FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Management of large volumes of data collected by wind monitoring sensors

**Pedro Filipe Agrela Faria**


Mestrado Integrado em Engenharia Informática e Computação

Supervisor: João Correia Lopes

Second Supervisor: José Carlos Matos

July 18, 2019

# Management of large volumes of data collected by wind monitoring sensors

**Pedro Filipe Agrela Faria**

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Professor Doutor Sérgio Sobral Nunes
External Examiner: Professora Doutora Maria Benedita Campos Neves Malheiro
Supervisor: Professor Doutor João Correia Lopes

July 18, 2019

# Abstract

Nowadays, virtually everything around us generates large amounts of data. With this comes the need to store such large volumes of data in a scalable and efficient way. Conventional databases, which follow relational models, are no longer scalable when applied to large amounts of data since it follows the properties: Atomicity, Consistency, Isolation, Durability — ACID. In these situations, losing the ACID properties, NoSQL databases are preferred due to easy scalability. There are also time series databases that are designed to store pairs of timestamp and value attributes.

In the case of the renewable energy, in particular, the wind energy area, the decision making is often — if not always — sustained by large data sets collected by different instruments and sensors. The constant fluctuation of the meteorological variables, such as wind speed, direction, air temperature, etc. increases the importance of accurate measurements of such variables. This can be done during prospect phase, where local wind conditions are assessed to ensure wind turbines suitability, or post construction, to track the performance of the wind farms, monitor the conditions of critical components and optimize the wind turbines control. These data can be recorded at different frequencies, typically with sampling rates of 1 Hz and integration times of 10 minutes where several statistics are generated. In certain circumstances, the sampling frequencies can reach higher values and the integration time may not even be applied, meaning all data needs to be stored for later processing, which creates a large amount of data.

The main objective of this work is to create an information system capable of storing and managing large volumes of data, processing, cleaning invalid records, and subsequent reporting of that data. This challenge was proposed by INEGI, who has an implemented system currently, although with some limitations, namely in what concerns scalability and flexibility. The created platform aims to solve those problems and is designed to support three database systems: PostgreSQL, MongoDB and Influx. We studied the performance by inserting and querying the raw data, and also the disk space occupied by each database system.

The performed tests not only helped us to choose the best database to do such operations but also allowed us to improve the platform during these tests. The study revealed that a Relational Database Management System — RDBMS — and a Time Series Database — TSBD — can coexist in the same system, using PostgreSQL to store and handle all the meta information of the system and Influx to store the raw data provided by the wind towers.

To develop the platform, we studied the different Actors in the system, refering to the different access levels in the platform by each user. We analyzed the different User Stories that contain the individual interactions in the platform by those users, and we also designed the Model of the Domain to describe the entities and their subsequent relationships present in the system. The platform was implemented in Django, a Python framework, that follows an Modal-View-Controller — MVC — architecture to separate the different layers in the Web system.

# Resumo

No decorrer dos dias de hoje, e cada vez mais, estamos rodeados de dados, o que gera uma necessidade de armazená-los de uma forma eficiente e escalável. As base de dados convencionais, que seguem o esquema relacional, deixam de ser escaláveis quando se deparam com grande volumes de dados, pois seguem as propriedades: Atomicidade, Consistência, Isolamento e Durabilidade — ACID. Nestas situações, perdendo as propriedades ACID, são preferidas as base de dados NoSQL devido à sua fácil escalabilidade. Existem também base de dados especificamente desenhadas para guardar pares de *timestamp* e valor, que são designadas por base de dados *time series*.

No caso da energia eólica, a tomada de decisão é muitas vezes — se não sempre — sustentada por grandes conjuntos de dados recebidos por diferentes instrumentos e sensores. A constante flutuação das variáveis meteorológicas, como a velocidade e direção do vento, temperatura do ar, etc., aumenta a importância de ler essas variáveis de uma forma mais precisa. Isso pode acontecer durante a fase prospetiva, onde as condições locais de vento são avaliadas para garantir o correto funcionamento das turbinas eólicas, ou pós-construção, para monitorizar o desempenho dos parques eólicos, as condições dos componentes críticos e otimizar o controle das turbinas eólicas. Esses dados podem ser registados em frequências diferentes, geralmente com taxas de amostragem de 1 Hz e tempos de integração de 10 minutos, onde são geradas várias estatísticas. Em determinadas circunstâncias, as frequências de amostragem podem atingir valores mais altos e o tempo de integração pode nem ser aplicado, o que significa que todos os dados precisam de ser armazenados para processamento posterior, o que cria uma grande quantidade de dados.

O principal objetivo desta dissertação passa por criar um sistema de informação capaz de armazenar e manipular grandes volumes de dados, processá-los, limpar registos inválidos e posteriormente gerar relatórios desses dados. Este desafio foi proposto pelo INEGI, que atualmente tem um sistema implementado, contudo possui limitações de escalabilidade e de flexibilidade. Esta plataforma visa resolver esses problemas e é projetada para suportar 3 sistemas de base de dados: PostgreSQL, MongoDB e Influx. Para isso, estudou-se o desempenho, inserindo e consultando os dados brutos, e também o espaço em disco ocupado por cada sistema de base de dados.

Os testes realizados, ajudaram a escolher qual a melhor base de dados para realizar essas operações e também permitiram melhorar a plataforma durante os mesmos. O estudo revelou que as base de dados relacionais e as *time series* podem coexistir no mesmo sistema, usando o PostgreSQL para armazenar e manipular a meta-informação do sistema e o Influx para armazenar os dados brutos fornecidos pelas turbinas eólicas.

Para desenvolver a plataforma, estudou-se os diferentes Atores do sistema, no qual permitenos diferenciar os níveis de acesso na plataforma por cada tipo de utilizador. Analisou-se as diferentes *User Stories* que contêm as interações individuais na plataforma por cada utilizador e desenhou-se também o Modelo Conceptual com o objettivo de descrever as entidades e as suas subconsequentes relações no sistema. A plataforma criada foi implementada utilizando o Django, uma framework Python, que segue uma arquitetura *Model-View-Controller* — MVC —, no qual separa as diferentes camadas de um sistema *Web*.

# Acknowledgements

I first would like to thank all the people who, directly and indirectly, helped during the academic course.

To my family, I express my gratitude for the support, not only financially, but also for all the advice and motivation during this phase of my life.

To my girlfriend, Andreia Gomes, I'm thankfully for the patience and understanding shown throughout the entire course.

To my colleagues, because without them, this wasn't possible, helping me daily through academic difficulties.

And especially to my supervisors, João Correia Lopes, José Carlos Matos and Amândio Ferreira, for all the guidance, orientation and high availability through these months.

Pedro Faria

*"You can have data without information, but you cannot have information without data"*

By Daniel Keys Moran

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| API | Application Programming Interface |
| B | Byte |
| BSON | Binary JavaScript Object Notation |
| CPU | Central Processing Unit |
| CSS | Cascading Style Sheets |
| DBMS | Database Management System |
| GHz | Gigahertz |
| GIS | Geographic Information System |
| GWh | Gigawatt hour |
| HTML | HyperText Markup Language |
| JSON | JavaScript Object Notation |
| INEGI | Institute of Science and Innovation in Mechanical Engineering and Industrial Engineering |
| ISAM | Indexed Sequential Access Method |
| KB | Kilobyte |
| LiDAR | Light Detection And Ranging |
| MB | Megabyte |
| MHz | Megahertz |
| MVC | Model-View-Controller |
| NoSQL | Not Only SQL |
| PB | Petabyte |
| RAM | Random Access Memory |
| RDBMS | Relational Database Management System |
| SQL | Structured Query Language |
| TB | Terabyte |
| TSDB | Time Series Database |
| UI | User Interface |
| URL | Uniform Resource Locator |
| UML | Unified Modeling Language |
| XML | Extensible Markup Language |
| WWW | World Wide Web |
| ZB | Zettabyte |

# Chapter 1

# Introduction

In the last 20 years, data has increased on a large scale in several areas. According to a report from International Data Corporation (IDC[1]), in 2011, the world created and copied an amount of 1.8 ZB data volume, that increased by almost 9 times in the next five years [13].

## 1.1 Context and Motivation

Recently, industries become interested in the real potential of big data and data e-Science, forcing many government agencies to declare major plans in big data research [20]. With that, comes a great need to store large data in a dynamic, reliable and efficient way [29]. If we consider the big company Google[2] as an example, they process data of hundreds of Petabytes, Facebook[3] generates log data of over 10 Petabytes per month, and Taobao[4] from Alibaba group[5], generates data of tens of Terabytes for online trading per day [6].

In the case of the renewable energy, in particular of the wind, where electric generation is achieved under conditions of constant variability, it is relevant to record and store the evolution of the various meteorological variables over time, as well as data generated by wind turbines [3]. This data should be stored in a reliable, efficient and scalable system, ensuring that managers handle that data easier to achieve goals and profit with their investors and clients.

It is also important to mention the impact of the renewable energies to avoid severe problems of environment pollution, energy shortage and climate change. A study made on India concludes that the forecast of total electricity consumption in the year 2020 and 2030 will be 944 524 GWh and 1 395 754 GWh, respectively in this country. This means an increase of energy consumption by 47 %. The same study predicted renewable energy sources would replace the fossil fuels to the

[1] https://www.idc.com/
[2] https://www.google.com
[3] https://www.facebook.com
[4] https://world.taobao.com/
[5] https://www.alibabagroup.com/

extent of 32 % in the year 2020 [22]. We have China as the most energy consumption country, representing a consumption of 20.3 % worldwide [28]. According to Enerdata, only 25 % of the total energy created in the year 2017 worldwide was produced by renewable energy [40], which means our world has a long way to change, fighting against financial and political strengths.

This work was proposed by the Institute of Science and Onnovation in Mechanical Engineering and Industrial Engineering (INEGI[6]), particularly by the Engineer José Carlos Matos from the renewable energy department. The Institute was created in 1986 and develops its activity in four fundamental services: as investigation, as innovation and technology transfer, as scientific and technological consulting, and as service supply.

The main goals from INEGI renewable energy department its to make equipment management, process data, clients interfaces and operations support.

## 1.2 Goals and Results

In the wind energy field, there are a lot of data sets being collected continuously during the whole project lifetime. With that comes the need to store such large amount of data in databases, in a reliable and scalable way. That's why NoSQL (Not only SQL) is very popular nowadays.

The main goal of this work it's the creation of a Web based Information System, focusing on performance and scalability to store, handle and export high volumes of data provided by meteorological masts, turbines and other equipment's. This platform will store all the meta information and the raw data provided from wind met masts. That means the platform will be divided into two sections, one to store all the meta information using a relational database, other to store the largest data set — all the raw data — using a NoSQL and a time series databases. Another functionality to be created is a classification scheme to flag and clean the raw data and the possibility of creating plots and statistic tables from that cleaned data. There are a lot of other functionalities regarding how data is created, such as data collection, complex reporting exportation, field report, that due to lack of time, will not be implemented.

Even with the popularity of the NoSQL databases in these days to store large amounts of data, no Web platform uses time series database (TSDB) to store, handle and export data provided by wind energy. With this, it is expected that this prototype compares performance and scalability with the different solutions, on the different databases that will be implemented, with the actual INEGI solution.

## 1.3 Document Structure

The remainder of this document is structured as follows:

- Chapter 2, *State of the Art*, refers to the existing platforms in e-Science field, as theoretical concepts of renewable energy, especially in wind energy and an important review on existing databases.

[6] http://www.inegi.pt

- Chapter 3, *Problem Statement and Solution Proposal*, describes the actual problem in the INEGI platform to store and handle a large amount of data and a possible solution to solve that problem.
- Chapter 4, *Requirements Analysis and Architecture*, refers to the requirements analysis that gives us a top overview of the system functionalities, the selected technologies and also the system architecture.
- Chapter 5, *Implementation*, describes how the application was implemented, presenting the approaches made for the most critical features.
- Chapter 6, *Tests and Results*, refers to the tests methodology applied, what and how we made these tests, and also discussion of the results from those tests.
- Chapter 7, *Conclusions and Future Work*, describes the conclusions of the work and also the future work to be done.

# Chapter 2

# State of the Art

In this chapter, we will review the existing e-Science platforms for data management, handle some theoretical concepts of renewable energy, especially the wind energy, and approach some existing databases, relational, NoSQL and time series databases, and how the database make transactions safely.

## 2.1 Research Data Management

This section analyzes the already existing solutions of e-Science platforms to manage their data. There are a few platforms already developed across the world.

### 2.1.1 Introduction

Unfortunately, similar institutions in Portugal that store and handle data provided from wind meteorological masts also develop their own system and no documentation on this topic is disclosed as such systems are regarded as a potential competitive advantage. For this reason, we research for worldwide platforms to handle that data. In the next topics, we will refer to some e-Science platforms that exists worldwide in this context and also mention the INEGI platform to manage wind tower data.

### 2.1.2 HubZero

The HubZero platform aims for researchers to access and share scientific simulation and modeling tools and results in an easy way. This platform was developed at Purdue University by researchers to support the nanoHUB.org, an online community for the Network for Computational Nanotechnology (NCN), founded in 2002 by the US National Science Foundation [30]. In 2007, nanoHUB.org served more than 56 000 users from 172 countries [24] and since that year, more Sites or HUBs were created, thanks to the high demand for more scientific areas. Some of these

areas are pharmaceutical engineering, heat transfer, microelectromechanical systems, health care, cancer care, and engineering education [31]. HubZero offers to their client's functionalities as, creation of datasets and interactive simulation tools using RStudio, Jupyter Notebooks, and other web applications, it also gives the researches the possibility to publish products including, datasets, tools, and white papers on the system. HubZero also provides online forums for researchers teams and collaborators to discuss data concepts, track progress, and share files through Google Drive, GitHub, or Dropbox.

This infrastructure can have an impact on scientific discovery [24] thanks to the creation of different hubs to serve various communities, to support collaborative development and dissemination of scientific models that run in a cloud system [31]. The primary function of this platform is the Middleware for hosted execution. The format of each tool published on the hub is like a journal paper, containing the title, abstract, list of authors and list of references. It also includes an option to launch a live session. The simulation runs on a cluster of executions hosted near the Web server, projected to the user's browser using a virtual network computing. To control access to file systems, network and many other server processes, the tool runs in a restricted lightweight virtual environment implemented using OpenVZ[1]. Each user has a unique and private home directory, requiring ownership, access controls and quota limitations.

The main goal for HubZero future is on the development of new functionalities to connect related content so that tools published on one hub can be shared and easily found by other hub's [31].

### 2.1.3 EUDAT

The EUDAT[2] project is a pan-European data initiative that started in October 2011. The project involves 25 partners – including research communities, national data and high-performance computing (HPC) centers, technology providers, and funding agencies [25].

In last years many investments have been made by the European Commission and European member states to create a pan-European e-Infrastructure supporting multiple research communities. EUDAT aims to build a sustainable cross-disciplinary and cross-national data infrastructure that provides a set of shared services for accessing and preserving research data [14] helping to overcome these challenges by laying out the foundations of a Collaborative Data Infrastructure (CDI). It's known that research communities from different disciplines have different ambitions, data and approaches, but they have in common many shared basic service requirements and EUDAT aims to establish common data services, designed to support multiple research communities.

The main characteristic of this platform is the CDI framework (Figure 2.1 adapted from [19]), that ensure the trustworthiness of data, provide for its curation, and permit an easy interchange among the generators and users of data [19]. This will enable the communities to focus a more significant part of their effort and investment on discipline-specific services. The CDI also provides to the smaller communities or individual researchers solutions with access to sophisticated

[1] www.openvz.org
[2] https://eudat.eu/

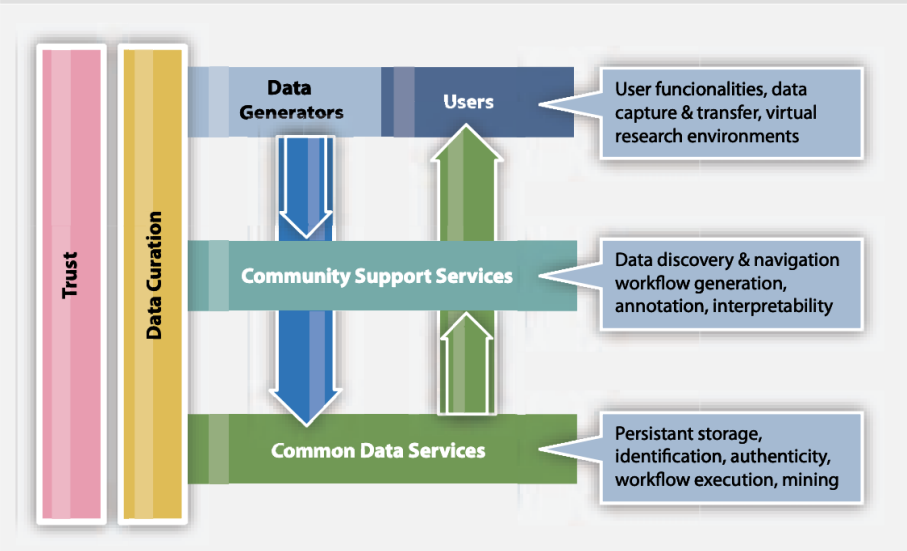



Figure 2.1: The Collaborative Data Infrastructure

shared services, thus removing the need for large-scale capital investment in infrastructure development [25].

The main services that EUDAT offers to their community are:

- Data Replication — to guard against data loss in long-term archiving and preservation.
- Data processing — ability to share a huge amount of data between the EUDAT storage structures.
- Metadata — making data from various disciplines available in one collaborative infrastructure can be extremely beneficial.
- Storage and Sharing of Scientific Papers — ability to share between researchers their data and scientific papers.

### 2.1.4 WindScanner.eu

The WindScanner.eu is a distributed Research Infrastructure facility (RI) from a European initiative, led by the Technical University of Denmark in association with the European Strategic Energy Technology Plan [33]. This infrastructure is a remote sensing-based research facility capable of providing new fundamental knowledge about the detailed three-dimensional atmospheric wind flow and turbulence around huge wind turbines, when extracting data from wind turbines [33]. The European WindScanner project consists of a measurement system and analysis through laser sensors — LiDAR. These sensors will measure and generate maps on wind conditions, in which a large amount of data is created over time. With this comes the necessity to develop the WindScanner.eu platform, that allows collection, processing and visualization of data provided from

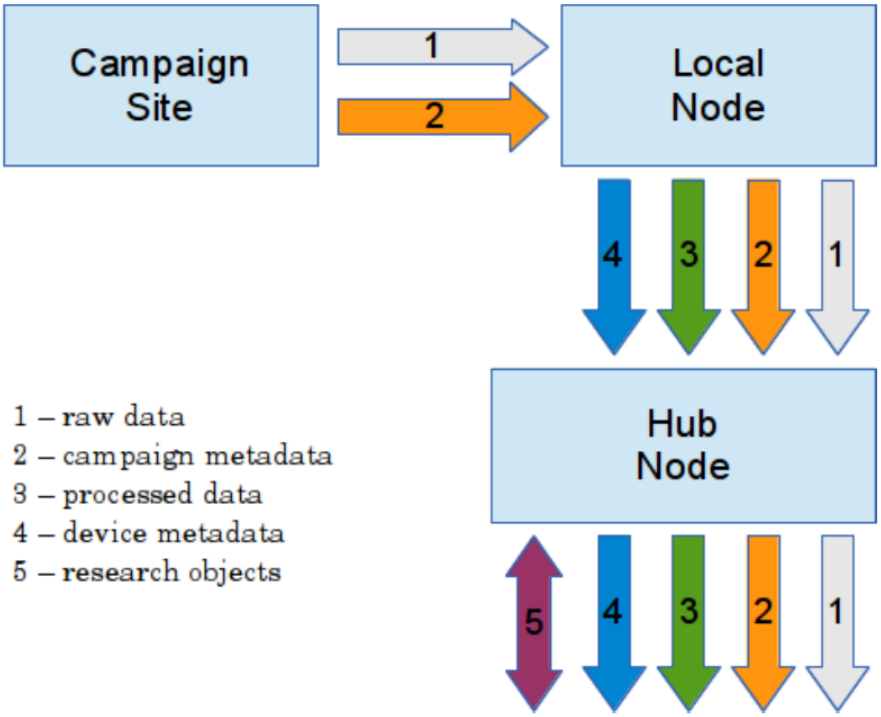



Figure 2.2: Data flow between nodes

those laser sensors, that will be later manipulated by researchers. This platform is also designed for three types of users to interact: data-curator — the user that clean and classify the raw data — data provider — the user that provides the raw data — and researcher — the user that export data and creates reports.

One main characteristic on this platform is the centralized architecture that contains three nodes [15]:

- The campaign site node — place where data created by the sensors will be collected.
- The local node — place where data will be collected from each Campaign Sites and then apply Quality Assurance (QA) process.
- The hub node — place where the e-Science platform is placed that receives data, making them available for other researchers

These nodes will ensure availability of the data anywhere to everywhere, the Figure 2.2 — adapted from [33] — represents the data flow between that three nodes.

This process will send processed data and the raw data to the Hub Node. It's essential to keep the raw data, as errors or inconsistent data could be created on the processed data. The hub node it's a place where the e-Science platform receives data from the Local Node, making them available for researchers [33].

### 2.1.5 Wind2Data

The Wind2Data (W2D) was developed in 2011 by INEGI staff, António Oliveira[3] at INEGI wind energy department. Unfortunately, no documentation was made, as this software was made in-house for that department and the main responsible left the institute during its development. This platform is Excel based and is used to store, retrieve, manipulate and classify wind data from different data loggers making use of a MySQL database for raw data insertion and querying. It also makes use of another database, the Skiron application that stores all the metadata — regarding equipment's, operation and maintenance of the met masts interventions.

### 2.1.6 Conclusion

This Section gives us a review of some existing platforms for data e-Science management worldwide. As already described in Section 2.1.1, unfortunately, most of the companies develop their own software and don't share publicly any content or documentation.

HubZero and EUDAT projects are made for different communities and areas, which is a significant advantage to handle different data and concepts. The Windscanner.eu project is explicitly made for handling data provided from laser sensors — LiDAR but may have scalability and performance issues when adding a high amount of data to the system. Finally, we have Wind2Data. It's the actual platform in INEGI renewable energy department. The MySQL database has scalability problems, as with the need for new towers in the system, is necessary to create new tables in that database. The database also present some limitations when populated with records over $5 * 10^5$, as no indexes was created.

## 2.2 Wind Energy

In this Section, we will introduce some of the theoretical concepts of wind energy, how this energy is generated, how important is the wind resource assessment campaign and how its possible to obtain data from this renewable energy.

### 2.2.1 Introduction

The Sun is the resource that most renewable energy creates in the world, and it is possible to produce energy through this resource directly and indirectly. The directly way most know is through plant cells, and the indirect way most know is trough winds. Global winds are caused by pressure differences across the Earth's surface due to the uneven heating of the planet by solar radiation. These thermal effects combine with dynamic effects from the earth's rotation to produce prevailing wind patterns [38], being one of the fastest-growing and cheapest forms of clean and renewable energy worldwide [4]. In 2017, in Portugal, 40% of the electricity production came from renewable energy and from that value, 25% was from wind source. Worldwide, in the same

[3] https://www.linkedin.com/in/ant%25C3%25B3nio-oliveira-6875b474/

year, 25% of the total electricity production was generated using renewable energies and from that value, only 5% comes from wind source [40]. This is a clear indicator that Portugal is entirely above the average when it comes to the clean generation of electricity.

### 2.2.2 Wind Resource Assessment

"For any power plant to generate electricity, it needs fuel. For a wind power plant, that fuel is the wind." [32]

The wind resource assessment is one of the most critical phases in the development of utility-scale wind farms. Like every other technical project, it requires planning, coordination and can be summarized by a budget and a schedule. The final success depends on the quality of the program's assembled assets [3]. Developing a wind project doesn't always depend on the wind resource. Other aspects need to be taken into account, such as accessibility, grid connection, environmental impact, among others [9].

There are at least 3 stages that are important when approaching a wind resource assessment [9]:

- Initial large-area assessment — this stage tries to identify regions with a good possibility of accepting a wind resource. It's studied especially using Geographic Information System (GIS), but wind atlases or meteorological stations are also used. This stage will provide important information in the selection of new wind measurements sites.
- Evaluation of a specific resource — after choosing a potential region, an area or site is chosen with potential development for a wind farm. At this stage its possible to compare resources with other areas, confirming that this area is justifiable for investigations and also gives a possibility to estimate data production for further analysis.
- Micrositing — it's considered as the smallest scale in wind resource assessment — a few kilometers distance only. The main goal is to analyze, in a specific site of interest, the characteristics of the wind resource to ensure turbine compatibility with reigning conditions and to maximize the energy production of the wind farm.

When studying a wind resource assessment, it's important to have good practices to have the most reliable data. The following points should be taken into action in order to obtain the data best suited to this goal [9]:

- The minimum monitoring duration should be at least two years. This way will be possible to assess the extra-seasonality of the wind.
- The location of the mast should be chosen to provide the best data possible.
- The wind speed should be measured at least at two heights and at each height should be used an anemometer and a wind vane to couple information of wind at that height.
- At each human intervention, preventive or corrective — for example, to replace faulty equipment— a report should be created with full details of the site visit.

- The measuring equipment should be of high quality if possible, maintained and calibrated at the correct campaign intervals.
- The data logger should be chosen, giving special care to accuracy, precision, memory capacity, and reliability. It should also be checked regularly for quality or defects over time [12].

A meteorological mast is composed by several devices or sensors — a mast can have more than 20 sensors, collecting data from several environmental variables such as wind speed, direction, air temperature, atmospheric pressure, and many others. The data logger conversion of main signals into measures is detailed below [2]:

- Anemometer sensor — it's the sensor type most commonly used to measure near-horizontal wind speed. The characteristic curve of the anemometer is given in the form of the linear equation: $v = a * I + B$, where $v$ it's equal to the wind speed in m/s, $a$ equal to slope of characteristic, $I$ equal to the number of impulses registered ( $I > 0$ ) and $B$ equal to the offset of characteristic at 0 m/s.
- Wind vane sensor — this sensor gives the wind direction. Are important when used at the same height as an anemometer. The formula obtains the wind direction: $d = U + O$, where $d$ it's equal to the wind direction in degrees, $U$ equal to the measured voltage and $O$ the offset value (angle of deviation relative to North).
- Humidity/temperature sensor — this sensor gives the air temperature and humidity. The formula obtains the air temperature: $T = a * U - b$ where, $T$ it's equal to the temperature in Celsius, $a$ equal to the slope of characteristic, $U$ equal to the measured voltage and $b$ represents the beginning of measurement range. The formula obtains the humidity: $F = a * U$, where $F$ represents the relative humidity in percentage, $a$ is equals to the slope of characteristic and $U$ is the measured voltage.
- Barometric sensor — this sensor gives the air pressure obtained by the formula: $P = a * U + B$, where P represents air pressure in $hPa$, $a$ equals to the slope of characteristic, $U$ equals to the measured voltage and $B$ represents the offset of characteristic.
- Pyranometer sensor — this sensor measures the solar radiation obtained by the formula: $R = a * U$, where $R$ represents the global radiation in $W/m^2$, $a$ it's equal to the slope of characteristic and $U$ equals to the measured voltage.
- Precipitation sensor — measures the precipitation and its obtained by the formula: $N = a * I$, where $N$ represents the amount of precipitation in $mm$, $a$ equals to the slope of characteristic and $I$ equals to the number of impulses registered.

Data is collected in the field by sensors in a met mast, a data logger stores the data and then this data is sent by a communication device to servers typically on a daily to weekly basis.

As already stated, the purpose of this work is to build a prototype able to run the necessary tasks to manage, store, clean and classify the data and also to run the subsequent reporting as well.

### 2.2.3 Conclusion

This section gives us a brief on how renewable energy, especially wind energy, embraces our quotidian. It is quite positive the fact that Portugal produces 40% of the electricity from renewable sources while the global average is 25%, and from the renewable sources, Portugal produced 25% from wind source while the worldwide average is only 5%. We described some essential stages when developing utility-scale wind farms and their best practices to collect and generate the most reliable data as possible.

## 2.3 Database Systems

In this Section, we will give a review of the theoretical concepts of the database systems — Relational, NoSQL and Time Series Databases — and how these databases are essential in today's systems when dealing with high volumes of data.

### 2.3.1 Introduction

Databases are present in almost every application and Web sites we visit daily, and it's getting more importance in e-Science field as more data is being generated daily and new databases are created to handle these massive amount of data. Databases can be Relational (Relational Model) or NoSQL (Not Only SQL). The main use of the databases is to store, update and retrieve information from a collection of data.

It's crucial to approach ACID (Atomicity, Consistency, Isolation, and Durability) properties as this ensures and maintain consistency in a Relational database and BASE (Basically Available, Soft state, Eventually consistent) properties are in favor of availability and performance in a NoSQL database [5].

### 2.3.2 Relational Databases

We will analyze the systems that follow the Relational Model of a database, being these the MySQL Database and the PostgreSQL Database.

#### 2.3.2.1 Introduction

Relational Database Management Systems (RDBMS) is a technology that is almost mandatory for storing structured data in a system. E. F. Codd first introduced the theoritical basis in 1970 [8], and since that time many software developers start using this DBMS, which is a relational model based.

This model makes the possibility to organize data into tables — also called relations – of columns and rows. Each row — also called records or tuples — have a unique key identification and its where the actual data is "stored". Each column — also called attributes, represents values attributed to an instance.

There are more than 100 Relational Database systems, but MySQL and PostgreSQL are the two most used open source systems to store structured data. In the next sections, a review will be made over those two database systems [23].

#### 2.3.2.2 MySQL Database

MySQL it is one of the most know databases in the world, it is considered as an RDBMS, and it is free and open source software. MySQL AB was created by a Swedish company, that was founded by David Axmark, Allan Larsson and Michael "Monty" Widenius in the early of the 90s, but the first version that comes to the public was on 23 May of 1995 [11].

Initially was designed to work with small and medium applications, but today, attending to the need to creating bigger applications, have a few advantages related to other RDBMS. MySQL has all the characteristics that a database system need and follows the ACID properties. MySQL database stores the data in tables in a indexed sequential access method — ISAM (low-level code) — mode that ensures high performance to tables that have an average space of 100 MB — around $10^6$ records of 1 kB each.

The following features characterize MySQL as a good database:

- Being relational, as it organizes data into one or more tables.
- SubSELECT ensuring the possibility of making two selects in the same query. Example: SELECT * FROM table1 WHERE x IN (SELECT y FROM table2).
- Triggers ensures that a command is automatically executed by the DBMS when a database operation is made, as an insert, update or delete.
- Full-text search gives the possibility to the developer to search for words that are located within a text field and a fast and simple way.

#### 2.3.2.3 PostgreSQL Database

PostgreSQL is a free and open source database system. It's considered as one of the most powerful and advanced databases related to other RDBMS. This system was created from a University project called Postgres in Berkeley University — California, in 1986. At the end of that decade, the first official version was released but didn't have huge popularity. In 1996 new improvements were made ensuring ACID transactions and now is getting more popularity due to his functionalities and performances [34].

There are a few characteristics that make this database one of the most powerful RDBMS:

- Like MySQL, it's a relational database system, as it organizes data into one or more tables.
- ACID transactions that ensure consistency in the database.
- Replication between servers.

- Multithreads that ensures high performance and multi-access from two or more users when accessing the database.
- SSL Security and encryption that ensures safety channels using SHA1 and MD5 encryption algorithms.
- Open source — is free to use by any developer or company.

#### 2.3.2.4 ACID Properties

In 1983 Andreas Reuter and Theo Härder created the ACID acronym as shorthand for Atomicity, Consistency, Isolation, and Durability [17] properties. But this work was firstly named by Jim Gray [16] who enumerated Atomicity, Consistency, and Durability but left out Isolation. These four properties are important because they allow a system to maintain consistency in a database, before and after a transaction, which has influenced many aspects of development in database systems. These properties are characteristic of Relational Databases. The characteristics of these four properties as defined by Andreas Reuter and Theo Härder, are as follows [17]:

- Atomicity — ensure that a database, when making a successful transaction, change their data, or upon a failed transaction it's left unchanged. An atomic system must guarantee atomicity in several situations, including power failures, errors and crashes.
- Consistency — ensures that the transaction can not leave the database in an inconsistent state. If a transaction violates any defined rule, including constraints, cascades and triggers, this transaction is reverted, and the database is left in a known consistency state. This prevents database corruption by an illegal transaction but does not guarantee that a transaction is correct.
- Isolation — ensures that two or more transactions are executed concurrently, leaving the database in the same state that would have been obtained if the transactions were executed sequentially.
- Durability — ensures that upon insertion of data in the database, this action isn't reverted even in the case of a system failure.

### 2.3.3 NoSQL Databases

We will analyze the system that follows the NoSQL Model of a database, being that the MongoDB.

#### 2.3.3.1 Introduction

Nowadays, there's a significant need to store a lot of data in a reliable system. In last years, to improve the performance of the database system, as more data was being entered — called as vertical escalation —, managers had to buy more powerful platforms instead of distributing the database through of multiple servers — called as horizontal escalation. RDBMS have tremendous

difficulty on vertical escalation and the NoSQL DBMS comes to solve that problem, as they are designed to scale out as they grow [36].

These databases have existed since the late 1960s [35], but only in the early of the 21st century they got immense popularity triggered by the needs of the web 2.0 companies like Amazon, Facebook, Google and Yahoo. The reason for their creation was due to the demand of storing more data in a faster way, as the transfer rate is higher than the RDBMS — being free of the join operation helps on transfer rate. But the DBMS has a few disadvantages related to the conventional RDBMS, as there is no standard query language for the DBMS and are BASE compliant, not ACID.

#### 2.3.3.2 MongoDB

MongoDB it's a document database developed by 10gen[4] — now MongoDB. Started to be implemented in 2007 and was initially released in 2009, it was developed using C++, and it's open source. The main advantage of MongoDB it's his high performance — query speed is 10 times faster than MySQL [18] — and efficiency, providing features like consistency fault tolerance, persistence, aggregation and indexing. The documents are mainly stored in BSON format — similar to an XML or JSON document but represented in binary to achieve efficiency, and contain an ordered list of elements consisting of a field name, type and value [27].

Another advantage of the MongoDB is the fact that the documents are polymorphic — fields or attributes can vary from document to document within a single collection. This means that there is no need to declare a structure of the document to the system.

MongoDB has a few disadvantages as it can turn unreliable due no ACID properties, and indexation takes a lot of RAM [36].

#### 2.3.3.3 BASE Properties

When a system requires availability and performance, as eBay or Amazon sites, BASE properties are applied, as the system needs to be available for their customers. Brewer introduced the BASE properties [5] and lost the ACID properties of strict consistency and isolation in favor of availability and performance is a characteristic of NoSQL Databases.

The characteristics of these properties, as defined by Brewer, are as follows:

- Basically Available — ensures that the underlying system remains functional and responsive even upon some of components failure. The returned data may not be consistent or the most recent.
- Soft state — means that even without any data entry, the database may change over time due to internal operation to achieve consistency.
- Eventually consistent — means that when dealing with a lot of data insertion, the database sooner or later will become consistent when no more data is entered. All received data will

[4] https://www.mongodb.com/press/10gen-announces-company-name-change-mongodb-inc

be disseminated to all nodes and the system remains available to receive new entries, not checking the consistency of each transaction before processing the next one.

### 2.3.4 Time Series Databases

In the following sections, we will analyze the Time Series databases, that are specially designed to deal with timestamp or time series data.

#### 2.3.4.1 Introduction

In recent years, many e-Science platforms start to giving special attention to the Time Series databases (TSDB) to store and handle timestamped data. In fact, TSDB, are not new, since it was primarily focused to financial data, the volatility of stock trading, and systems built to solve trading [21]. These databases are built specifically for handling metrics and events or measurements that are timestamped, focusing on the following features [37]:

- Scalability. Many databases even support clustering for high availability.
- Optimized for large scans over many records.
- High performance on writing information.
- Challenges specified to time-based queries simplified — such as manipulating time zones or providing filters for days of the week.

#### 2.3.4.2 InfluxDB

InfluxDB is one of the most recent TSDB. This database is composed of measurements, series, and points. It's designed specially to handle time series data, as each point consists of several key-value pairs — field-set and a timestamp. Series are defined by a group of points, that at the end determine a measurement. This database start to be developed by InfluxData as a project in late 2013 and was specially made for storing large amounts of timestamped data [21]. Based on GitHub stars [5], InfluxDB emerge as the open source tools with the strongest traction when compared to other TSDB [37].

There are a few features that make this database a must use for time series storage [21]:

- It has a datastore written specifically for time series data allowing high ingest speed and data compression.
- Written in Go language, compiling into a single binary with no external dependencies, offering the developer a great performance on a query.
- It's SQL-like query language to query aggregated data easily.
- Allows indexation in series for fast and efficient queries.

[5] https://help.github.com/en/articles/about-stars

### 2.3.5 Conclusion

In this Section, we described how databases are essential when it comes to storing information in the nowadays systems and also some of the most used open source databases. We refer to how NoSQL databases are more efficient when it comes to data insertion and querying over the Relational databases when working with large volumes of data. We additionally described the TSDB to manipulate timestamped data.

It was also important to refer to the ACID transactions when we want consistency over availability in an RDBMS system and to the BASE properties when we want a system that needs to be available almost every time over consistency in the NoSQL databases.

# Chapter 3

# Problem Statement and Solution Proposal

As already described, nowadays there is a great need to store a large volume of data in a scalable and efficient way. A reliable system is necessary for storage and display such amount of data.

This chapter describes the problems to be solved and presents the approaches designed for its resolution.

## 3.1 Problem Statement

Over time, a wind met mast generates a lot of data that needs be to stored in a system. These data can be recorded at different frequencies, typically with 1 Hertz sampling and time integration of 10 min. In certain circumstances the sampling frequencies can reach 100/200 Hertz and integration might not be applied [32] — meaning all records need to be stored — which creates a huge amount of data if we take into account that some of these measurement campaigns can last for several years.

Currently, INEGI's Wind Energy department has a system that imports, processes and allows the visualization of the data generated from the wind towers. The system uses a MySQL database, but has many limitations, as scalability with the number of wind tower and as new metrics, or new data logger information comes with a new time series. If a new wind tower is entered on the actual system, a group of tables are necessary to be created. To manage a wind tower, the system needs four to five tables. INEGI's department has around 300 active stations, this means, a total of 1 200 to 1 500 tables are necessary to manage all stations. Another 900 to 1 000 stations are inactive but should, ideally, migrate to the same system as historical data is often used to establish correlations, long term exercises and other types of analysis. From a disk space view, each time series entry occupy around 90 B. With an average integration time of 1 min, a day will generate 129 600 B of data for a wind tower — In a year the total is around 45 MB. The system has about

1 300 towers, and this means approximately 57 GB of space disk is used to store all wind towers raw data over a year. Due to the enormous amount of data that is produced by each wind tower, it's inefficient to make use of a relational database to store the time series produced [27]. This has mostly to do with the inherent scalability problems described above, as the relational model presents limitations when it is necessary to manage a large number of tuples — sometimes in the order of the billions [27]. When dealing with a few rows in a MySQL database, the system is quite fast, but when querying with a large number, like thousand millions of rows, it takes more than 24 h to return the results [27]. The difficulty of MySQL is not in storing the data, but returning them when we are faced with large amounts of data, meaning that MySQL has scalability problems [27].

Regarding the data visualization, this is done in Wind2Data application — Excel based — which further limits the efficiency of the system. The Excel starts by querying the database and then populates the spreadsheets; this process generates a very heavy operation. For example, when loading one month of data for a single wind tower, the time to query the database and view in Excel takes about 30 s. A request of two or three years volume data becomes very time-consuming.

The data logger stores temporally the signal provided from the different sensors in each channel. If a change of channel is made, during a technical intervention, for example, the data logger configuration may not be up to date, leading to inconsistencies between the real mast configuration and the data logger configuration. This raises a difficulty, meaning it is necessary to be flexible enough to change/update/edit the access to each data set along time as several situations as the one previously described will certainly occur.

## 3.2 Solution Proposal

From the analysis of the problem described previously in Section 3.1, it becomes clear that there is a need to build a new application that handles a large amount of data, storing it efficiently and able to display, classify and export these data. The Figure 3.1 — adapted from [39] — describes a comparison with MySQL and MongoDB when managing a textbook system. It's possible to confirm that MySQL, a relational database, has a few disadvantages regarding insertion and querying, comparing to a NoSQL database. It's possible to verify that MongoDB is almost three times faster when inserting and nearly two times faster when querying a large amount of data [39]. Other tests made prove that MongoDB have 10 times better query performance when comparing to MySQL [18].

From a performance view, when comparing InfluxDB over MongoDB for time series data, it's possible to conclude from the Figure 3.2 — adapted from [7] — that InfluxDB had a better performance by 2.4x when it came to data ingestion. The same study proves that InfluxDB has 65x better performance when comparing disk space used to store the data — InfluxDB occupies 145 MB, and MongoDB occupy 9 420 MB — and it is 5.7x better when comes to query performance — From 87 264 000 records in the dataset.

Many other features were not considered due to the limited amount of time available to accomplish this work. As in many different areas, the wind sector is progressively adapting and

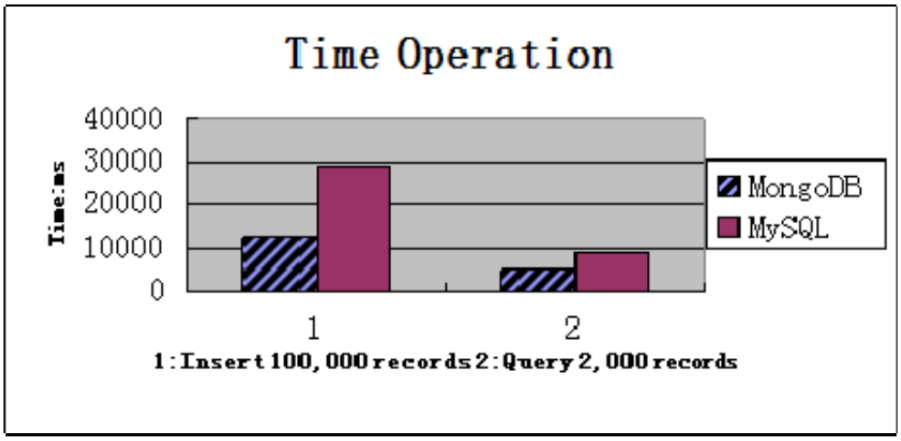



Figure 3.1: MySQL and MongoDB comparison

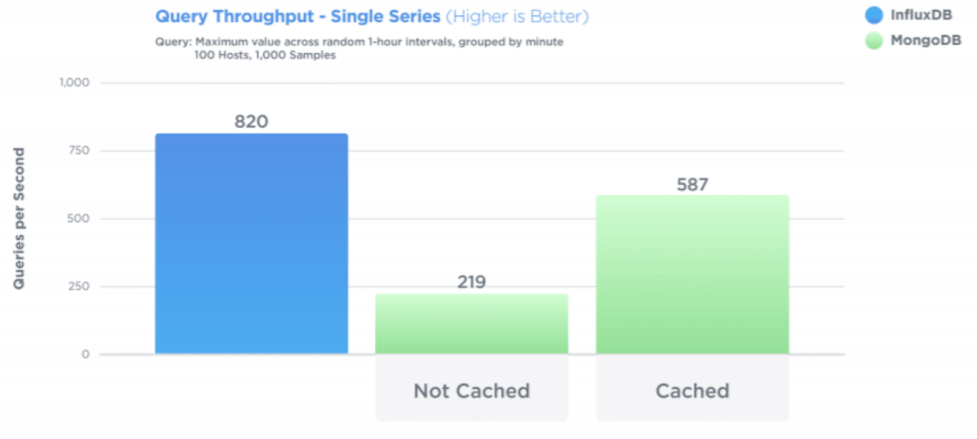



Figure 3.2: InfluxDB and MongoDB comparison

taking advantage of the digital transformation that is currently taking place worldwide. Thus, the integration of the information of several areas of interest brings additional added value to software solutions like this one. This topic will be addressed later in the Chapter 7.2, concerning future works.

The new system will be Web-based and built using Django, a Python framework. With PostgreSQL database, we will manage static information, such as tower information, sensors information and classification information. The raw data will be stored in a NoSQL database using MongoDB and InfluxDB. From a performance point of view, the raw data can also be stored in PostgreSQL and then compare the query performance with the MongoDB and the InfluxDB — it will also be possible to compare the performance between MongoDB and InfluxDB.

As already described in Section 3.1, data can be inconsistent, as raw data need the data logger configuration to know what values belong to each sensor and metrics. To ensure data integrity, the system will store the raw data as received from the data logger and then apply different configurations to that data. Those configurations will be saved in the database to easily use them if new data logger configurations come over time or if data exportation or reports are necessary. Data can also be invalid during a period, the system will be able to classify that period with the corresponding activity, as the data can be classified as no problem, frozen, extreme heat and many others.

## 3.3 Conclusion

In this chapter, it was possible to identify and describe the problems to be solved and the approaches designed for their resolutions. It was possible to conclude that Wind2Data have an inadequate database design and also presents scalability problems. It's essential to study the impact that a relational database system will have on the performance of the platform. To make this study possible, it will be necessary to approach the problem using three different databases: one using a relational database for the meta-information — as sensors, towers, configuration and classification period — and other two using a NoSQL and a Time Series databases for the raw data — for a scalability and performance comparison.

# Chapter 4

# Requirements Analysis and Architecture

In this chapter, we will describe the Actors and User Stories of the system and approach the architecture and technologies that will be used to implement this work. It's important to study and analyze the user needs to arrive at a definition of the problem domain and system requirements. This will help us to get a better description of the problem and the solution, such as detection of conflicts between user requirements, prioritize and triage of requirements, define different levels of access between users and especially to estimate costs and work necessary to develop the project.

## 4.1 Actors

An actor represents an entity that interacts with the system that is being specified. This can include people, external systems, and other organizations. Actors are always foreign to the system that is being modeled and never part of the system [1]. The Figure 4.1 represent the actors in the system.

- User — Generic user. Can consult project public information.
- Visitor — Non authenticated user. Can consult project public information and do login.
- Authenticated — Authenticated user, can be a manager, administrator or a customer. Can edit their personal information.
- Manager — Can view all wind stations in the system and edit those associated to him. Can add equipment, such as data loggers and sensors (anemometer, wind vane and others). From the wind stations associated to the manager, it's allowed to add and edit period of configuration's, sensor settings, measurement information, classification periods and comments.
- Administrator — Can manage all authenticated users. Will assign winds stations to a manager and will also assign masts to the clients.

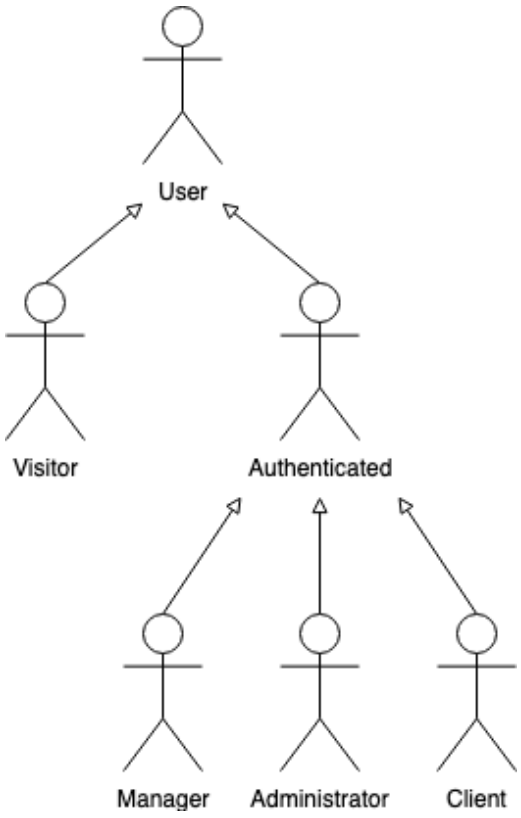



Figure 4.1: Actors

- Client — Can export data and reports from a wind station during a period.

## 4.2 User Stories

A user story is a high-level definition of a requirement, containing only the information necessary for developers to produce a reasonable estimate of the effort required to implement it [1]. It's a brief description of potential interaction with the system by one of its users focuses only on behavioral requirements rather than on design aspects.

The Table 4.1 describes each user story with their priority, low or high, giving priority to the high level to develop first. The low priority can be developed in the future.

Table 4.1: User Stories

| Identifier | Name | Description | Priority |
|---|---|---|---|
| US01 | Login | As a visitor, I want to make the login, so that I have access to the all private information in the system, like wind towers, equipment's and raw data | High |
| US02 | Profile | As an authenticated, I want to edit my personal information, so that I can keep them updated | High |
| US03 | Add station | As a manager, I want to add a new station, so that later I can add raw data and equipment's information | High |
| US04 | Edit station | As a manager, I want to edit information of an station, associated to me, so that I can keep them up-to-date. This information can be the name or the position of a station | High |
| US05 | Remove station | As a manager, I want to remove a station, associated to me, so that I can keep my system up-to-date | High |
| US06 | Add cluster | As a manager, I want to create a cluster of stations, so that I can have a group for stations | High |
| US07 | Edit cluster | As a manager, I want to edit the information from a cluster, so that I can keep them up-to-date | High |
| US08 | Remove cluster | As a manager, I want to remove a cluster, so that I can keep my system up-to-date | High |
| US09 | Add equipment | As a manager, I want to add new equipment with a name, so that I keep tracking equipment information. This equipment can be a data logger or a sensor. | High |
| US10 | Edit equipment | As a manager, I want to edit the static information of equipment, so that I can keep them up-to-date | High |
| US11 | Remove equipment | As a manager, I want to remove equipment, so that I can keep my system up-to-date | High |

Continues on next page...

Table 4.1 – continued from previous page

| Identifier | Name | Description | Priority |
|---|---|---|---|
| US12 | Add equipment characteristic | As a manager, I want to add a new characteristic to equipment with static information as, manufacturer, model, version, designation and serial number, so that I keep tracking equipment information. | High |
| US13 | Edit equipment characteristic | As a manager, I want to edit the static information of a characteristic of equipment, so that I can keep them up-to-date | High |
| US14 | Remove equipment characteristic | As a manager, I want to remove an equipment characteristic, so that I can keep my system up-to-date | High |
| US15 | Add calibration | As a manager, I want to add new calibration information, like offset, slope, date, reference, belonging to equipment, associated to a dimension type so that I keep tracking equipment information. | High |
| US16 | Edit calibration | As a manager, I want to edit the static information of an calibration, so that I can keep them up-to-date | High |
| US17 | Remove calibration | As a manager, I want to remove an calibration, so that I can keep my system up-to-date | High |
| US18 | View stations | As a manager, I want to view a list of all stations in the system, so that I can get an overview of all the stations and select one of them to view all information | High |
| US19 | View station | As a manager, I want to view information of a chosen station, so that I can see the static information, period of configuration's and comment's | High |
| US20 | Add comment | As a manager, I want to add a new comment, internal or external, to a certain station, with the possibility to associate equipment to this comment, so that I keep all users updated with any extra information | Medium |
| US21 | Edit comment | As a manager, I want to edit the information from a comment, so that I can keep them up-to-date | Medium |
| US22 | Remove comment | As a manager, I want to remove a comment, so that I can keep my system up-to-date | High |
| US23 | Add equipment configuration | As a manager, I want to add a new equipment configuration, that contains logger offset, logger slope and height information, so that later I can complete data raw information | High |

Continues on next page...

Table 4.1 – continued from previous page

| Identifier | Name | Description | Priority |
|---|---|---|---|
| US24 | Edit equipment configuration | As a manager, I want to edit information of an equipment configuration, so that I can keep them up-to-date | High |
| US25 | Remove equipment configuration | As a manager, I want to remove an equipment configuration, so that I can keep my system up-to-date | High |
| US26 | Add classification | As a manager, I want to add a new classification period to an equipment configuration, that contains status information (OK, frozen, damaged and others), so that I can classify data raw information | High |
| US27 | Edit classification | As a manager, I want to edit information of a classification period, so that I can keep them up-to-date with correct information | High |
| US28 | Remove classification | As a manager, I want to remove a classification, so that I can keep my system up-to-date | High |
| US29 | Add dimension | As a manager, I want to add dimension information to an equipment configuration, that contains information as row (help for reading raw data), and a dimension type, so that later I can complete data raw information | High |
| US30 | Edit dimension | As a manager, I want to edit information of dimension information, so that I can keep them up-to-date with correct information | High |
| US31 | Remove dimension | As a manager, I want to remove a dimension, so that I can keep my system up-to-date | High |
| US32 | Add period configuration | As a manager, I want to add a new period of configuration to a certain station and associate equipment configuration's to this period, so that I can have different periods to different data logger configuration's | High |
| US33 | Edit period configuration | As a manager, I want to edit information of a period of configuration, so that I can keep them up-to-date with correct information | High |
| US34 | Remove period configuration | As a manager, I want to remove a period of configuration, so that I can keep my system up-to-date | High |

Continues on next page...

Table 4.1 – continued from previous page

| Identifier | Name | Description | Priority |
|---|---|---|---|
| US35 | View period configuration | As a manager, I want to view information of a period of configuration, so that I can see the static information and a list of configuration of equipment's from this period of configuration | High |
| US35 | Add auto-configuration | As a manager, I want to add a new period of configuration to a certain station by a data logger file configuration, so that each period of configuration knows where to read each position on the time series automatically | Low |
| US36 | View raw data | As a manager, I want to visualize raw data and classifications in two plots by choosing a certain station and a period time, so that I can see and classify that data and clean it | High |
| US37 | View neighbors data | As a manager, I want to visualize data from the neighbor's stations, by choosing a period time and a radius, so that I can compare the clean data between them | Low |
| US38 | Insert raw data | As a manager, I want to upload raw data, so that I can keep my platform updated with the last existing raw data from wind towers | High |
| US39 | Export INEGI | As a manager, I want to export data in INEGI native format, so that I keep the clean data ready to be saved in an INEGI server | Low |
| US40 | Create report | As a manager, I want to create a report, that contains plots, as wind rose, diagram rose, energy rose, and tables with general and important statistics of monthly campaigns, so that I can have a better visualization of the wind station data and deliver them to the clients | Medium |
| US41 | Add equipment type | As a manager, I want to add a new equipment type (Example: anemometer), so that I can use this new equipment in the system | High |
| US42 | Edit equipment type | As a manager, I want to edit information about an equipment type, so that I can keep them up-to-date with correct information | High |
| US43 | Remove equipment type | As a manager, I want to remove an equipment type, so that I can keep my system up-to-date | High |

Continues on next page...

Table 4.1 – continued from previous page

| Identifier | Name | Description | Priority |
|---|---|---|---|
| US44 | Add dimension type | As a manager, I want to add a new dimension type, that contains component, statistic, unit and metric types, so that I can use this new dimension type in dimensions and/or calibrations of equipment's | High |
| US45 | Edit dimension type | As a manager, I want to edit information of a dimension type, so that I can keep them up-to-date with correct information | High |
| US46 | Remove dimension type | As a manager, I want to remove a dimension type, so that I can keep my system up-to-date | High |
| US47 | Add component type | As a manager, I want to add a new component type (Example: horizontal), so that I can use this new equipment in the system | High |
| US48 | Edit component type | As a manager, I want to edit information of a component type, so that I can keep them up-to-date with correct information | High |
| US49 | Remove component type | As a manager, I want to remove a component type so that I can keep my system up-to-date | High |
| US50 | Add statistic type | As a manager, I want to add a new component type (Example: average), so that I can use this new equipment in the system | High |
| US51 | Edit statistic type | As a manager, I want to edit information of a statistic type, so that I can keep them up-to-date with correct information | High |
| US52 | Remove statistic type | As a manager, I want to remove a statistic type, so that I can keep my system up-to-date | High |
| US53 | Add unit type | As a manager, I want to add a new unit type (Example: m/s), so that I can use this new equipment in the system | High |
| US54 | Edit unit type | As a manager, I want to edit information of a unit type, so that I can keep them up-to-date with correct information | High |
| US55 | Remove unit type | As a manager, I want to remove a unit type, so that I can keep my system up-to-date | High |

Continues on next page...

Table 4.1 – continued from previous page

| Identifier | Name | Description | Priority |
|---|---|---|---|
| US56 | Add metric type | As a manager, I want to add a new metric type (Example: speed), so that I can use this new equipment in the system | High |
| US57 | Edit metric type | As a manager, I want to edit information of a metric type, so that I can keep them up-to-date with correct information | High |
| US58 | Remove metric type | As a manager, I want to remove a metric type, so that I can keep my system up-to-date | High |
| US59 | Add status type | As a manager, I want to add a new status type (Example: Frozen), so that I can use this new equipment in the system | High |
| US60 | Edit status type | As a manager, I want to edit information of a status type, so that I can keep them up-to-date with correct information | High |
| US61 | Remove status type | As a manager, I want to remove a status type, so that I can keep my system up-to-date | High |
| US62 | Add user | As an administrator, I want to add a new user, so that this user be part of the system | High |
| US63 | Ban user | As an administrator, I want to ban a user, so that this user doesn't have more access to the system | High |
| US64 | Add user type | As an administrator, I want to add a new user type, so that later I can associate the type of user in a user | High |
| US65 | Edit user type | As an administrator, I want to add edit information from a user type, so that I can keep them up-to-date | High |
| US66 | Remove user type | As an administrator, I want to remove a user type, so that I can keep my system up-to-date | High |
| US67 | Associate stations | As an administrator, I want to associate an access period to a user and a station, so that this user manage only stations associated with him | High |
| US68 | Edit station association | As an administrator, I want to edit the information from an association, so that I can keep them up-to-date | High |
| US69 | Remove stations association | As an administrator, I want to remove an association so that I can ensure the user doesn't have more access to the station | High |

Continues on next page...

Table 4.1 – continued from previous page

| Identifier | Name | Description | Priority |
|---|---|---|---|
| US70 | Export data | As a client, I want to export cleaned data from a wind station associated with me, so that I can analyze and create reports | Medium |
| US71 | Create report | As a client, I want to create a report from a wind station associated with me, that contains plots, as wind rose, diagram rose, energy rose, and tables with general and important statistics of monthly campaigns, so that I can have a better visualization of the wind towers data | Medium |

## 4.3 Non-Functional Requirements

Some requirements actually aren't necessary to implement any line of code but have a crucial impact when developing a system. These are called "non-functional requirements" and relates important aspects as performance, security, usability and compatibility. It is vital to identify these requirements because they force limitations in the architecture of the platform to be developed [1].

The platform must provide these non-functional requirements:

- Accessibility — the platform should be user-friendly on the Web interfaces
- Availability — the platform must be available almost every time.
- Cost — the platform must use open source components.
- Data Integrity — the platform must maintain accuracy and consistency of the data over its entire life-cycle.
- Fault tolerance — the platform must continue operating correctly in case of the failure of some of its components.
- Performance — the platform must respond in a considerable time to the user actions.
- Scalability — the platform must provide enough storage and be scalable for the raw data collected from wind towers and meta-information that will be entered in the system over time

## 4.4 Conceptual Model

The class diagram is one of the most important UML (Unified Modeling Language) diagrams. The main objective of this diagram is to demonstrate the identification and description of the entities

of the problem domain and also the relationships between them. The class diagram represents a static view of how classes are organized, emphasizing on how to define their logical structure [10].

The Figure 4.2 represents the class diagram designed for the system to be implemented. The *Station* entity will handle meta-information from a *Tower* or a *Machine*, storing attributes as code, name and geographical coordinates of its position. A *Manager User* can handle *Station*'s associated to him during a period and a *Client User* can see the information of the *Station*'s associated to him also during a period. The *Station* entity will also contain *Period_Configuration*'s to represent the begin and end date of a campaign period. To a *Period_Configuration* entity several *Equipment_Configuration* are related to store information about the configuration of the equipment as height, orientation, offset and slope from the data logger. A *Equipment* containing a serial number has *Equipment_Characteristic* relation that includes static information as the manufacturer, model, version and designation. A *Calibration* containing the offset and slope is related to a *Equipment* to be used in *Equipment_Configuration*'s. To the *Equipment_Configuration* entity two relations will be made, a *Classification_Period* entity, to classify raw data from a begin and end data containing a *Status*, and a *Dimension* entity, containing auxiliary attributes and relation as column to read in the raw data and *Dimension_type*. *Comment*'s can be given to a *Station* and a *Classification_Period*. In the *DataSet* we will store the raw data.

## 4.5 Architecture

The platform to be created will follow the architecture MVC (Model-View-Controller), developed by Trygve Reenskaug for the Smalltalk platform in the late 1970s [26]. Since then, it has played an influential role in most UI (User Interface) frameworks and in the thinking about UI design. Almost every web framework is based on this architecture, because separate the user interface from the underlying data represented by the user interface. The Figure 4.3 represents this architecture.

In MVC, the Model represents an entity of the domain and maps in a table of the database. The View is the UI, and it's actually what the user sees in the Browser and are represented by HTML/CSS/JavaScript files. The Controller is the middleman that connects the View and Model, meaning that it is the one passing data from the Model to the View.

The Figure 4.4 represents the components of the system and how they interact with each other to achieve the proposed objectives. The Controller Layer takes cares of actions made by a user, calling the correct service by GET and POST request calls. The Business Logic makes the separation from the Controller Layer and the Data Layer and takes care of the calling the correct functions and methods — available in the Service Layer — chosen the by the user in the Controller Layer. The Data Access Object Layer will provide access to the databases in the system, and it's used to separate the low-level API from the highest-level layers. The Data Layer is where the databases will be present, and this databases will be PostgreSQL, Mongo and Influx.

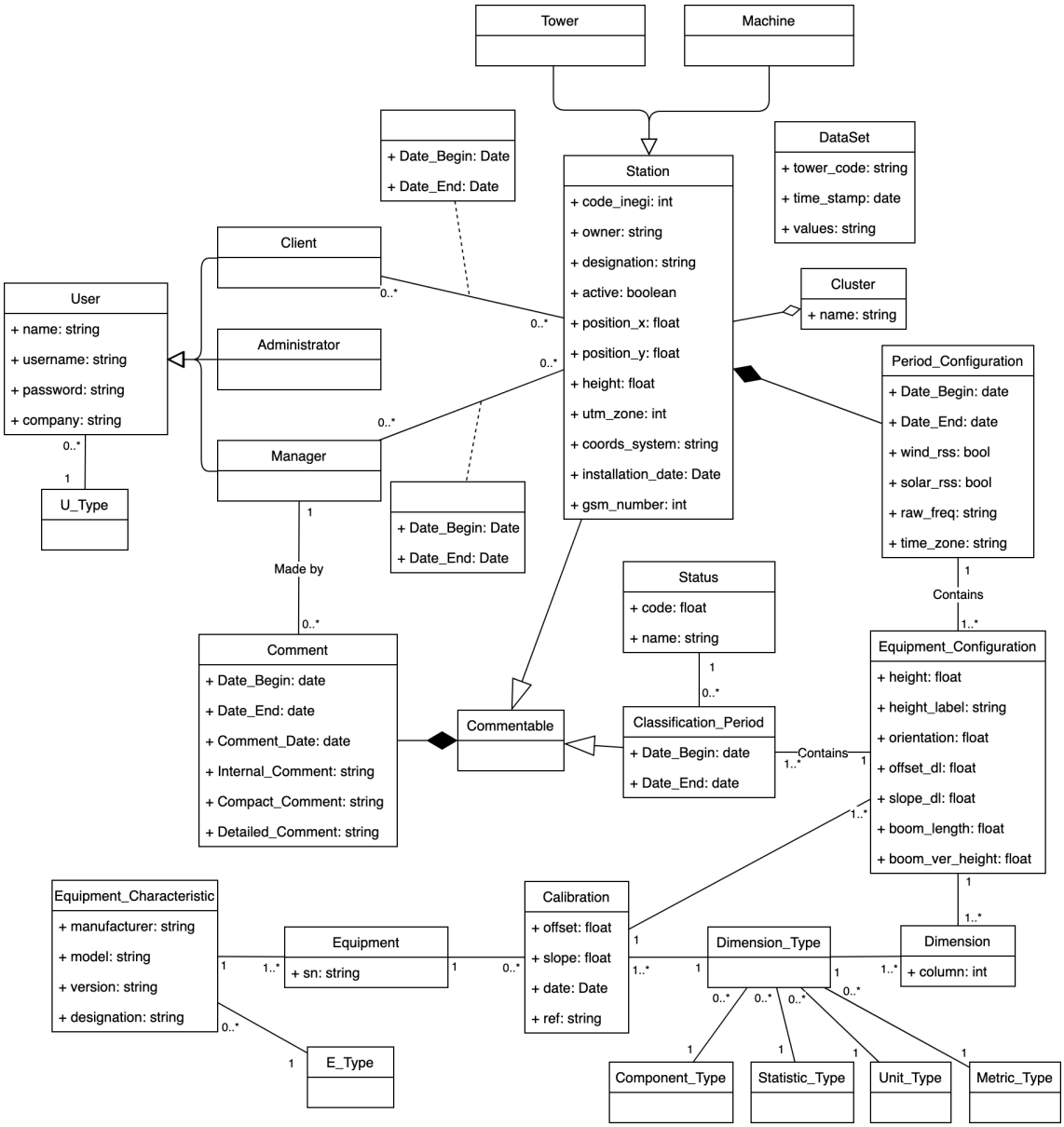



Figure 4.2: Model of the Domain

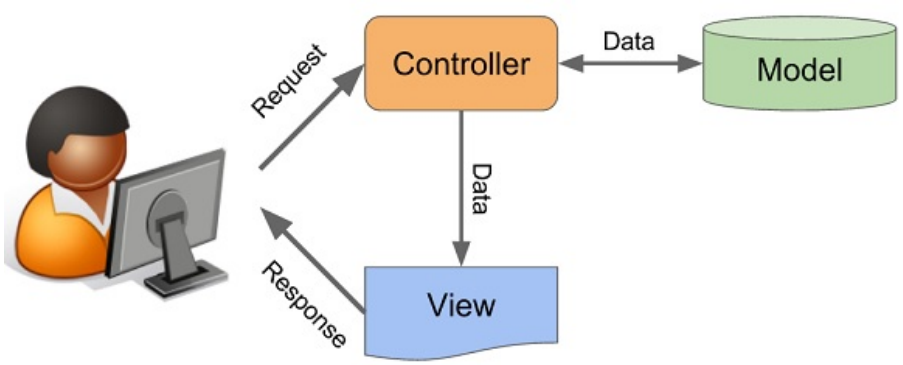



Figure 4.3: Model View Controller

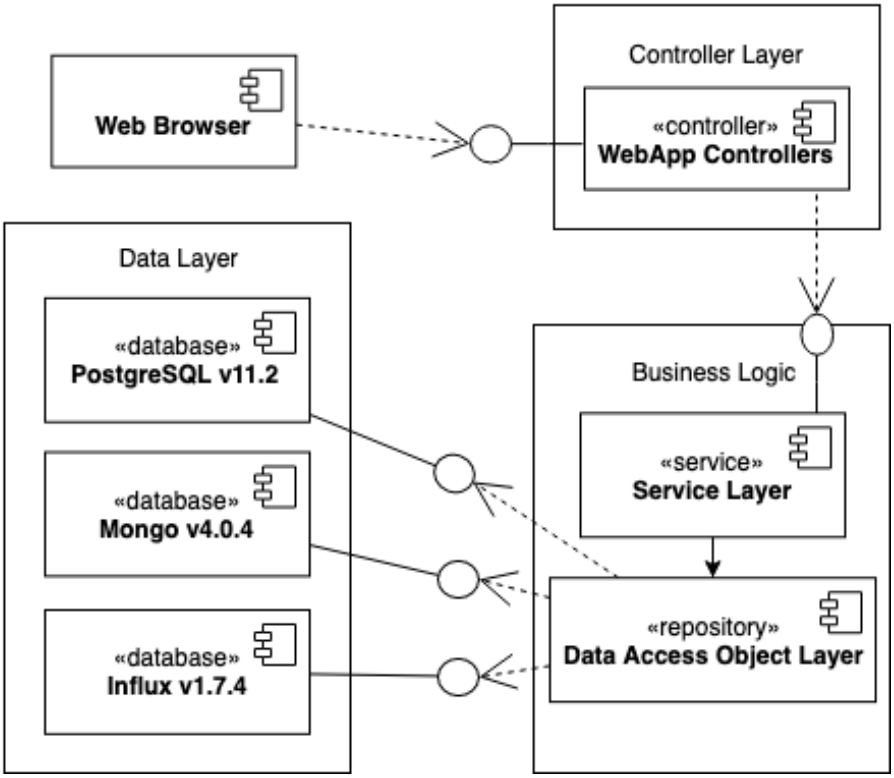



Figure 4.4: Component diagram

## 4.6 Selected Technologies

In this Section, we will describe the selected technologies to develop the system. These technologies were chosen specially for being open source, free of use and developer friendly:

- Django[1] — is a high-level Python Web framework giving the developer the possibility of rapid development and pragmatic, clean design. A web framework is a toolkit of components all web applications need, making life easier for developers, as it is "out of the box", having everything necessary to create a web platform in one product, working seamlessly together, following consistent design principles. It's free and open source. Django uses the MVC architecture, thus separating the UI from the logic layer, what is a plus for this choice. One main advantage of using Django are the packages, allowing the developer to choose according to their needs. It also has a good online community and documentation, which makes more points in favor to choose this Web framework.

- Pandas[2] — is an open source software library written for the Python programming language. Gives the developer the ability to data manipulation and analysis, offering data structures and operations for manipulating numerical tables and time series.

- PostgreSQL[3] — as already described in Section 2.3.2.3, is a powerful open source object-relational database system that uses a SQL language. Ensures the developer's reliability, data integrity and extensibility, enforcing the ACID properties.

- MongoDB[4] — as already described in Section 2.3.3.2, is an open source and cross-platform document-oriented database system. It is classified as a NoSQL database system, that uses documents similar to JSON with schemas.

- InfluxDB[5] — as already described in Section 2.3.4.2, is an open source time series database developed by InfluxData that uses a SQL-like language. It is written in Go language and offers the developer a great performance on querying, high-availability storage and retrieval of time series data. It is classified as a NoSQL database system.

## 4.7 Summary

In this chapter, we described the Actors, that will help us to understand the different access levels in the system by each user. We illustrated the different User Stories, that contain different interactions in the system by those users, helping the developer to prioritize the requirements to be developed first. We also described the architecture and technologies that will be used for the development of the system.

[1] https://www.djangoproject.com/
[2] https://pandas.pydata.org/
[3] https://www.postgresql.org/
[4] https://www.mongodb.com/
[5] https://www.influxdata.com/

# Chapter 5

# Implementation

In this chapter, we will present the actual implementation. We will describe several approaches on how to store and handle the raw data with equipment's configurations and finally showing it in charts.

## 5.1 Introduction

After careful deliberation on the architecture in Chapter 4, we started by implementing the application in Django. It was necessary to maintain all the code readable, easy to maintain and consistent to simplify the development of new features.

We start by creating a basic Django project using an open source template. The content was chosen to give the customer needs, being responsive and clean using bootstrap and jquery frameworks.

The application separates the service layer and data layer, where the service layer works independently of the data layer that is taking care of. This architecture means that the service layer is a higher top layer level and will handle the three databases systems that are in a lower layer level, what is a good programming practice, as we can avoid replicated code and also gives the user the ability to access any of the three database systems easily within the same application. It was also necessary to make this approach as we will compare the performance of each database system.

It was also made good programming practices during the development, as commenting the essential views/functions and classes, handling the exceptions and errors properly and giving the correct attributes, functions, views and URL routes names.

## 5.2 Django Framework

Django is a robust web framework using Python language that gives the programmer the necessary tool-kits to start coding a web application faster and easier.

Table 5.1: Routes for Tower entity

| URL | Type | Description |
|---|---|---|
| tower/new | GET | Form to add a new tower. |
| tower | POST | Add a new tower. |
| towers | GET | List all towers. |
| tower/{id} | GET | Show tower information, period's of configuration and comments from that tower. |
| | PUT | Update tower information. |
| | DELETE | Remove a tower. |

As described in Section 4.5, Django separates different layers using a Model View Controller architecture, that fits perfectly in the application. We need to show the user a User Interface — UI — the Controller handles the different user actions and the Model stores and load the data to the application using, in turn, the three different databases systems.

The Django framework works in three different stages:

1. When the Browser makes POST or GET requests to Django — Web Server — this return responses in HTML or JSON.

2. Web Server — start by looking in the *urls.py* the URL called by the browser and then call the corresponding view in *views.py*. The *views.py* handle all actions — load template HTML code, load forms, handle data and database calls — and then returns a *HttpResponse*. When making a database call in the middle, Django looks for the *settings.py* for the database connection properties.

3. Database — Save, update and load, all information sent and received by the Web Server.

### 5.2.1 Naming Resource Routes

Defining routes allows the Django dispatcher to call the controllers with the respective views from the introduced URL. Giving clean route names is an essential detail in a high-quality Web application, helping the following developers to understand the calls easily and on creating new features.

These routes are defined in an array called *urlpatterns* in *urls.py*. When a user requests a page, Django will search in the *urlpatterns* the route name, if it's matched, the controller will call the corresponding view or function. If it isn't matched, or if an exception is raised during any point in this process, Django invokes an appropriate error-handling view. It's also possible to send values through the URL; this is defined with regular expressions on each route.

The Table 5.1 shows the available URL in the platform for *Tower* entity with their call types — POST or GET — and description using Web services RESTful. In the Appendix A, we can see all the URL for all entities in the platform.

### 5.2.2 Django Packages

One of the main advantages of Django is being Python based, which gives the developer the possibility of using any Python packages. To help to develop the project, we make use of some open source packages created by the Django and Python community.

Django already comes with the basic standards to start a web application, but for our needs, we installed the next packages:

- django-autocomplete-light[1] — allow inputs in forms to make autocomplete with information or objects present in the database.
- django-bootstrap-datepicker-plus[2] — displays in the input form a modal with date picker making use of Bootstrap and jQuery.
- django-bootstrap4[3] — package of Bootstrap 4 that allows to build responsive web pages.
- django-chartjs[4] — allows to display charts from Chart.js and HighCharts in the HTML page using JavaScript.
- django-widget-tweaks[5] — helps the developer to make full forms customization with several forms fields like, Text, Email, Dates, Radio and Check box's and many more.
- django-pandas[6] — allow the developer to make use of Pandas framework to manage data structures and make data analysis more easily.
- influxdb[7] — connector to the Influx database.
- pymodm[8] — connector to the MongoDB database.

These packages not only help to create web applications faster but also allow to connect to different databases systems in the same application. Some other packages were also tried to handle the connection between databases and showing charts that will be described in the next section.

## 5.3 Databases and Data Visualization Approaches

The web application makes use of three different database systems using the same service layer. This type of implementation is essential because it makes the separation of each data layer when using the same interface. This implementation was also crucial as we had to compare insertion and

[1] https://github.com/yourlabs/django-autocomplete-light
[2] https://github.com/monim67/django-bootstrap-datepicker-plus
[3] https://pypi.org/project/django-bootstrap4/
[4] https://github.com/peopledoc/django-chartjs/
[5] https://pypi.org/project/django-widget-tweaks/
[6] https://github.com/chrisdev/django-pandas
[7] https://github.com/influxdata/influxdb-python
[8] https://github.com/mongodb/pymodm

```
1 class Cluster(models.Model):
2     name = models.CharField(unique=True, max_length=100)
3     towers = models.ManyToManyField('Tower', verbose_name="list of towers", blank=
          True)
```

Listing 5.1: Example of Cluster model in Django

querying performance of each database system and helps the developer to avoid code replication what is a good programming practice.

The first approach uses a PostgreSQL database system. This database stores all the meta information of the system referred on Figure 4.2: as towers, periods of configuration's, equipment's and much more. It is also storing all raw data provided from the wind towers. The second approach is using a MongoDB database system, where is storing only the raw data. The third approach is also for storing only raw data, and it's the Influx database system.

It's expected that these database systems store at least 156 millions of records, what corresponds to raw data of then years with rates of a timestamp of then minutes over 300 stations.

### 5.3.1 Database — PostgreSQL

We choose PostgreSQL database system to store all meta information of the system for being an RDBMS, what helps us to manage all information provided from the wind towers, equipment's and periods of configuration's that the system will face over periods. The main goal of this structure was to avoid scalability problems as the current solution from INEGI's application is facing: instead of creating a new group of tables to manage a new tower, our solution only needs one row in some tables to handle that same information. All of the meta information was described in the conceptual model of Section 4.4. Detailed knowledge of this RDBMS was described in Section 4.6.

One of the main difficulties that we faced while studying this architecture was to avoid replicated information inside the tables, as we are taking care of several equipment's that can be used in several periods of configuration, but normalizing the database helped us to avoid this problem.

Django already provides a connector to the PostgreSQL Database, and we only had to describe all of our relations into Python classes inside the *models.py* file, then the Django automatically converts into the PostgreSQL database tables. For example, the *Cluster* model has the syntax of Listing 5.1 in Django. This syntax means a *Cluster* has a field with a name and towers associated and towers that can be used in this or other clusters. This relationship is a ManyToManyField in Django, in an RDBMS it's referred to a Many-To-Many relationship. The table definition in PostgreSQL from the *Cluster* model is represented in Listing 5.2. As we can see, Django helps the developer creating the models for the database without writing any SQL language. It also creates constraints automatically to give rules for the data in a table.

```
1  TABLE cluster (
2      id SERIAL PRIMARY KEY,
3      name character varying(100) NOT NULL UNIQUE
4  );
5
6  TABLE cluster_towers (
7      id SERIAL PRIMARY KEY,
8      cluster_id integer NOT NULL REFERENCES cluster(id) DEFERRABLE INITIALLY
           DEFERRED,
9      tower_id integer NOT NULL REFERENCES tower(id) DEFERRABLE INITIALLY DEFERRED,
10     CONSTRAINT cluster_towers_cluster_id_tower_id_403c5ce9_uniq UNIQUE (cluster_id,
           tower_id)
11 );
```

Listing 5.2: Example of a table definition in SQL

Like a ManyToManyField relationship, Django also offers other associations to refer to other tables:

- OneToOneField — in PostgreSQL it's referred to a One-To-One.
- ForeignKey — in PostgreSQL it's referred to a Many-To-One.

Table 5.2 help us understand how all the classes defined in *models.py* were implemented into the PostgreSQL tables.

Table 5.2: Relational Schema

| **Relation** | **Description** |
|---|---|
| R01 | datasetpg(**id**, tower_code NN, time_stamp, value NN) |
| R02 | tower(**id**, code_inegi UK NN, code_aux1, code_aux2, code_client, designation, position_x NN, position_y NN, utm_zone, coords_system NN, installation_date, client -> affiliationtype, project, parish, district, country NN, gsm_number) |
| R03 | machine(**id**, code_inegi UK NN, designation, position_x NN, position_y NN, utm_zone, coords_system NN, installation_date, client -> affiliationtype, project, parish, district, country NN) |
| R04 | cluster(**id**, name UK NN) |
| R05 | cluster_towers(cluster_id -> cluster, tower_id -> tower) |
| R06 | myuser(**id**, password NN, username UK NN, full_name NN, is_staff NN DF False, is_client NN DF False, is_manager NN DF False, group_type_id -> usergrouptype, affiliation_id -> affiliationtype) |
| R07 | usergrouptype(**id**, name UK NN) |
| R08 | affiliationtype(**id**, name UK NN) |

Continues on next page...

Table 5.2 – continued from previous page

| **Relation** | **Description** |
|---|---|
| R09 | myuser_towers(myuser_id -> myuser, usertowerdates_id -> usertowerdates) |
| R10 | usertowerdates(**id**, begin_date NN, end_date NN CK end_date > begin_date AND end_date <= Today, user_id -> myuser NN) |
| R11 | usertowerdates_tower(usertowerdates_id -> usertowerdates, tower_id -> tower) |
| R12 | periodconfiguration(**id**, begin_date NN, end_date CK end_date > begin_date AND end_date <= Today, wind_rss NN DF False, solar_rss NN DF False, raw_freq NN, time_zone NN, tower_id -> tower NN) |
| R13 | equipmentconfig(**id**, height NN, height_label, orientation, boom_length, boom_var_height, offset_dl NN DF 1, slope_dl NN DF 1, calibration_id -> calibration NN, conf_period_id -> periodconfiguration NN) |
| R14 | calibration(**id**, offset NN, slope NN, calib_date, ref NN, equipment_id -> equipment NN, dimenstion_type_id -> dimensiontype NN) |
| R15 | equipment(**id**, sn UK NN, model_id -> equipmentcharacteristic NN) |
| R16 | equipmentcharacteristic(**id**, manufacturer, model, version, designation, output NN, gama, error, sep_field, sep_dec, sep_thousand, type_id -> equipmenttype NN) |
| R17 | equipmenttype(**id**, name UK NN) |
| R18 | classificationperiod(**id**, begin_date NN, end_date NN CK end_date > begin_date AND end_date <= Today, equipment_configuration_id -> equipmentconfig NN, status_id -> status NN, user_id -> myuser NN) |
| R19 | status(**id**, code UK NN, name UK NN) |
| R20 | dimension(**id**, column NN, dimenstion_type_id -> dimensiontype NN) |
| R21 | dimenstiontype(**id**, metric_id -> metrictype NN, statistic_id -> statistictype NN, unit_id -> unittype NN, component_id -> componenttype NN) |
| R22 | metrictype(**id**, name UK NN) |
| R23 | statistictype(**id**, name UK NN) |
| R24 | unittype(**id**, name UK NN) |
| R25 | componenttype(**id**, name UK NN) |
| R26 | comment(**id**, begin_date NN, end_date NN CK end_date > begin_date AND end_date <= Today, comment_date NN DF Today, internal_comment, compact_comment, detailed_comment) |
| R27 | comment_tower(**id**, comment_id -> comment NN, tower_id -> tower NN) |
| R28 | comment_classification(**id**, comment_id -> comment NN, classification_id -> classificationperiod NN) |

```
1 class DataSetPG(models.Model):
2     tower_code = models.CharField(max_length=20, null=False)
3     time_stamp = models.DateTimeField(default=datetime.now, null=True, blank=True)
4     value = models.CharField(max_length=200)
```

Listing 5.3: DataSet model for PostgreSQL in Django

To complement the Table 5.2 we have the following legend:

- UK — Unique Key
- NN — Not Null
- DF — Default
- CK — Check

In PostgreSQL, we are also storing all raw data provided from wind towers. In Django models, our class is called as *DataSetPG*, that is represented in Listing 5.3. This class is represented by a tower code, a timestamp and a string with values — coming from each data logger channel — separated by commas. This structure means that we will have a table with millions of rows as we will store all raw data from all towers into one table. We decided to keep and store all values of the corresponding timestamp and tower code in a string rather than a dynamic array. The reason for this is because later we parse this value field with Pandas framework and its easier to do in a string rather than an array. Another reason for this was on to keep all the values as received from data-logger as much as possible, so our model can receive and store any values coming from any data logger.

We could also use an approach of storing the corresponding raw data from each tower on separated tables using dynamic models[9] — creating and using models in runtime, as new towers with raw data is entered in the system — this would help us to avoid millions of records into the same table. But unfortunately, we discover this option to late and due lack of time we couldn't implement it.

### 5.3.2 Database — MongoDB

In our second approach, we make use of a NoSQL database system to store only raw data provided by wind towers. The database we choose is MongoDB, and it's detailed in the Section 4.6.

The MongoDB model has the same structure as the one as PostgreSQL, meaning that we have one MongoDB collection with documents and each document have a tower code field, a timestamp and a string with values separated by commas. In Django models, our class is called as *DataSetMongo*, that is represented in Listing 5.4. We can see that *MongoModel* is the base class for the *DataSetMongo* model and not the base class model from Django database system. This

[9] https://pypi.org/project/django-dynamic-model/

```
class DataSetMongo(MongoModel):
    tower_code = fields.CharField(max_length=20)
    time_stamp = fields.DateTimeField(default=datetime.now)
    value = fields.CharField(max_length=200)
```

Listing 5.4: DataSet model for MongoDB in Django

is because Django, at the current version, doesn't support MongoDB database by default. So, to solve this problem, we need to use a package that handles the connection between the Django and MongoDB server.

The package we first used to handle connection between Django and MongoDB server was MongoEngine[10], but unfortunately this package has a few performance issues when inserting and/or querying high number of documents in the same collection[11], so we ended by using other package called PyMODM, that have better inserting and querying performance than MongoEngine.

In the beginning, our model structure was different from the one we referred at Listing 5.4. We were trying a more structured model, where our document was represented by a tower code and a list of documents with timestamp and value. The Listing 5.5 represents this type of structure in the Django model. This structure ensures that each document stores raw data from an individual tower, meaning we will have a good performance querying when compared to the structure in Listing 5.4. But we fast stand with a big problem as MongoDB documents can only have a maximum size of 16 MB, which was easily occupied with our raw data. Because of this problem, we ended by using the structure where a document saves only a record of a value belonging to a timestamp and a tower code, as described in Listing 5.4.

### 5.3.3 Database — InfluxDB

Like in the MongoDB approach, the Influx database will only store the raw data provided from wind towers. The information of this database it's detailed in Section 4.6.

[10] https://github.com/MongoEngine/mongoengine
[11] https://github.com/MongoEngine/mongoengine/issues/1230

```
class TimeValue(EmbeddedMongoModel):
    time_stamp = fields.DateTimeField(default=datetime.now)
    value = fields.CharField(max_length=200)

class DataSetMongoPyMod(MongoModel):
    tower_code = fields.CharField(max_length=20)
    time_value = fields.EmbeddedDocumentListField(TimeValue)
```

Listing 5.5: First DataSet model structure for MongoDB in Django

```
point={
    "measurement": tower_code,
    "time": time_value,
    "fields": {
        "value": values
        }
}
```

Listing 5.6: JSON structure to insert into Influx Database

Because Influx it's oriented to time series, it's structure store information into *Measurements*, and each *Measurement* receive a time and a value field to store value or values. The structure we are using at Influx to store raw data is different from PostgreSQL and MongoDB, where instead of saving all raw data into one table, we store the raw data from each wind tower into individual *Measurements*, meaning that each *Measurement* stores only raw data of one wind tower.

At present, we only have one connector from Django server to Influx database called influxdb[12]. This connector its more straightforward than the others used in PostgreSQL and MongoDB, as wasn't necessary to initiate any class model in the *models.py* file to handle Influx. To insert new instances into the Influx database, we need to create a JSON document, as referred at Listing 5.6. This structure it's called as a *point* and receive a measurement field that will contain a tower code, the corresponding timestamp and a string with values separated by commas. After creating an array of *point's*, we can insert into Influx using *write_points* function provided from Influx connector. To make queries we need to use *query* function from the connector that uses SQL code.

The connector also offers the developer the option to use a class model to insert instances into the Influx database using a base class called *SeriesHelper*, but unfortunately using this approach bring us inefficiency when inserting values into the database when compared to the *write_points* function.

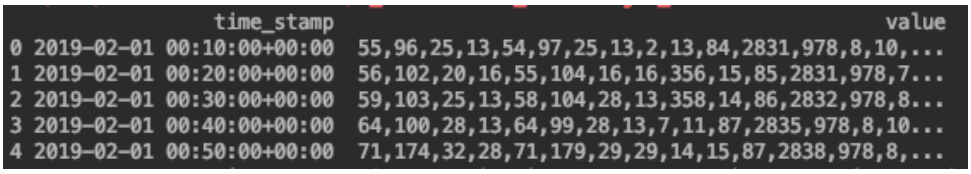
### 5.3.4 Data Visualization

After successfully store the raw data into the database, we need to query it, manipulate and then show it on charts. This is where *django-pandas* and *django-chartjs* packages are tacked into action.

#### 5.3.4.1 Pandas

Pandas is one of the most known framework in Python community to manipulate a large amount of data, as it requires only a few lines of code to load data, split it and do mathematical operations. So we tacked advantage of this powerful framework and used it to handle the data in our application, what is one of the most important requirements in our application.

[12] https://github.com/influxdata/influxdb-python

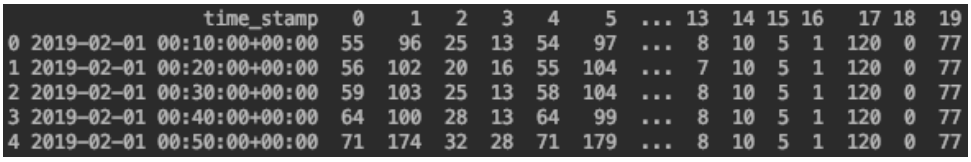


| | time_stamp | value |
|---|---|---|
| 0 | 2019-02-01 00:10:00+00:00 | 55,96,25,13,54,97,25,13,2,13,84,2831,978,8,10,... |
| 1 | 2019-02-01 00:20:00+00:00 | 56,102,20,16,55,104,16,16,356,15,85,2831,978,7... |
| 2 | 2019-02-01 00:30:00+00:00 | 59,103,25,13,58,104,28,13,358,14,86,2832,978,8... |
| 3 | 2019-02-01 00:40:00+00:00 | 64,100,28,13,64,99,28,13,7,11,87,2835,978,8,10... |
| 4 | 2019-02-01 00:50:00+00:00 | 71,174,32,28,71,179,29,29,14,15,87,2838,978,8,... |

Figure 5.1: DataFrame without value field splited

After making the query to the raw data we use *read_frame* function from Pandas to create a *DataFrame* with the raw data. The *DataFrame* is composed by rows and two columns, the timestamp and a value field that have all values separated by commas, as represented in Figure 5.1. Using the *apply* function from Pandas we can split one column from the *DataFrame* and create many columns as many commas the column value have — the Figure 5.2 illustrate that representation.

Having the *DataFrame* composed by a timestamp and values separated by columns, we need to query the *periodconfiguration* table to get the different configurations present in that raw data. When iterating the Period Configuration's we create temporary *DataFrames* to manipulate and change columns if necessary to get the correct values. At this point, we have access to the Equipment Configuration's that belongs to the Period Configuration. Each Equipment Configuration has access to several Dimensions that will tell us what is the correct value column that the Equipment Configuration is reading at that point and then associate the correct value column on the temporary *DataFrame*. Equipment Configuration is linked to all Equipment information, as static information and calibrations. At this point, we can apply mathematical operations to the data to get the final results using the correct offset and slope values. To do these mathematical operations, we need two steps. The first step is to get the default value that the sensor read before sending to the data logger. The default value is taken using the formula: $default = RawData - (offset\_DL/slope\_DL)$, where $RawData$ is the raw data stored in the database, $offset\_DL$ is the *offset_dl* attribute value stored in the Period Configuration, $slope\_DL$, is the *slope_dl* attribute value stored in the Period Configuration. The second step is to get the final value using the Calibration values present in the Equipment Configuration. To get the final value we use the formula: $final = default * (slope + offset)$, where $default$ is the value obtained in the first step, the $slope$ is the slope attribute value present in the Calibration and the $offset$ is the offset attribute value existing in the Calibration. Using Pandas, we only need one line of code to apply each step to the *DataFrame*.



| | time_stamp | 0 | 1 | 2 | 3 | 4 | 5 | ... | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2019-02-01 00:10:00+00:00 | 55 | 96 | 25 | 13 | 54 | 97 | ... | 8 | 10 | 5 | 1 | 120 | 0 | 77 |
| 1 | 2019-02-01 00:20:00+00:00 | 56 | 102 | 20 | 16 | 55 | 104 | ... | 7 | 10 | 5 | 1 | 120 | 0 | 77 |
| 2 | 2019-02-01 00:30:00+00:00 | 59 | 103 | 25 | 13 | 58 | 104 | ... | 8 | 10 | 5 | 1 | 120 | 0 | 77 |
| 3 | 2019-02-01 00:40:00+00:00 | 64 | 100 | 28 | 13 | 64 | 99 | ... | 8 | 10 | 5 | 1 | 120 | 0 | 77 |
| 4 | 2019-02-01 00:50:00+00:00 | 71 | 174 | 32 | 28 | 71 | 179 | ... | 8 | 10 | 5 | 1 | 120 | 0 | 77 |

Figure 5.2: DataFrame with value field splited

After having the temporary *DataFrames* appended to a final *DataFrame*, we only have to fill empty raw data with NaN with frequencies of 10 min — as requested by INEGI — and convert the timestamp to UNIX time. At this point, we are ready to send the *DataFrame* with the raw data to display in a chart.

#### 5.3.4.2 Charts

We make use of charts to display the raw data, choosing a period from a wind tower, and the corresponding classifications status that the equipment's have for that period, to fully implement the User Story US36 from Table 4.1.

To display the raw data, we make use of *django-charts* to send this data from the view to the HTML page. This package gives the developer the possibility to use charts from HighCharts[13] and Chart.js[14], but to display the raw data we use the Line Chart[15] from HighChart because of their functionalities and simple configuration. HighChart gives the developer the possibility to customize the charts using JavaScript, like functions upon a directly click on the chart, zoom in and out options and much more. We tried other packages that the Django community offer, like *django-nvd3*[16] that makes use of the d3.js[17] JavaScript library, but unfortunately, it's limited in functionalities and outdated. We also tried *django-graphos*[18] this package gives the developer the ability to use charts from Google[19], Flot[20], Morris.js[21] and many others, but is also limited in functionalities, as we can't make use of all of the powerful features and customization's from those APIs. This happens because *django-graphos* doesn't implement them and we can't customize using JavaScript, as the template is being rendered by *django-graphos* tags and not by JavaScript code directly on that HTML source code.

We also make use of X-Range Chart[22] from HighCharts to display the classifications status from equipment's in a period entered by the user.

## 5.4 User Interfaces

In this section, we will describe the most important interfaces of the platform. Our main goal was to create a web application to replace the platforms from INEGI wind resource team. While the INEGI team use two applications to manage meta information from wind towers and classify data, the web application was implemented to make that two functionalities in one only, adding more features like clients access, comment a classification or a station, etc. To create the web application

[13] https://www.highcharts.com/
[14] https://www.chartjs.org/
[15] https://www.highcharts.com/docs/chart-and-series-types/line-chart
[16] https://github.com/areski/django-nvd3
[17] https://d3js.org/
[18] https://github.com/agiliq/django-graphos
[19] https://developers.google.com/chart/
[20] http://flotcharts.org/
[21] http://morrisjs.github.io/morris.js/
[22] https://www.highcharts.com/docs/chart-and-series-types/x-range-series

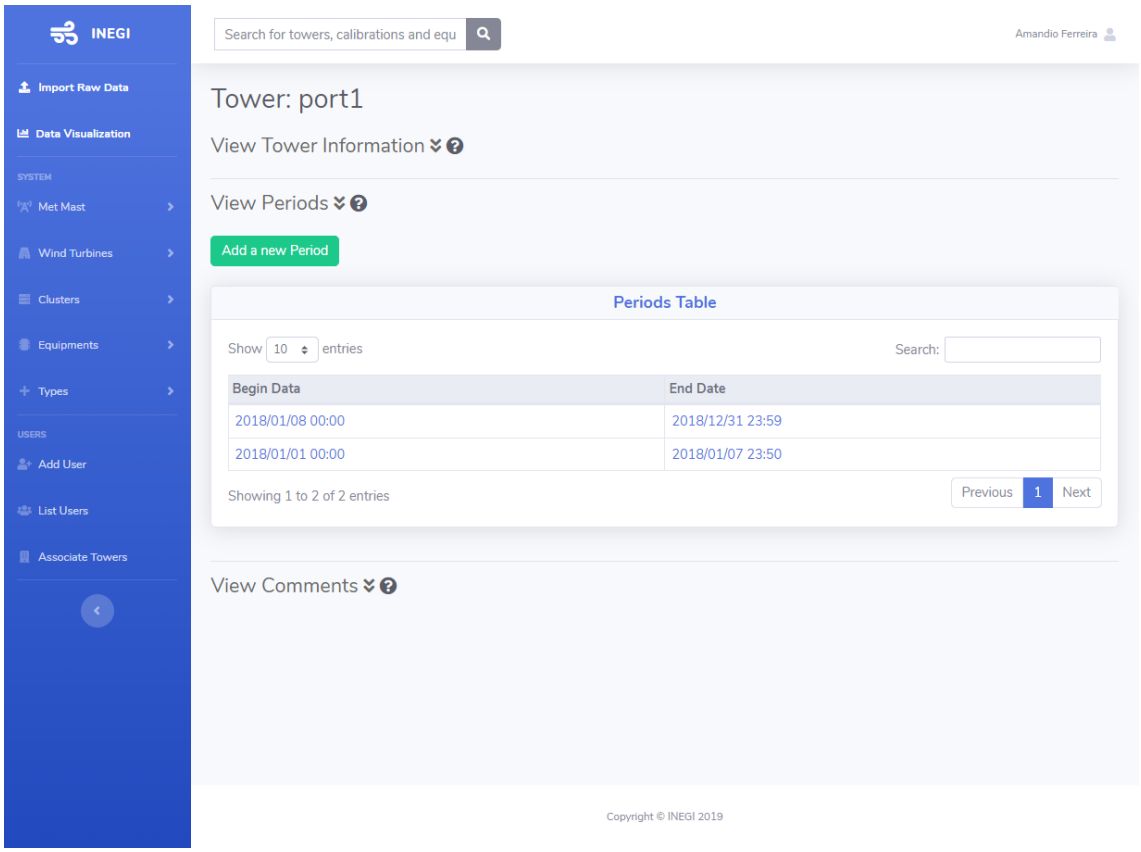



Figure 5.3: Interface containing information from a Wind Tower

we used a template derived from SB Admin 2[23], that use Bootstrap and jQuery, ensuring that is responsive and easy to use and that the user only needs a few clicks on the interface to get the desired actions.

The Figure 5.3 represent the URL $tower/\{id\}$ from routes Table 5.1. This interface shows the meta-information from wind tower — by default, the form is collapsed — the periods of configuration and the comments belonging to that tower — also collapsed by default to avoid full fill the page with information. This interface also represents the next User Stories from Table 4.1:

- US04 — Edit station
- US05 — Remove station
- US19 — View station
- US20 — Add comment

The Figure 5.4 represent the URL $tower/\{id\}/conf\_period/\{id\}$ from routes Table A.1. This interface is one of the most important in the platform, as it shows the meta-information from a period of configuration — by default, the form is collapsed — and also a table with equipment's

[23] https://startbootstrap.com/themes/sb-admin-2/

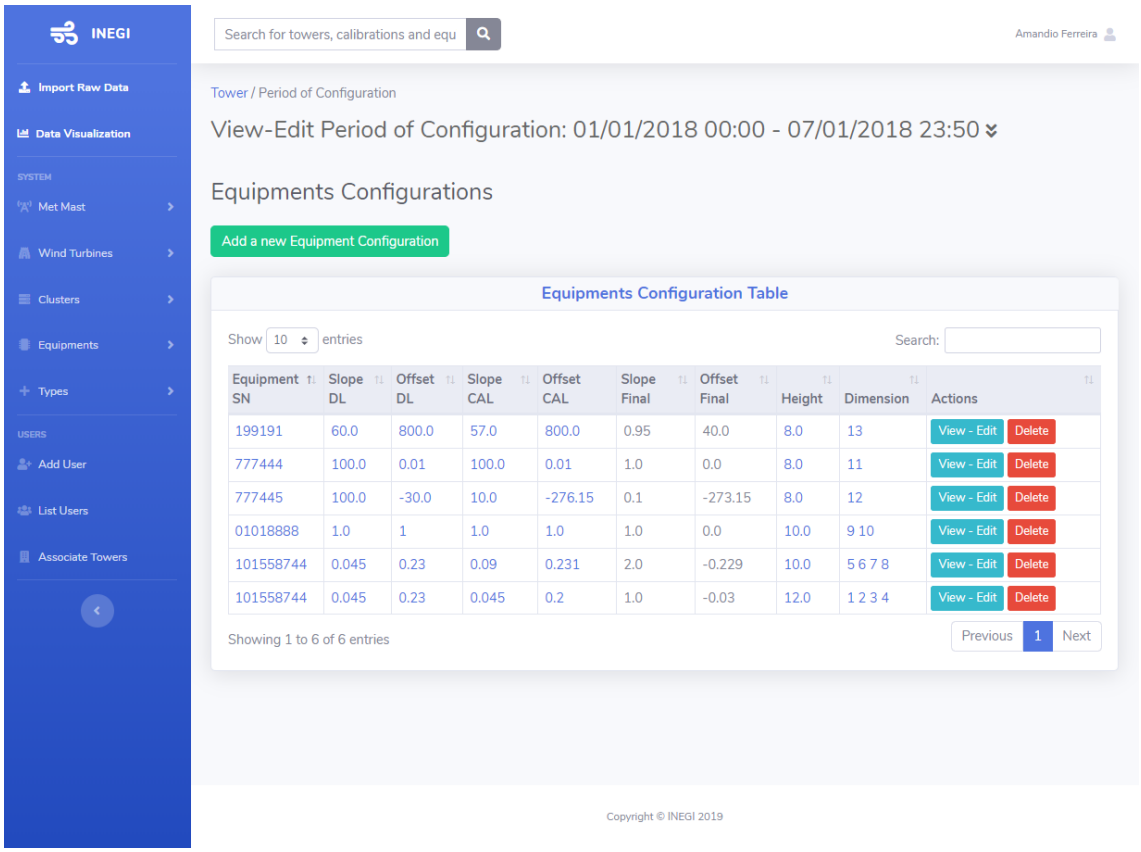



Figure 5.4: Interface containing information from a period of configuration and it's equipment's

present in that period of configuration. On this interface, we can see some important attributes: as the equipment serial number, the slope's and offset's from the data logger and calibration and also the dimension that equipment is reading in the string value from raw data. This interface also represents the next User Stories from Table 4.1:

- US23 — Add equipment configuration
- US33 — Edit period configuration
- US34 — Remove period configuration
- US35 — View period configuration

The Figure 5.5 represent the URL $tower/\{id\}/show$ from routes Table A.14. This interface is the most important in the platform, as it shows the classifications status and the raw data from a period of a wind tower in charts. It's also possible to classify and comment on this interface by choosing a period of dates directly from the raw data chart. This interface also represents the next User Stories from Table 4.1:

- US20 — Add comment

- US26 — Add classification
- US36 — View raw data

## 5.5 Conclusion

In this chapter, we described the most important details of the implementation. Choosing a good architecture initially makes the development quick and facilitates new features to be implemented.

We described how Django makes use of three databases in the same application to measure the insertion and query performance of the raw data and how we pass from the Conceptual Model from Section 4.4 to tables in PostgreSQL database that is represented by a Relational Schema in Table 5.2. We also described how important was the use of Pandas framework to structure and manipulate the raw data from wind towers and their configurations. The essential User Interfaces were also presented in this chapter.

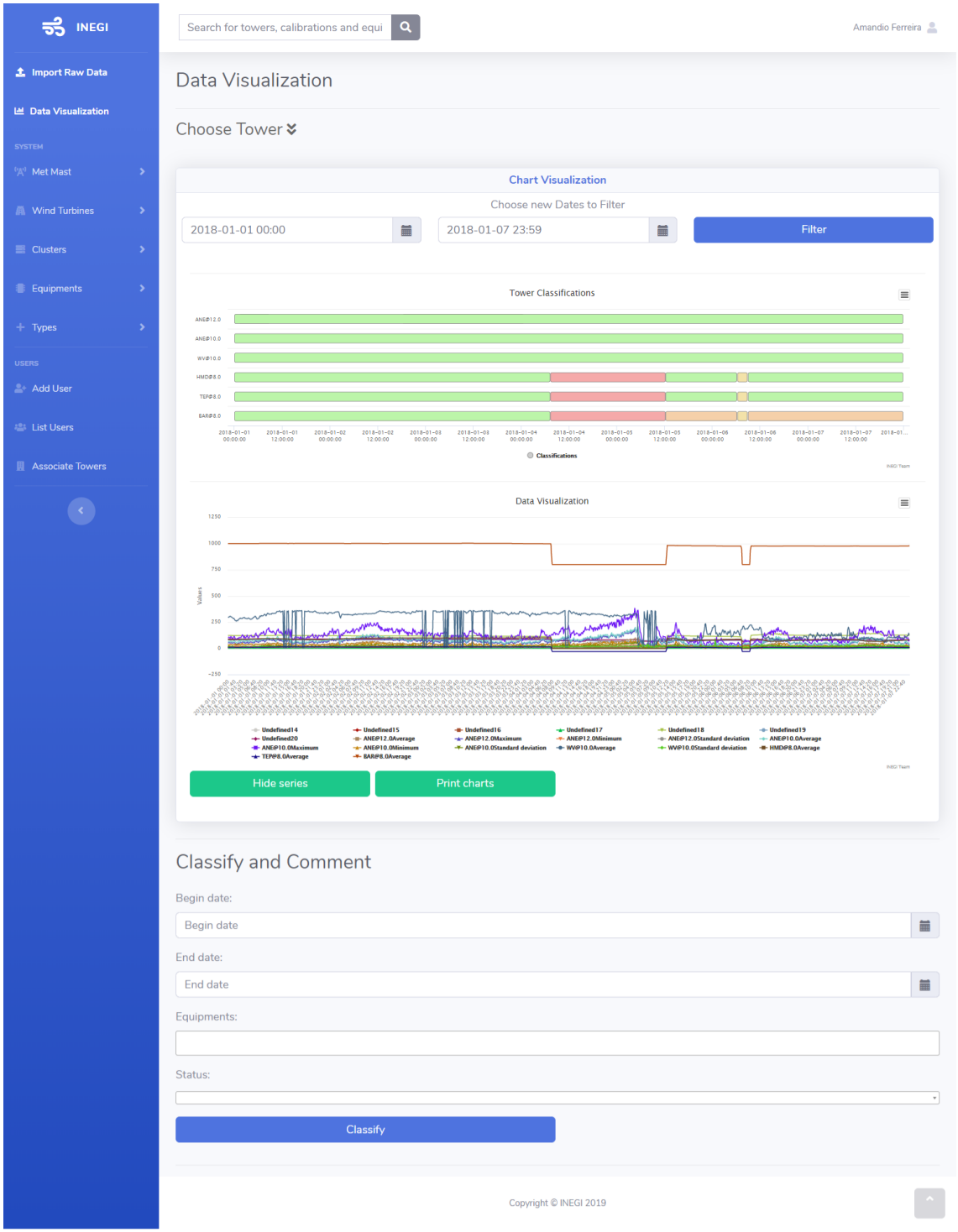



Figure 5.5: Interface showing classifications and raw data from a wind tower

# Chapter 6

# Tests and Results

In this chapter, we will measure the performance of the insertion and selection of the raw data in three different database systems and also describe how validation tests were made during the development of the platform. The performance test will give us a better understanding of what database system should we use to store the raw data provided from the wind towers.

## 6.1 Methodology

One of the main goals of this work was to understand what database system best suits the application. To obtain the conclusions about the tests, we will perform load tests, meaning we will create iterations rising the amount of data to study the behaviour of the system.

The tests made are divided into two groups, insertion and selection to the three different databases. The raw data inserted follows a logarithmic base of 10, and we start adding one thousand rows of the series — $10^3$ records — and finishing at 1 thousand million rows of the series — $10^9$ records —, this means we will have seven iterations, where each iteration has an amount of data ten times greater than the old one. We also compared the selection performance and chart load between this new platform and the actual INEGI platform, that works as the baseline. We repeated each test several times, to achieve an average to represent the bests results as possible and also — to give consistency on the results —, we kept the same CPU processes running, killing all unnecessary applications and avoiding any scheduled tasks.

To measure the performance of the databases in our platform, we used the *time.time()* function from Python, that returns the current time in seconds since the Epoch by measuring the wall-clock time. Python also offers other function to measure the time called *time.clock()*, but this function returns the CPU time or real time since the start of the process or since the first call of *time.clock()*. The code in Listing 6.1 gives us a better understanding of the difference between *time.time()* and *time.clock()* functions, where we called each function and give a *sleep* of one second and then called each function again. We can see that *time.time()* is different of *time.clock()*, as we need

```
print(time.time(), time.clock())
time.sleep(1)
print(time.time(), time.clock())
> 1560874948.752479 2.793988
> 1560874949.756335 2.824063
```

Listing 6.1: time vs clock function

to measure, not only the CPU time, but also the access to the database storage. To measure the execution time of the INEGI platform — Wind2Data — we used the Timer function from VBA.

To make these tests, we used two machines, one to insert and query the three databases and one other to query and load the chart with raw data as we needed to compare with the Wind2Data. The first machine has the next hardware specifications: 2.0 GHz dual-core Intel Core i5, Turbo Boost up to 3.1 GHz with 4 MB shared L3 cache, 8 GB of 1866 MHz LPDDR3 onboard memory, 512 GB PCIe-based onboard SSD, running a macOS Mojave OS with version 10.14.5 (18F132). The second machine has the next hardware specifications: Intel(R) Core(TM) i7-7500U CPU @ 2.70 GHz, 2901 MHz, 2 Core(s), 4 Logical Processor(s), 16 GB of memory RAM, 512 GB of SSD disk space, running a Windows 10 OS with version 10.0.17134 Build 17134. The databases installed on each machine was PostgreSQL v.11.2, MongoDB v4.0.4 and Influx v1.7.4. The Python version used was the 3.7.1.

## 6.2 Performance Tests

In this section, we will describe the performance tests made. One of the main questions was to study the most used operations by the INEGI team, the insertion time, the partial selection and the disk space that the raw data was occupying into each database system. We made a total selection to study the performance of each database but it is not the most used operation. We also compared this new solution with the actual INEGI solution by making a partial selection and a chart load. PostgreSQL and MongoDB databases will be tested with and without indexes, and Influx with indexes only, as it already comes with indexes by default on its *Measurements*. In this section, we will also describe some optimization's made to try to obtain better results when making selections.

### 6.2.1 Insertion Tests

The insertion represents the User Story US38, the ingestion of time series into the database systems. What we expect with this tests it's to measure the insertion time for each database system. To successfully get most homogeneous and reliable results, we insert the same raw data into the three database systems with the following configurations:

- $10^3$ records — contains approximately seven days of different time series values from one tower, with a timestamp frequency of ten minutes.

- $10^4$ records — contains approximately 70 days of different time series values from one tower, with a timestamp frequency of ten minutes.
- $10^5$ records — contains approximately two years of different time series values from one tower, with a timestamp frequency of ten minutes.
- $10^6$ records — contains approximately 12 years of different time series values from one tower, with a timestamp frequency of one to ten minutes.
- $10^7$ records — contains approximately 12 years of different time series values from ten towers, with a timestamp frequency of one to ten minutes.
- $10^8$ records — contains approximately 12 years of different time series values from 100 towers, with a timestamp frequency of one to ten minutes.
- $10^9$ records — contains approximately 12 years of different time series values from 1 000 towers, with a timestamp frequency of one to ten minutes.

Each record is composed by a tower code, a timestamp and a string with values separated with commas. We also repeated the insertion tests several times, to get the most reliable results, having the following repetitions configurations:

- $10^3$ to $10^6$ records — repeated 50 times
- $10^7$ records — repeated 20 times
- $10^8$ records — repeated 5 times
- $10^9$ records — repeated 2 times

We needed to reduce the repetition times with the increment of the working load because the insertion time was increasing and was getting impractical to repeat such amount of times. But from Figure 6.1, we can confirm that the measuring times was practically constant. Calculating the coefficient of variation, for $10^6$ records without indexes, we get 2.75 % for PostgreSQL and 1.45 % for MongoDB, what is considered a low dispersion — below 15 % —, meaning we are working with homogeneous data, giving us more security on the results for records $10^7$ to $10^9$. For records of $10^3$ and $10^4$ we get high dispersion — over 30 % —, as we are facing with milliseconds operations, meaning we have a high rate of different time values. The coefficient of variation values are present in Table 6.1.

It's crucial that we verify the initial conditions for each insertion test and, therefore, before each test, we place the database in its initial state, meaning that no time series are stored. To insert into the PostgreSQL database we used the *bulk_create* function from Django models, using a *batch_size* of 100 000, to avoid any in-memory problem, and for the MongoDB database we used the *bulk_create* function from MongoModel, but unfortunately we had no *batch_size* parameter, so we ended by forcing a max of 100 000 records to be inserted each time on the MongoDB database.


Figure 6.1: Insertion of $10^6$ records with 50 repetitions

Table 6.1: Coefficient of variation over records in insertion tests

| | **Without Indexes** | | **With Indexes** | | |
|---|---|---|---|---|---|
| **Records** | **PostgreSQL** | **MongoDB** | **PostgreSQL** | **MongoDB** | **Influx** |
| $10^3$ | 32.05 % | 40.45 % | 32.51 % | 20.40 % | 28.89 % |
| $10^4$ | 5.71 % | 8.55 % | 4.08 % | 6.87 % | 45.26 % |
| $10^5$ | 3.11 % | 4.76 % | 2.84 % | 2.32 % | 5.66 % |
| $10^6$ | 2.75 % | 1.45 % | 8.86 % | 1.97 % | 2.41 % |
| $10^7$ | 2.99 % | 1.31 % | 5.76 % | 0.33 % | 1.50 % |
| $10^8$ | 0.80 % | 0.26 % | 2.84 % | 0.64 % | 2.48 % |
| $10^9$ | 0.61 % | 0.65 % | NA | 0.13 % | 0.52 % |

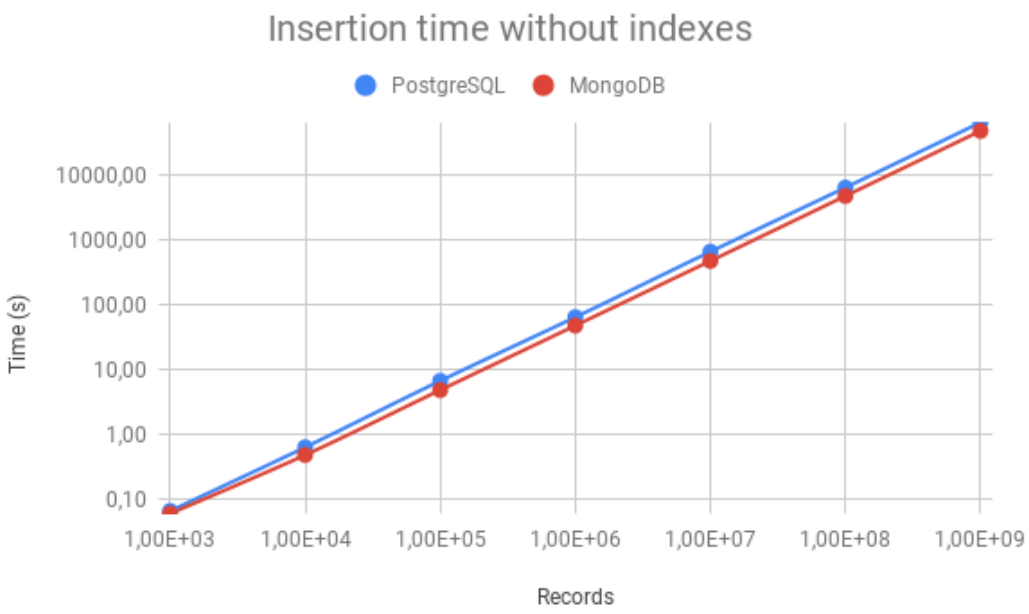



Figure 6.2: Insertion performance for PostgreSQL and MongoDB without indexes

Finally to insert into the Influx database we used the *write_points* function from Influx connector, using a *batch_size* of 100 000.

It's also important to refer that these tests helped us to measure the database in extreme situations, as the regular insertion will be one day of raw data for 300 wind stations — around 43 200 records — and rarely we will insert $10^9$ records.

#### 6.2.1.1 Without Indexes

The Figure 6.2 shows us the comparison of insertion without indexes for PostgreSQL and MongoDB, increasing the number of records inserted into the databases. We couldn't make insertion tests without indexes for Influx, as that database structure already comes with indexes.

#### 6.2.1.2 With Indexes

To improve the selection performance, we configured the PostgreSQL and MongoDB with a B+Tree index on *tower_code* and *time_stamp* attributes. Consequently, we also had to rerun the insertion tests to have the updated results, as using indexes influence the insertion time, because it's necessary to fill the index with information from those two attributes. The Figure 6.3 shows us the comparison of insertion with indexes for PostgreSQL, MongoDB and Influx increasing the number of records inserted into the databases.

We can see that we are unable to insert $10^9$ records into the PostgreSQL due disk space limitation, stopping at nearly $0.7 * 10^9$ records and taking around 358 600 s — three days. It would be expected to take approximately four days to insert $10^9$ records into the PostgreSQL.

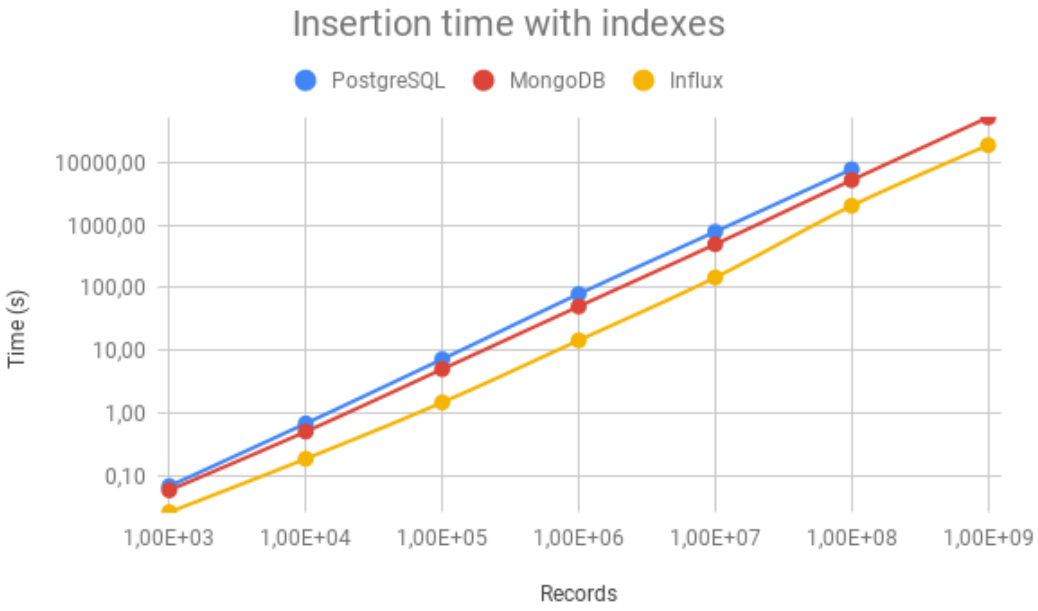



Figure 6.3: Insertion performance for PostgreSQL, MongoDB and Influx with indexes

We also faced one problem inserting $10^9$ records into the Influx database due to insufficient RAM. Influx uses two stages — using wal and data directories — to insert the data into the database. In the first stage, Influx uses the wal — Write-Ahead-Log — directory to append new writes and deletes, and also store in-memory cache until the cache is snapshotted. Based on the cache-snapshot-write-cold-duration — the time interval at which the engine will snapshot the cache and write it to a new TSM (Time-Structured Merge) Tree file if the shard hasn't received writes or deletes — value a new level 1 TSM file is written and the wal segments associated to that snapshotted are removed. This is where the problem was. The database was always storing using in-memory without releasing it because the cooldown time of ten min — by default — wasn't never expired, so we ended by changing that value to twenty s to solve the problem. At this stage, Influx is fast to write but not to query. The second stage — using data directory — takes care of compacting the TSM files into more compressed forms to improve query performance.

#### 6.2.1.3 Results

The Table 6.2 shows a resume of results obtained when inserting raw data without and with indexes. The performance results are expressed in seconds, rounded to 2 decimals, for each database system. We also used a speedup measurement, comparing the worst and the best execution, where PostgreSQL is represented by (P), MongoDB by (M) and Influx by (I). For the speedup measurement, we used the original measured time.

We can see that MongoDB had, an average of 1.32 times, better performance inserting without indexes when comparing to PostgreSQL, mainly because MongoDB, upon an insertion, don't have

Table 6.2: Insertion without and with indexes

| **Records** | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $10^7$ | $10^8$ | $10^9$ |
|---|---|---|---|---|---|---|---|
| **Without Indexes** | | | | | | | |
| MongoDB | 0.06 s | 0.62 s | 6.59 s | 63.30 s | 649.27 s | 6 342.88 s | 64 154.51 s |
| PostgreSQL | 0.06 s | 0.47 s | 4.74 s | 46.95 s | 466.96 s | 4 703.66 s | 47 444.40 s |
| Speedup | 1.10 (M) | 1.32 (M) | 1.39 (M) | 1.35 (M) | 1.39 (M) | 1.35 (M) | 1.35 (M) |
| **With Indexes** | | | | | | | |
| PostgreSQL | 0.07 s | 0.67 s | 7.19 s | 79.29 s | 785.29 s | 7 832.81 s | NA |
| MongoDB | 0.06 s | 0.50 s | 4.98 s | 50.10 s | 495.31 s | 5 248.05 s | 52 863.88 s |
| Influx | 0.03 s | 0.18 s | 1.46 s | 14.38 s | 145.71 s | 2 041.70 s | 19 028.08 s |
| Speedup (P)/(I) | 2.61 | 3.67 | 4.92 | 5.51 | 5.39 | 3.84 | NA |
| Speedup (M)/(I) | 2.24 | 2.72 | 3.40 | 3.48 | 3.40 | 2.57 | 2.78 |

to look for foreign keys and also for ACID properties. Using indexes, we can see that Influx is faster inserting than PostgreSQL and MongoDB. If we compare with PostgreSQL, Influx is faster with a speedup average of 4.32, and 2.94 when compared with MongoDB. This happens because Influx use in-memory to insert and it's well optimized for time series data composed by a key pair of time/value(s).

We also compared the insertion from each database system without and with indexes. The results are represented in Table 6.3. It's expected that the insertion time is higher with indexes than without indexes, as the database need to fill information for those indexed attributes. PostgreSQL, with indexes, is slower 1.15 times inserting when we compare the same operation without indexes. MongoDB is also slower 1.06 times when we insert with indexes. We didn't get any value for $10^9$ records in PostgreSQL insertion with indexes — as explained in Section 6.2.1.2 — but we could estimate it would take 4 days, meaning that would take around 5.38 times more than the insertion without indexes. We noticed on PostgreSQL that the time was increasing when dealing over $10^6$ records, as that database recommends the creation of indexes only after insertion to speed up the insertion process. With MongoDB, we noticed a slight increase in insertion time when dealing with records over $10^8$.

Table 6.3: Time relation on each database system with and without indexes upon insertion

| **Records** | **PostgreSQL** | **MongoDB** |
|---|---|---|
| $10^3$ | 1.04 | 0.98 |
| $10^4$ | 1.08 | 1.06 |
| $10^5$ | 1.09 | 1.05 |
| $10^6$ | 1.25 | 1.07 |
| $10^7$ | 1.21 | 1.06 |
| $10^8$ | 1.23 | 1.12 |
| $10^9$ | NA | 1.11 |

```
1 SELECT relname AS objectname, relkind AS objecttype, reltuples AS "#entries",
      pg_size_pretty(relpages::bigint*8*1024) AS size
2 FROM pg_class
3 WHERE relpages >= 8
4 ORDER BY relpages DESC;
```

Listing 6.2: Get size of tables

### 6.2.2 Disk Space

One of the goals was to know how much disk space each database occupy. For this, we measure the PostgreSQL database with and without indexes using the query at Listing 6.2 through Postico[1] — a client for PostgreSQL. The query returns the size for data, primary and foreign keys, and indexes on each table from the database.

For MongoDB we used the *stats* and *storageSize* functions from the collection, directly in MongoDB Shell, to get, respectively, the indexes and unique indexes — for _id —, and the data size. For Influx, we had to get the actual size in the *data* directory, as Influx don't offer any query or function to get the disk space being used at the *Measurements*.

The Figure 6.4 show us the comparison of the data and the primary key disk space — in kB — for PostgreSQL and MongoDB, over records.

The Figure 6.5 show us the comparison of disk space — in kB — of the data, the primary key and the indexes for PostgreSQL, MongoDB and Influx, over records.

#### 6.2.2.1 Results

We can see by the Table 6.4 that MongoDB (M) had — with an average of 1.82 times — less disk space when comparing to PostgreSQL. This has to be how the data is stored, while MongoDB stores the data on the disk as a BSON[2]— binary-encoded serialization of JSON-like documents, PostgreSQL, by default, writes blocks of data — pages — to disk in 8 k chunks.

[1] https://eggerapps.at/postico/
[2] http://bsonspec.org/

Table 6.4: Disk space without indexes

| Records | PostgreSQL | MongoDB | Better x |
|---|---|---|---|
| $10^3$ | 133 kB | 97 kB | 1.37 (M) |
| $10^4$ | 1 317 kB | 724 kB | 1.82 (M) |
| $10^5$ | 13 406 kB | 7 135 kB | 1.88 (M) |
| $10^6$ | 136 080 kB | 70 906 kB | 1.92 (M) |
| $10^7$ | 1 400 167 kB | 758 996 kB | 1.84 (M) |
| $10^8$ | 16 168 661 kB | 7 764 471 kB | 2.08 (M) |
| $10^9$ | 142 938 347 kB | 78 207 475 kB | 1.83 (M) |

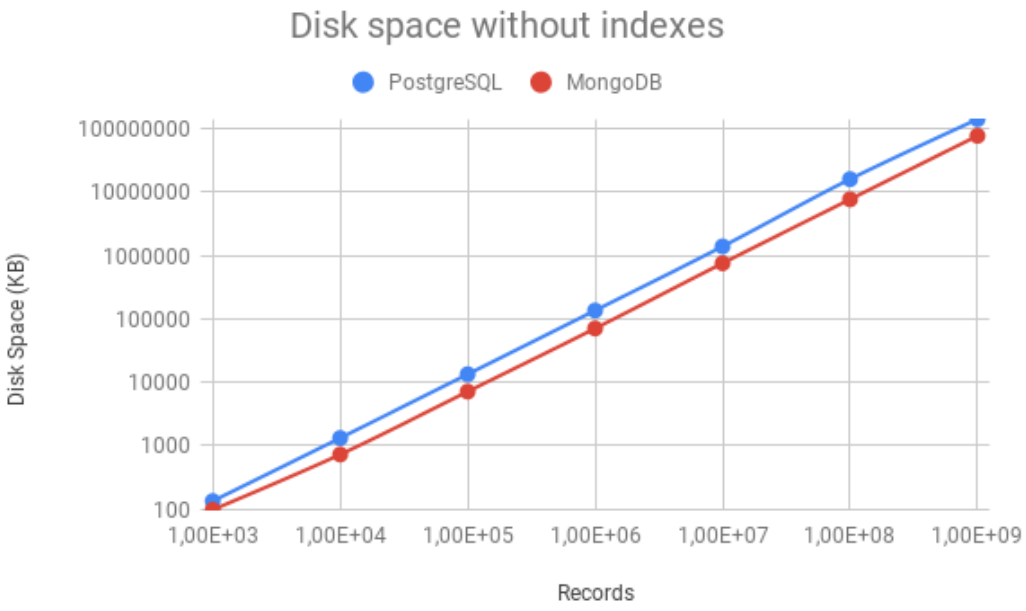



Figure 6.4: Disk space without indexes

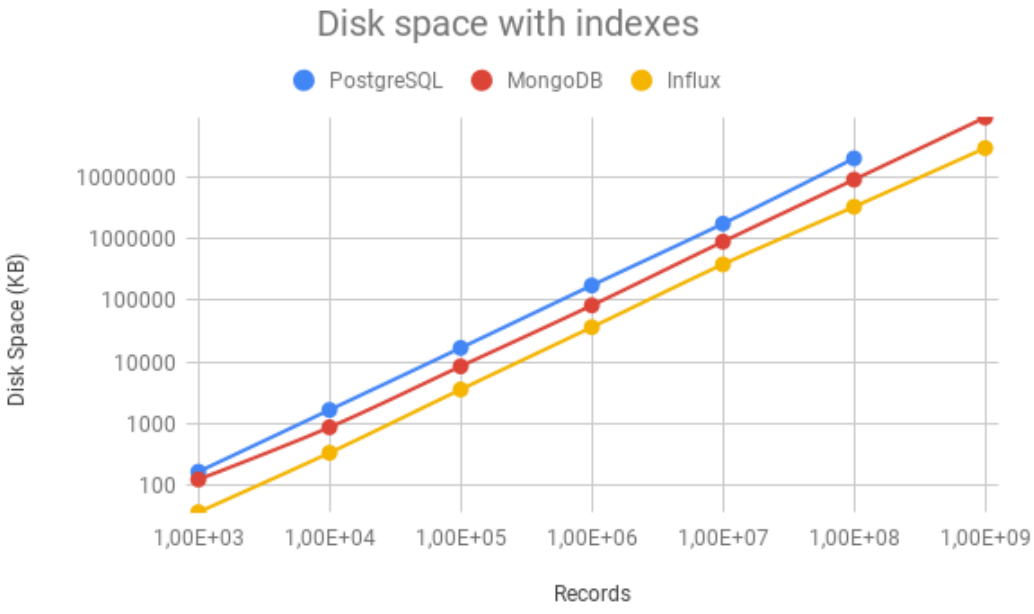



Figure 6.5: Disk space with indexes

Table 6.5: Disk space with indexes

| Records | PostgreSQL | MongoDB | Influx | Better (P)/(I) | Better (M)/(I) |
|---|---|---|---|---|---|
| $10^3$ | 167 kB | 125 kB | 37 kB | 4.51 | 3.38 |
| $10^4$ | 1 671 kB | 875 kB | 338 kB | 4.94 | 2.59 |
| $10^5$ | 16 823 kB | 8 482 kB | 3 573 kB | 4.71 | 2.37 |
| $10^6$ | 172 821 kB | 82 108 kB | 36 457 kB | 4.74 | 2.25 |
| $10^7$ | 1 726 582 kB | 881 949 kB | 378 318 kB | 4.56 | 2.33 |
| $10^8$ | 19 664 402 kB | 8 917 146 kB | 3 244 919 kB | 6.06 | 2.75 |
| $10^9$ | NA | 90 413 333 kB | 28 865 885 kB | NA | 3.13 |

By analyzing the Table 6.5, we can compare the three databases systems with indexes. We can see that Influx occupy 4.92 times less disk space when compared with PostgreSQL and when compared with MongoDB, require 2.69 times less disk space. This is because Influx is good compressing data into TSM files.

If we compare PostgreSQL and MongoDB with and without indexes, we can see that PostgreSQL, by an average of 1.25, and MongoDB, by an average of 1.19, occupy more disk space when using indexes. We could expect that PostgreSQL when dealing with $10^9$ records, with indexes, would occupy around $167 * 10^6$ kB. This means almost 1.85 more disk space when comparing that same number of records without indexes.

### 6.2.3 Selection Tests

The selection tests represent the User Story US36. In the next two sections, we will analyze the time that the platform takes to select the raw data from the three databases systems. This tests will be divided into two groups: total and partial selection. Like the insertion tests, selection tests will follow a logarithmic base of 10, and we will start selecting one thousand lines — $10^3$ — and iterating until 1 thousand million of records — $10^9$. We will also repeat the selection tests several times to get an average of execution times — we will describe the repetition configurations in the next two Sections.

#### 6.2.3.1 Total Selection

In this section, we will analyze the total selection. The total selection consists of getting all the records from the *DataSet* table, independently of the wind towers. This operation isn't the most used but will help us measure the performance of the database in extreme situations. For PostgreSQL, the total selection have the following code: $DataSetPG.objects.all()$, that is the same as $SELECT * FROM datasetpg$ in SQL language. For MongoDB, we used the following code: $DataSetMongo.objects.all()$, that is the same as $db.data\_set\_mongo.find()$. For Influx we used the *query* function from Influx connector to make the following query: $SELECT * FROM /.* /$. Because influx stores each raw data from each tower into separated *Measurements*, we had to select all of them with the code: $/.*/$.

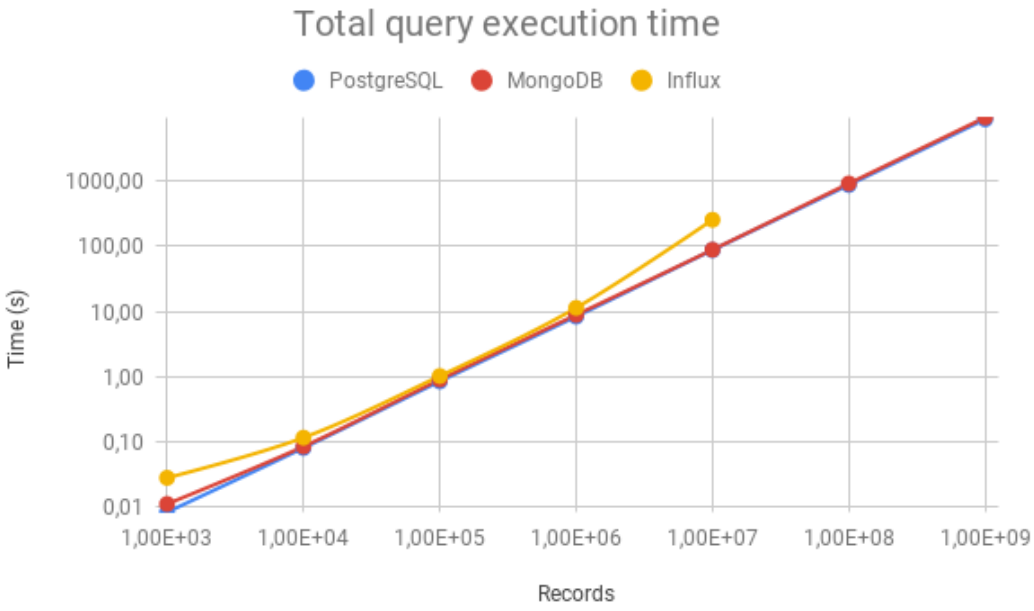



Figure 6.6: Total query execution time

During the total selection tests, we faced in-memory problems when selection records over $10^8$. This problem occurred because we were selecting huge amounts of values into memory without releasing it. In order to solve this problem, we had to use the *iterator* function for PostgreSQL with *chunk_size* equal to 100 000, and the *aggregate* function for MongoDB with a *batchSize* equal to 100 000. This means both solutions will load into memory, small records of 100 000 and return them each time. Unfortunately, the Influx connector doesn't offer, at the moment, any solution to load into memory, small records each time, so we had to stop selecting at $10^7$ records. To get the most reliable results, we repeated the total selection tests several times with the following configurations:

- $10^3$ to $10^6$ records — repeated 50 times
- $10^7$ records — repeated 20 times
- $10^8$ records — repeated 10 times
- $10^9$ records — repeated 5 times

The Figure 6.6 show us the comparison of the total selection in PostgreSQL — without indexes —, MongoDB — without indexes — and Influx — with indexes —, increasing the number of records to be selected from the databases.

```
1 DataSetPG.objects.filter(tower_code=value1, time_stamp__gte=value2, time_stamp__lte
    =value3)
```

Listing 6.3: Django partial query for PostgreSQL

#### 6.2.3.2 Partial Selection

A partial selection is the most used and vital operation, as it consists in getting raw data from a tower between a period. For PostgreSQL query, the partial selection has the following code at Listing 6.3 in Django. For MongoDB, the partial selection is represented by the following code at Listing 6.4 in Django. For Influx, the partial selection has the following code at Listing 6.5 in Django. Where, for the three cases, the value1 is the code tower, and the value2 and value3 are, respectively, the beginning and end date of the period to be searched. We choose a period within 90 days, that corresponds to 8 209 records, to be returned in the queries. We had to start the database populated with $10^4$ records, as the chosen period don't fit in $10^3$ records. It's also essential to refer that we don't order by *time_stamp* or time during the selections, as we are doing it in the Pandas framework.

We can compare in the Figure 6.7 the measured time without indexes for PostgreSQL and MongoDB when making a partial selection.

At this point, by looking at the results, we quickly realized the need to create indexes. A final user shouldn't have to wait several minutes to get the raw data belonging to a period from a tower, especially when it's the most used operation in the platform. To fully understand what attributes should we use to create indexes, we study the query, using the query planner. In Django, we can get the query planner from PostgreSQL database using the *explain* function — Unfortunately, PyMODM connector doesn't offer that function, but we could get an idea where to create the indexes in MongoDB using the query planner from Django. We tested with $10^5$ records and by looking for the results at Listing 6.6, we can see only one step made by the database, a sequential scan in all the table to find the raw data from a *tower_code* within a *time_stamp* period.

This step can be easily optimized by creating a composite index with the attributes *tower_code* and *time_stamp*, as we will always be looking for a tower and a period in the same query. This way, it wouldn't be necessary to search sequentially in all *tower_code's* and then in all *time_stamp's* from that *tower_code* to find the desired results. After creating the indexes and running the *explain* function, we can confirm — by checking the results at Listing 6.7 — that the database only hit the

```
1 DataSetMongo.objects.raw({'tower_code': value1, 'time_stamp': {'$gte': value2, '
    $lte': value3}})
```

Listing 6.4: Django partial query for MongoDB

```
INFLUXCLIENT.query("select * FROM value1 where time >= " + value2 + " and time <= "
    + value3)
```

Listing 6.5: Django partial query for Influx

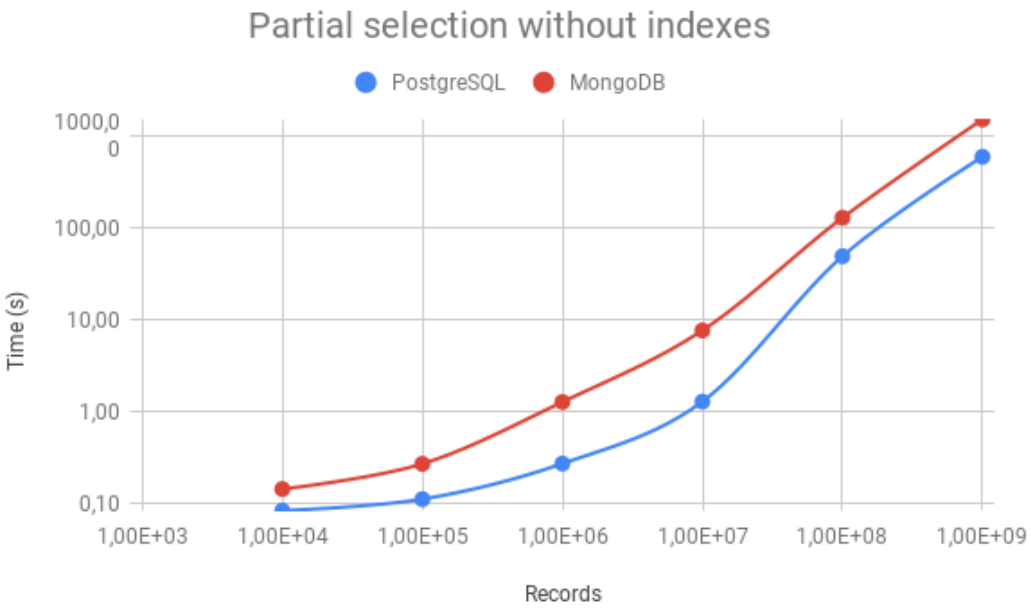



Figure 6.7: Partial selection without indexes

```
Seq Scan on datasetpg  (cost=0.00..3119.00 rows=8147 width=73)
  Filter: ((time_stamp >= '2019-01-05 04:10:00+00'::timestamp with time zone) AND (
      time_stamp <= '2019-04-05 04:10:00+00'::timestamp with time zone) AND ((
      tower_code)::text = 'port1'::text))
```

Listing 6.6: Query planner without indexes

```
1 Bitmap Heap Scan on datasetpg  (cost=232.29..1743.86 rows=8147 width=73)
2   Recheck Cond: (((tower_code)::text = 'port1'::text) AND (time_stamp >= '
        2019-01-05 04:10:00+00'::timestamp with time zone) AND (time_stamp <= '
        2019-04-05 04:10:00+00'::timestamp with time zone))
3   ->  Bitmap Index Scan on main_app_da_tower_c_6a0ada_idx  (cost=0.00..230.25 rows
        =8147 width=0)
4         Index Cond: (((tower_code)::text = 'port1'::text) AND (time_stamp >= '
              2019-01-05 04:10:00+00'::timestamp with time zone) AND (time_stamp <= '
              2019-04-05 04:10:00+00'::timestamp with time zone))
```

Listing 6.7: Query planner using indexes

raw data from the *tower_code* and *time_stamp*, not being necessary to go through the entire table to search for it.

In the Figure 6.8, we can compare the measured time for the partial selection with indexes for PostgreSQL, MongoDB and Influx. We can notice in the Influx database that, over $10^7$ records, the measured time start to increase, by a small portion, as we start to get more *Measurements* — more towers with raw data — in the system. PostgreSQL and MongoDB maintain a constant measured time while increasing the total of records.

#### 6.2.3.3 Results

Through the selection tests, we analyzed the time the platform takes to get the raw data from the different database systems.

By looking at the Table 6.6, we can discuss the results of the execution time for the total selection operation. It's possible to confirm that PostgreSQL over Influx have a speedup average of 2.06. PostgreSQL is also better when making a total selection when comparing to MongoDB, but only with a speedup average of 1.09. One of the main reason for Influx being slower than MongoDB and PostgreSQL has to do with the fact that the query stores all returned data in-memory, instead of storing small records and returning them each time. In fact, we tested PostgreSQL and MongoDB without using, respectively, the *chunk_size* and *batchSize* parameters in the query call, and it was possible to conclude that both databases are 1.9 times slower without those two parameters.

By looking at the Table 6.7, we can compare the execution time for the partial query. It's possible to analyze that PostgreSQL (P) it's better than MongoDB with a speedup average of 3.32 when our databases have no indexes. But at this point, our system was slow when selecting a few results from a table with million of records, so we created indexes and had the following performance improvements for each database system:

- PostgreSQL — improved an average of 113.38 times over the solution without indexes.
- MongoDB — improved an average of 1883.69 times over the solution without indexes.

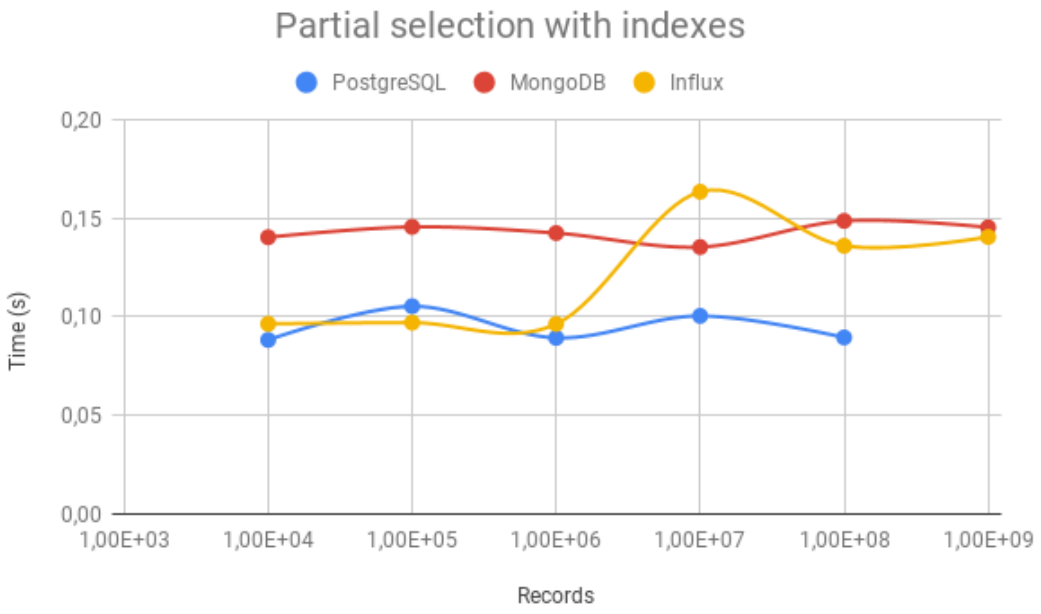



Figure 6.8: Partial selection with indexes

Table 6.6: Execution time for total query

| Records | PostgreSQL | MongoDB | Influx | Speedup (M)/(P) | Speedup (I)/(P) |
|---|---|---|---|---|---|
| $10^3$ | 0.01 s | 0.01 s | 0.03 s | 1.33 | 3.36 |
| $10^4$ | 0.08 s | 0.09 s | 0.12 s | 1.05 | 1.44 |
| $10^5$ | 0.85 s | 0.90 s | 1.03 s | 1.07 | 1.21 |
| $10^6$ | 8.37 s | 8.89 s | 11.38 s | 1.06 | 1.36 |
| $10^7$ | 87.59 s | 88.78 s | 255.51 s | 1.01 | 2.92 |
| $10^8$ | 873.31 s | 920.44 s | NA | 1.05 | NA |
| $10^9$ | 8 867.57 s | 9 487.63 s | NA | 1.07 | NA |

Table 6.7: Execution time for partial query

| **Records** | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $10^7$ | $10^8$ | $10^9$ |
|---|---|---|---|---|---|---|---|
| **Without Indexes** | | | | | | | |
| PostgreSQL | NA | 0.08 s | 0.11 s | 0.27 s | 1.29 s | 49.14 s | 591.74 s |
| MongoDB | NA | 0.15 s | 0.27 s | 1.28 s | 7.68 s | 128.66 s | 1 506.62 s |
| Speedup | NA | 1.71 (P) | 2.42 (P) | 4.67 (P) | 5.93 (P) | 2.62 (P) | 2.55 (P) |
| **With Indexes** | | | | | | | |
| PostgreSQL | NA | 0.09 s | 0.11 s | 0.09 s | 0.10 s | 0.09 s | NA |
| MongoDB | NA | 0.14 s | 0.15 s | 0.14 s | 0.14 s | 0.15 s | 0.15 s |
| Influx | NA | 0.10 s | 0.10 s | 0.10 s | 0.16 s | 0.14 s | 0.14 s |
| Speedup (M)/(P) | NA | 1.59 | 1.38 | 1.59 | 1.35 | 1.66 | NA |
| Speedup (I)/(P) | NA | 1.09 | 0.92 | 1.08 | 1.63 | 1.52 | NA |
| **Improvement With Indexes** | | | | | | | |
| PostgreSQL | NA | 0.96 | 1.07 | 3.07 | 12.89 | 548.89 | NA |
| MongoDB | NA | 1.03 | 1.87 | 8.99 | 56.74 | 865.87 | 1 0367.64 |

If we compare the three database systems with indexes, our results are almost similar on each, giving an advantage to PostgreSQL. It's, with a speedup average of 1.25, faster than Influx, and, with a speedup average of 1.51, faster than MongoDB.

### 6.2.4 Comparison with Wind2Data

One goal was also to compare this new solution with the actual INEGI solution. For this, we used the second machine described at Section 6.1, to get direct access to the Wind2Data. This test aims to make a partial query — in PostgreSQL, MongoDB and Influx — and a chart load using the HighCharts in our platform. Then we do the same in the Wind2Data, that uses MySQL for database and Excel for chart visualization. Each system had $10^6$ records on each database, and we selected the raw data with the following configurations:

- 2 160 records — around 15 days — with 10 repetitions. Having a coefficient of variation of 3.39 % on Wind2Data, and an average of 12.53 % on our system, between the three databases.
- 4 320 records — around 30 days — with 10 repetitions. Having a coefficient of variation of 2.38 % on Wind2Data, and an average of 10.33 % on our system, between the three databases.
- 103 680 records — around 2 years — with 5 repetitions. Having a coefficient of variation of 3.21 % on Wind2Data, and an average of 4.00 % on our system, between the three databases.

In Figure 6.9, we can compare the results and see that Wind2Data is slower than our platform. The main reason for this is because Excel needs to make a query to the MySQL database and then fill the Excel spreadsheet's with all the information from the query, which is a very heavy process.

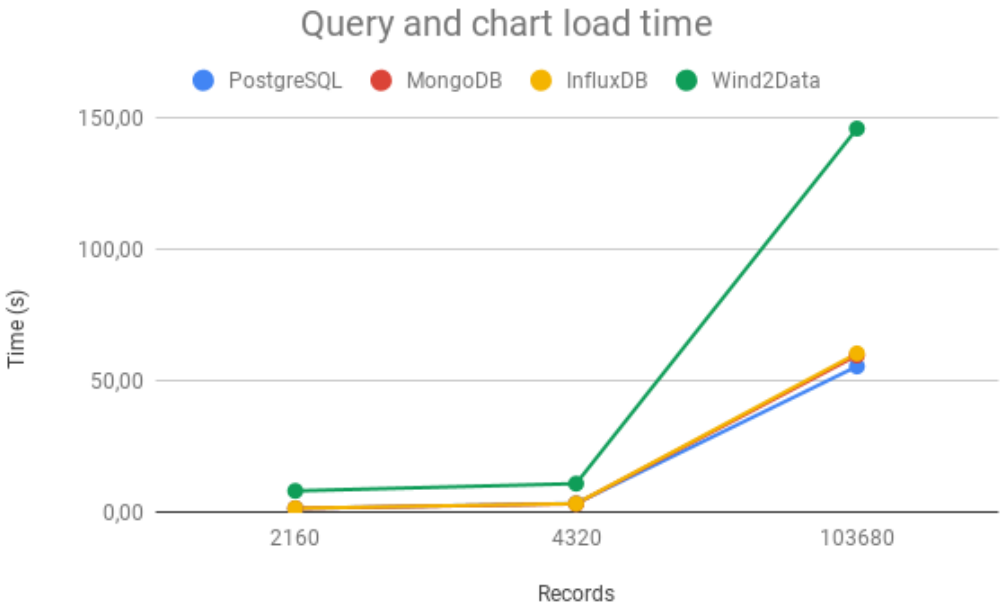



Figure 6.9: Partial selection with indexes

Looking at the Table 6.8, we can confirm that our solution, regardless of the database, is almost 5.6 times better when selecting 2160 records, 3.3 times better when selecting 4320 records, and 2.6 times better when selecting 103 680 records.

Table 6.8: New vs INEGI solution

| Records | PostgreSQL | MongoDB | Influx | Wind2Data |
|---|---|---|---|---|
| 2 160 | 1.43 s | 1.44 s | 1.47 s | 8.08 s |
| 4 320 | 3.31 s | 3.16 s | 3.24 s | 10.84 s |
| 103 680 | 55.40 s | 59.57 s | 60.32 s | 145.77 s |

## 6.3 Validation Tests

During the development of the platform, more precisely on each User Story development, we made validations tests by introducing real data into each form from each User Story. We also tested the classification and raw data chart's with different equipments and configurations to fully validate those implementations. These tests ensure that the product actually meets the client's needs. We create this project "dockerized"[3] to quickly run the platform independently of the SO being used. This way, the client could test the new implementations that were developed. These tests not only helped to reinforce the client's needs but also helped us to find bugs to be fixed during the development.

[3]https://www.docker.com/

## 6.4 Final Results

On the previous sections of the performance tests, we described what is the best database system for insertion, selection and also what is the one that occupies less disk space.

We could conclude that Influx is faster inserting when compared with PostgreSQL and MongoDB — with an average of speedup of 4.32 and 2.94, respectively. For disk space management, we can conclude that Influx occupies 4.92 times less disk space when compared to PostgreSQL and 2.69 times less when compared with MongoDB. Upon a total selection, even if isn't the most used operation by the client, the PostgreSQL is 1.09 times faster against MongoDB and 2.06 times faster against Influx. For a partial selection, using indexes, we got almost the same results, less than 0.2 seconds for each database system, even when the databases had a total of $10^9$ records to search. But PostgreSQL had, slightly, the better results when compared with MongoDB and Influx — better 1.51 and 1.25 times, respectively. It was also astonishing the comparison of PostgreSQL and MongoDB upon the creation of indexes to use on partial selections. PostgreSQL had a speedup improvement of 113.38 and MongoDB an improvement of 1 883.69 without indexes.

With these results, we can conclude that, in overall, Influx is 2.66 better than PostgreSQL and 1.47 better than MongoDB, being able to store and handle raw data from up to 30 years. For this scenario we choose Influx to handle the raw data and PostgreSQL to handle the meta information of the system. Many companies already use Influx in their services[4], for example, Adobe[5] use it as a microservice based on the Adobe EchoSign integration for Microsoft SharePoint online. Cisco[6] for other side uses to monitor their ecommerce application that tracks all Cisco Service Renewals. Huawei[7] uses Influx to collect and show the Linux computing cluster performance data.

We also made performance tests to compare the new solution with the INEGI solution. We conclude that our solution is, on average, 3.86 faster, making the partial selection and loading the chart with raw data.

[4] https://www.influxdata.com/customers/
[5] https://www.adobe.com
[6] https://www.cisco.com
[7] https://www.huawei.com

# Chapter 7

# Conclusions and Future Work

In this Chapter we will summarize the work done and also analyze what future work can be done to improve and complement the implemented system.

## 7.1 Summary

Nowadays, everything around us generates data, specially in the wind energy, a strongly data-driven domain. With all this data comes the need to store and handle it in an efficient and scalable way.

The main goal of this work was the creation of a Web based Information System, that not only stores the raw data, but also handles all meta information associated to the wind meteorological masts in order to ensure traceability and scalability. Flexibility is also a requirement, particularly in what concerns the ability to deal with a vast variety of data sources and data formats. Despite the lack of time to implement and test a large amount of data types, this requirement was taken into account in the solution designed to store and manage the data — making it compatible with, virtually, any time series based data structure. The findings of this work might help INEGI wind energy department understand which solutions shall be followed in future developments of their internal tools.

In the first phase, focus on studying the actual problem was the main goal. And it was quickly realized that it would be a big and complex project. Not only INEGI has to make the data collection but also applies different transfer functions to different data series along time, classifies the data with a diverse range of status and subsequently uses the processed and cleaned data as input to prepare several complex and detailed reports. After getting an overview, we studied the database systems that do not follow the relational model, as an alternative to the RDBMS.

In a second phase, the platform was designed, focusing to storing the raw data, wind meteorological tower and equipment's information. The platform had also to be able to store data classification and present it in charts.

In a third phase, the actual implementation of the platform was accomplished. The platform development followed two big stages, in the first stage we designed all Models for PostgreSQL — a RDBMS — to handle all meta information from wind towers, equipment's and classifications. The second stage focus to insert the raw data using three different storage solutions, PostgreSQL, MongoDB and Influx. The second stage is combined with the first, as the platform is able to access any of the three data bases created. This implementation not only helped us developing the project faster, but also allowed us to compare insertion and selection performance from the three technologies. During the development of the project, the client was able to do validations tests, to confirm that the product actually meets the client needs.

In a fourth stage, we made performance tests and conclude that Influx was the faster database inserting and making partial selections of the raw data. We also measured the disk space to compare what approach uses less disk space storing data, and we could conclude that Influx occupies less disk space when compared with PostgreSQL and MongoDB. These tests not only helped us to deduce which database was faster, but also allowed us to improve the platform during these tests. We also compared our platform with the INEGI platform and could conclude that our platform was, 3.86 times faster when loading raw data from the database to charts.

At the end we conclude that the platform actually meet the client needs being scalable and with a good performance. The platform is able to handle meta information from all designed domain entities, and also to insert and query raw data and show it in charts. We also conclude that RDBMS and TSDB can coexist in the same system, using PostgreSQL to handle the meta information and Influx to store the raw data. This platform is only a small portion of what can be in the future, requiring further studies and also a base for new implementations.

## 7.2 Future Work

As already described on the previous chapters, the developed platform was limited to the core functionalities defined by INEGI as time was limited — in short raw data storage and subsequent processing. Despite that, an effort was made during the design and development of this solution to fulfill the main requirements — traceability, scalability, flexibility and time effectiveness. The flexibility is regarded here as the ability of this system to deal with diverse data types. Thus, despite not fully tested, a considerable effort was made to ensure that the system is suitable to store data of different sources, formats and time scales — not only from wind meteorological masts but also from wind turbines, solar power plants, inverters, meso-scale models, reanalysis data, etc. To make it possible, the approach was to store all time series in a standard structure within the database and on top of each data series, a layer of meta-data is necessary to accurately describe its content, source, provider, owner and many other attributes. As only limited sources of data series were tested, in future work, as other sources of data are considered, it will be necessary to understand which attributes are required to describe it. Apart from the "multi-data-types" ability, the integration of this tool with management and support to the field operations, management of the equipment and clients portfolio, the introduction of advanced analytics, creation of customized

template reports, etc. is still to be done in the future provided the solution developed in the context of this work proves robust and reliable.

PostgreSQL, MongoDB and Influx were used to store the raw data, but these are only a few of the many existing databases. There are other databases, mainly, oriented to timeseries, like TimeScale[1] and OpenTSDB[2] that require study and should be tested in order to improve the system.

[1] https://www.timescale.com/
[2] http://opentsdb.net/

# Appendix A

# URL Tables

At this appendix we can view all the available URLs routes for each entity in the platform that was described at Section 5.2.1.

Table A.1: Routes for Period_Configuration entity

| URL | Type | Description |
|---|---|---|
| tower/{id}/ conf_period/add | GET | Form to add a new period of configuration to a tower. |
| tower/{id}/ conf_period | POST | Add a period of configuration. |
| tower/{id}/ conf_period/{id} | GET | Show tower information, period's of configuration and comments from that tower. |
| | PUT | Update period of configuration information. |
| | DELETE | Remove a period of configuration. |

Table A.2: Routes for Equipment_Configuration entity

| URL | Type | Description |
|---|---|---|
| tower/{id}/ conf_period/{id}/ equi_conf/add | GET | Form to add a equipment configuration to a period of configuration that belongs a station. |
| tower/{id}/ conf_period/{id}/ equi_conf | POST | Add a new equipment configuration. |
| tower/{id}/ conf_period/{id} equi_conf/{id} | GET | Show equipment configuration information. |
| | PUT | Update equipment configuration information. |
| | DELETE | Remove a equipment configuration. |

Table A.3: Routes for Classification_Period entity

| URL | Type | Description |
|---|---|---|
| tower/{id}/ conf_period/{id}/ equi_conf/{id}/ classification/add | GET | Form to add a classification period to a equipment configuration that belongs to a period of configuration from a tower. |
| tower/{id}/ conf_period/{id}/ equi_conf/{id}/ classification | POST | Add a new classification period. |
| tower/{id}/ conf_period/{id} equi_conf/{id} classification/{id} | GET | Show classification period information from an equipment configuration |
| | PUT | Update classification period information. |
| | DELETE | Remove a classification period. |

Table A.4: Routes for Dimension entity

| URL | Type | Description |
|---|---|---|
| tower/{id}/ conf_period/{id}/ equi_conf/{id}/ dimension/add | GET | Returns a form to add a new dimension to a equipment configuration that belongs to a period of configuration's of a tower. |
| tower/{id}/ conf_period/{id}/ equi_conf/{id}/ dimension | POST | Add a new dimension. |
| tower/{id}/ conf_period/{id} equi_conf/{id} dimension/{id} | GET | Show dimension information from an equipment configuration |
| | PUT | Update dimension information from an equipment configuration. |
| | DELETE | Remove a dimension. |

Table A.5: Routes for Comment entity

| **URL** | **Type** | **Description** |
|---|---|---|
| tower/{id}/ conf_period/{id}/ equi_conf/{id}/ classification/{id}/ comment/add | GET | Returns a form to add a comment to a classification that belongs to a equipment configuration from a period of configuration's of a tower. |
| tower/{id}/ conf_period/{id}/ equi_conf/{id}/ classification/{id}/ comment | POST | Add a new comment to a classification that belongs to a equipment. |
| tower/{id}/ comment/add | GET | Returns a form to add a comment to a tower. |
| tower/{id}/ comment | POST | Add a new comment to a tower. |
| tower/{id}/ comment/{id}/ type | GET | Show the comment information from a type — tower or an equipment. |
| | PUT | Update the comment information. |
| | DELETE | Remove a comment. |

Table A.6: Routes for User entity

| **URL** | **Type** | **Description** |
|---|---|---|
| user/new | GET | Form to add a new user. |
| user | POST | Add a new user. |
| users | GET | List all users. |
| user/{id} | GET | Show user information. |
| | PUT | Update user information. |
| | DELETE | Remove a user. |
| user/{id}/ban | PUT | Ban a user. |

Table A.7: Routes to associate User to Tower's entity

| **URL** | **Type** | **Description** |
|---|---|---|
| associate/new | GET | Form to add a new tower's association to an user. |
| associate | POST | Add a new association. |
| associations | GET | List all towers associations. |
| associate/{id} | GET | Show tower association information. |
| | PUT | Update tower association information. |
| | DELETE | Remove a tower association. |

Table A.8: Routes to Cluster entity

| URL | Type | Description |
|---|---|---|
| cluster/new | GET | Form to add a new cluster. |
| cluster | POST | Add a new cluster. |
| clusters | GET | List all cluster's. |
| cluster/{id} | GET | Show cluster information. |
| | PUT | Update cluster information. |
| | DELETE | Remove a cluster. |

Table A.9: Routes to Equipment entity

| URL | Type | Description |
|---|---|---|
| equipment/new | GET | Form to add a new equipment. |
| equipment | POST | Add a new equipment. |
| equipments | GET | List all equipment's. |
| equipment/{id} | GET | Show equipment information. |
| | PUT | Update equipment information. |
| | DELETE | Remove a equipment. |

Table A.10: Routes to Calibration entity

| URL | Type | Description |
|---|---|---|
| equipment/{id}/ calib/add | GET | Form to add a new calibration to the equipment ID. |
| equipment/{id}/ calib | POST | Add a new calibration. |
| equipment/{id} calib/{id} | GET | Show calibration information from an equipment. |
| | PUT | Update calibration information from an equipment. |
| | DELETE | Remove a calibration from an equipment. |

Table A.11: Routes to any Type's entity

| URL | Type | Description |
|---|---|---|
| type/add/typex | GET | Form to add a new type of typex — equipment, model, status, unit, statistic, metric, dimension, component, affiliation, user_group. |
| type | POST | Add a new type of typex. |
| type/{id}/typex | GET | Show type information of a typex. |
| | PUT | Update type information of a typex. |
| | DELETE | Remove a type of typex. |
| dimension_type/add | GET | Form to add a new dimension_type. |
| dimension_type | POST | Add a new dimension_type. |
| dimension_type/{id} | GET | Show dimension_type information. |
| | PUT | Update dimension_type information. |
| | DELETE | Remove a dimension_type. |

Table A.12: Routes for Machine entity

| **URL** | **Type** | **Description** |
|---|---|---|
| machine/new | GET | Form to add a new machine. |
| machine | POST | Add a new machine. |
| machines | GET | List all machine. |
| machine/{id} | GET | Show machine information. |
| | PUT | Update machine information. |
| | DELETE | Remove a machine. |

Table A.13: Routes for Status entity

| **URL** | **Type** | **Description** |
|---|---|---|
| status/new | GET | Form to add a new status. |
| status | POST | Add a new status. |
| statues | GET | List all status. |
| status/{id} | GET | Show status information. |
| | PUT | Update status information. |
| | DELETE | Remove a status. |

Table A.14: Routes for Data Management

| **URL** | **Type** | **Description** |
|---|---|---|
| tower/{id}/show | GET | Show charts of raw data and classifications made |
| classify_from_charts | POST | Make a new classification viewing charts |
| view_classifications_chart | GET | Ajax call to initiate and update the classification chart |
| view_raw_data | GET | Ajax call to initiate and update the raw data chart |

# References


[1] Scott W Ambler. *The object primer: Agile model-driven development with UML 2.0*. Cambridge University Press, 2004.

[2] Ammonit. *Ammonit manual datalogger*. Ammonit, 2009.

[3] Bruce H Bailey, Scott L McDonald, D W Bernadett, M J Markus, and K V Elsholz. Wind resource assessment handbook: Fundamentals for conducting a successful monitoring program. Technical report, AWS Scientific, 1997.

[4] Asian Development Bank. *Guidelines for wind resource assessment: Best practicces for countries initiating wind development*. Asian Development Bank, 2014.

[5] Eric Brewer. Towards robust distributed systems. In *Symposium on Principles of Distributed Computing (PODC)*, page 7, 2000.

[6] Min Chen, Shiwen Mao, and Yunhao Liu. Big data: A survey. In *Mobile Networks and Applications*, volume 19, pages 171–209, 2014.

[7] Chris Churilo. InfluxDB is 2.4x Faster vs. MongoDB for Time Series Workloads, 2018.

[8] E F Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, 1970.

[9] Tom Cronin, Niels-erik Clausen, B. Frydenber, E. Huard N. Tuan, S. Hernando, and T.T Lien. *Best practice guidelines for the development of wind energy projects*. EC-ASEAN Energy Facility, 2007.

[10] John Daniels and J Cheesman. *UML components: a simple process for specifying component-based software*. Addison-Wesley, 2001.

[11] Paul DuBois, S Hinz, J Stephens, M Brown, and T Bedford. MySQL 5.1 reference manual. *Oracle Corporation*, 2008.

[12] Jason B Dunham, Gwynne L Chandler, Bruce Rieman, and Don Martin. Measuring Stream Temperature with Digital Data Loggers: A User's Guide. *USFS Technical Report*, pages 1–18, 2005.

[13] John Gantz and David Reinsel. Extracting value from chaos. *IDC iview*, 1142(2011):1–12, 2011.

[14] Wolfgang Gentzsch, Damien Lecarpentier, and Peter Wittenburg. Big data in science and the EUDAT project. In *Global Conference (SRII), 2014 Annual SRII*, pages 191–194. IEEE, 2014.

[15] Filipe Gomes, João Correia Lopes, José Laginha Palma, and Luís Frölén Ribeiro. WindS@UP: The e-science platform for WindScanner.eu. In *Journal of Physics: Conference Series*, 2014.

[16] Jim Gray. Readings in Database Systems. In Michael Stonebraker, editor, *Readings in Database Systems*, pages 140–150. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[17] Theo Haerder and Andreas Reuter. Principles of Transaction-oriented Database Recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[18] Jing Han, Haihong E, Guan Le, and Jian Du. Survey on NoSQL databases. In *2011 6th International Conference on Pervasive Computing and Applications*, pages 363–366, 2011.

[19] High Level Expert Group on Scientific Data. Riding the wave: How Europe can gain from the rising tide of scientific data. *Final report to the European Commission*, 2010.

[20] White House. Fact sheet: Big data across the federal government. *Office of Science and Technology Policy*, 2012.

[21] Influxdb.com. InfluxDB - Open Source Time Series, Metrics, and Analytics Database, 2019.

[22] S Iniyan, S Jebaraj, L Suganthi, and Anand A Samuel. Energy Models for Renewable Energy Utilization and To Replace fossil fuels. *METHODOLOGY*, 2016.

[23] Solid IT. DB-Engines - Knowledge Base of Relational and NoSQL Database Management Systems, 2019.

[24] Gerhard Klimeck, Michael Mclennan, Sean Brophy, George Adams, and Mark Lundstrom. nanoHUB.org: Advancing Education and Research in Nanotechnology. *Computing in Science & Engineering*, 10:17–23, 2008.

[25] Damien Lecarpentier, Peter Wittenburg, Willem Elbers, Alberto Michelini, Riam Kanso, Peter Coveney, and Rob Baxter. EUDAT: a new cross-disciplinary data infrastructure for science. *International Journal of Digital Curation*, 8(1):279–287, 2013.

[26] Avraham Leff and James T Rayfield. Web-application development using the model/view/-controller design pattern. In *Enterprise Distributed Object Computing Conference, 2001. EDOC'01. Proceedings. Fifth IEEE International*, pages 118–127. IEEE, 2001.

[27] Yishan Li and Sathiamoorthy Manoharan. A performance comparison of SQL and NoSQL databases. In *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pages 15–19. IEEE, 2013.

[28] Xiao Liu, Dequn Zhou, Peng Zhou, and Qunwei Wang. Factors driving energy consumption in China: A joint decomposition approach. *Journal of Cleaner Production*, 172:724–734, 2018.

[29] Sebastian Maneth and Alexandra Poulovassilis. Data science, 2017.

[30] Michael McLennan, Steven Clark, Ewa Deelman, Mats Rynge, Karan Vahi, Frank McKenna, Derrick Kearney, and Carol Song. HUBzero and Pegasus: integrating scientific workflows into science gateways. *Concurrency and Computation: Practice and Experience*, 27(2):328–343, 2015.

[31] Michael McLennan and Rick Kennell. HUBzero: A Platform for Dissemination and Collaboration in Computational Science and Engineering. *Computing in Science & Engineering*, 12(2):48–53, 2010.

[32] Jeremy Tensen Michael Brower, Mike Marcus, Mark Taylor, Dan Bernadett, Matthew Filippelli, Philippe Beaucage, Erik Hale, Kurt Elsholz, Jim Doane, Matt Eberhard. *Wind Resource Assessment Handbook*. Aws TruePower, 2010.

[33] Torben Mikkelsen, Søren Knudsen, M Sjoholm, Nikolas Angelou, and A Tegrmeier. WindScanner.eu — A New Remote Sensing Research Infrastructure for On-and Offshore Wind Energy. In *International Conference on Wind Energy: Materials, Engineering and Policies (WEMEP2012), Hyderabad, India, Nov*, pages 22–23, 2012.

[34] André Milani. *PostgreSQL-Guia do Programador*. Novatec Editora, 2008.

[35] C Mohan. History Repeats Itself: Sensible and NonsenSQL Aspects of the NoSQL Hoopla. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 11–16, New York, NY, USA, 2013. ACM.

[36] Ameya Nayak, Anil Poriya, and Dikshay Poojary. Type of NOSQL databases and its comparison with relational databases. *International Journal of Applied Information Systems*, 5(4):16–19, 2013.

[37] Rachel Stephens. State of the Time Series Database Market, 2018.

[38] John Twidell and Tony Weir. *Renewable energy resources*. Routledge, 2015.

[39] Z Wei-ping, L Ming-xin, and C Huan. Using MongoDB to implement textbook management system instead of MySQL. In *2011 IEEE 3rd International Conference on Communication Software and Networks*, pages 303–305, 2011.

[40] Enerdata World Energy Statistics. Total Energy Consumption, 2018.