INFO - H - 515
2018 - 2019

3 May 2019

Big Data – Data Management Part

Project Assignment - Phase I

Kaïs Albichari	000395807	M-INFOS
Tanguy d'Hose	000409718	M-INFOS

Introduction

The project for the course on Big Data is split into two parts. For this first part, students are required to design a Big Data Management Pipeline that is to be used for storage and management of sensor data as well as the computation of batch and streaming queries. The second phase will consist of applying Machine Learning methods on the collected data in order to build a predictive model.

BDMA architecture

Our smart city consists of a total of 10,000 locations in which 100,000 sensors were dispersed. As mentioned in the assignment, every sensor emits a reading approximately every 30 seconds. In other words, in a live context, the architecture should expect over 3,000 readings every second. Consequently, our use-case has immense data velocity.

1. Key assumptions

- Our sensors are part of a very well-maintained network and do not emit faulty readings in such a way that previously recorded data is to be altered.
- All data sent by sensors is correct and correspond to the present (no readings from the past or future can be introduced).
- Data sent over the pipeline is sent in chronological order.
- The project allows for 10,000 spaces and 100,000 sensors. Spaces are assigned IDs ranging from 1 to 10,000 and sensors are assigned IDs from 1 to 100,000.
- The pipeline will solely consider spaces and IDs described in the locations folder. If no additional files are provided, the lab will be only space to send data over the pipeline (mote_locs.txt).
- Municipalities in which spaces are located are assigned randomly. This is also the case for the privacy setting associated to this space (private or public).
- Benchmarking requires additional data. For this purpose, additional data (sensors and spaces) is generated randomly.

2. Characteristics of a BDS

As a refresher, here are the properties of a BDS (obtained from the course):

- Robust and fault-tolerance
- Scalable
- Low latency answering of pre-defined queries
- Support ad-hoc querying
- Low-latency updates
- Extensible/general

a. Robustness and fault-tolerance

An assumption of our use-case is that our sensors are part of a well-maintained network. In case of any malfunction of a sensor, it is assumed to stop emitting data. If one was to assume that malfunctions would result in the emission of unprobable readings, the system would compute wrong statistics about the network (robustness remains, but faults are introduced). All readings emerge from IoT devices that automatically measure and send the data. As this data is assumed to be correct and about a specific time, we are not confronted with possible mutability issues (data is immutable from start).

However, the sample of readings that was provided for this assignment shows that it is possible sensor-values are missing in the readings. In this case, the system avoids any errors by simply omitting these readings. By doing so, fault-tolerance is ensured.

b. Scalability

In any professional implementation project, it is important to consider possible scaling of the infrastructure and associated problem. From an external point of view, the use case can scale in several ways: more sensors could be fitted into spaces, more spaces could be added in our municipalities, the project could be extended to reach municipalities outside Brussels, etc.
Due to the fact that new spatial information can quickly be introduced into the system, structural issues are very small. Indeed, such operations solely require the assignment of new IDs to sensors and spaces, a mapping between sensors and spaces as well as a mapping between spaces and municipalities. If the project extends outside Brussels, more complicated modifications have to occur as the assumption that any municipality is located within Brussels will no longer hold.

From an architectural point of view, one should ensure new incoming data is handled adequately. As the format of information would not be altered by scaling the project, it all comes down to the frequency of incoming messages. Indeed, an increased frequency will increase the load on the system. Hence, one must ensure computational power as well as data management techniques are suited to this frequency. By correctly constructing the original data architecture, scalability of this element should not be too complicated.

c. Low latency answering of pre-defined queries

The project describes three different queries that are to be implemented. These form the set of pre-defined queries of the system. As we assume the data arrives in a streaming fashion, elements of the stream will be separated in several batches that are analyzed separately before storing the associated result. It comes without surprise that these results form the core of the dashboard and will need to be computed and displayed with low latency.

d. Ad-hoc querying

Our smart city project enables the user to consult data that was produced by the three queries that are to be implemented. However, the user might like to consult information according to a particular granularity in time and space. The rendering of these statistics relies on ad-hoc queries of previously computed data. Indeed, the data infrastructure computes very rough results for several sensor types and times along the day. When the user will display data, he will want to display information for a particular granularity in time and space. This operation will require queries on pre-computed data and new storage of intermediate results.

e. Low latency updates

Users that consult the dashboard will not want to wait a long period of time before having any information made visible or obtain updates previously requested information. Consequently, the system must provide information on regular basis to update the data displayed in the dashboard. As the assignment mentions each sensor emits readings with averaged intervals of 30s. Consequently, the dashboard will use the same frequency for refreshing the displayed data.

f. Extensible/General

Extensibility or generality of a project is related to the previously described structural scaling. Indeed, for the system to be as general as possible, assumptions that are made cannot be too closely related to the current state of the project. In our case, this means a loss of generality emerges when computations are specifically aimed towards a particular municipality for instance (hard-coded queries). To avoid this, the structure of the system should be such that adding sensors, spaces or maybe even additional municipalities would not impact the functionality of the system. Also, the possibility exists that the project manager changes the meaning of the values that are emitted. If such modification occurs, an extensible system should not be harmed by it.

3. Finding a suitable architecture

a. Broad description

As mentioned previously, we assumed incoming data is immutable. Consequently, the question of defining our architecture relies on what model would be most suitable for the tasks that we would like to perform. In the assignment, these tasks are described in the requirements section. In short, these refer to the possible queries that are to be implemented as well as their granularities in time and space. Additionally, we should consider that data arrives in streaming fashion and needs to be stored afterwards.

The system we have in mind consists of immutable data streaming in every second. Subsequently, this data is submitted to queries that will pre-compute useful information. For instance, statistics will be computed on batches and stored every 30 seconds. Using the information that was already stored, as well as newly computed information, the dashboard will be able to use precomputed batch views as well as precomputed (near) real-time views to execute the ad-hoc queries.

In more precise terms, incoming data should be stored alongside all other data that will be handled by system. In terms of architecture, this forms the what is called a batch layer. Additionally, we want to extract useful information from readings that come streaming in and provide them as support for queries on already collected data. This forms our architectural speed layer. Finally, we would like to gather a series of precomputed views (query results on batches) to ensure disposal over data that can be displayed in the dashboard. This type of mechanism corresponds to a serving layer. Hence, the obtained model can be formalized as a Lambda architecture.

b. Why lambda?

We are aware of the fact that Lambda architectures are not the only BDMA architectures that exist. In fact, we briefly considered the implementation of a Kappa architecture. Indeed, Kappa architectures essentially function in the same manner Lambda architectures do. The key difference lies within the fact that Kappa architectures omit the batch layer. If our dashboard was not required to be able to access earlier computations (e.g. 10 years back), the Kappa architecture would have been sufficient. Unfortunately, our need for storage and more complete views over the data directed us towards an architecture with batch layer, namely the Lambda-architecture.

Our research has also shown that data lake architectures exist. Contrarily to the currently discussed solutions, data lakes allow storage of structured, semi-structured, and unstructured data. Additionally, data lakes do not use conventional ways of storing data: usually data is stored in files and folders, whereas data lakes provide a flat hierarchy where different records essentially share a common space. Though the principle of these kinds of architecture seems very interesting and could provide efficiency advantages in some use-cases, we know our data is very structured and, hence, systems designed to store structured data are more suited to our architecture.

c. Components

The fact that our BDM should be scalable and generalizable will have to be translated into our specificities. Indeed, if one was to assume a single 'client' could be able to manage all sensor information, the system would suffer when the load increases. Hence, we know that multiple client type workers are required. As we dispose over several types of sensors, we choose to assign one worker to each sensor category. By doing so, we know the model will withstand possible increased loads. Additionally, generalization and scalability are maintained as an increase (decrease) in sensor types can be managed by adding (removing) associated workers. Also, if the load was to increase dramatically, assigning multiple workers to each sensor type would make the overload manageable. All in all, the system would remain scalable.

In terms of implementation, we have opted for Apache Kafka. Indeed, Kafka gives us the possibility to create consumers that obey the "publish-subscribe" message queue paradigm. More precisely, for each sensor type we can instantiate a consumer that will solely consume messages of that type (topic = sensor id). However, dramatical increase in load could result in a paradigm shift to a hybrid consumption model (with consumer groups). If this is the case, some modifications in implementation will have to be carried out.

Scalability is enforced by Kafka through partitioning of topics. Aside from a scalability, opting for Kafka also makes querying lighter: most queries will not require data from all types of sensors, but rather only a given sensor type. Consequently, having access to consumption per sensor type is very advantageous in such conditions. Kafka's implementation is also very beneficial to increase fault-tolerance. Indeed, as partitions can be replicated, the probability of faults occurring diminishes. Topics themselves are not fully managed by Kafka. In reality, Kafka builds on Apache Zookeeper that essentially acts as a centralized service that provides all necessary operations to manage our topics, as well as their distribution and synchronization.

There exist multiple alternatives for Kafka. For instance, Apache also designed Apache Pulsar. Similar to Kafka, Pulsar is an open-source publish-subscribe messaging system. In fact, both systems are said to be very similar. Of course, the newer Pulsar has some key differences, such as different subscription types, namespaces possibly having multiple topics, support for multitenancy, etc. However, even though Pulsar has low latency, Kafka is reported to have a greater processing capacity thanks to consumption techniques that do not involve any copies. Also, the mature community surrounding Kafka provides greater support and documentation for correct implementation of that part of the pipeline.

Managing streaming data will be done using the Spark Streaming framework. This framework will enable us to collect short-span data to subsequently produce RDDs. A great advantage of using this mini-batch method is that we can handle multiple data instances simultaneously, as opposed to one-by-one management. There does exist alternatives to Spark Streaming that are able to handle incoming data streams. For instance, the same company offers Apache Storm. The main difference between the two frameworks is that the latter processes data in real-time, whereas the former introduces a small delay between arrival and processing of data. However, it is precisely that delay that makes it possible for Spark Streaming to satisfy the *exactly once* in normal circumstances and *at least once* conditions in less favorable circumstances.

Data that is streamed through the pipeline by producers and subsequently captured by consumers needs to be stored somewhere. Of course, results computed by Spark Streaming cannot simply be discarded. The main purpose of obtained data is to be consulted by the consumer. As a reminder, it precisely that aspect that guides us towards a Lambda architecture, as opposed to a Kappa architecture. To handle storage requirements of our pipeline, we opted for a Python implementation of MongoDB.
As there are uncountable possibilities for data management in software projects, we really needed to thoroughly analyze the different data storage methods and frameworks. The assignment describes a typical IoT project. The Internet of Things is characterized by elements that can easily be found in this project: a possible large number of concurrent users, need for responsiveness to distributed users, possibly unstructured data, no downtime, and flexibility. Though the implementation we have in mind does not necessarily involve semi- and unstructured data, we do know that opting for relational databases is not in our interest if we have concurrence in mind. Additionally, relational databases would impact the scalability aspect of our project. Our research has shown that a NoSQL is more convenient given the characteristics of this assignment. In fact, they are said to be best suited for real-time analytics and high-speed operations as well as scalability. As we had previous experience with

MongoDB and, similar to Kafka, there exists a very mature community surrounding the framework, we decided to opt for this DBMS.

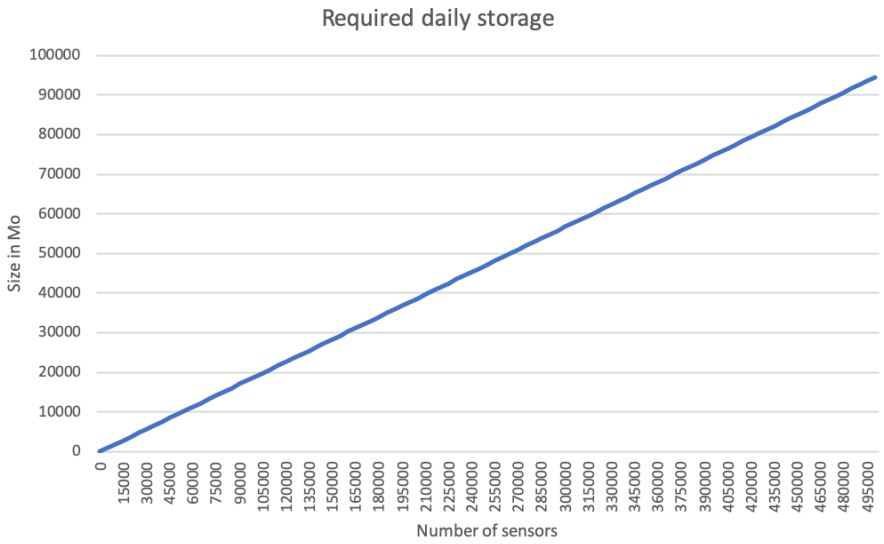
The last component of the pipeline consists of having a dashboard that can be used by the user to query and display data. The assignment specifies that it should be a web-based dashboard. In this case, we have downloaded an extension of Jupyter notebooks called Jupyter dashboards. This enables us to use python technologies to create a dashboard with which the user can interact.

In first instance, we wanted to create a custom dashboard starting from scratch. Unfortunately, such implementations are time consuming and do not necessarily yield the expected results. Hence, we researched the several possibilities that exist to create a web-based dashboard. Similar to the choice of data infrastructure, the possibilities for making web-based dashboards are endless. We found several candidates, such as Pusher for instance. Most solutions, like the latter, require installation of numerous (Python) modules as well as a lot of time dedicated to the design of the interface. As this project is more about data processing and management, we wanted to opt for something simpler. In these simpler frameworks we stumbled upon plotly that also has dashboard creation features, but with reduced installation and configuration times. Though plotly offers a great solution with associated performance and lots of possibilities, it is a commercial solution of which the open source version does not provide enough documentation, and therefore ill-suited to this project.

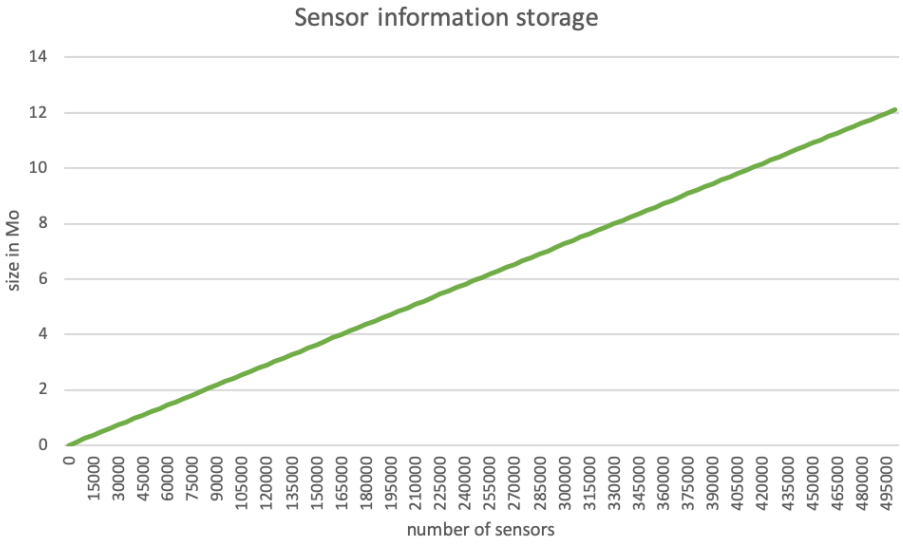
Further research showed Jupyter Dashboard Layout Extension (JDLE) is perfectly suited for the task of creating a dashboard for our smart-city use-case. Indeed, the assignment mentions three queries (or types of information) that can be consulted by the user. As these can potentially vary in time and space granularity, a lot of different visualizations would be required. However, JDLE dramatically reduces the number of visualizations by introducing interactions.

Analysis of data volumes

Data originally provided with the project only considers a single space, namely a laboratory of 54 sensors. The associated file contains a month worth of readings, that add up to around 300Mo of required storage. Our calculations show that, given the frequency of updates, the system processes around 0,189Mo per sensor per day.



The immediate first question is the following: leaving aside data storage how would the processed data evolve if the scope of the project was to increase. We cannot possibly hypothesize in terms of spaces as sensor distributions amongst them depend on predefined files. Also, we noticed that spaces and sensors utilize a very limited space in our database.

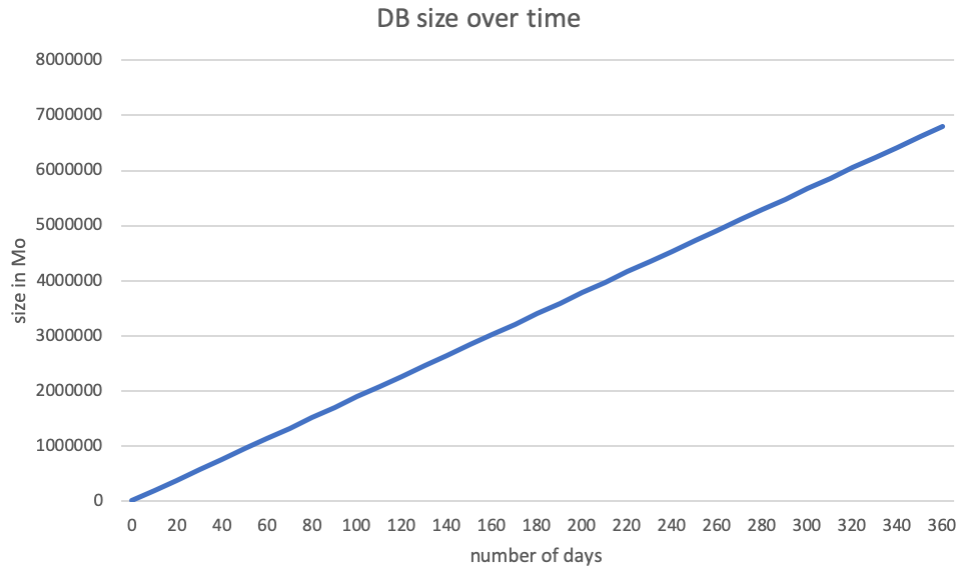


Indeed, we see that scaling the number of sensors would not impact the need for storage in dramatic ways. In fact, assuming the project scales to a capacity of 500,000 sensors, only 14Mo would be required to include new sensors. From a space point of view, the data

requirements are roughly the same. Assuming the number of spaces would scale in proportional manner (to 50,000), barely over 1Mo could store all data.

It becomes obvious the challenge of this project lies within data storage and management of the readings. Hence, from now on, we decided to express the processed amount of data only in terms of number of sensors. Our findings show that a small smart city project will face a small load. Indeed, if we assume our project has 10,000 fully active sensors, it would require stream below 2Gb of data on a 24-hour period. As the number of sensors increases, the load the system faces follows in a linear fashion. Consequently, a very large projects of 500,000 sensors will face a load of 1TB on a 24-hour period.

In the case of this assignment, we hypothesized that a maximum number of 10,000 spaces will totalize a maximum number of 100,000 sensors. Hence, when reaching full capacity, the pipeline streams 18,9Gb per day. As our lambda architecture requires storage of information to be able to compute information that is displayed in the dashboard, it becomes of interest to see how much space the database would occupy over time for this project. For this purpose, we decided to plot data requirements over a 1-year period at full capacity:



Unsurprisingly, the tendency is linear. What is much more interesting to know is that the total storage requirements on a one-year period at full capacity is close to 7TB. Hence, the project administrator would require a large and consistent data-storage solution. That being said, the project was implemented having scalability in mind. The use of NoSQL databases takes care of concurrence and frequency of access to the data. This means that there would be few constraints on the project growth. However, data storage will be a huge issue.

If further growth does indeed happen, we suggest the project administrator implements a few changes. First, downtime should be avoided and data should be spread across multiple servers. In this context, the administrator should look for solutions that match the needs of the project. As a reminder, these needs are mainly centered on processing time and processing quantity.

We are aware of the fact that greater data volumes will also impact other facets of this project. For instance, the data frames returned by some queries will be quite large in size. Hence, these queries will have to be redesigned to avoid having to deal with frames that are that large.

Queries

Before thorough description of the queries and their implementation, it is important to understand that each provided query has to be divided in two entities:

- The streaming and batch processing: using Spark Streaming, this part deals with continuously incoming data and manages them in such a way that relevant information for this specific query can be stored in the database. As stated in the assignment specification file, each record has the structure « date, time, sensor-id, measurement voltage ».

- The representation in the dashboard: The second part is mainly using the database (MongoDB) and data frames in order to apply operations on results and report dynamically the requested information with their granularity.

1. Basic statistics

The first query requires computation of basic statistics about the sensor readings (per type of sensor). These basic statistics are the minimal reading, the maximal reading and the average reading. As the smart city that yields readings is hierarchical (space < municipality < city), the computed statistics are to be made available for each element of each level of the hierarchy. Additionally, these should be computed for several granularities in time.

1.1. Implementation

The design of the code is relatively simple. The underlying idea is to store all information of a record in a collection (table in MongoDB) as a document. It is not required to keep the sensor voltage, but as the aim of the assignment is to obtain a system that acts in real conditions, this information is can potentially be important to store to detect when a sensor is close to stop functioning due to battery issues for instance.
The computations of minimum, maximum and average measurement values could not have been done in batch for two reasons: first, there is no information about the space and municipality in the streaming data. Therefore, this information is assumed to be stored in database and the grouping can only be made afterwards. Secondly, the granularities in time are numerous, therefore it is logical to only have one process which stores relevant information in the database. Hence, granularity and representation options are delegated to the dashboard rather than overloading the batch processing with every possible of granularity option.

2. Characterization of temperature timeslots

For every temperature sensor, the second query should characterize every 15-minute timeslot into daytime and nighttime temperature. The former is described as an average temperature starting from 19,5°C, whereas the latter is described by any average temperature below this number. Again, characterization of timeslots is to be executed for all possible granularities in space as well as several granularities in time.

2.1. Implementation

A timeslot must be assigned to every temperature sensor's reading, with respect to its time value. 24h is divided into 96 slots of 15min each. Again, different granularities in time and space, but also in space type (private/public) must be available to classify the groups into daytime or nighttime.

Space parameters (space, municipality) and space types are not specified in the streaming data. Consequently, this data, alongside the associated timeslots, have to be stored in the database. This database contains other associations (sensor-id, space, municipality) that were previously stored. Computed information is loaded in the Dashboard with the specified granularity options and is joined with the space and municipality in which it lies, if needed. The reason why this join operation is not mandatory is that if the requested information is to be grouped for the entirety of Brussels, obviously we can ignore the records about the space/municipality.

3. Frequent readings in sliding window

The last query computes readings that are frequent on a sliding window of one hour, as well as an estimate of their frequency. For this query, students are allowed to choose their own threshold of what is considered to be frequent. Additionally, the result of the query is allowed to be approximate.

3.1. Implementation

For this last query, the batch processing takes care of most of the job. A window of 1h with a slide duration of 10 seconds has been used directly on the streaming data. In other words, the batch processing will handle a window of a maximum of 1h, with the most recent data being updated every 10 seconds. As there is not any granularity specified, the computation can be made during batch processing and stored as-is in the database.

Assuming that the temperature measurements are within -10°C and 30°C and that these values are rounded to a precision of one fractional digit, there exist 400 different values that can be observed. Hence, the frequency threshold must be coherent with this wide range, in the sense that if it is too low, a lot of different temperatures would be recorded. If some of them are too high, then there won't be any temperature above this frequency threshold.

As the frequency depends on the number of measurements within the window of 1h, when starting the new process (assuming there is no information stored in the database), the number of distinct measurements will only grow. During the first hour, and then stabilize.

Keeping that in mind, the choice has been made to have a frequency threshold that depends on the number of distinct measurements available in this window. Therefore, this frequency threshold is $\frac{1}{N}$, where N is the number of distinct values in the current window.

Dashboard implementation

As mentioned in the section about the components of the pipeline, we have opted for a dashboard implementation using Jupyter. More precisely, Jupyter disposes over a layout extension that is known as Jupyter Dashboard. This extension adds functionality to jupyter notebooks to enable its users to generate interactive dashboards using Jupyter's report-style functionality implementation that can alternate between markdown and code.
This extension adds several visualization possibilities under the "Dashboard View" section. The first one permits to see the actual implementation of the dashboard. Next, a choice gives the possibility to visualize using either grid layout or report layout. It must be noted that in our case, grid layout seems to be malfunctioning. Finally, it allows dashboard preview, in which only results and interactions are displayed.

By running simple commands[1], this extension can be added:

```
MBP-de-Tanguy:~ tanguydhose$ pip3 install jupyter_dashboards
```

```
MBP-de-Tanguy:~ tanguydhose$ jupyter dashboards quick-setup --sys-prefix
```

The implementation itself is close to identical to regular Jupyter notebooks. However, this extension introduces ways of interacting with the reader. For this purpose, the module Ipywidgets has to be imported. This module contains a series of UI elements as well as event handlers that, together, enable interaction with the user.
We use a lot of well-known modules to compute our queries. In this context, we use Seaborn, Matplotlib, and Ipywidgets for visualization. Pandas is used to handle the collected data using the DataFrame object. The contents of these frames are obtained through queries made possible by the PyMongo module. Finally, we are able to handle dates and times thanks to the datetime module.

[1] https://jupyter-dashboards-layout.readthedocs.io/en/latest/getting-started.html