Integration of technologies for Big and Open Geodata handling.

Integración de tecnologías para tratamiento de datos masivos y abiertos en el área de geomática.

Final Grade Project

Computer Engineering

by

Alejandro Sánchez Medina

Tutors: Agustín Trujillo Pino and Pablo Fernández Moniz

Las Palmas de Gran Canaria, November, 2016

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Description and motivation of the project

The topics of Big Data, distributed databases and Open Data were introduced to the student while working on a grant for a research and development team of the University of Las Palmas de Gran Canaria.

After collaborating on different projects related with these technologies, it became interesting for the student to expand on some of the tasks worked to make a complete final degree project and deepen on the concepts learned.

Therefore, the project is focused in researching and experimenting with Big Data handling technologies and methodologies, with special interest on applications where geolocated data is an important asset.

1.2 State of the art

From the first decade of the 2000s, a new trend has appeared in the field of data management and manipulation which has been named **Big Data**.

A common description of Big Data technologies is that they deal with the **3 Vs of data**[1]:

- **Volume**: Big Data technologies bring new solutions to the storage and management of high volumes of data, usually composed of small registers, for example, individual readings of a network of sensors.

- **Velocity**: One of the reasons of the generation of Big Data is normally the high rate at which it is generated. In this field it is important to process as fast as possible the new data collected which can even be needed to take some reactive actions in almost real time.
- **Variety**: This field also addresses the issue of managing different sources of data or even providing better flexibility regarding structure changes on an already known data source.

Another interesting point of view is to bring the focus on the different processing approachs in Big Data technologies compared with traditional relational databases [2]. Traditional databases use a **Schema-on-Write** (**SOW**) approach, where data structure must be clear before storing it. On the other hand, Big Data workflows provide a **Schema-on-Read** (**SOR**) paradigm, where the data is stored in raw format (typically text files) and the structure is applied at the moment it is read or processed(Figure 1.1).

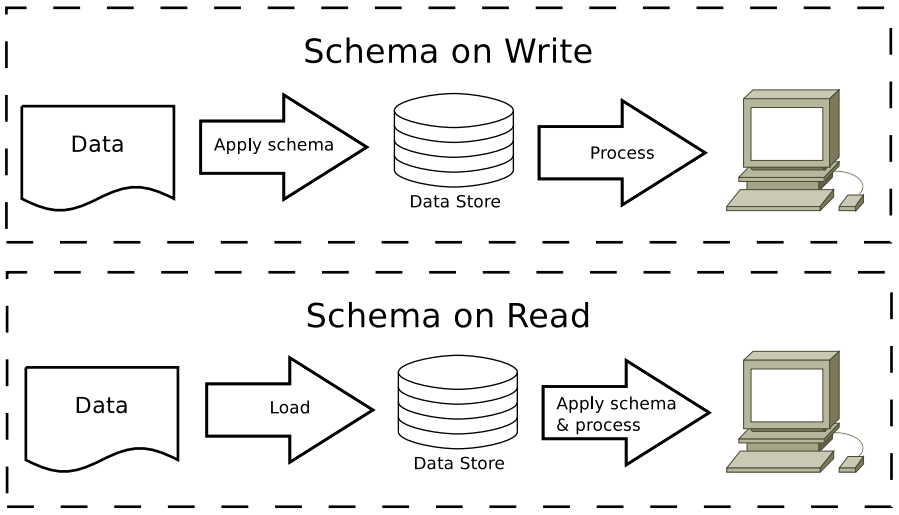


FIGURE 1.1: Schema-on-Write and Schema-on-Read paradigms.

The following sections aim to give a general overview of the technologies that have been used in this project, which are all free and open source software.

1.2.1 Relational databases

Relational Database Managements Systems (**RDBMS**) have been the base of multiple businesses and applications since the 1970s. In a relational database the data is structured in tables, where every table is defined by a set of columns and rows. Normally a table is a representation of an entity of the domain being modelled, where each column

represents different attributes of that entity. Finally, each row or record represents a single instance of the modelled entity.

Relational databases are based on the set theory and the **Standard Query Language** (**SQL**) defines the operations that can be made over their data structures.

A key point of the power of RDBMS is data integrity. In order to guarantee it, these systems implement **ACID** (**A**tomicity, **C**onsistency, **I**solation and **D**urability) transactions (sets of data instructions such as insert data, delete data and transform data). The ACID characteristics can be described as:

- **Atomicity**: If any instruction of the database transaction fails, the whole transaction fails and nothing is changed.
- **Consistency**: The system has to be coherent with the rules defined over its model. In this case, it is achieved by not allowing half-completed transactions.
- **Isolation**: Each transaction is independent from each other, avoiding interferences between them.
- **Durability**: When a transaction is finished, it cannot be undone and it will survive any system failure.

1.2.1.1 PostgreSQL

FIGURE 1.2: PostgreSQL logo.

PostgreSQL [3, 4] is an object oriented RDBMS widely used for resource management software at enterprise level. Its main advantages are its stability, constant development, features coverage and extensibility.

PostgreSQL provides the following abstractions to model the storage of our data models:

- **Tables**: The information stored is modelled and presented in tabular form where each **column** represents an attribute of the entity it represents. Each **row** of the table represents an instance stored of said entity and queries are executed by referencing them.

- **Schemas**: All tables are grouped in schemas, which can serve a functionality or thematic purpose.
- **Databases**: Finally, schemas are contained within databases, which are normally dedicated to a single application.

Since 2011, PostgreSQL also has support for the **SQL/MED** [5, 6] (**M**anagement of **E**xternal **D**ata) which establishes a standard to access data stored in remote database engines via **foreign data wrappers** (**FDW**). There are foreign data wrappers available to connect to other PostgreSQL instances (**Postgres FDW** [7]) and also to other database management systems like MongoDB (**MongoDB FDW** [8]).

PostgreSQL also allows developers to extend their database implementations by defining custom types, custom procedural functions and even packaging new functionality into extensions. To develop stored procedures, which can then be triggered to add extra validations or data transformations, PostgreSQL offers several procedural languages syntaxes. **PL/PgSQL** [9] is the procedural language developed by PostgreSQL developers but they also support other external languages such as Python with **PL/Python** [10].

1.2.2 Non relational databases

Non relational databases (also known as **NoSQL**, Not only SQL) have been recently gaining popularity given the shift in computing and storing paradigms to distributed environments and cloud platforms.

As explained in 1.2.1, relational databases rely in the set theory for their functioning; however, non relational databases search for other underlying mathematical theories and data structures (such as graphs, associative arrays and tuples) to provide different means to tackle problems. Having the option to model on a system that takes advantage of nature of the data to be stored helps speeding response times and can ease software development.

Non relational databases focus on **horizontal scalability** (that is adding more nodes in a distributed configuration) instead of **vertical scalability** (making the machine *bigger* by adding more CPU, RAM and disk space). This feature makes it harder to guarantee the ACID requirements for which relational databases are so important.

In distributed systems, it is important to take into account the **CAP** (**C**onsistency, **A**vailability and **P**artition tolerance) **theorem** which consists of two parts. The first part defines three requirements to make successful distributed systems:

- **Consistency**: It is equivalent to the consistency presented in ACID systems. It also covers the need of the data contained in each node to be aligned with the state of the rest of the nodes.
- **Availability**: A system is available if it returns a response for all he requests that it receives, whether it is a success or a failure.
- **Partition tolerance**: This requirement refers to the ability of the system to keep working when a node fails and becomes unavailable.

The second part of the theorem states that distributed systems can only guarantee two of the three requirements at the same time. Since those systems are distributed, the weak link of the equation falls on the consistency of the system.

The CAP theorem them gives insight about the unsuitability of ACID characteristics on distributed systems, such as the set-ups that non relational databases aim to cover. In order to overcome this limitation, consistency needs to be postponed during the phase of data storage.

Following these ideas, the **BASE** (**B**asically **A**vailable, **S**oft state and **E**ventual consistency) characteristics were designed to achieve system reliability. Those requirements can be defined as:

- **Basically Available**: Every request sent to the system will receive a reply. Sometimes a successful reply might return inconsistent data at that specific time.
- **Soft state**: The system has a soft state in the sense that, while it stabilizes, the system can change its state without having received any input.
- **Eventual consistency**: While the system receives input, the consistency between transactions is not checked. However, from the moment the system stops receiving input data will start to propagate to all the nodes and the system will become consistent.

It is important to note that not all distributed databases have to be designed around BASE principles, but it is important to know this new paradigm and how it contrasts with the classical ACID [11]. This aspect and many others have been noted on several comparison tables in Appendix A.

1.2.2.1 MongoDB

FIGURE 1.3: MongoDB logo.

MongoDB [12] is a non relational **document** database that aims to provide high performance, high availability and automatic scaling storage solution.

In order to organize the data stored, MongoDB provides the following abstractions:

- **Documents**: A document is a recursive data structure of key-value pairs. Every field can contain other single values, documents and arrays of different types. This data structure is a text representation of objects from diverse language programming and MongoDB represents them with JSON syntax. The storing format of MongoDB documents is **BSON** (**B**inary **JSON**). Documents can be associated with PostgreSQL rows.
- **Collections**: Every document is associated to a MongoDB collection. Since all data queries are executed over collections, they could be associated with PostgreSQL tables. The difference is that collections can have documents with different structure, since the structure is stored in the same abstraction where data is stored, while tables already define a static structure and rows must comply with it.
- **Databases**: Databases contain all the collections of the model. They do not provide an equivalent alternative to PostgreSQL schemas.

1.2.3 Hadoop ecosystem

The **Hadoop ecosystem** is one the most extended and known Big Data ecosystems at the present. Based on Apache Hadoop framework, we can find many projects in the market that try to complement or improve it [13]. In the next sections a collection of them will be presented, starting from Hadoop itself. All of them are licensed under the Apache v2 license.

1.2.3.1 Hadoop

FIGURE 1.4: Hadoop logo.

Apache Hadoop [14] is a dedicated software for distributed data processing. Its architecture was designed to offer **horizontal scalability** by using hardware without advanced features, cheap and easily interchangeable (usually referred as commodity hardware).

Hadoop can be divided in three different modules [15], which leverage its distributed computing and storage nature:

- **Hadoop Distributed File System** (**HDFS**), where data is stored across the different nodes of a Hadoop cluster. HDFS follows a master-slave model where the master node (**NameNode**) is responsible of tracking which data blocks are stored in each slave node (**DataNodes**). This system achieves resilience to disk failure by making redundant copies of blocks across different nodes. When a block is under the desired level of replication, the NameNode schedules the creation of an extra block copy inside the cluster.
- **MapReduce** [16], which serves as the processing framework for the Hadoop cluster. It follows the **divide and conquer** paradigm, where computations are divided in smaller sets of computations which results are later recombined. All the smaller computations can them be sent to different DataNodes to process and combine the result in parallel execution. The name of the framework comes from the two basic phases to apply in this processing approach:
  - The **Map** phase takes a data record in (key, value) format and produces a list (key, value) pairs: `map(k1,v1) -> list(k2,v2)`
  - The **Reduce** phase takes all the different lists from the Map output and groups them to create a distinct list for each key: `reduce(k2, list (v2)) -> k3,list(v3)`
- **Yet Another Resource Negotiator** (**YARN**), which is responsible for the allocation of the MapReduce jobs computation into the DataNodes.

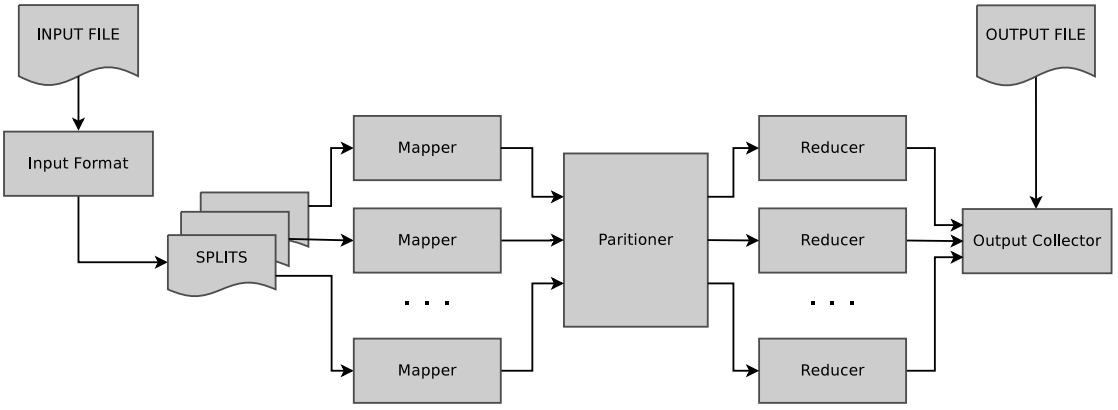


FIGURE 1.5: General description of the components of a MapReduce job execution flow.

Figure 1.5 shows a simplified process flow for a MapReduce job from the point of view of the classes normally involved. In order to make custom MapReduce jobs, the developer needs to implement the map and reduce function defined on their respective classes to process the desired files. If the computation has some more complex requirements, it is also possible to override the rest of the classes to adapt its behaviour to the specific problem. HDFS is used to read the input file and write the output file at the end.

Hadoop is implemented using Java and the MapReduce framework provide APIs for Java, Python and R programming languages. Some recognised Hadoop commercial implementations are **Hortonworks** [17], **Cloudera** [18] and **MapR** [19].

1.2.3.2 Hive

FIGURE 1.6: Hive logo.

Hive [20] is a data warehouse tool built on top of distributed storage systems like HDFS. It provides a SQL-like interface called **HiveQL** and indexing features.

Hive can be executed in console mode, where **HiveQL** queries are interpreted, translated to MapReduce jobs and sent for execution to the underlying Hadoop instance.

HiveQL is also a more suitable syntax for scripting and through the **HiveServer** it can also serve as remote entry point to query a Hadoop cluster.

1.2.3.3 Spark

FIGURE 1.7: Spark logo.

Spark [21] is a distributed computing framework compatible with Apache Hadoop. It can be used as a standalone library or in conjunction with different storage engines. Its most famous set-up is substituting MapReduce in a Hadoop cluster.

Spark is implemented in Scala, an object oriented functional language for the **Java Virtual Machine** (**JVM**), which makes it compatible with Java and the thousands of libraries on the Java ecosystem.

The project claims to be from 10 to 100 times faster than MapReduce applications and due to its Scala implementation it also offers a higher and more powerful abstraction model by using the functional programming paradigm.

Although Scala offers parallel data structures, they are intended for multi-core CPUs and cannot be remotely distributed. To cover multiple node clusters, Spark provides a custom data type named **Resilient Distributed Dataset** (**RDD**) which has the following properties [22]:

- **Immutability**: once an RDD is created, it cannot be changed. This property is inspired by immutable objects in functional programming, which make this type of data suitable for distributed computation as it guarantees that the underlying data is not changed by side effects.
- **Distributability**: when run on a cluster of nodes, RDDs can be separated in smaller chunks and sent to different contexts. The dataset also stores metadata information that allows Spark to recover from system failures.

- **On memory storage**: Spark tries to keep all the RDDs in memory as much as possible and this optimisation is the reason of its performance improvement.
- **Strong typing**: like Scala types, RDDs are strongly typed, which minimizes the number of runtime errors. It is also very valuable to receive data type issues on compile time for processes that run on large datasets.

  RDDs also implement most of the native higher order functions available in Scala, which include map and reduce variations. The use of Scala higher order functions in Spark leads to very expressive and concise code, in contrast to the verbose nature of Java and MapReduce.

1.2.4 Open Data portals

In recent years there has been an increasing awareness to promote **Open Data** access. In the Open Data movement, a work (e.g: a dataset) is defined [23] as open when it complies with the following requirements:

- The work is distributed under an **open license** or under the **public domain**.
- It needs to be accessible and downloadable via the Internet.
- It is able to be processed, accessed and modified by a computer
- It is provided in an open format, without limiting on the number of times to be retrieved, no extra monetary costs and it needs to be able to be processed by at least one free/open source software tool.

Open Data access is, therefore, of key importance for fair collaborations, equality of opportunities and increase of transparency between organisations, institutions, governments and individuals.

1.2.4.1 CKAN

FIGURE 1.8: CKAN open data portal logo.

CKAN[24] is the most extended open source Open Data Portal at the time. It is being developed by the non-for-profit organisation **Open Knowledge Foundation** [25].

The advantages provided by CKAN can be summed up in the following list:

- It is highly customisable, allowing each organisation to develop their own visual identity.
- It is compatible with well established **Content Management Systems** (**CMS**) like Wordpress and Drupal.
- It provides many extensions for file uploads, data conversions, visualisations and many more. It also provides and extensive API to develop custom extensions if needed.
- Its design revolves around the easy of access both for human interaction and machine communication trough REST APIs.

CKAN is being used to serve open data collections by several governments like the European Union, USA and Australia [26].

1.2.5 FIWARE platform

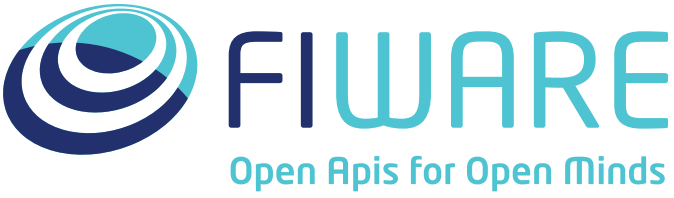


FIGURE 1.9: FIWARE platform logo.

The **FIWARE platform** [27] is a collection of APIs, named **Generic Enablers** (**GEs**), aiming to facilitate the development of Smart Applications. The platform defines in open specifications how these GEs should behave and work between them and also provides publicly available reference implementations, named **Generic Enabler Reference Implementation** (**GEris**).

The project allows developers to also use private testing virtual machines in order to test new application concepts based on the technology provided by the own platform. This way, the project aims to promote the tools and lower the requirements from developers to develop their initial concept of application.

In the next sections, some deeper insight on the most meaningful GEris for the project is provided. However, the platform hosts dozens of varying tools for different needs.

1.2.5.1 Orion Context Broker

FIGURE 1.10: Orion Context Broker logo.

Orion Context Broker [28] is the GEri oriented on data context management built on top of MongoDB. The project implements the NGSI9 and NGSI10 standards [29] to provide an API that allows creation, editing and deleting entities and subscriptions to changes on those entities.

Among its features, it also provides means to store geolocated data and implements some basic geoquery operators, although work is still in progress in this aspect.

Orion Context data can be persisted to a Hadoop instance thanks to the FIWARE related project **Cygnus** [30], which was designed to work as a data stream processor.

1.2.6 Docker

FIGURE 1.11: Docker logo.

Docker ([31, 32]) is an open source engine aimed to the virtualisation of software. Docker provides **containers** as virtualisation tools instead of virtual machines.

Both containers and virtual machines pack next to the target software all its binary dependencies and libraries, but the main difference between them is that virtual machines also include a complete guest operating system while containers share the base kernels and run on separate user spaces, which make them considerably more lightweight.

Docker containers are configured by writing the necessary installation instructions on special configuration files called **Dockerfiles**. Once the user is satisfied with the configuration the Dockerfile can be run to create a static **image** which can be afterwards used to spawn several containers without having to compile all the necessary dependencies form scratch. Both Dockerfiles and Docker images can be published in the **Docker Hub** [33] repository, from which other users can reuse already working configurations.

These features make Docker a very useful tool for tasks like deploying cloud applications, continuous integration (allowing to use containers as development and production machines) and creating microservices.

1.2.7 Geospatial tools

FIGURE 1.12: OSGeo and OGC logo.

In the field of geospatial tools and solutions, the **Open Source Geospatial Foundation** (**OSGeo** [34]) and the **Open Geospatial Consortium** (**OGC** [35]) non-for-profit organisations play an important role on the open geospatial software scene. OSGeo promotes free and open geospatial software, the open publication of data managed by governments and the open education on geospatial technology. On the other hand, OGC works on designing open standards [36] to facilitate the sharing of geospatial data.

As the focus of the project is to work with Big Data technologies in use cases with georeferenced data, it is important to describe the main geospatial technologies used in the following subsections.

1.2.7.1 PostGIS

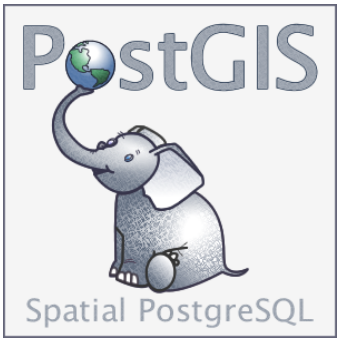


FIGURE 1.13: PostGIS logo.

PostGIS [37] is an extension developed for PostgreSQL databases included in the OSGeo geographic projects. It provides to a PostgreSQL instance with geographic objects, functions and predicates implementations, making it a very powerful tool for geographic data analysis.

1.2.7.2 Java Topology Suite

The **Java Topology Suite** [38] (**JTS**) is a Java library which provides an extensive collection of spatial functions, algorithms and utilities.

The library implements the standard **Simple Features Specification for SQL** [36] for the representation of geometric entities. Given its varied and completed API, it is used by numerous GIS software in the Java ecosystem.

1.2.7.3 OpenStreetMap

FIGURE 1.14: OpenStreetMap logo.

OpenStreetMap [39] (**OSM**) is an Open Data collaborative project to create free and editable maps. OSM map layers are edited by volunteers around the world and its vision

is to provide free and open georeferenced data in contrast to other services like Google Maps, where data is distributed under proprietary license.

The maps and metadata hosted in OSM servers can be used to provide a background map representation for applications with geolocated data.

1.2.7.4 QGIS

FIGURE 1.15: QGIS logo.

QGIS [40] is a multiplatform **Geographic Information System** (**GIS**), that is a set of tools with means for the organisation, storage, analysis and editing of geographic data.

QGIS is able to manage geographic **raster data** (which are bitmaps and images like TIFF, JPG or GeoTIFF), **vector data** (e.g.: Shapefiles) and connections to PostGIS-enabled PostgreSQL databases.

Besides providing a wide variety of utilities in its core implementation, QGIS is also easily extended by developing and installing custom Python plugins to solve domain specific problems. The development of said plugins are possible thanks to the **PyQGIS** [41] and **PyQT** [42] APIs available in the system.

Chapter 2

Objectives

2.1 Direct objectives of the project

Since geodata handling libraries for with Big Data technologies have not been as developed as in other fields, the project aims to show some of their current possibilities and limitations.

Another important objective is showing publication processes to automatically make data of interest accessible to the citizenship via integration with Open Data portals.

These objectives will be implemented focusing in three different scenarios where different combinations of technologies will be put into practice. The next sections of this chapter will provide the initial analysis and requisites of every scenario.

The fist case to tackle is the parsing of GPS logs collected from the cars of the private transportation service named Uber. The main idea is to upload the logs into a Hadoop cluster and analyse the data using MapReduce jobs. Afterwards the same solution will be provided but making use of the Spark framework in order to provide a comparison between both workflows.

For the second case, the focus will be on collecting AIS messages and parsing them to be inserted into an Orion Context Broker instance. For persistence storage, the data will be then connected to a Hadoop instance and after some simple transformations it will be accessible through a CKAN Open Data Portal

The last case consists on providing a water management system with higher scalability by storing the history of its changes over time in MongoDB, a non relational database.

In order to reach the highest audience possible for this project document, English was chosen as the language to be used.

2.2 Indirect objectives of the project

As Big Data analysis and technologies have become a trend, this project aims to introduce the student in the topic as a possibility for specialisation, which is of high personal interest given their relationship with distributed systems and distributed programming.

The project will also allow the opportunity to practice server administration skills including management of dependencies, compilation of source code as the requirements arise and managing services.

On the side of the software used, the project provides a perfect environment to analyse, adapt and implement different software architectures tailored to the use case in sight.

Finally, the student wanted to study and put in practice LaTeX. Thus, the current document has been written using it.

Chapter 3

Requirements

3.1 Hardware requirements of the project

This project has been developed using a personal laptop and a server provided by the faculty for the student to configure. The majority of the development and testing was done on the laptop while the server has been used to run the software developed uninteruptedly during several days.

In both cases the hardware used was very modest, which makes the project easy to be replicated for educational purposes:

- Latop hardware description:
  - **CPU**: 64 bits, Intel Core i7-4500, 1.8 GHz.
  - **RAM**: 8 GB DDR3, 1600 MHz .
  - **Disk**: 500 GB hard drive.
- Server hardware description:
  - **CPU**: 64 bits, Intel Xeon L5320, 1.86 GHz.
  - **RAM**: 4 x 2 GB DDR2, 667 MHz .
  - **Disk**: 126 GB hard drive.

3.2 Software requirements of the project

The software requirements to be installed in the machines has been kept at minimum thanks to the use of virtualisation.

Therefore, to run this project the bare minimum software required in the working machines was:

- **Ubuntu Desktop 14.04** for the laptop and **Ubuntu server 14.04** for the server machine.
- **Docker** engine to virtualise through containers the three use cases to develop.
- **QGIS Desktop** to be used as a client program to explore geolocated data.

All the databases and frameworks used in the project have been presented and described in the state of the art. Given those technologies, the solutions have been implemented using the programming languages **Python**, **Java**, **Scala** and **PHP**.

On the side of the software used to develop the project document, the tools used have been:

- **LaTeX** and **Overleaf** to write this document.
- **Dia**, **KolourPaint** and **Inkscape** to create the figures and prepare pictures.
- **Microsoft Excel** to create graphs.

From all the previous software, Microsoft Excel is the only proprietary software used in the development of this project and associated memory.

Chapter 4

Case 1: Parsing and processing GPS data.

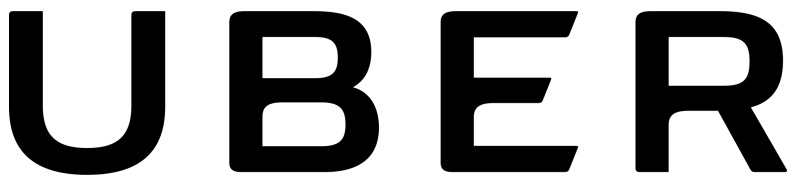


FIGURE 4.1: Uber logo.

This first use case is inspired by the on-demand car service company Uber [43]. Through their user and driver apps, Uber connects users which the need of moving around a city and private drivers willing to provide such service. The advantages that the application provides are:

- It allows the users to request for a driver from a simple tap on their smartphones.
- Once an available driver is located, the user is able to see in their smartphone the location of the driver and how much time is left to be picked up.
- In many cities, the total expense is much cheaper than the local cab service.
- Both driver and user can check on real time the route to follow to the destination.
- The payment process is handled by the application, which transfer the money from the user to the driver once the trip is confirmed to have finished. This brings the advantage for the users to avoid carrying cash to pay for such services.

The objective of this use case is to deploy a Hadoop-based architecture to store, process and visualise GPS data.

FIGURE 4.2: Driver using the Uber app as a GPS navigator.

4.1 Analysis

The dataset used is an anonymised collection of 1128663 GPS data entries from trips registered by the Uber application in the city of San Francisco, California [44]. Each entry of the dataset gives information about the position of an Uber car and it is formatted as a tab-separated line indicating the ID of the trip, the time at which the position was read, the latitude and the longitude of the car (see Figure 4.3). The default GPS data reading update refresh is 4 seconds, but it can register higher intervals, specially if the car has stopped for a long period of time. The dataset has also protected when the trips were really registered by translating all the dates to the first week of January 2007, keeping untouched both the time and the day of the week.

trip_id	iso_timestamp	latitude	longitude
	[...]		
00008	2007-01-03T01:00:56+00:00	37.750103	-122.443784
00008	2007-01-03T01:01:02+00:00	37.750390	-122.444010
00008	2007-01-03T01:01:08+00:00	37.750448	-122.444013
00008	2007-01-03T01:01:14+00:00	37.750536	-122.444040
00008	2007-01-03T01:01:20+00:00	37.750493	-122.444141
00009	2007-01-07T05:05:40+00:00	37.790859	-122.402808
00009	2007-01-07T05:05:46+00:00	37.790864	-122.402768
00009	2007-01-07T05:06:22+00:00	37.790995	-122.402539
00009	2007-01-07T05:06:28+00:00	37.791148	-122.402172
00009	2007-01-07T05:06:34+00:00	37.791385	-122.401312
00009	2007-01-07T05:06:40+00:00	37.791405	-122.400776
	[...]		

FIGURE 4.3: Fragment of the Uber dataset used.

Given the manageable size of the complete dataset, it is interesting to start working with it using a sequential approach in order to get familiar with its results and have a reference implementation. In this way, it is easier to validate and compare future distributed solutions. The programming language chosen to implement the sequential solution is Python because it is an already familiar language for the student, which makes it a perfect candidate for prototyping. The distributed solutions will be developed using the MapReduce and Spark frameworks, where both will read the input and write the output using HDFS.

Taken all the previous considerations into account, the tasks to be worked in this use case are:

1. Configure Docker files to create a Hadoop and Spark container.
2. Clean-up unwanted entries from the dataset.
3. Develop Python script (sequential solution).
4. Develop MapReduce program (distributed solution).
5. Develop Spark program (distributed solution).
6. Develop a Web API service to work with data stored in HDFS.
7. Develop a simple QGIS plug-in to allow the interaction with the previously develop Web API and the visualisation of results.

4.2 Design and development

4.2.1 Configuring and installing server side technologies using Docker

All the back-end configuration and installation has been prepared using a CentOS based Docker container. In order to allow this case to be executed in a general computer instead of a dedicated and expensive cluster of servers, we only use a single machine acting as a master and slave in **pseudo-distributed mode**, which simulates a cluster by allowing to execute more than one MapReduce jobs in the same machine.

Therefore, as can be seen in Figure 4.4, the software installed in this single container is Hadoop (which deploys an instance of HDFS and YARN), Spark (which is configured to make use of HDFS), Hive for basic querying and Apache Web Server for the minimal web service. The rest of the software is loaded to accessible locations and executed on top of the previous basic software.

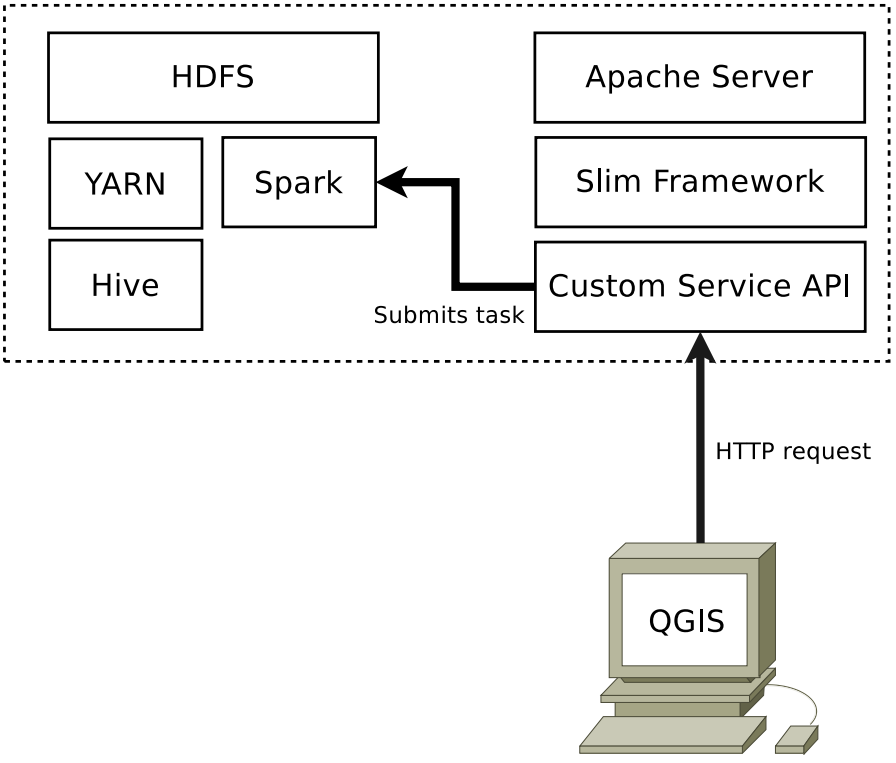


FIGURE 4.4: Diagram of the docker container configuration for case 1.

4.2.2 Dataset clean-up

Once all the needed software is configured and ready to be used, the first important step taken was cleaning up the dataset to avoid undesired behaviour. Since all the dates were transformed into the equivalent day of the week of the first week of January 2007, the raw

dataset presented undesired jumps "back in time" when a trip took place on the night from a Sunday to a Monday. The following IDs presented that kind of exceptions and have been cleaned since they were a source of inconsistency: 7444, 12446, 7048, 13869, 272, 15901, 15710, 5743, 15566, 17577, 24593, and 1381. The *data/all.tsv* file provided in the Docker repository for this test case is already cleaned up and in Figure 4.5 we can see the visualisation of the dataset using QGIS and OpenStreetMap as a source for the reference map.

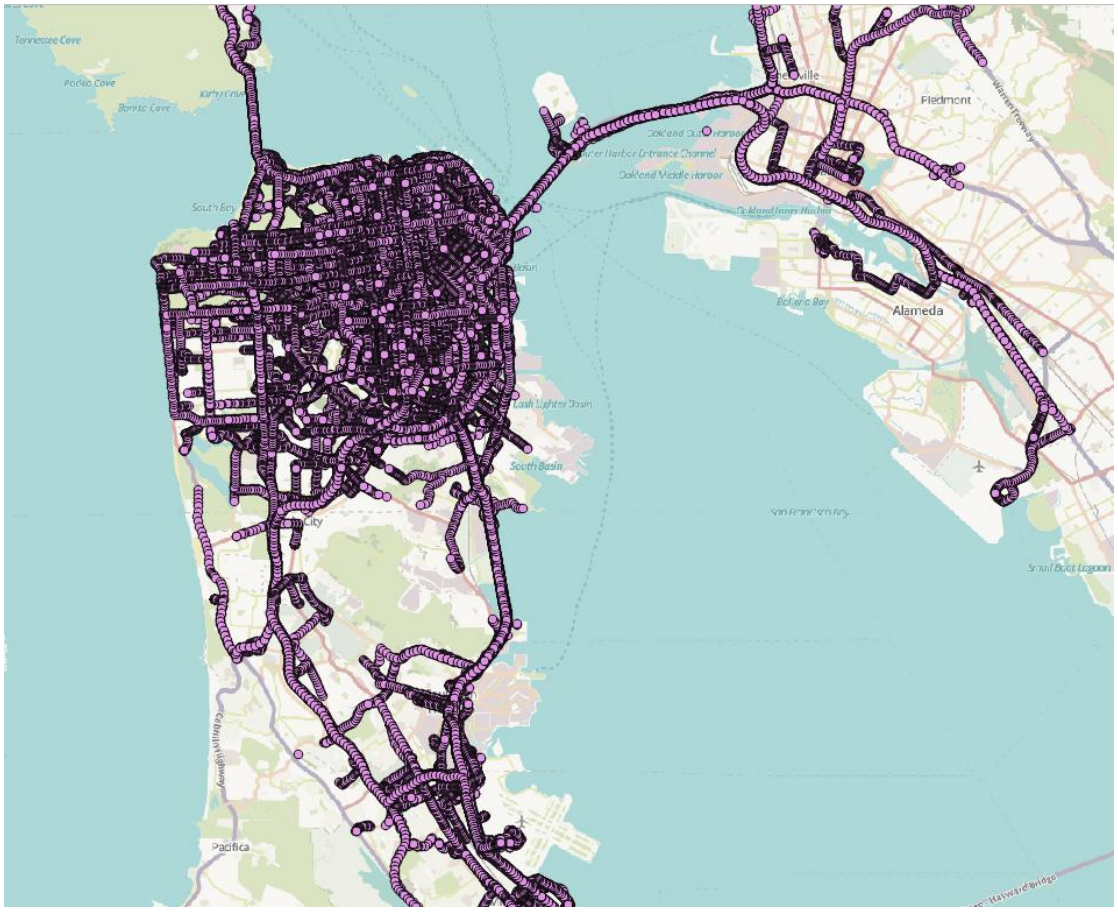


FIGURE 4.5: Coordinates of Uber car trips displayed using QGIS.

4.2.3 Implementing sequential solution in Python

The sequential solution takes a very straightforward approach. The Python script developed simply takes the file containing the dataset and loops over every line . When reading every line in order, the program can safely construct the geometry representation by concatenating the coordinates in every step. Since every GPS reading was sent every 4 seconds on average, the script estimates the total time by adding 4 seconds in every iteration. This calculation was simplified because the total trip time is not interesting for the geometric analysis we want to make on the dataset, but it is useful to keep the

field as a counter for later checks. The distance is calculated by aggregating the direct distance between each pair of contiguous coordinates by using simple mathematical conversions taking into account an average earth radius and later converting into meters . Every time the IDs of two consecutive entries are different, we can save all previously collected data in a list and start the data composition for the new trip.

```
trip_id  time  distance  geom
                          [...]
00008    312   3940.672  EPSG:4326;LINESTRING(37.773043 -122.422118, ...)
00009    176   1857.461  EPSG:4326;LINESTRING(37.790864 -122.4027688, ...)
00010    96    483.242   EPSG:4326;LINESTRING(37.782616 -122.400599, ...)
                          [...]
```

FIGURE 4.6: Fragment of processed output.

In Figure 4.6 we can observe a tab formatted fragment of the output of the reference implementation. Every entry of the output file represents each Uber car trip in the dataset with its ID, the estimation of the trip duration in seconds, the distance covered in meters and its geometry in **Extended Well Known Format** (**EWKT** [45]). In Figure 4.7 we can see a visualisation of the linear geometries calculated using this process in QGIS.

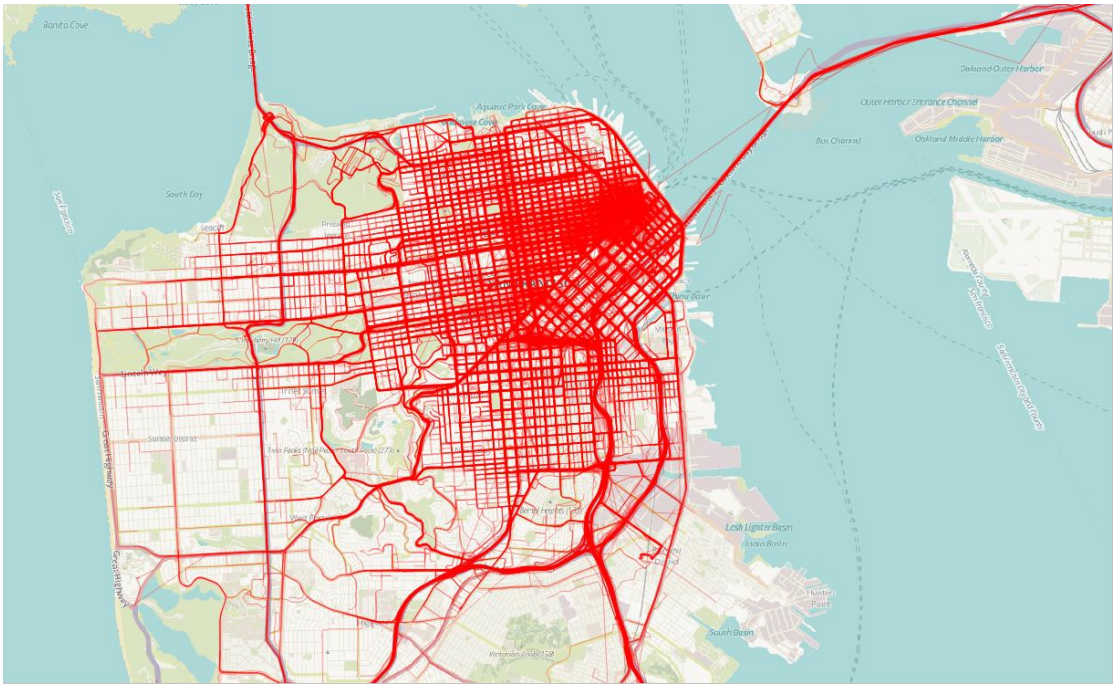


FIGURE 4.7: Reconstructed Uber car trips displayed as line geometries using QGIS.

Now we are ready to proceed with the MapReduce and Spark distributed solutions and to compare their results from those generated by the sequential Python script.

4.2.4 Implementing distributed solutions using MapReduce and Spark framework

The distributed solution follows the same implementation logic as the Python solution already described. In this case, the raw file is directly read from HDFS and the processing power is distributed among the simulated Hadoop workers.

When using the MapReduce framework, the program needs to be configured to specify where data will be read from, which map and reduce processes will be used and to where the output will be saved. All this set-up is specified in the main method of the program.

The execution of this solution is completed in 4 main steps:

1. The input file is splitted and distributed across the different workers.
2. Every worker maps each line of the partition they receive into a key-value pair. In this case the key is the trip ID and the value are the timestamp value, the latitude and the longitude.
3. All the key-value pairs are then sent to a grouping phase where they separated into groups by key. The value of each key will now consist of a list of values. In this step there is also a sorting made based on the key of each pair.
4. Finally, the list of values for each group is reduced by calculating the estimated trip time, the distance travelled and the geometry of the trip. After this step all the key-value pairs are written to an output file as tab-separated lines.

These steps are the same used when developing in the Spark framework. The main difference is found in the flexibility of coding thanks to Scala, which is the language in which Spark is implemented and the native language to execute Spark programs. One of the advantages of Scala is the ability to code mixing imperative and functional programming. This feature makes it easier to transition to Scala from Java environments, adding also the compatibility with existing Java libraries.

The biggest change is in the way the computation is initiated and the syntax of the implementations. Since Scala is a functional object oriented programming language, it allows the developer to use higher order functions to achieve the same map and reduce procedures provided as classes in the MapReduce framework.

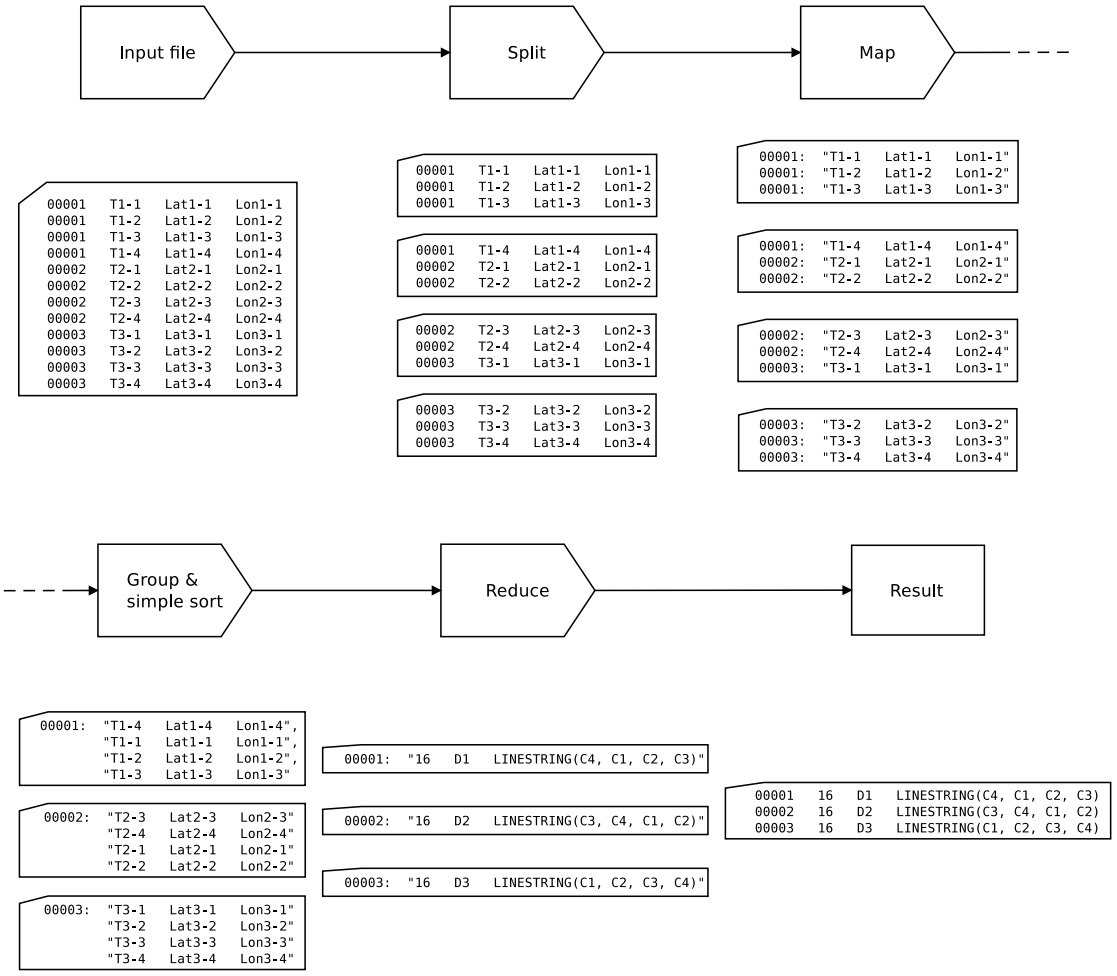


FIGURE 4.8: Distributed implementation using simple sorting execution overview. T represents time, D represents distance and C represents coordinate.

Figure 4.8 shows a graphical representation of the process in which we can detect an undesired effect of this implementation: the grouping stage can provide an unordered list of values to the reduce stage. This unordered list leads to incorrect results as the change of the order in which the coordinates are traversed change the geometry of the trip and also the distance covered. We can visually see the problem with incorrect coordinates order in Figure 4.9.

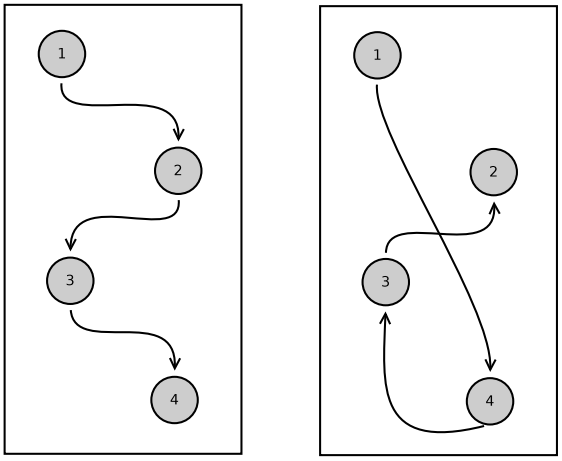


FIGURE 4.9: Ordering problem when calculating the geometry and total distance covered by a set of coordinates.

4.2.5 Fixing the incorrect ordering problem in the distributed implementations

The ordering problem detected in the previous implementation has a simple and efficient solution by changing the map and grouping stages:

- First, the map stage needs to be updated to produce a composite key which first part is the trip ID and its second part is the timestamp. The value part of the key is again the timestamp value, the latitude and the longitude.
- Finally, in the grouping phase, the new implementation indicates the framework to use a secondary sorting approach. In this case all the mapped values are first ordered by the first part of the key (the trip ID) to form the groups but then the groups are grouped in order by the second part of the key (the timestamp value). This way all the value lists are already in the correct order in an efficient manner, instead of delegating the reordering task to the reduce phase.

Both solutions for each framework have been kept in the project in order to allow later reproductions of the simple sorting problem.

4.2.6 Comparing implementations

Before implementing the web service, it is needed to decide which technology will be used for the back-end processing.

To visualise the errors made by the implementations without secondary sorting, a set of Hive tables have been created to read the results from HDFS. Figure 4.10 shows the SQL statements that let us know the number of trips with the same ID but different geometry. In the comment after each statement, the actual count calculated can be seen. We can check the fact that the simple sorting implementations have errors while the implementations with secondary sorting give the same results as the Python solution.

```
1  SELECT count(*) FROM python_output py, mapreduce_unordered_output muo
2  WHERE py.id = muo.id AND py.geom <> muo.geom; -- Count: 18540
3
4  SELECT count(*) FROM python_output py, scala_unordered_output suo
5  WHERE py.id = suo.id AND py.geom <> suo.geom; -- Count: 10519
6
7  SELECT count(*) FROM python_output py, mapreduce_ordered_output moo
8  WHERE py.id = moo.id AND py.geom <> moo.geom; -- Count: 0
9
10 SELECT count(*) FROM python_output py, scala_ordered_output soo
11 WHERE py.id = soo.id AND py.geom <> soo.geom; -- Count: 0
```

FIGURE 4.10: SQL statements to compute errors between distributed and sequential implementations.

In Figure 4.11 we can see the comparison of execution time for all the implementations. As expected, for this small dataset the sequential implementation using Python is the fastest. It can also be seen that Scala solutions have worst performance for this specific case, probably because the set-up overhead for the Spark programs was considerably slower (around 25 seconds).

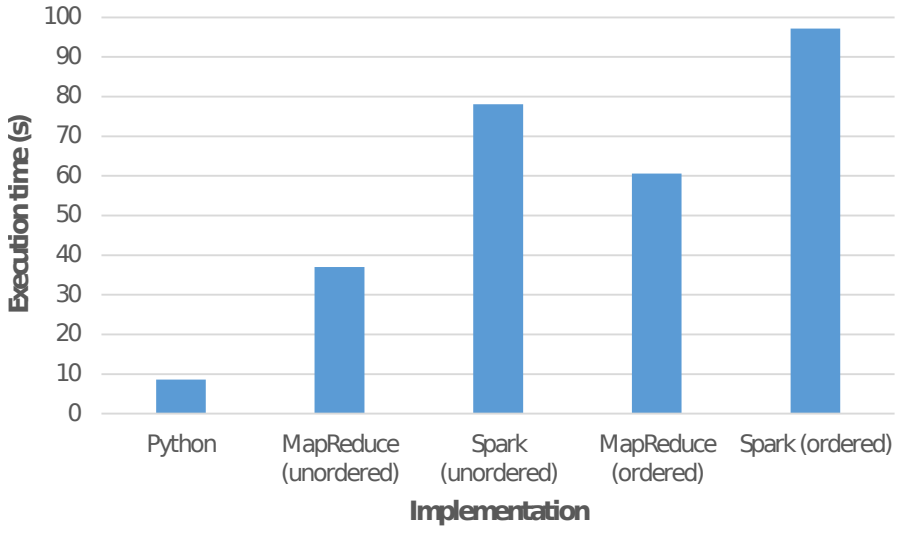


FIGURE 4.11: Execution time for Python, MapReduce and Spark implementations of trips reconstruction.

Despite its worse performance, Spark is the framework chosen to implement the last task of this use case because it offers an improvement in development comfort thanks to the Scala functional programming paradigm.

4.2.7 Developing a simple web service

In order to develop a web service to allow interactive processing of the parsed dataset, it was needed to find geoprocessing libraries that were compatible with Spark and with the Java Topology Suite (JTS, [38]), which provides a vast library of transformations and operations for georeferenced geometries. The most interesting libraries found were **GeoTrellis** [46], **Magellan** [47], **GeoSpark** [48] and **SpatialSpark** [49]. After reading the documentation available and trying the libraries, SpatialSpark was the chosen library to develop the service example:

1. GeoTrellis functionality was focused on working with raster data instead of vector data.
2. Magellan does not offer compatibility with JTS objects or EWKT format, but it only interfaces with **GeoJSON** format [50]. Since no libraries to convert from EWKT format to GeoJSON were found, that would have meant an important extra development.
3. GeoSpark offers compatibility with JTS objects but the library is limited to distance calculation and inclusion operations (i.e. the geometrical *contains* operation).
4. SpatialSpark offers compatibility with JTS objects and slightly bigger variety of operations such as distance calculation, closest neighbours, intersections and inclusions.

Using JTS and SpatialSpark as program dependencies now it was possible to develop a simple example to be executed remotely. In this case, the data to be processed by service is the reconstructed geometries of the Uber car trips. Therefore, the program was designed to accept as input two trip IDs, fetch them from HDFS and calculate the intersection from both trips using spatial operators implemented in SpatialSpark. In order to trigger a more complex execution, the service also allows only one trip ID to be specified. In that case, the program will calculate the intersection of the specified trip with the rest of the dataset.

It is important to note how JTS and SpatialSpark complement each other in the solution implemented. Using the function **BroadcastSpatialJoin** from SpatialSpark library, the

program is able to compute the trips that intersect with the target trip. However, the previous function does not provide the value of the intersection (either a point or multi-point geometry) for which we use the **intersection** function from JTS library between each pair of intersected trips calculated by **BroadcastSpatialJoin**.

From the web service point of view, the only piece missing is the HTTP entry point. After searching for remote Spark execution, the **Livy** project [51] was found. However, the project appeared to be unreliable and with little support because it is in its early stages of development and is constantly being updated.

Given the lack of options found inside the Hadoop/Spark ecosystem, the entry point was developed using **Apache server** and **PHP** with the **Slim** framework [52]. With the Slim framework and PHP, it was straightforward to implement a very minimal REST-like API to allow the execution the previously compiled Spark programs. The API provides two entry points, **/uber/trips** and **/uber/intersections**, which return the complete reconstructed trips dataset and the result of the requested trip intersections respectively.

4.2.8 Developing a QGIS plug-in to interact with the web service

Finally, a QGIS plug-in is developed to serve as a user interface in order to command the execution of the example service, retrieve data and load the results in the QGIS Desktop application.

For this purpose, a QGIS project configuration is provided where a layer with the base OpenStreetMaps map is already set.

In Figure 4.12 can be seen the result of the intersection between the trips with ID 15297 and 11862. In the right corner can also be seen the plug-in minimalistic interface: at the very top the user can introduce the server IP to where the requests will be sent, below there are buttons to trigger the retrieval of both the point dataset and the linear reconstruction, and finally the user can specify two IDs and trigger the intersection calculation.

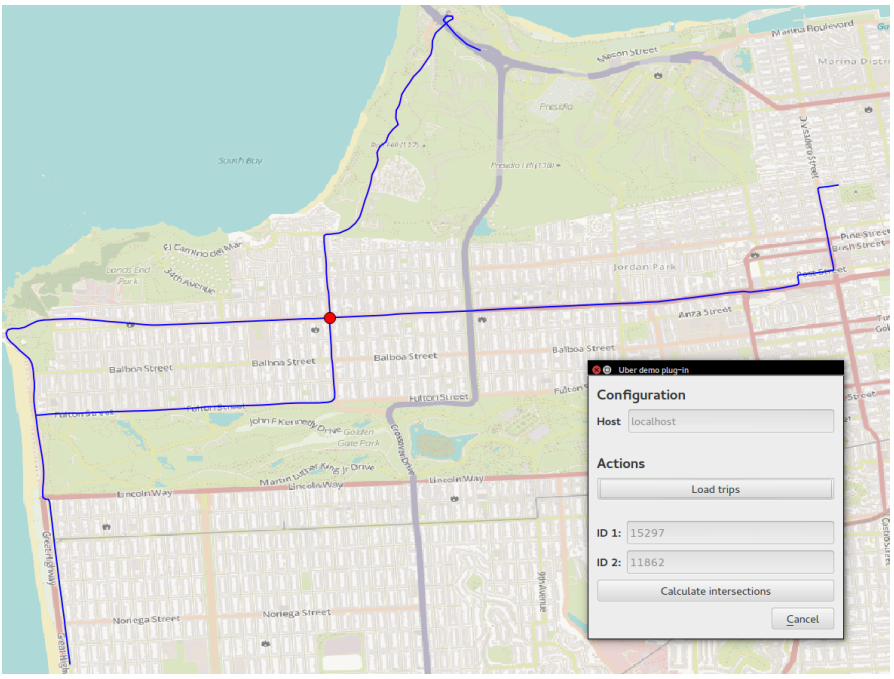


FIGURE 4.12: Result of the intersection between trips 15297 and 11862.

As it was explained in the previous section, it is also possible to efficiently compute the intersections between a specific trip and the rest of the dataset. In Figure 4.13 can be seen the result of the intersections with the trip with ID 15297.

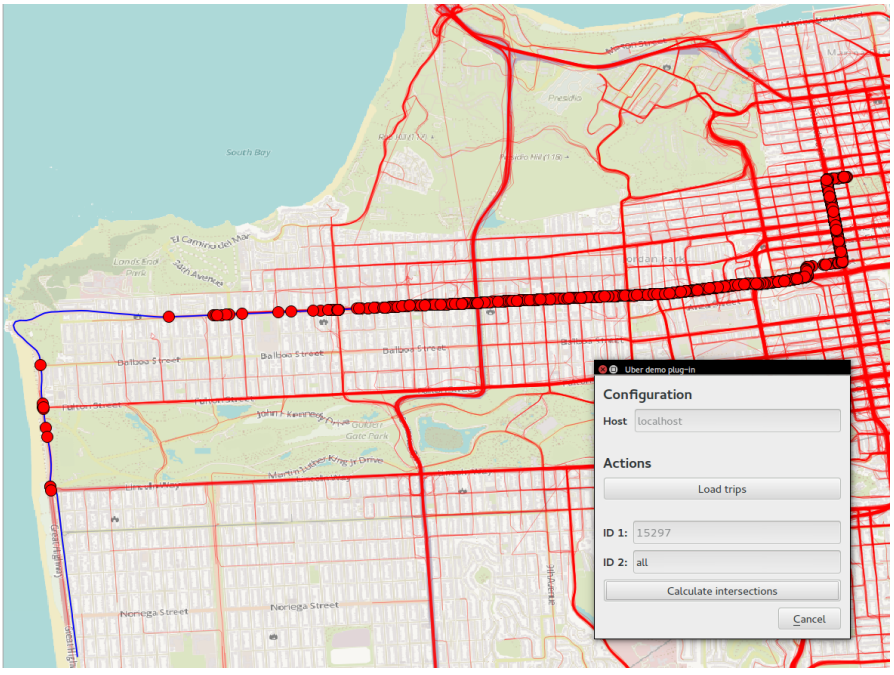


FIGURE 4.13: Result of the intersection between trip 15297 and the rest of the trips

4.3 Results analysis

In this use case, a complete stack for processing and serving data stored in HDFS has been developed. The processing has being iteratively improved from a sequential solution to a distributed solution, taking into account issues appearing when data is processed in a concurrent fashion.

By developing a REST-like interface to run the processes the result of the computations can be easily consumed in a user friendly interface and used by GIS technicians.

However, the response time of the REST service and plugin is quite high (from 30 seconds to 1 minute, depending on the amount of data to be represented on the map). While part of the issue is due to the amount of data that can be printed at the same time in the QGIS application, it is an interesting point to take into account to improve the architecture developed.

Chapter 5

Case 2: Handling data from acquisition to publication in an Open Data portal.

FIGURE 5.1: Port Authority of Las Palmas de Fran Canaria logo.

The **Automatic Identification System** (**AIS**) is a maritime communication system based on VHF radio signal broadcast. It was designed to serve as a complementary navigation system to avoid vessel collisions [53].

The objective of this use case is to develop a collecting, processing and publication chain for AIS related data. The dataset used is a fragment from an AIS data stream facilitated by the Port Authority of Las Palmas de Gran Canaria [54] saved during 2 weeks.

5.1 Analysis

AIVDM/AIVDO is the communication protocol for AIS and all its messages can be divided between class A and class B systems. Class A systems are compulsory and reserved for big international vessels, cargo ships and passenger ships with more than 12 passengers. On the other hand, class B systems are meant for recreational ships, other small vessels and inland stations. An example of a codified AIS message can be seen in Figure 5.2

```
!AIVDM,1,1,,A,33Emhr?P2cNqMQD@6<CVGgw@01h1,0*5D
```

FIGURE 5.2: AIS message codified.

Although it is hard to obtain information about the protocol, there are different pages that provide a decoding service to test small streams of codified AIS messages ([55]). Besides, a good resource of AIS protocol documentation and implementation is the AIVDM page under the GPSD open source project ([56]).

Using the AIVDM/AIVDO specification we can analyse the different AIS messages available. In the specification there are 27 different types of messages described from which this simulation only treats the following:

- Types 1 and 3, which report the position of class A vessels.
- Type 5, which reports static and voyage related data (e.g.: ship name and destination) for class A vessels.
- Type 8, which is binary encoded and also reports static and voyage data but for inland vessels.
- Type 18, which reports the position of class B vessels.
- Type 24, which reports static and voyage related data for class B vessels

FIGURE 5.3: AIS portable receiver.

Given the high frequency nature of the AIS messages (specially when vessels are moving [57]), it is interesting to provide a complete storage pipeline from which data could be accessed as soon as possible but also some processed data could be published to be used in third party applications and analysis. With that objective in mind, the tasks defined for this use case are the following:

1. Configure Docker files to create and link the system using containers(Orion, Cygnus, Hadoop, CKAN).
2. Integrate different technologies to capture AIS data and store it into HDFS.
3. Develop a Spark program to process the stored AIS data.
4. Publish the processed data to a CKAN Open Data portal.
5. Create a heat map program and visualise data using QGIS

5.2 Design and development

5.2.1 Configuring and installing server side technologies using Docker

This use case requires the combination of a slightly more complex stack of technologies. In this case, the back-end configuration (see Figure 5.4) consists on 5 Docker containers:

- A MongoDB Docker container to store context data. This container uses the official MongoDB version 3.2 image from Docker Hub.

- An Ubuntu based Docker container with the CKAN open data portal. This container will serve to publish data to be consumed by interested users and access to the portal through HTTP.
- A CentOS based container where the Hadoop platform is installed. In this container the data processed from the AIS stream will be stored using HDFS. A Spark and Apache service, similar to one deployed in the Uber use case, is provided to make remote processing requests. Finally, a data publishing process is also configured and executed in this container, which will manage and publish data to the CKAN container.
- A CentOS base container with the Cygnus data connector configured for this use case. This container is connected to both the Orion container and the Hadoop container as it is responsible to receive data form Orion and store it into HDFS.
- An Orion Context Broker container to manage context data and serve as the the entry point to the AIS stream storage processing. This container uses the official Orion version 1.1 image from Docker Hub. This container is connected the Cygnus container to persist data to the Hadoop container.

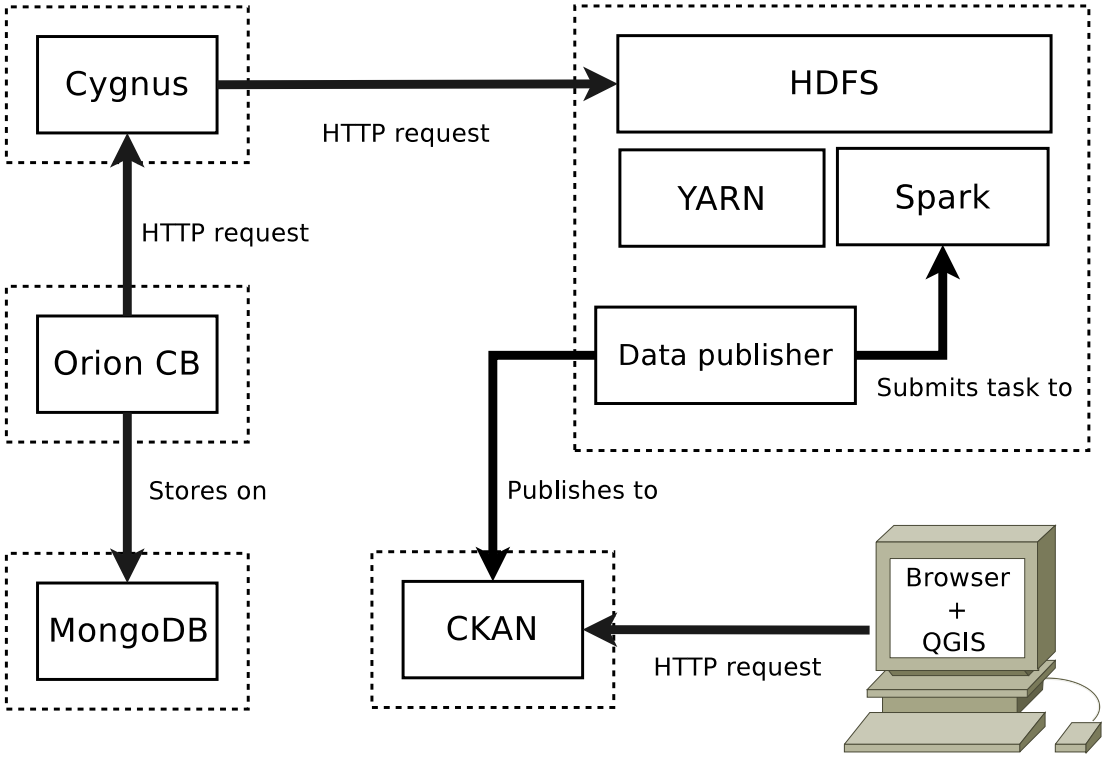


FIGURE 5.4: Diagram of the docker container configuration for case 2.

5.2.2 Capturing and storing AIS messages

The original AIS data stream shared by the Port Authority is a text stream sent over HTTP. However, for this project a portion of the stream was saved to a text file during 2 weeks and said file will be the data source.

The AIS decoding process [58] has been taken from the GPSD open source project and used as an external library. Two other small utility libraries have been developed to further process the AIS messages (**aisUtils.py**) and manage connections to Orion Context Broker (**orionUtils.py**).

The AIS decoder used does not provide support for Type 8 AIS messages, which can be used to broadcast environmental status such as weather metrics or sea level [59]. Since the stream captures some Type 8 messages, the AIS utilities library provides with a Type 8 decoder function which translates the decoded character string into the different message bit fields taking as a reference the Type 8 message specification in the AIVDM/AIVDO documentation.

Using these three utility libraries, the workflow for the data acquisition is very straightforward:

1. The script **subscribe_ais_entities.py** is run to register the subscriptions to the different messages that will be stored in Orion. The subscriptions are defined in a way that the state of the different type of messages is notified to the Cygnus deployed in another container whenever the data changes.

2. Once the subscriptions are created, the utility **parse_push_ais_local.py** is run. This script takes every line from the raw data file, convert them into JSON format and push them to the Orion Context Broker to make it available for real time applications. Every change in the notifications generates a notification that is sent to the Cygnus service.

3. The Cygnus service, which was already configured when deploying the container, receives the Orion notifications, converts them to a one line JSON string and saves them into the HDFS deployed via its web API. At that point, the messages can be processed inside the Hadoop platform to inspect them more deely.

At normal speed, that is without forcing delays between the messages read, the complete file is processed in about 60 hours. Although the Dockerfiles provided allow to repeat the process from scratch, it has been decided to also provide the processed data exported from HDFS to the compressed file **/data/ais.tar.gz** to give the opportunity to skip this time consuming step.

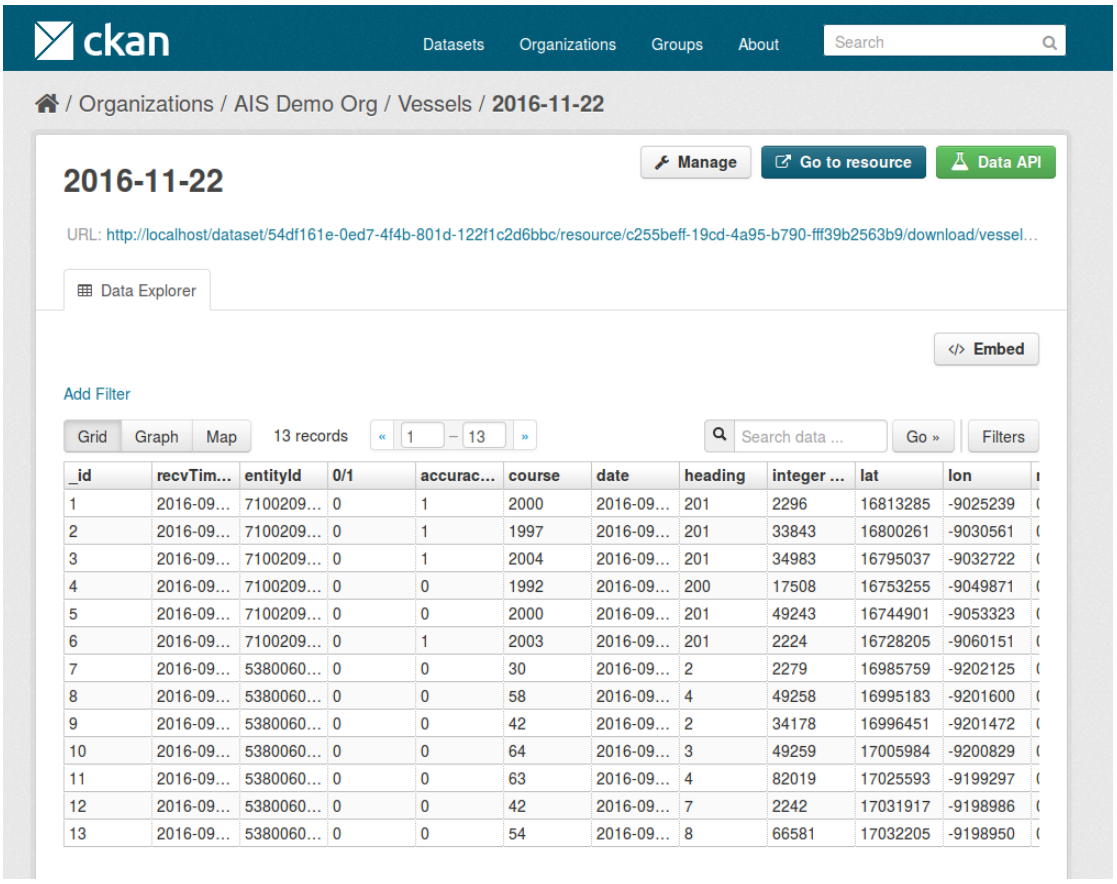
5.2.3 Merging AIS messages and publishing to CKAN Open Data portal

The previous set-up takes charge of the persistence of AIS data stream inside the Hadoop instance. While having the data stored and accessible through HDFS is convenient to perform batch processing on the dataset, it is interesting to bring access to third parties in a common format like CVS.

In order to provide this interface, a script is run at consistent intervals of time (every 5 minutes) to join the different types of messages and publish the combined records to the CKAN instance. Below are described the steps taken to achieve it:

1. Data is originally stored in a folder called *input*.
2. Every time the scripted program is launched, the current messages stored message are moved to a working folder named *processing*.
3. The script then iterates over different partitions of the documents read and pairs them. In this case, the messages merged are those indicating position information and the vessel static description by searching by its ID.
4. Two special datasets are created with the IDs 1111111 and 999999 as they are the result of corrupted IDs and non identified ships (since it is not compulsory to broadcast the ID for small vessels).
5. With both pieces of information being merged, we can publish the resulting JSON document to the Demo dataset created during the machine compilation. Each resource is actually uploaded as a CSV file for the current day of the year.
6. If it is the first time data is being pushed in a given day then a new resource is made with the current day of the year. The identification is stored in a *metadata* folder inside HDFS to know to which resource the data must be updated.
7. After the messages are processed, they are moved to a *processed*, where they will remain archived for future batch processing.

Figure 5.5 shows a daily published dataset being accessible through the Open Data Portal, ready to be consumed by third party software and users interested in the data.



_id	recvTim...	entityId	0/1	accurac...	course	date	heading	integer ...	lat	lon
1	2016-09...	7100209...	0	1	2000	2016-09...	201	2296	16813285	-9025239
2	2016-09...	7100209...	0	1	1997	2016-09...	201	33843	16800261	-9030561
3	2016-09...	7100209...	0	1	2004	2016-09...	201	34983	16795037	-9032722
4	2016-09...	7100209...	0	0	1992	2016-09...	200	17508	16753255	-9049871
5	2016-09...	7100209...	0	0	2000	2016-09...	201	49243	16744901	-9053323
6	2016-09...	7100209...	0	1	2003	2016-09...	201	2224	16728205	-9060151
7	2016-09...	5380060...	0	0	30	2016-09...	2	2279	16985759	-9202125
8	2016-09...	5380060...	0	0	58	2016-09...	4	49258	16995183	-9201600
9	2016-09...	5380060...	0	0	42	2016-09...	2	34178	16996451	-9201472
10	2016-09...	5380060...	0	0	64	2016-09...	3	49259	17005984	-9200829
11	2016-09...	5380060...	0	0	63	2016-09...	4	82019	17025593	-9199297
12	2016-09...	5380060...	0	0	42	2016-09...	7	2242	17031917	-9198986
13	2016-09...	5380060...	0	0	54	2016-09...	8	66581	17032205	-9198950

FIGURE 5.5: Vessel information served using CKAN Open Data.

5.2.4 Visualising common vessel positions displaying a heatmap in QGIS

It is also important to validate the data registered by visualising most of the positions at which the messages are registered. For this purpose the AIS messages are analysed by observing a heatmap of the surroundings of the Port of Las Palmas de Gran Canaria.

In order to achieve this result, a grid of 10x10 rectangles was created and uploaded into HDFS to serve as input for the the heatmap calculation. Once the grid is accessible as an HDFS file, it can be crossed with the position messages for type A and B vessels using the SpatialSpark *contains* spatial operator. This operator returns all the ships that are contained in each grid cell, which allows us to return a matrix of cell and ship count.This matrix is afterwards loaded using QGIS and a base map to have a visual representation of the distribution of AIS location messages.

In Figure 5.6 can be seen the heatmap obtained for the processed AIS messages in this use case. It can be clearly seen that the highest number of messages are registered in the shore and inside the dams of the port (red and orange tones), which is expected as those

are the places where many vessels stay docked for long periods of time. The heatmap also shows that the following highest count cells (light green coloured) are spread downwards, towards the south of the island which indicates that the vessels registered during that period of time were mostly traversing that route.

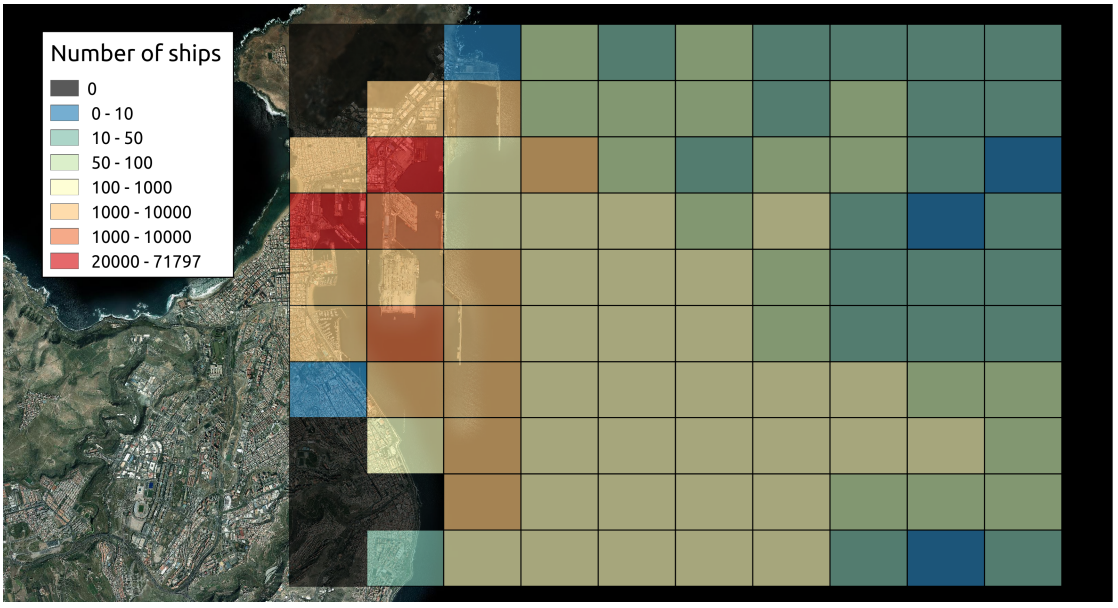


FIGURE 5.6: Heat map representing the number of vessels inside the area near the port.

5.3 Results analysis

In the presented use case, a single sensor pusher has been simulated. However, the system could be later extended by feeding the system with sensors pushing directly AIS information from different listening stations instead of having a centralized structure. Such a set-up would be closer to the concept of **Internet of Things** (**IoT**), since it can be quite complicated to have the vessels themselves sending the information directly to the processing system.

The Data Portal adds an important publication interface to make accessible data valuable for research and also to increase the transparency of the management of the port.

The biggest advantage of the architecture deployed is that it is now trivial to keep adding different data stream pushers, data generators or sensors to enrich the data being collected. When the system stores a higher variety of related data types (like weather and traffic measures around the port, to name a few) it can be updated by developing more useful and elaborate analysis processes which generating valuable information for the Port Authorities.

Chapter 6

Case 3: Integration of a water management system with MongoDB

This third and final use case is based on a real request from a local water management company for a history management subsystem in its custom water management GIS.

Therefore, the objective in this project section is to experiment with the generation and storage of said history database and the implementation of a **Point-In-Time Recovery** (**PITR**) functionality to allow the system to be recovered to a previous valid state.

6.1 Analysis

One of the most important requisites in this system is to provide to the company digitizers the ability to work on an **editing workspace** database while all the changes get tracked on a **history** database. This set-up allows to implement a procedure to recover the digitised network to any valid previous state (the PITR process) .

In order to simplify the test case development and the comparison between technologies and implementations, the code developed for this document is based on a simplified entity with linear geometry, which represents a network pipe. In the complete system, there exists many other entities based on the company customised model with different geometries (points, lines and polygons representing nodes, pipes and working places, for example) and spatial relationships between them (Figure 6.1). However, all their relationships with the history database are reduced to the same table to table operations presented in this simplified case study.

FIGURE 6.1: Visualising the water management network in QGIS.

The technologies chosen to implement the proposed history management are **PostgreSQL** with its geographic extension **PostGIS**, and MongoDB. The reason to directly compare the performance of both databases is because MongoDB is a common choice for distributed and redundant databases, due to its built-in partitioning and balancing solutions; and PostgreSQL paired with PostGIS is a very useful and powerful technology to develop a GIS.

To successfully cover this case, the following steps were followed:

1. Design different solutions for the history management and PITR functionality.
2. Configure Docker files to create containers for the different solutions proposed.
3. Implement the solutions designed.
4. Compare results given by each implementation.

The solutions designed for the described system are implemented using two **Docker** containers to keep separated the *editing workspace* and *history* space. The technologies to be used for the three solutions thought are:

1. The first solution uses **PostgreSQL** for both the *editing workspace* and the *history* database using the **PostgreSQL Foreign Data Wrapper** (**FDW**).
2. The second solution uses **MongoDB** for the *history* database, connecting the **PostgreSQL** database to the MongoDB one using the **MongoDB FDW**.

3. The last solution also uses **MongoDB** to store the *history* database, but in this case the connection is done via **direct connection** made in a custom database function.

In order to perform the comparison between all three solutions, two different measures will be collected when executing them. The first measure to compare is the time taken to insert different network sizes. The second measure is the total time required to recover a past state of the network.

6.2 Design and development

First of all, the implementation started by defining the sample entity to be used. This entity will represent a water pipe since it is one of the most important elements of a water network. The fields of the entity table would then be:

- **id**: An auto incremented integer value to reference the register.
- **geom**: The geometry field indicating the type of geometry and its shape. In this case the type of geometry is linear.
- **digitization_date**: The date at which the register was first introduced.
- **update_date**: The date at which the registry was last updated.
- **comments**: A text type field to allow the user to introduce general comments about the register.

In the following sections the different implemented cases and the process of automating and executing the comparison experiments are explained.

6.2.1 History management and recovery mechanism

In the previous section we could see the structure of the entity table. In order to keep track of changes and start managing the history of the registers of the entity table, the system needs a **history table**. In this history table we have the same fields of the original entity table and the following extra fields:

- **register_date**: The date at which the action was registered.

- **operation_type**: The type of change made to the register. The possible values correspond with the common SQL operations *INSERT*, *UPDATE* and *DELETE*.
- **validation_range**: This field indicates the date range where the data for the current register is valid.

Once the fields of both tables were decided, the process to log the changes in the entity was defined as the following steps:

1. A digitizer makes a change in the *editing workspace* database.
2. The change triggers an INSERT operation in the *history* database where the entity field values are copied and the extra fields are filled accordingly:
   - The **register_date** is set to the current time of the operation.
   - The **operation_type** is set to the INSERT, UPDATE or DELETE, depending on which was the original operation on the editing table.
   - The **validation_range** is set to start at **register_date** and end on infinity.
3. If the operation on the original table was an UPDATE or a DELETE, the history process searches for the previous history entry of the register and updates its **validation_range** field to set the upper bound to the new **register_date**.

The third step of the log process is needed to clearly specify the date range when the history entry is relevant. This field is then used in the recovery process, which can be summed up in the following steps:

1. The user starts the PITR process by providing the date at which the system should recover.
2. The recovery process searches for all the history entries which **validation_range** contains the input date.
3. For every entry recovered from the history table, the process searches for it in the editing table by its **id**. From this search there are 2 possible recovery actions to perform on the editing table:
   - If the recovered register is not found, it means that the register was deleted at some point and the process triggers an INSERT operation.
   - If the recovered register is found, then the operation triggered is an UPDATE with the values provided

4. Finally, for all the registers in the editing table whose **id** has not been found in the recovered entities, the recovery process triggers a DELETE operation, because they were created after the point in time to be recovered.

It is important to take into account that the PITR process triggers ordinary INSERT, UPDATE and DELETE operations. Therefore, all recovery operations get registered in the history table, which allows the user to easily restore the state previous to the recovery action by executing the recovery process again with the desired date.

6.2.2 PostgreSQL editing workspace with PostgreSQL history database

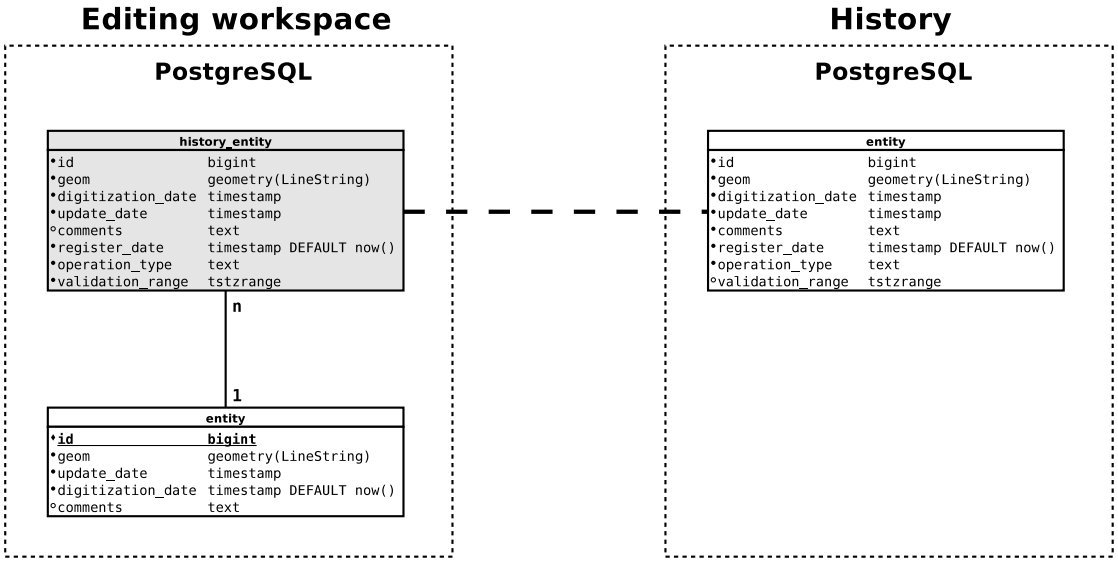


FIGURE 6.2: Implementation diagram using PostgreSQL for both editing workspace and history database using a foreign table as interface.

Figure 6.2 shows the representation of the history management using separate Docker containers for editing and history databases, both using PostgreSQL as the database engine.

In order to allow the communicating between both databases, the editing workspace database has installed a PostgreSQL FDW. With this FDW we can create a foreign table (named **history_entity**) pointing to the log table in the history database. This way all the SQL operations made on the foreign table get transmitted and executed in the history database remotely and synchronously.

The logging process described in Section 6.2.1 is implemented by defining a trigger function over the editing entity table written using **PL/pgSQL**. The recovery process is also implemented by a **PL/pgSQL** function named **recover_entity** which accepts as input

the date to recover the entity table to. With this approach, the logic for logging and recovering past states is contained within the system.

6.2.3 PostgreSQL editing workspace with MongoDB history database via FDW

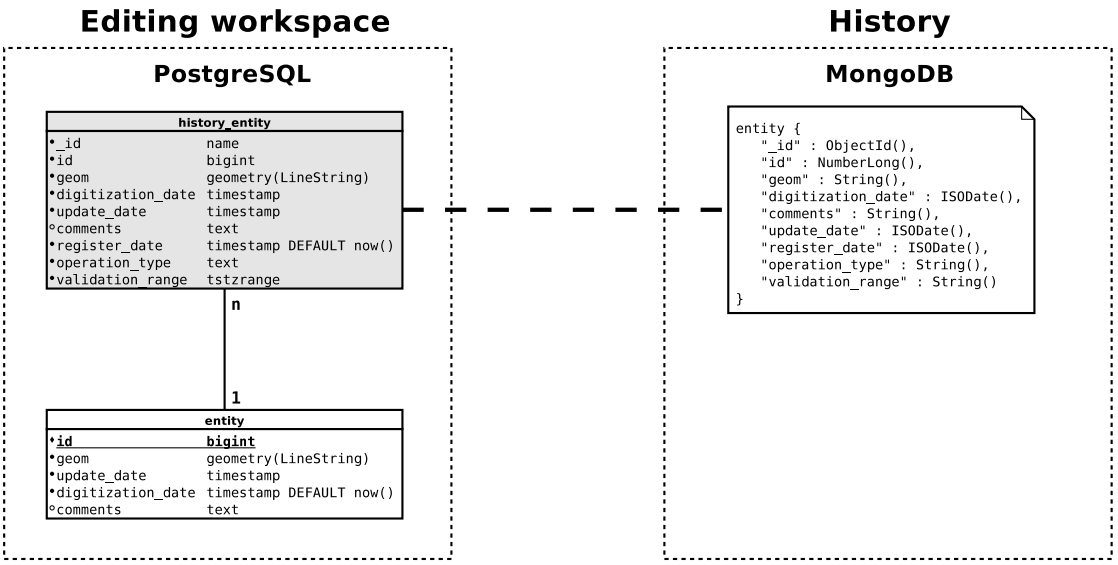


FIGURE 6.3: Implementation diagram using PostgreSQL for editing workspace and MongoDB for history database using a foreign table as interface.

In Figure 6.2 can be seen the implementation of the history system but now using MongoDB for the history database. In this approach, the approach of using the FDW techonology was also chosen by installing in the editing workspace the MongoDB FDW provided by EnterpriseDB.

The most important change compared to the PostgreSQL to PostgreSQL implementation is that MongoDB does not provide the option to write triggers for changes in collections. In the PostgreSQL to PostgreSQL, the responsibility to update the validation range for every register changed relied on the history database by triggering the update function after each operation on the history table. This is worked around in this implementation by registering the range update trigger function directly on the foreign table.

Another important change is the addition of the extra **_id** column in the foreign table. This change is a compulsory requisite by the MongoDB FDW as the fields in the foreign table must match the keys of the target MongoDB document and the **_id** attribute, which is automatically generated in MongoDB documents in order to make them uniquely identifiable.

The implementation of the rest of the functions and triggers is almost identical to the implementation used in the PostgreSQL to PostgreSQL system, only adding trivial adaptations to take into account the compulsory **_id** field.

6.2.4 PostgreSQL editing workspace with MongoDB history database via direct connection

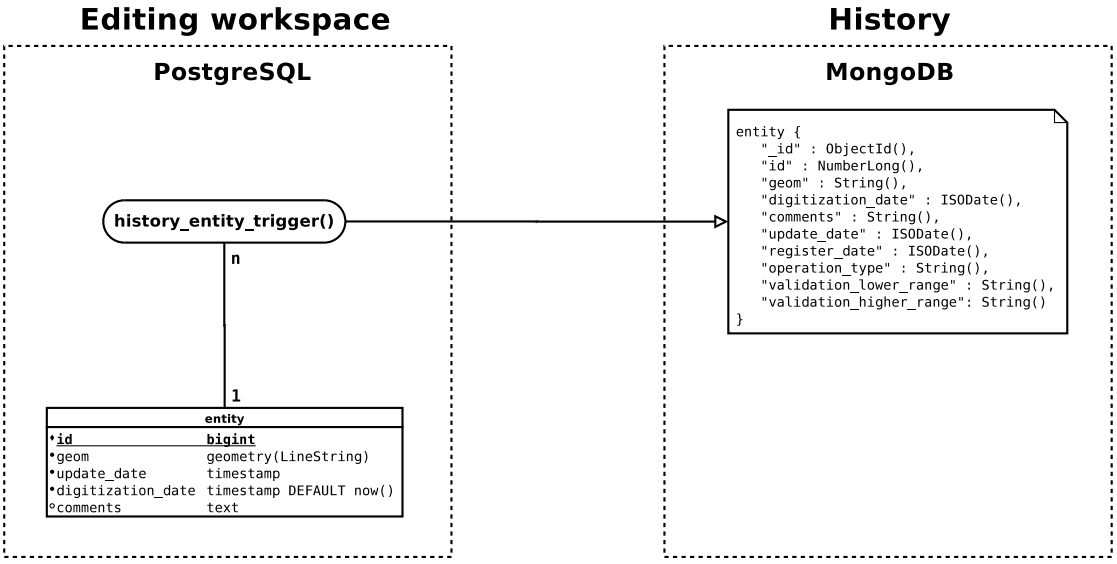


FIGURE 6.4: Implementation diagram using PostgreSQL for editing workspace and MongoDB for history database using a stored procedure as interface.

The third and last implementation made builds the history system using again PostgreSQL for the editing database and MongoDB for the history database, as can be seen in Figure 6.4.

However, this time the connection is managed using only stored procedures instead of interfacing via FDW and foreign tables. In order to achieve these direct connections through code, the technologies used were the **PL/Python** language and the **PyMongo** Python module [60].

The PL/Python functions implemented are responsible for inserting the changes made in the history document inside MongoDB (**history_entity_trigger** function) and to retrieve all the document entries valid at the given point in time (**get_history_entities** function).

If we compare the documents stored in Figure 6.3 and Figure 6.4 we can notice that the FDW implementation stores the validation range as a single string while the direct connection approach stores the lower bound and the upper bound in separated strings. The reason for the separation is to allow the use of the native filtering functions provided

by MongoDB, which could not be made over the complete range expression due to its lack of range operations unlike PostgreSQL [61].

Finally, an extra PL/Python function is needed to set-up the connection to MongoDB and to load all the necessary Python modules, the **_meta** function. The reason to use this approach is that, by default, PL/Python functions do not share any resources and it heavily punishes the performance of the load of libraries and connection set-ups, which has to be done for all the function calls. By executing the **_meta** function at the beginning of the PostgreSQL session, we can simulate the well-known sharing of loaded libraries in a normal program execution.

6.2.5 Comparison experiments

In order to make the tests isolated and easily reproducible, **Docker containers** were extensively used. For this purpose, a simple Python module to manage the creation and deletion of testing Docker containers was made.

For each system implementation the script creates an output file where the total number number of history registers, the total time to generate the whole history load and the time to recover the first network introduced is saved as a tab separated file.

Each implementation inserts the same **n registers**, makes a total of **X** maximum updates and increments the number of updates to be made on each iteration by **d**. At each iteration, a new container is created, after the operations described are made and their execution time is correctly stored in the output file. Once the statistics are saved, the container is destroyed to leave the same amount of resources for the containers created in future iterations.

The test script described was run twice to compare between lower and higher loads in the three different implementations:

- The lower loads graphs were generated by inserting 100 original entities and then incrementing the number of updates done by 10 during 10 iterations. The smallest history database generated stored 1100 registers and the biggest stored 10100 registers.
- The higher load graphs were generated by inserting the same 100 original entities but incrementing the number of updates by 100 during 10 iterations. The smallest history database generated stored 10100 registers and the biggest stored 100100 registers.

6.3 Results analysis

First of all, we analyse the insert performance based on the number of entities stored in the *history database*. Figure 6.5 shows that when the test generates from 1000 to 10000 updated entities in the history database, the implementation using MongoDB FDW **quickly diverges**. In the other two implementations we can observe a more **controlled growth** and, also, that up to 10000 history entities the PostgreSQL FDW implementation is faster than the direct MongoDB connection implementation. What is more, we can start distinguishing a **linear growth** for the implementation using the MongoDB direct connection and a **exponential growth** for the implementation using PostgreSQL FDW.

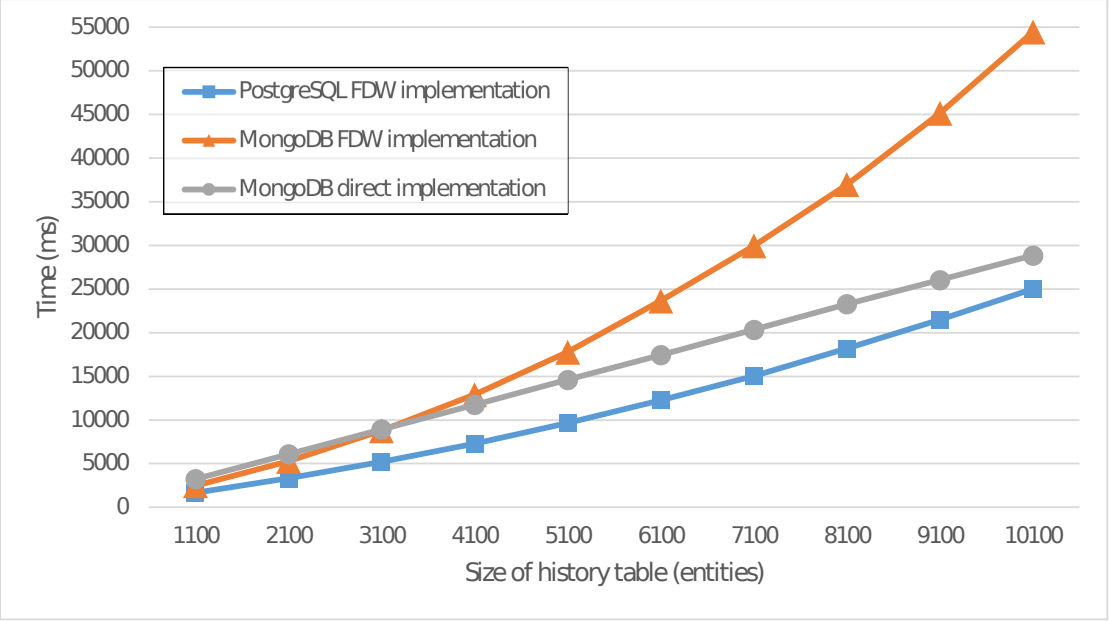


FIGURE 6.5: Execution time to generate a lower load on the history subsystem of the water management GIS.

On the other hand, Figure 6.6 shows that when the test generates from 10000 to 100000 updated entities in the history database, the implementation using the direct MongoDB connection displays a very **slow growth** while the other implementations are very noticeably less efficient.

If we now focus on the performance results for past states recovery, we can see in Figure 6.7 and Figure 6.8 that the MongoDB FDW implementation **diverges very quickly again** and that the MongoDB direct implementation has a very slow growth, which compared with the other graphs almost appears like a **constant line**. In the case of the PostgreSQL FDW, it is only more efficient than the direct MongoDB implementation for loads lower than 8000 history entities.

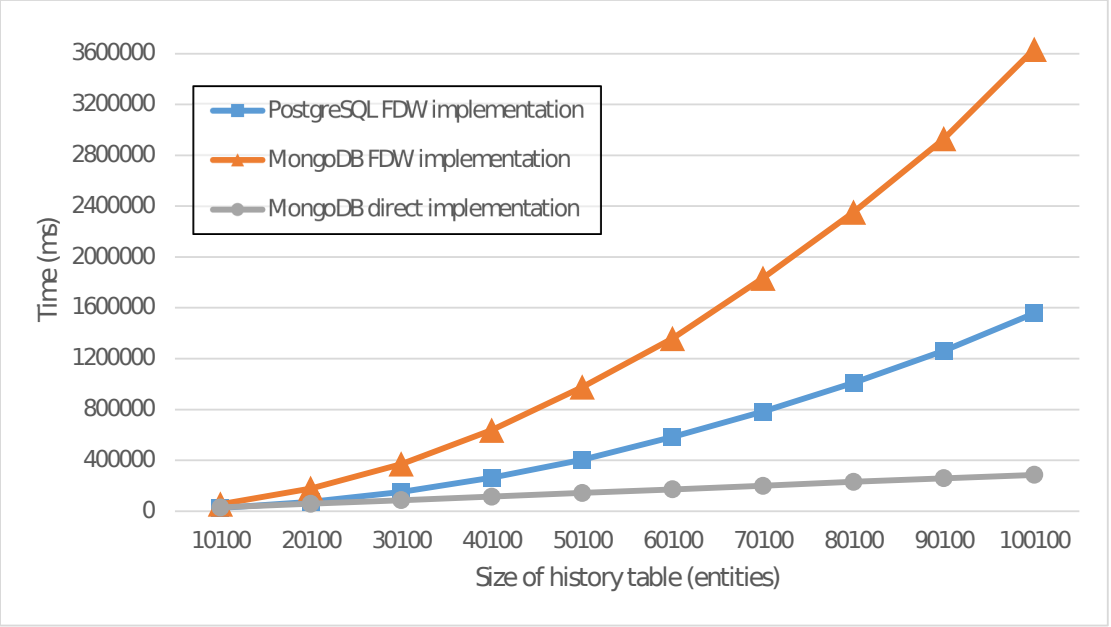


FIGURE 6.6: Execution time to generate a higher load on the history subsystem of the water management GIS.

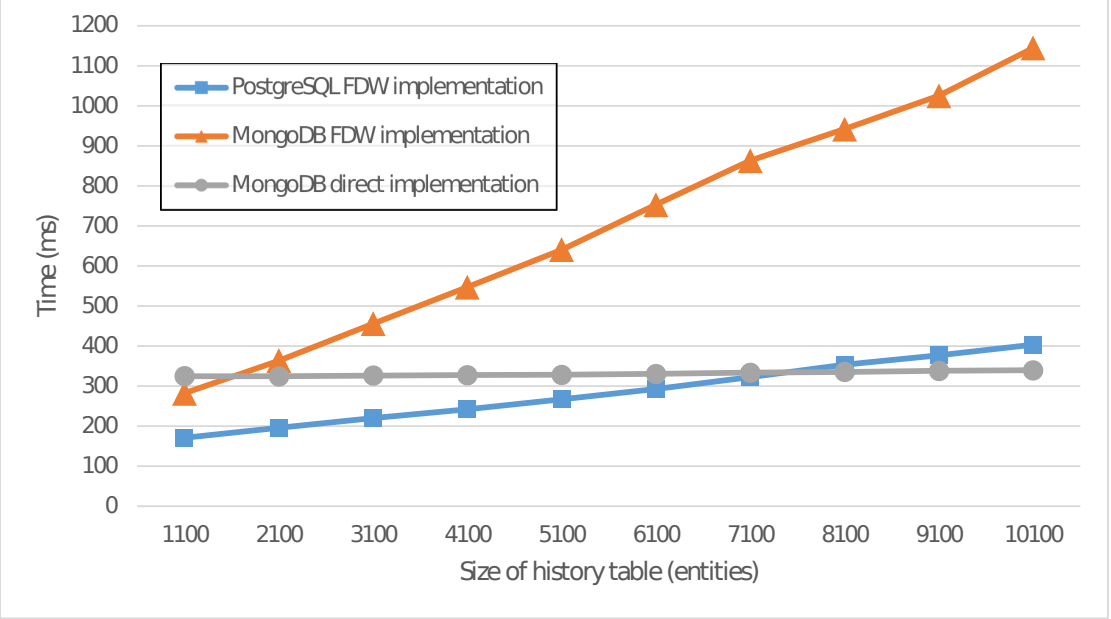


FIGURE 6.7: Execution time to recover the first insert from a lower load on the history subsystem of the water management GIS.

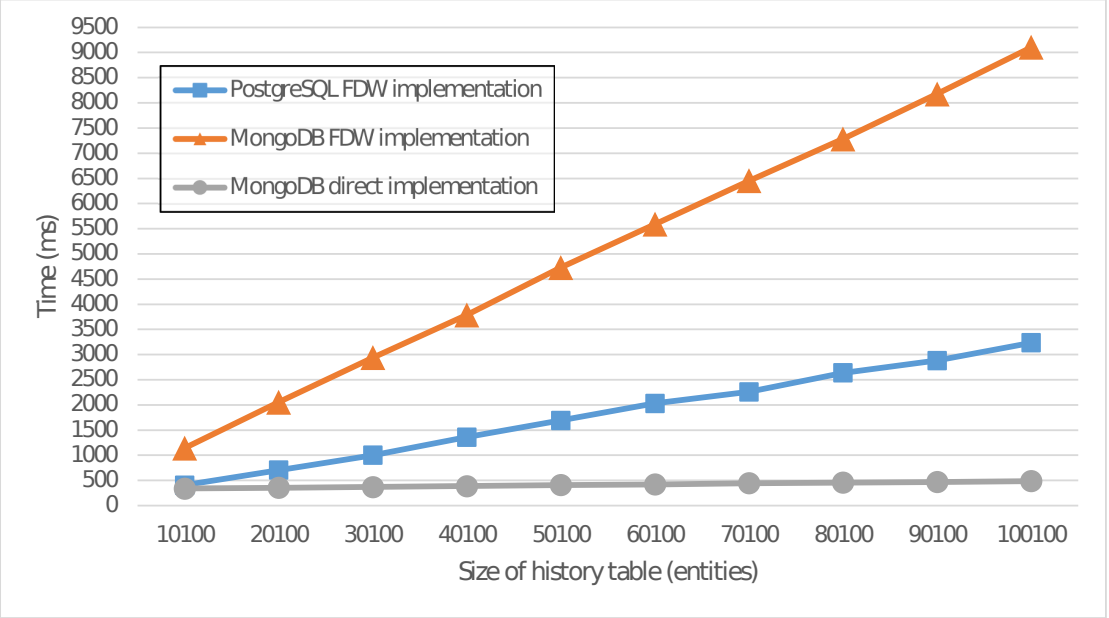


FIGURE 6.8: Execution time to recover the first insert from a higher load on the history subsystem of the water management GIS.

The reason to this noticeable difference relies on the way FDW are implemented. When a query is made over a foreign table, data must be sent from origin to the destination database, and then the query is executed locally. However, in the customised direct connections made, the queries are executed remotely and then their results are brought to the destination database.

Despite providing a much higher efficiency both in terms of history entries generation and point-in-time recovery, the direct MongoDB connection implementation does not include control over the database transaction. This means that if the update operation fails in the middle of a network change, the history database does not revert the intermediate changes to the correct state. This feature is provided by the PostgreSQL FDW implementation and is also missing from the MongoDB FDW implementation given the different approaches taken by the PostgreSQL and MongoDB engines regarding data transactions.

Finally, it is interesting to note the usefulness of using Docker containers to develop and execute the presented tests. This tool has simplified the process for collecting the relevant information and guaranteeing the same starting conditions for each separate test.

Chapter 7

Future work

Although during the development of this project many different technologies and approaches have been used, there are still many lines of work that could be worked in the future.

The market of Big Data technologies is dominated by the Hadoop ecosystem. It can be interesting to research about Hadoop alternatives and perform a comparative between them. This could not only bring insight about different distributed models, but also put on the spotlight potential new tools to combine with data processing architectures like the ones worked on this project.

Regarding the first use case (Uber cars GPS processing), the architecture developed is insufficient for real world application since its response time is too high to be accepted as it is. This brings the opportunity to work on new approaches to decrease user waiting time, probably by adding intermediate services to cache results. It can also be interesting to expand on geoprocessing libraries compatible with MapReduce, as Spark was focused in order to not broaden the topic too much.

The second use case (AIS data stream processing and publication) already presents a very flexible architecture. An interesting point of improvement could be adding more data streams, specially from sensors, which would present different challenges to the simulated data pusher created. It can be also worth trying other means and alternatives to CKAN for data publishing.

Finally, for the third case (history management for a water management GIS) the most interesting path to follow would be to find a development mechanism to be able to revert unfinished operations in a direct MongoDB connection, and analyses the trade off with the functions implemented currently.

Chapter 8

Conclusions

The presented project has served as a valuable introduction to many new topics. The most difficult aspect to tackle has been dealing with a huge field like Big Data, which is in constant and active development.

However, the selection of tools and different use cases to cover the field has helped on studying and comprehend a varied set of related skills, which are now a solid basis to keep on specialising.

It is also worth noting that the concepts and methodologies described have been successfully applied to research and development projects of the University of Las Palmas de Gran Canaria [62–64]. The new paths explored can feedback the original projects, which inspired the present one, to add further value.

Appendix A

Comparing non relational databases

In this appendix, a collection of tables comparing several non relational databases is presented to be used as a concise reference. The content of this comparison tables was chosen by examining the most common attributes provided by the content curation site DB-Engines [65]. The information was double checked with the respective databases official documentation.

The criteria used to compare the databases is detailed in the following subsection. The concepts described will serve to have a general idea on the different possibilities that non relational databases provide.

The databases to be compared are: **MongoDB**[12], **CouchDB**[66], **DynamoDB**[67], **HBase**[68], **Cassandra**[69], **Accumulo**[70], **Redis**[71], **Riak**[72], **Neo4j**[73], **OrientDB**[74], **Aerospike**[75] and **Hypertable**[76].

A.1 Comparison criteria

A.1.1 Model

Document store

The document stores or document-oriented database management systems organize data in a schema-free fashion. The records stored are named documents and they compile a set of attributes or columns. Every attribute can have a variable number of values (the case of an array). What is important is that documents of the same category might have a different number of attributes, attributes can be of different types and values are allowed to be documents themselves, in the case of nested documents. All these features compose the definition of a schema-free data management.

Key-Value store

The key-value stores database management systems only store tuples with two fields named key and value. In these systems, a value can only be retrieved when the related key is known.

Wide Column store

The wide column or extensible record stores database management systems also organize data in a schema-free fashion. In this case, their implementation consists on records, the columns, with variable number of entries, normally being a very large set. They can be seen as a two dimensional key-value stores for this reason.

Graph Oriented

The graph-oriented stores database management systems hold data as nodes and edges, being the edges the relationships between the nodes. This structure make easier the calculation of properties of the graph like the number of hops between nodes.

A.1.2 Features

Data Storage

Volatile memory: The database is optimised to make use of RAM memory for storage.

File system: The database persists data to the operating system file system.

SSD: The database is optimised to work Solid States Disks instead of spinning disk.

HDFS: The database is built on top of the Hadoop Distributed File System and persists the data on it.

Bitcask: An Erlang application for key/value storage on disk. It is based on log-structured file systems.

LevelDB: A Google project for key/value storage on disk. It is based on log files and sorted by key tables.

Query language

JavaScript: The database employs a query language which syntax is based on JavaScript expressions.

API Calls: The database does not provide a concrete syntax to query data. Instead it provides a query API which is implemented in several programming languages.

HBase Shell commands: The shell program for HBase provides with custom commands to do basic queries.

CQL: The Cassandra Query Language is an inspired SQL variation for the Apache Cassandra project.

Lucene: The database employs a query language which syntax is based on Lucene expressions.

Cypher: The database provides Cypher Query Language to make data queries. It is inspired in the SQL syntax but is oriented towards graph representation.

Gremlin: The database provides Gremlin Query language to make data queries. It is oriented towards graph representation.

SQL: The database implements a variant of SQL for its implementation language.

Lua UDF: The database provides querying by User Defined Function coded in the Lua programming language.

HQL: The Hypertable Query Language is an inspired SQL variation for the Hypertable project.

Protocol

Which communication protocols does the database system use?

Conditional entry updates

Does the database allow natively to update data based on conditional queries? (i.e: set column a to true when column b is greater than 10)

Implementation language

Programming language in which the database is implemented.

A.1.3 Integrity

Integrity model

ACID: It stands for the following set of properties; Atomicity (failed transactions are undone safely), Consistency (transactions take the database from a valid state to another valid state), Isolation (concurrent transactions provide the same result as if they were run sequentially) and Durability (once a transaction is finished, its result will be permanent no matter what happens afterwards).

BASE: It stands for the following set of properties; Basically Available (the system always replies, even if the reply indicates data could not be retrieved due to issues or updates), Soft-state (the system might be changing even if no commands are being issued, as it might be converging into a consistent state

) and Eventual consistency (the system will converge to a consistent state some time after the last input received).

MVCC: It stands for Multiversion Concurrency Control. In general, readers of the database are able to see snapshots of the data while it is being read. When a writer finishes updating data, the previous state is marked as obsolete and the writers are able to know it is not current data, but are still able to access it.

Transactions

Does the database provide transaction mode commands?

Referential integrity

Does the database provide mechanisms to reference values from the fields of another entity (document, row, key-value pair)?

Revision control

Does the database provide features to deal with data versioning on every record?

A.1.4 Indexing

Secondary indexes

Can we define several indexes in the entities, besides the primary keys, in order to optimize searches?

Composite indexes

Are we able to use several columns, pairs or fields to define an index?

Geospatial indexes

Does the database allow for the definition and use of geospatial indexes?

Graph support

Is the database able to provide graph oriented queries?

A.1.5 Distribution

Horizontal scalable

Is the database system able to provide scalability by adding more nodes to the architecture?

Replication

Master-Slave: In master to slave replication set ups, the replication of data happens from a main server/node called the master to every established slave. Data cannot be synced in the other way around. Some systems define two modes for this replication to happen:

- **Continuous:** Whenever a change happens in a master, it is replicated to the slaves as soon as possible.
- **One-shot:** The replication is triggered by sending a replication command to the system.

Horizontal scalable

Replica sets are groups of nodes that keep the same data set. The replication of data inside the set happens in a master-slave fashion but when the master is unavailable a new master is chosen via a voting system.

Master-Master

In master to master replication set ups, the replication of data happens from a master to a slave node. The difference is that now the slave can behave as a master to a third now and so on.

Cyclic

In cyclic replication set ups, the master and the slave nodes can change roles and the data flows in both ways.

Multi master

In a multi master or master-less replication set up, any node of the system is able to receive write commands and propagate the changes to the rest of nodes.

Synchronous

In synchronous replication set up replication happens to any node of the defined replication cluster. The main difference here is that a write command is not acknowledged to be successful until all the replicas confirm that the data has been propagated.

Sharding

The concept of sharding refers to horizontal partitioning of a database storage. Therefore, a shard is defined as a horizontal partition of data that is stored on a particular node of the database. The concept of horizontal partitioning refers to splitting the dataset by rows instead of by columns or attributes.

Shared nothing architecture

A shared nothing architecture is type of distributed architecture in which each node is independent and self-sufficient. Therefore, the system does not store any type of centralized state information in order to coordinate its nodes. The advantages of

such systems are avoiding single points of failures, the possibility to add self-healing capabilities and the possibility of making non-disruptive upgrades.

A.2 Comparison tables

TABLE A.1: Features comparison for non relational databases

DB	Model	Implementation Language	Data storage
MongoDB	Document	C++	Volatile memory File System
CouchDB	Document	Erlang	Volatile memory File System
DynamoDB	Document Key-Value	Java	File System (SSD)
HBase	Column	Java	HDFS
Cassandra	Column	Java	File System
Accumulo	Column	Java	HDFS
Redis	Key-Value	C C++	Volatile memory File System
Riak	Key-Value	Erlang	Volatile memory Bitcask LevelDB
Neo4j	Graph oriented	Java	Volatile memory File System
OrientDB	Document Key-Value Graph oriented	Java	Volatile memory File System
Aerospike	Key-Value	C	Volatile memory File system (SSD)
Hypertable	Column	C++	HDFS

TABLE A.1: (Continued) Features comparison for non relational databases

DB	Query language	Protocol	Conditional entry updates	MapReduce
MongoDB	Javascript	Binary	Yes	Yes
CouchDB	Javascript	HTTP/REST	Yes	Yes
DynamoDB	API calls	HTTP/REST	Yes	Yes
HBase	API calls HBase shell	HTTP/REST Thrift	Yes	Yes
Cassandra	API calls CQL	Binary CQL Thrift	No	Yes
Accumulo	API calls	Thrift	Yes	Yes
Redis	API calls	Telnet-like Binary	No	No
Riak	Lucene	HTTP/REST Binary	No	Yes
Neo4j	CypherQL Gremlin	HTTP/REST	No	No
OrientDB	API Calls SQL	HTTP/REST Binary	Yes	Yes
Aerospike	Lua UDF	Propietary	Yes	Yes
Hypertable	API calls HQL	Thrift	Yes	Yes

TABLE A.2: Integrity comparison for non relational databases

DB	Integrity model	Atomicity	Consistency	Isolation	Durability
MongoDB	BASE	Conditional	Yes	No	Yes
CouchDB	MVCC	Yes	Yes	Yes	Yes
DynamoDB	ACID	Yes	Yes	Yes	Yes
HBase	ACID MVCC	Yes	Yes	No	Yes
Cassandra	BASE	Yes	Yes	No	Yes
Accumulo	MVCC	Conditional	Yes	Yes	Yes
Redis	BASE	Yes	Yes	Yes	Yes
Riak	BASE	No	No	Yes	Yes
Neo4j	ACID	Yes	Yes	Yes	Yes
OrientDB	ACID BASE MVCC	Yes	Yes	Yes	Yes
Aerospike	ACID	Yes	Yes	Yes	Yes
Hypertable	MVCC	Conditional	Yes	Yes	Yes

TABLE A.2: (Continued) Integrity comparison for non relational databases

DB	Transactions	Referential integrity	Revision control
MongoDB	No	No	No
CouchDB	No	No	Yes
DynamoDB	No	No	Yes
HBase	Yes	No	Yes
Cassandra	No	No	No
Accumulo	Yes	No	Yes
Redis	Yes	No	No
Riak	No	No	Yes
Neo4j	Yes	Yes	No
OrientDB	Yes	Yes	Yes
Aerospike	Yes	No	No
Hypertable	No	No	Yes

TABLE A.3: Indexing comparison for non relational databases

DB	Secondary indexes	Composite keys	Geospatial indexes	Graph support
MongoDB	Yes	Yes	Yes	No
CouchDB	Yes	Yes	No	No
DynamoDB	No	Yes	No	No
HBase	Yes	Yes	No	No
Cassandra	Yes	Yes	No	No
Accumulo	Yes	Yes	Yes	Yes
Redis	No	No	No	No
Riak	Yes	Yes	No	Yes
Neo4j	Yes	No	Yes	Yes
OrientDB	Yes	Yes	Yes	Yes
Aerospike	Yes	No	No	No
Hypertable	Yes	Yes	No	No

TABLE A.4: Distribution comparison for non relational databases

DB	Horizontal scalable	Replication	Sharding	Shared nothing Architecture
MongoDB	Yes	Master-Slave Replica sets	Yes	Yes
CouchDB	Yes	Master-Slave (Continuous and one-shot)	Yes	Yes
DynamoDB	Yes	Master-Slave	Yes	Yes
HBase	Yes	Master-Slave Master-Master Cyclic	Yes	Yes
Cassandra	Yes	Master-Master	Yes	Yes
Accumulo	Yes	Master-Slave Master-Master Cyclic	Yes	Yes
Redis	Yes	Master-Slave	No	Yes
Riak	Yes	Multi master	Yes	Yes
Neo4j	No*	Master-Slave	Yes	No*
OrientDB	Yes	Multi master	Yes	Yes
Aerospike	Yes	Synchronous	Yes	Yes
Hypertable	Yes	Master-Slave	Yes	Yes

Glossary

ACID Atomicity, Consistency, Isolation and Durability. These are normally the characteristics of relational databases like PostgreSQL.

AIS Automatic Identification System.

API Application Protocol Interface.

BASE Basically Available, Soft state and Eventual consistency. These are normally the characteristics of non relational databases like MongoDB.

BJSON Binary JavaScript Object Notation. Binary format for data storage in MongoDB based on JSON.

CAP Consistency, Availability and Partition tolerance. The CAP theorem states that only two of those three characteristics can be achieved at the same time in a distributed system.

CMS Content Management System.

DataNode Slave node in a Hadoop instance.

Dockerfile Configuration file with instructions to create a Docker container.

EWKT Extended Well Known Text. OGC standard format for the exchange of geographic data.

FDW Foreign Data Wrapper. It is a software extension that allows a database to connect with and interact with remote data sources..

GE Generic Enabler. It is an free and open API and design description of a FIWARE solution to a specific problem, for example, the Context Broker GE to manage context information.

GeoJSON Geospatial data interchange format based on JSON.

GEri Generic Enabler Reference Implementation. It is a specific example implementation of a GE, FIWARE Orion is a reference implementation of the Context Broker GE.

GIS Geographic Information System.

GPS Global Positioning System.

GPSD Open source project to monitor GPS and AIS data streams and receivers.

HDFS Hadoop Distributed File System.

Image (Docker) A Docker image is a snapshot of the state of a container, used to create quickly create new containers.

IoT Internet of Things. It is a concept representing the ability of a different variety of devices to communicate and exchange data through the internet.

Job (MapReduce or Spark) A distributable program.

JSON JavaScript Object Notation. A text based format for data exchange.

JTS Java Topology Suite.

JVM Java Virtual Machine.

NameNode Master node in a Hadoop instance.

NoSQL Not only SQL. Another common name for non relational databases.

OGC Open Geospatial Consortium.

OSGeo Open Source Geospatial Foundation.

OSM OpenStreetMap.

PITR Point-In-Time Recovery.

Raster data Image or bitmap encoded geographic data.

RDBMS Relational Database Managements Systems.

RDD Resilient Distributed Dataset. A custom Spark datatype.

REST Representational State Transfer. A distributed software architecture.

SOW Schema-On-Read. It is a processing paradigm where data is normally stored in raw format and the structure is applied when it is read or precessed.

SOW Schema-On-Write. It is a processing paradigm where data structure must be decided before storing the data.

SQL Standard Query Language. The standard programming language to interact with relational databases.

SQL/MED Extension to the SQL protocol for the Management of External Data.

Transaction Unitary block of actions executed in a database. When an error is raised during a database transaction, all the operations of the block are undone.

Vector data Object representation of geographic collection of points.

WKT Well Known Text. OGC standard format for the exchange of geographic data.

YARN Yet Another Resource Negotiator. The software responsible to allocate jobs in a Hadoop instance.

Bibliography

[1] Big Data definition. URL http://decipherias.com/currentaffairs/big-data-whats-so-big-about-it.

[2] Venkat Ankam. *Big Data Analytics*. Packt Publishing, 2016. ISBN 9781785884696.

[3] PostgreSQL website, . URL http://www.postgresql.org.es.

[4] Salahaldin Juba. *Learning PostgreSQL*. Packt Publishing, Birmingham, UK, 2015. ISBN 9781783989188.

[5] Jim Melton, Jan Eike Michels, Vanja Josifovski, Krishna Kulkarni, and Peter Schwarz. Sql/med: a status report. *ACM SIGMOD Record*, 31(3):81–89, 2002.

[6] SQL/MED PostgreSQL documentation. URL https://wiki.postgresql.org/wiki/SQL/MED.

[7] PostgreSQL FDW documentation, . URL https://www.postgresql.org/docs/9.5/static/postgres-fdw.html.

[8] Mongo FDW repository, . URL https://github.com/EnterpriseDB/mongo_fdw.

[9] PL/PgSQL documentation, . URL https://www.postgresql.org/docs/9.5/static/plpgsql.html.

[10] PL/Python documentation, . URL https://www.postgresql.org/docs/9.5/static/plpython.html.

[11] ACID vs BASE characteristics in database transactions. URL http://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing.

[12] MongoDB website, . URL https://www.mongodb.com.

[13] Hadoop ecosystem table, . URL https://hadoopecosystemtable.github.io.

[14] Hadoop website, . URL https://hadoop.apache.org.

[15] Garry Turkington. *Learning Hadoop 2*. Packt Publishing, Birmingham, UK, 2015. ISBN 9781783285518.

[16] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. URL http://research.google.com/archive/mapreduce.html.

[17] Hortonworks website. URL http://www.hortonworks.com.

[18] Cloudera website. URL http://www.cloudera.com.

[19] MapR website. URL https://www.mapr.com/.

[20] Hive website. URL https://hive.apache.org.

[21] Spark website, . URL http://spark.apache.org.

[22] Rajanarayanan Thottuvaikkatumana. *Apache Spark 2 for Beginners*. Packt Publishing, City, 2016. ISBN 9781785885006.

[23] Open definition. URL http://opendefinition.org/od/2.1/en.

[24] CKAN website, . URL http://ckan.org.

[25] Open knowledge foundation website. URL https://okfn.org.

[26] Examples of CKAN being used by different governmens, . URL http://ckan.org/instances.

[27] FIWARE project website. URL https://www.fiware.org.

[28] Orion Contex Broker documentation, . URL http://catalogue.fiware.org/enablers/publishsubscribe-context-broker-orion-context-broker.

[29] Ngsi9/ngsi10 model documentation. URL https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/NGSI-9/NGSI-10_information_model.

[30] Cygnus repository. URL https://github.com/telefonicaid/fiware-cygnus.

[31] Pethuru Raj. *Docker: Creating structured containers*. Packt Publishing, 2016. ISBN 9781786465931.

[32] Docker website, . URL https://www.docker.com/what-docker.

[33] Docker Hub website, . URL https://hub.docker.com.

[34] OSGeo website. URL http://www.osgeo.org.

[35] OGC website, . URL http://www.opengeospatial.org.

[36] OGC standards documentation, . URL http://www.opengeospatial.org/docs/is.

[37] PostGIS website, . URL http://www.postgis.net/.

[38] Java Topology Suite website. URL http://www.vividsolutions.com/JTS.

[39] Openstreetmap website. URL https://www.openstreetmap.org.

[40] QGIS website. URL qgis.org.

[41] PyQGIS Developer Cookbook, . URL http://docs.qgis.org/testing/en/docs/pyqgis_developer_cookbook/intro.html.

[42] PyQT introduction, . URL https://riverbankcomputing.com/software/pyqt/intro.

[43] Uber website, . URL https://www.uber.com.

[44] Uber dataset source, . URL https://github.com/dima42/uber-gps-analysis/tree/master/gpsdata.

[45] WKT format specification. URL http://www.opengeospatial.org/standards/wkt-crs.

[46] GeoTrellis website, . URL http://geotrellis.io.

[47] Magellan website. URL https://github.com/harsha2010/magellan.

[48] GeoSpark website, . URL http://geospark.datasyslab.org.

[49] SpatialSpark website, . URL http://simin.me/projects/spatialspark.

[50] GeoJSON format specification, . URL http://geojson.org.

[51] Livy website. URL http://livy.io.

[52] PHP Slim website. URL http://www.slimframework.com.

[53] Australian Maritime Safety Authority website describing AIS, . URL https://www.amsa.gov.au/navigation/services/ais.

[54] Las Palmas de Gran Canaria port authority website. URL http://www.palmasport.es/web/guest/presentacion.

[55] MaritecTrust AIVDM/AIVDO decoder. URL http://www.maritec.co.za/?page_id=1051.

[56] AIVDM/AIVDO protocol specification. URL http://catb.org/gpsd/AIVDM.html.

[57] AIS reporting rates, . URL http://arundale.com/docs/ais/ais_reporting_rates.html.

[58] AIVDM/AIVDO protocol specification, . URL https://github.com/ukyg9e5r6k7gubiekd6/gpsd/blob/master/devtools/ais.py.

[59] AIVDM/AIVDO type 8 message specification. URL http://catb.org/gpsd/AIVDM.html#_type_8_binary_broadcast_message.

[60] PyMongo API documentation. URL https://api.mongodb.com/python/current.

[61] PostgreSQL range operators and functions documentation. URL https://www.postgresql.org/docs/9.5/static/functions-range.html.

[62] Pablo Fernández, José Miguel Santana, Sebastián Ortega, Agustín Trujillo, José Pablo Suárez, Conrado Domínguez, Jaisiel Santana, and Alejandro Sánchez. Smartport: A platform for sensor data monitoring in a seaport based on fiware. *Sensors*, 16(3):417, 2016.

[63] Pablo Fernández, Jose M. Santana, Sebastián Ortega, Agustín Trujillo, Jose P. Suárez, Jaisiel A. Santana, Alejandro Sánchez, and Conrado Domínguez. *Web-Based GIS Through a Big Data Open Source Computer Architecture for Real Time Monitoring Sensors of a Seaport*, pages 41–53. Springer International Publishing, Cham, 2017. ISBN 978-3-319-45123-7. doi: 10.1007/978-3-319-45123-7_4. URL http://dx.doi.org/10.1007/978-3-319-45123-7_4.

[64] Pablo Fernández, Jaisiel Santana, Alejandro Sánchez, Agustín Trujillo, Conrado Domínguez, and José Pablo Suárez. A gis water management system using free and open source software. 2016. URL http://mami.uclm.es/ucami-2016/sc-program.html.

[65] Db-engines website. URL http://db-engines.com.

[66] CouchDB website. URL http://couchdb.apache.org.

[67] DynamoDB website. URL https://aws.amazon.com/dynamodb.

[68] HBase website. URL http://hbase.apache.org.

[69] Cassandra website. URL http://cassandra.apache.org.

[70] Accumulo website. URL https://accumulo.apache.org.

[71] Redis website. URL http://redis.io/.

[72] Riak website. URL http://basho.com/products/riak-kv.

[73] Neo4j website. URL https://neo4j.com.

[74] OrientDB website, . URL http://orientdb.com/orientdb.

[75] Aerospike website. URL http://www.aerospike.com.

[76] Hypertable website. URL http://www.hypertable.org.