Implementation of a (Big) Data Management Backbone

Master in Data Science, FIB
Big Data Management

April 23rd, 2023

Ander Barrio Campos - Odysseas Kyparissis

Table of Contents

Introduction

The goal of this report is to present the conceptual design of a (big) data management backbone which will be used to manage and manipulate data related to Barcelona rentals, territorial income distribution, as well as the vehicle motorization index of the city. The described datasets are obtained from multiple sources and in various ways. To begin with, for the housing properties information, the API of the website *idealista*[1] *was* used. In addition, for gathering the data about the territorial income distribution as well as the vehicle motorization index of the city, the Ajuntament de Barcelona's open data service[2] was utilized. In the following figures, one can find the technologies selected for each specific part of the implementation (Figure 1) as well as the Business Process Modeling and Notation (BPMN) diagram (Figure 2) which includes the conceptual design of the system's operations. The implementation of the software can be found here, which is the link to the repository of the project on GitHub[3]. More details about the implementation and the assumptions made can be found in the next chapter.

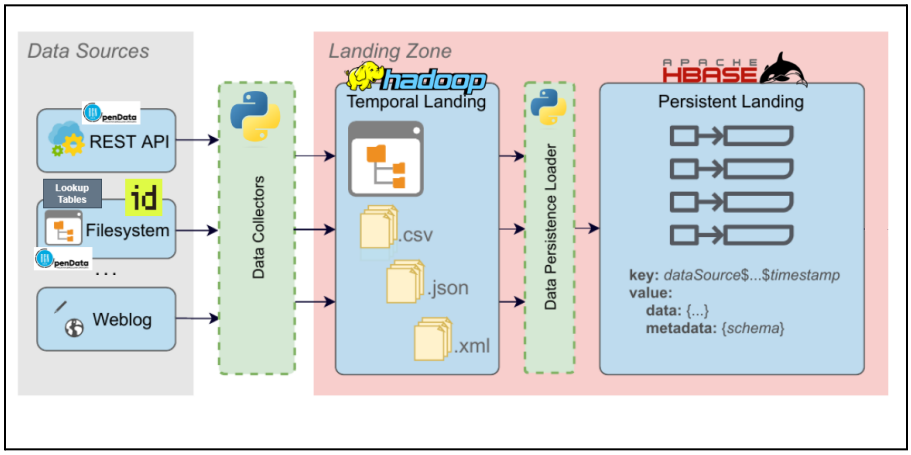

Figure 1. Technologies Applied During System Design

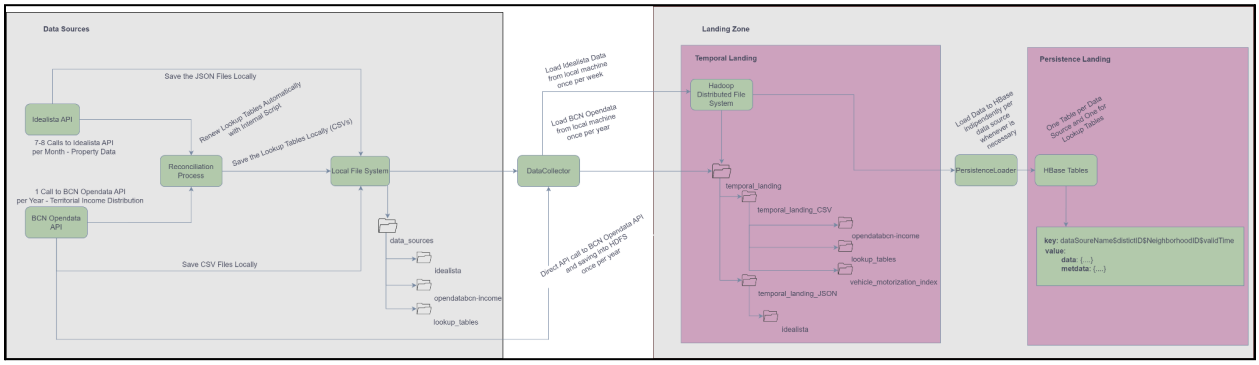

Figure 2. BPMN Diagram of the System

[1] https://www.idealista.com/
[2] https://opendata-ajuntament.barcelona.cat/en
[3] https://github.com/

Landing Zone

In this chapter, a deeper look is taken into the details of the *Temporal* and *Persistent Landing Zones*. The design of the system was implemented, having in mind that the specific solution is going to be used, in order to apply descriptive and predictive analyses of data related to different districts and neighborhoods of Barcelona. As said above, the available data are housing properties information, territorial income distribution as well as the vehicle motorization index of the city.

Temporal Landing Zone

This subsection presents a comprehensive overview of the decisions and assumptions that were made during the design and implementation of the temporal landing zone. As illustrated in Figure 1, the selection of Hadoop Distributed File System[4] (HDFS) was determined to be an efficient choice for the temporal landing zone. A detailed explanation of the rationale behind this decision is provided in the following sections. Information about the timeline of the processes and the conceptual design of the functionality of the system is presented in Figure 2.

Since the Temporal Landing Zone is designed to store raw data until it is processed, a distributed file system like HDFS is suitable for this zone. HDFS is a highly scalable and fault-tolerant distributed file system that is designed to handle large volumes of data efficiently, making it a suitable choice for Big Data storage and processing. Its support for data replication across multiple nodes ensures high availability and data redundancy. Additionally, it supports Big Data formats like SequenceFile, Avro, and Parquet, which offer efficient storage, compression, and fast data access. Furthermore, HDFS with Parquet or Avro format can support efficient data processing with tools like Apache Spark, which can read data from HDFS in a distributed manner. However, as the data will be stored persistently in HBase, and will not be retrieved from the Temporal Landing Zone in subsequent stages, the benefits of using advanced storage formats such as Parquet may not be fully realized. As a result, it may be more beneficial to keep the files in their original formats, such as JSON or CSVs, within the Temporal Landing Zone. By doing so, the processing of data in subsequent stages can be simplified, while also avoiding any potential performance penalties that may arise from using an advanced storage format unnecessarily. What is more, the system employs a single coordinator that divides files into chunks of 128 MBs for efficient storage and retrieval. Given a file each of its chunks is stored in a potentially different chunkserver (randomly chosen) and replicated in multiple nodes. As a default, every file is replicated three times to ensure data redundancy and high availability.

The negative aspects of this technology have also been taken into account. HDFS is not well-suited for random data access or updates due to its design for sequential data processing. The setup and maintenance of an HDFS cluster can be complex and time-consuming, requiring

[4] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

specialized knowledge and skills. Furthermore, HDFS performance may be impacted by factors such as network latency and disk I/O, which can result in slower data processing times.

Moving on to the design aspects of the project, one of the key design decisions was to have one data collector per data format, namely CSV and JSON. This design choice makes the solution more robust and flexible. With this approach, adding more external data sources is simply a matter of adding a new folder in the directory ***data_sources*** locally, which allows the data collector to easily access the new data. Additionally, if there is a need for adding a new data format like XML, we can simply update the ***data_collector*** class by adding a new function that implements this format. This modular design not only improves the overall maintainability of the project, but also enables it to be easily extended to accommodate new data formats as needed.

The general functioning of the code defines some methods which serve the purpose of uploading files from the local directory to the designated HDFS directory according to their respective formats. The methods first retrieve a list of all files present in the local directory, and then check if the specified HDFS directory exists. Following this, the methods iterate through the retrieved files and create a new directory with the same name as the containing folder of each CSV file in the HDFS directory. Once the directories have been created, the files are loaded and converted to bytes. Finally, the methods upload the files to their respective directories in the HDFS directory using another method named ***upload_file_to_hdfs***. Any exceptions that occur during the process are caught and logged. Overall, these methods facilitate the seamless transfer of data from the local directory to the designated HDFS directory, making it readily available for subsequent processing and analysis.

Once the execution of the data collectors is made, the folders within the HDFS follow a specific structure that organizes data by ***data source format***, ***data source***, and ***individual files***. This structured distribution of source files facilitates the movement of data from the temporal zone to the persistent zone by including important information such as ***format***, ***source name***, and ***transaction time*** in the file paths. This structure enables the optimal organization of HBase for the Persistent Zone by providing necessary details about the data source and its format. As a result, data is efficiently organized and easily retrievable, making the HDFS a suitable choice for large-scale Big Data storage and processing.

Persistent Landing Zone

In this subsection, a detailed description of the decisions and assumptions made during the design and implementation of the persistent landing zone is provided. As it can be seen in Figure 1, for the *Persistent Landing Zone* it was concluded that Apache HBase[5] is one of the best selections, since the temporal landing zone is built on HDFS.

HBase was selected in this case, for the reason that it is a distributed column-family key-value store that is optimized for random access to large data sets, making it suitable for handling big data. Also, HBase provides high availability and fault tolerance by replicating data across

[5] https://hbase.apache.org/

multiple nodes. Finally, it is scalable and can handle petabytes of data. Thus, for dealing with the specific project, the usage of HBase seemed appropriate based on the large amount of data which is being gathered from *idealista* and *BCN open data service*. Additionally, it was considered that with an appropriate design on the key and the value of the tables of HBase, we are gonna be able to achieve fast queries for the descriptive and predictive analyses which are gonna be conducted in the second part of the project. Finally, it is going to be easy to integrate the current solution well with other big data tools like Apache Spark[6] in the future.

On the other hand, MongoDB[7] could also be used here, but in general MongoDB can be slower for the queries as well as more resource-intensive compared to HBase. Also, it requires careful consideration of the data model and indexing strategy to achieve optimal performance. For those reasons it was discarded from our options. In the following sub-section, the design of the HBase database is taking place.

HBase Design

As it can be seen in Figure 2, for moving the data from the temporal landing zone into the persistent landing zone we implemented the *PersistenceLoader* component by using Python[8]. The *PersistenceLoader* consists of four main functionalities, one for each data source available in the current solution. For more details about the functionality of this component, it is necessary to present the design of the HBase database. It is important to note here that we assumed that the format of the files for each data source is the same in the temporal landing zone.

More specifically, in HBase we concluded that the most suitable design would include one table for each distinct data source. Consequently, we created the following four tables in HBase: ***opendatabcn-income***, ***veh_index_motoritzacio***, ***idealista*** and ***lookup_tables***. For the first three tables, we used two column families, one called ***data:*** and one called ***metadata:,*** while for ***lookup_tables*** we used three column families, as follows: ***idealista_data:***, ***income_data*:**, and ***metadata:***. As it is clear, the column family ***data:*** include the information about the data of the files, separated into different qualifiers (based on the available variables/columns of the files) and the column family ***metadata:*** stores for each file the appropriate schema information (number of rows, number of columns, column names, ingestion time, etc.).

Additionally, for the ***lookup_tables*** we decided that we can save the lookup information in the same table, but by using different column families (***idealista_data:***, ***income_data*:)**. For this decision, we made the assumption that the lookup tables are created by us with an internal script, for which we are sure that the columns are always the same and that each time the most updated version is in the temporal landing zone in HDFS, located under the directory ***temporal_landing/temporal_landing_CSV/lookup_tables***. Also we assume that we overwrite the file each time we update it.

[6] https://spark.apache.org/
[7] https://www.mongodb.com/
[8] https://www.python.org/

Key Design

During the design of the key for each unique table several assumptions were made. To begin with, the following key implementation was followed for each data source:

- **opendatabcn-income** & **veh_index_motoritzacio**: {table_name}_{district_id}_{neighborhood_id}_{valid_year}
- **idealista**: {property_code}_{district}_{neighborhood}_{valid_time}
- **lookup_tables**: {table_name}_{district_id}_{neighborhood_id}

where, *neighborhood_id* refers to the identifier of the neighborhood and *district_id* refers to the identifier of the district where the data is related to (for the sources coming from BCN opendata API). As for idealita, *district* and *neighborhood* refers to the names of the districts and neighborhoods respectively, but as they were saved into the JSON files. In order to join later the data sources, the usage of the lookup tables is necessary.

The composite key for each table described here, consists of a combination of district_id, neighborhood_id, and a timestamp (valid_year and valid_time, besides lookup_tables). This key design would allow for efficient querying of data based on district and neighborhood, and would also enable time-based queries to retrieve data for a specific time period by specifying a range of timestamps. Additionally, by including the district and neighborhood in the key, you can easily aggregate data at that level for analytical purposes. The idea also followed an approach that uniquely identifies the data files, and then can optimally spread the data in different machines for achieving analytical tasks in different districts and neighborhoods of the city. Thus, the combination of attributes used can be considered that are often looked up inside the key. In addition, the attributes have more uniform distribution and that enables efficient splitting into regions which improves the balancing.

As for the timestamp, in the schema design we used *valid_year* and *valid_time* which were extracted from the filenames and they refer to the time that the data was downloaded from each source (valid time). In that way the key can include information about the validity of the data. Using the HBase timestamp can also be useful if it is needed to track the time when a particular row was inserted or updated in HBase. This timestamp can be found inside each row of a table.

Moreover, as we are using a single coordinator in this solution we did not dive further into the configuration of different regions for the tables. But, since the row key is constructed using the format mentioned, a good idea would be to use a sharded approach based on *district_id* for region splitting. This will ensure that all data for a particular district is stored in the same region, which can improve read performance when querying data for a specific district.

To conclude, in order to maintain a history of the changes to the data, we enabled versioning for the HBase tables. This allows us to keep multiple versions of a row, each with a different timestamp. This can be useful for auditing purposes and for tracking changes over time. The number of versions to keep was configured to 3 (default value), but in general it is configured based on the storage capacity available and the desired level of detail for the history of the data.