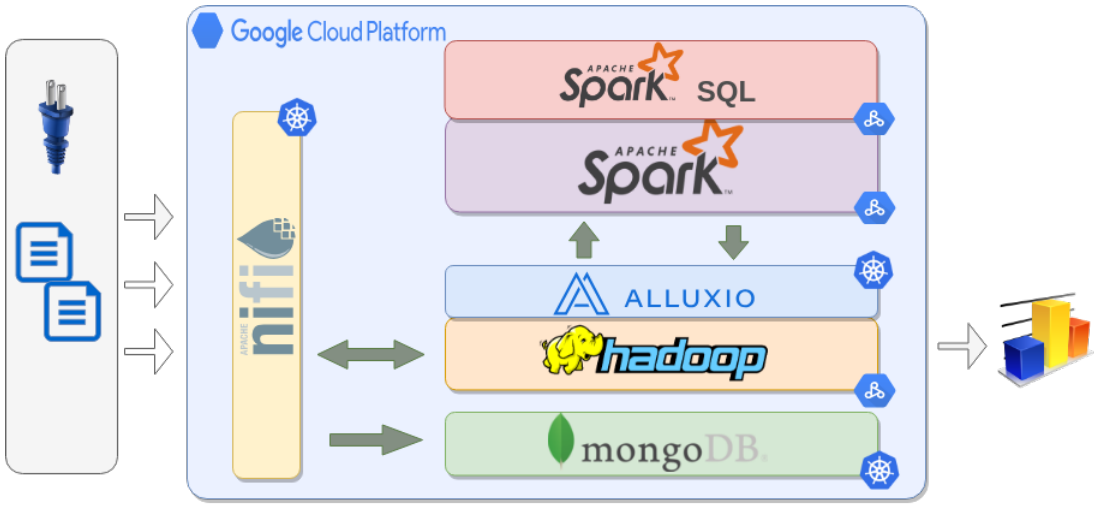

Figure 2: The System architecture overview

3.1 Query 1

The goal of the first query is to return the list of houses whose instantaneous load (considered as the sum of the loads of every plug placed in it) exceeds the 350 W threshold at any given moment. The query is structured in the following way:

- **flatMap**: if property value is 1 (load), map each measurement into a key-value tuple ((`house_id`, `timestamp`), `value`), else the input is discarded (implicit filtering)
- **reduceByKey**: for each key performs the sum of all the tuple values, so we have the total instantaneous load for every house at any given timestamp
- **flatMap**: if the value is greater or equal than the 350 threshold , map each tuple to its `house_id`, else the tuple is discarded
- **distinct**: return distinct values of `house_id`

Query 1 results are available here in JSON format.

3.1.1 *Using Spark SQL*

- **Filtering**: The first statement is performed with a *where* clause, discarding all rows that are not an instantaneous load measure.
- **Aggregation**: Records are grouped by `house_id` and `timestamp`. An aggregation is performed to sum values over the groups.
- **Retrieve results**: A SELECT statement extracts all rows with a total consumption given by the previous aggregation, with value greater than 350. Results are distinct and sorted.

3.2 Query 2

The goal of the second query is the computation of basic descriptive statistics (mean and standard deviation) of the energy consumption in each house, differentiated in four time slots: [00:00-05:59], [6:00-11:59], [12:00-17:59], [18:00-23:59].

We had to compute the statistics on increments of cumulative energy consumption values: we only consider those measurements with property flag set to 0, then for each hour we compute the increment as difference between the maximum and minimum value measured in that slot.

Considering the house energy consumption as the sum of its plugs' energy consumption, we first sum all increments in one of the four time slots to obtain the total consumption for that house in that slot for that day, then compute the statistics per time slot for all days.

We implemented a support function to retrieve the referring time slot given a timestamp value as an index from 0 to 3.

The query is composed of the following phases:

- **flatMap**: if the property flag is set to 0, map each measurement into a key-value tuple composed of:
  - key: (`house_id`, `household_id`, `plug_id`,`slot`,`day`, `month`) to identify each value measured by a given plug in a given house during a given time slot on a particular date
  - value: `MaxMinHolder` object to keep minimum and maximum value along incoming tuples, initializing both minimum and maximum to the cumulative consumption value

  else the tuple is discarded
- **reduceByKey**: for each key updates maximum and minimum values among all incoming values, resulting in a new tuple composed by the previous key and a new `MaxMinHolder` object containing max and min values of the consumption of a plug during a time slot in a specific day
- **map**: map each tuple in a new key-value tuple formed as following:
  - key: (`house_id`, `slot`, `day`, `month`) to identify house energy consumption measured in a specific day during a time slot

  - value: difference between maximum and minimum values saved in the `MaxMinHolder` object to compute effective energy consumption in that slot

- **reduceByKey**: returns the sum of all values per house per slot per date
- **map**: map each tuple into a new key-value tuple composed by:
  - key: (`house_id`, `slot`) to identify house energy consumption measured during a time slot
  - value: value of energy consumption, two initial values for counters to use to compute statistics
- **reduceByKey**: for each key sum all values of energy consumption,the number of measurements and of squared energy consumption
- **map**: maps each tuple in a new tuple composed by:
  - key: (`house_id`, `slot`) to identify house energy consumption measured during a time slot
  - average: mean statistics computed as: $avg = \frac{sum}{N}$ where *sum* is the sum of all consumption values and N is the tuples number given by incoming tuple
  - standard deviation: statistics computed as : $std\_dev = \sqrt{\frac{square\_sum}{N} - avg^2}$ where *square_sum* is the sum of the square of all values and N is the counter value of number of tuples

Query 2 results are available here in JSON format.

3.2.1 Using Spark SQL

The second query is implemented with Spark SQL exploiting tumbling time windows.

- **Type conversion**: The first statements are focused on data type conversion. The timestamp represented as a *Long* type is converted to a *TimestampType* provided by Spark SQL library. This conversion needs to exploit Spark SQL time windows functionality.
- **Plug's consumption**: Records are grouped by `house_id` and `plug_id` into a tumbling time window of 6 hours. By default the windows begin at 1970-01-01 00:00:00 UTC. The time window is added to the *DataFrame* in a new columns as a tuple with two elements representing the starting and ending point of the time slot. Therefore all records are grouped by different `plug_id` and different time slot. Since rows are sorted by timestamp the difference between the last record and the first one is computed for each group. The result is the energy consumption for each plug in a time slot for each day of the month and we refer to this amount as the plug's consumption.
- **House's consumption**: Records are grouped by `house_id` and time slot. Then the sum of the plug's consumption is performed in order to achieve the consumption for each house, into each time slot for each day of the month. We refer to this measure as the house's consumption and it represents the consumption of a house for each day divided by time slot.
- `Statistics`: Before computing the final statistics the starting and the ending point of each time slot is truncated by removing the date. Follows a group by the house's id and the time slot column. Each group contains records for a house into a given time slot over all the days of the month. We exploit Spark SQL aggregation functions to compute the mean and standard deviation of the house's consumption.

3.3 Query 3

The third query we resolved is aimed at finding a plug ranking, sorted following the criteria based on the difference of the average monthly energy consumption related to the high rate and to the low rate consumption times.

High rate consumption is the time period from 6:00 to 17:59 from Monday to Wednesday while the low rate consumption is from 18:00 to 5:59 from Monday to Wednesday, weekends (Saturday and Sunday) and holidays.

To compute monthly mean on increments of cumulative energy consumption values, we consider those tuples with property flag set to 0, then for each hour we compute the increment as the difference between the maximum and minimum value measured in that slot.

We sum all increments in one of the rates to obtain the total consumption for that plug in that rate for that day, then computing statistics per time slot for all days in every month.

To discover how much every plug consumes less in the low rate than in the high rate, we compute the difference between monthly mean consumption measured during high rate and the monthly mean consumption measured during low rate. The final ranking in given by a list of plugs sorted by value of difference among rates in decreasing order.

We implemented a support function to retrieve the relative rate given a timestamp value as an index from -12 to 12, except 0: if the index if positive, it indicates that the measurement belongs to a *high rate* time slot at month referenced by the index; if index is negative, it indicates that the measurement belongs to a *low rate* time slot at month referenced by the index.

We structured the query in the following way:

- **flatMap**: if the boolean property flag is set to 0, map each measurement into a key-value tuple composed by:
  - key: (`house_id`, `household_id`, `plug_id` `rate` `hour` `day`) to identify each value measured by a given plug in a given house during a given rate time slot at a given hour of a given day
  - value: `MaxMinHolder` object to keep minumum and maximum value along incoming tuples, initializing both minimum and maximum to cumulative consumption

  else the tuple is discarded
- **reduceByKey**: for each key perform a check to update maximum and minimum values for all incoming values, resulting in a new tuple composed by the previous key and a new `MaxMinHolder` object containing maximum and minimum values for consumption values of a plug during a rate time slot in a hour of a day

- **map**: maps each tuple in a key-value couple composed by:
  - key: (`house_id`, `household_id`, `plug_id`,`rate`, `day`) to identify each value measured by a given plug in a given house during a given rate time slot at a given day
  - value: tuple with value of the increment computed as difference between maximum and minimum value maintained in the `MaxMinHolder` object of incoming tuples and a counter initialized to 1
- **reduceByKey**: for each key perform sum of values of increment and counter, resulting into a new tuple composed by the key and as value a tuple with the sum of all increment values in a day for each rate and the counter of related tuples
- **map**: maps each tuple in a new key-value couple composed by:
  - key: (`house_id`, `household_id`, `plug_id`, `rate`) to identify each value measured by a given plug in a given house during a given rate time slot at a given month indicated by the absolute value of rate
  - value: tuple with value of the monthly mean computed as:
    $avg = \frac{sum}{N}$
    where N is the number of tuples coupled with the value of a counter initialized to 1
- **reduceByKey**: for each key performs sum of values of increment by day and counter, returning a new tuple composed of the old key and as value a tuple with the sum of all increment values in a month for each rate and the counter of related tuples
- **map**: maps each tuple in a key-value couple composed by:
  - key: (`house_id`, `household_id`, `plug_id`), absolute value of `rate` to identify each value measured by a given plug in a given house during a given rate time slot at a given month, described by the absolute value of the rate index
  - value:
    * if the rate of incoming tuple is more than 0 (high rate index) then the value is the monthly mean computed as:
      $avg = \frac{sum}{N}$
    * if the rate of incoming tuple is less than 0 (low rate index) then the value is the monthly mean computed as:
      $avg = \frac{sum}{N}$
      but changing sign to make possible subtract this value from the related high rate mean value in the next Phase
- **reduceByKey**: for each key performs algebraic sum of values of monthly mean by rate,generating a new tuple formed by the old key and as value a tuple with the score to use for ranking plugs
- , **sort**: sort list of tuples in decreasing order respect to value of score

Query 3 results are available here in JSON format.

3.3.1 Using Spark SQL

To perform the third query we have employed the **UDF** (user defined functions) feature of Spark SQL. It allows to define new column-based functions that extend the vocabulary of SQL DSL to transform DataFrames.

- Phase 1: Two new column are added to the DataFrame. The first suggests the time slot withing a given day (high or low), to which each tuple belongs. The second one indicates the day of the month.
- Phase 2: For each plug, the difference between the value of the last tuple and the first one is computed for every time slot of one hour resulting in the hourly consumption for that plug in each day of the month.
- Phase 3: The average of the hourly consumption grouped by day of the month and time slot returns the daily consumption for the two time slots.
- Phase 4: Daily consumption are averaged grouping by time slot to obtain the monthly consumption of each plug for the two time slots.
- Phase 5: Finally we compute the difference between the high slot and the low one resulting in a score value that is used to compute the final rank of the plugs.

4. SYSTEM DEPLOYMENT

The system components behave in the same way in local and cloud deployments. The Spark cluster can perform read and write operations on HDFS directly or through Alluxio. The dataset in different formats (along with the application code) need to be injected into HDFS (using NiFi) from where all the nodes in the Spark cluster acquire them. NiFi awaits for updates on a specific directory and starts a workflow that reads the files with the results and automatically stores them into MongoDB. More details on how to deploy it in both local and cloud environments can be found at the project repository.

4.1 Local deployment with Docker

For every system component (HDFS, Spark, NiFi, Alluxio, MongoDB) we created Docker images. Some were made from scratch to accommodate our needs (Spark, NiFi and Alluxio) while the others were already configured and built. In particular, Spark and Alluxio containers were built to run in pseudo-distributed mode. All components need to run on the same docker network.

4.2 Cloud deployment with DataProc and Kubernetes

The complete cloud architecture is described in Figure 2. Using the DataProc service offered by the *Google Cloud Platform* (GCP), we allocated a cluster of VMs running Hadoop and its ecosystem (along with Spark). DataProc allows to run Spark jobs on the cluster by simply specifying the application jar's URL (supports HDFS, Google Storage among others) and its arguments.The service also comes with a configured and working HDFS. Since the DataProc configures

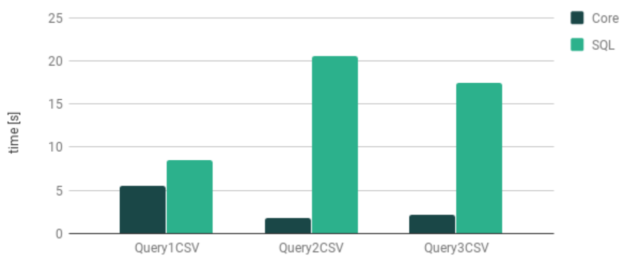

Figure 3: Comparison between Spark CORE and Spark SQL queries implementation on local Dockerized application with CSV format

clusters without any Docker or Kubernetes support, we deployed the remaining system components on the *Kubernetes Engine*, also offered by GCP. The service simplifies the process of configuring and deployment a Kubernetes cluster. We configured a memory optimized cluster on which we run Alluxio, MongoDB and NiFi using the Docker images mentioned in the previous section. The Kubernetes and DataProc cluster are on the same VPC network so every component can interact with each other.

5. PERFORMANCE RESULTS

We performed different experiments in both local and cloud environments to evaluate the performance of the System with Spark core and Spark SQL. Further evaluations were made to evaluate which data format is the better. The local experiment was carried out on a single machine with low hardware specifications:a MacBookAir with a single two-core CPU Intel i5 with 1.4 GHz speed, 256 KB L2 cache and 3MB L3 cache, 4 GB of RAM. The System runs also on the cloud in a DataProc cluster with a master and two workers, each one on n1-standard-4 (4 vCPUs) instance with 35 GB of memory available. The Kubernetes cluster includes three n1-standard-2 (2 vCPUs) nodes with a total 22.5 GB of memory.

As illustrated in Figures 3 and 4 the general trend is that Spark Core performs better both in local and cloud environments. The difference between the execution times is noticeable especially for the queries implemented on the local cluster where Spark core outperforms Spark SQL two to ten time faster. The trend is confirmed for the queries executed on the cloud where the difference is smaller but still considerable. All execution times are computed as the mean of 10 consecutive runs.

Because official benchmarks of Spark Core and Spark SQL on different data format have not been found, we decided to support input data coming in CSV, Avro and Parquet for performance comparison. While Spark SQL offers native support for Parquet format, Spark Core handles the creation of RDD only by CSV. Measuring the execution time on Spark Core for queries on Avro and Parquet files, we have also considered the overhead to transform a DataFrame into a RDD. The results are illustrated in figures from 5 to 7 where different system configurations are used. In Figure 5 there is only HDFS as the storage system with no Spark caching facilities, which has been added later with results in Figure 6. Finally 7 shows the results using Alluxio on top of HDFS with no Spark caching, still in cloud environment.

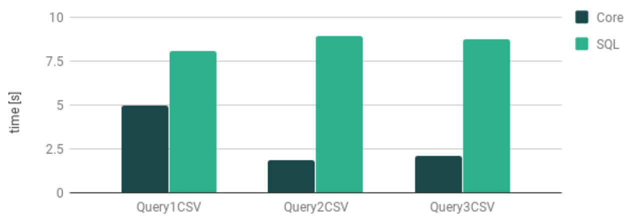

Figure 4: Comparison between Spark CORE and Spark SQL queries implementation on Cloud with CSV format

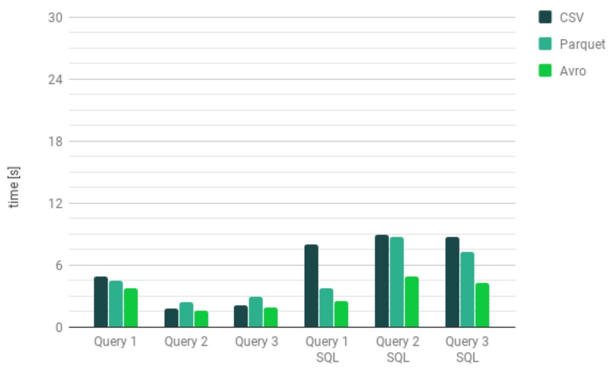

Figure 5: HDFS storage on Cloud

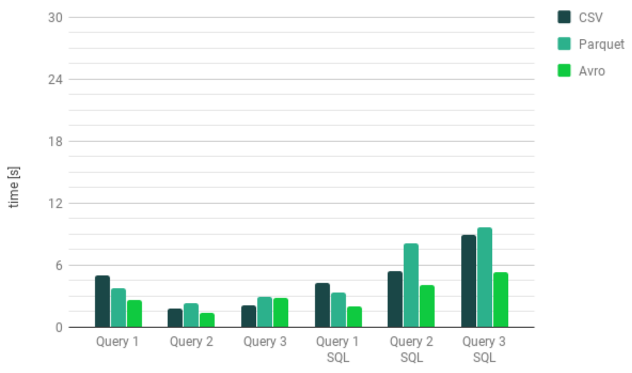

Figure 6: Alluxio storage on Cloud

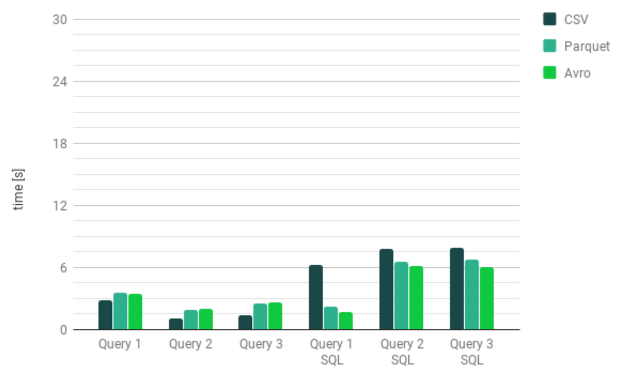

Figure 7: Cache activity and only HDFS storage on Cloud

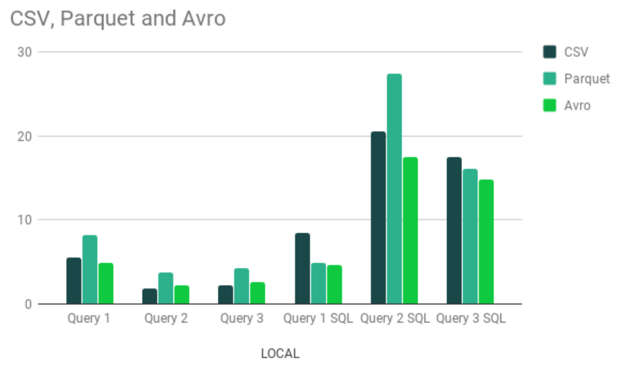

Figure 8: HDFS storage on local Dockerized application

6. CONCLUSIONS

In conclusion, the System provides services to resolve the proposed queries with execution times in the order of seconds that is considered efficient for batch systems even though the dataset is small.

From the results, it can be seen that the best system configuration is using the Avro format, Alluxio as the distributed file system with no Spark caching.

It is possible to improve performance for query 2 and 3 considering a bigger interval between which computes the single value of increment instead that by every hour, by day; on the other side, this change will decrease statistics precision.

Another possible improvement concerns the implementation of User Defined Functions as alternative implementation for query 2.

7. REFERENCES

[1] Benchmarking alluxio on hdfs. https://www.alluxio.com/blog/using-alluxio-to-improve-the-performance-and-consistency-of-hdfs-clusters.

[2] Benchmarking apache parquet. http://blog.cloudera.com/blog/2016/04/benchmarking-apache-parquet-the-allstate-experience.