Implementation of a (Big) Data Management Backbone

Eemi Vihavainen
Liam James Glennie England

Introduction

The aim of the project is to determine a pipeline to make an organized and properly structured data backbone to be accessed later for descriptive and predictive analysis. This part of the project focuses on the data collecting and the landing zone of the pipeline, which are discussed in depth later in the report. The steps of this project are visualized in Figure 1.

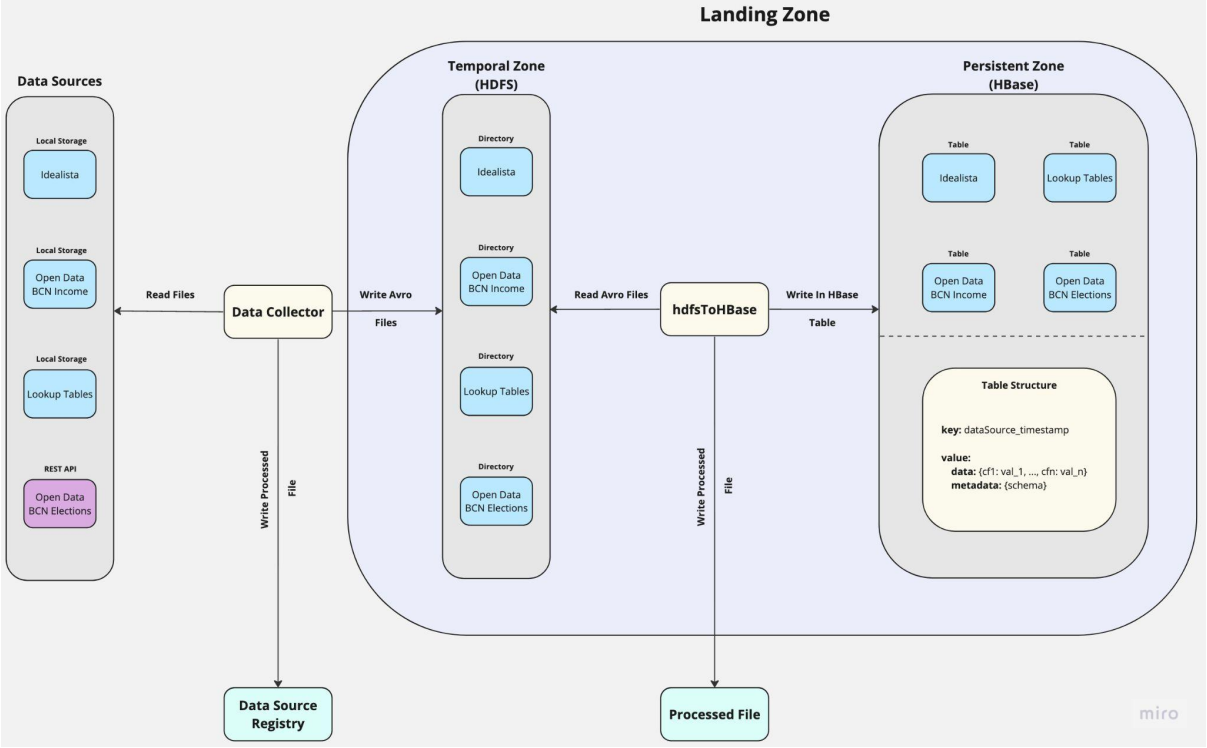

Figure 1: BPMN of the project stages.

Data sources

The data consists of two datasets from two different websites. It is related to Barcelona's housing and the relationship with its economy. First dataset (*idealista*) contains rental related data, and the other (*openBCN*) contains data regarding

territorial distribution of income. The data consists of columns like names of districts, neighbourhoods, and statistics related to those. Furthermore, a lookup table is given to combine the two datasets, since the datasets don't necessarily share the same names for the connecting attributes.

A third dataset was selected and included as well. The content of the data is related to election habits in the districts of Barcelona, and it is obtained from the *OpenBCN* website. Since the data is from the same source as the second dataset, the lookup tables will work on this dataset as well. In order to access the data, data collectors were implemented to fetch the data from the website's API.

KPIs

The diversity of the data, i.e. the wide range of columns, allows for many variations for the KPIs. Few examples of either descriptive and predictive analysis KPIs will be considered and explored.

Voting related KPIs:
With the added dataset regarding voting habits combined with the other datasets, several useful and informative KPIs could be formed, which could include instances like: voting degree of a district/neighbourhood, and their trends between the election seasons, or variation of party endorsement in a district/neighbourhood.

Housing related KPIs:
The variation of the *idealista* data offers a wide range of options for KPIs. One example could be like: occupancy rate per district/neighbourhood.

Temporal landing zone

Temporal landing zone is the initial part of the landing zone. Its purpose is to receive the data from the sources for a temporary storage of the data, and organize it to a structure to be forwarded to the persistent landing zone. In the Big Data context, the key feature for this part is to be able to handle large amounts of data points efficiently. In practice, this means fast read times, which leads us to the first design choice.

The format of the files stored in the distributed file system plays a big role in the efficiency of the whole temporal zone. Three different formats were considered: *Avro*, *Parquet,* and *SequenceFile*. Out of the three, Parquet can be excluded rather easily, since the column-based structure is not needed in this use case as we are not interested in column-based partitioning. We are interested more in horizontal partitioning because it offers faster reading and writing times and we only want to move the data between zones, we will not be performing any queries on the data that would favour vertical partitioning.

The two options left, *Avro* and *SequenceFile*, are both a good fit to be used in the temporal landing zone. Both store the data in a similar way: the data stored horizontally in key-value pairs. However, the key differences are the schema and data type. Avro uses JSON format, while SequenceFiles are just binary key-value pairs. Comparing just the data types, JSON is better from the two, since it's human-readable, and half of the data is in JSON format already, so only the CSV files need to be formatted to data type accepted by Avro.

The schema used by Avro is defined using JSON-like notation. The key factor with Avro schema is that it is easily evolvable, i.e., it is easily modified/updated in case there's a change in the structure. So if the source data is changed, the schema can be easily modified to handle it. This brings flexibility for future decisions. SequenceFiles, on the other hand, doesn't have any schema. It only stores the records (or blocks) in a sequence. Both technologies use compressing, which is preferable for this use case. Avro compresses (binary encoded) rows, and SequenceFile compresses either records or blocks (of records). Since dealing with big data, compressing the data is desirable, since it takes less storage space

To conclude the choice for the temporal zone, both Avro and SequenceFile can be used. However, based on Avro's schema and the fact that the data is partly the same type as Avro's default, Avro will be the technology used in the temporal zone. Additionally, the lack of SequenceFile documentation also played a role in our decision process.

Code Assumptions

- We created a dictionary containing information of the format and schema of each source. The code is written in a manner that by adding a new source to the dictionary, it will be uploaded to HDFS without any further modifications to the code. If a new file format arrives that is not JSON or CSV, certain modifications would need to be done. We could not account for all file formats for obvious reasons.
- We also created a dictionary for API sources. This dictionary contains a key for each API source and its values are a list of the relevant URLs to fetch.
- We have included feedback messages in case of failure.
- The schema is assumed to be the same for all files of the same source.
- -We wanted to run the code in the VM. However, the python version was too old for fastavro, the library we used to process Avro files. Apparently, it is quicker than the normal Avro library. To get an updated version of python we used conda. There were alternatives, such as, using the Avro Python library or running the code in our local machine. For performance reasons we want to run it all in the same machine.

Persistent landing zone

The second part of the landing zone is the persistent landing zone, which acts as a persistent storage for the data. The persistent landing zone is the final part of the landing zone. The purpose of the landing zone is to organize all of the data in a way that it could be queried and utilized in the later stages of the pipeline. Therefore, it's important to choose the correct technology to be able to achieve a consistent workflow to the next phase of the pipeline.

The two storage options considered to be used in the persistent landing zone were *HBase* and *MongoDB*. Out of the two, HBase is the more suitable technology to be used instead of MongoDB for various reasons. The justifications for this choice are explored in the following paragraphs.

The data model of Hbase is a column-family-oriented database that is optimized for handling large, structured data sets. It is designed to store and manage large volumes of data with high write and read throughput. On the other hand, MongoDB is a document-oriented database that is optimized for handling unstructured data sets. While MongoDB can store structured data, it may not be as efficient as HBase in handling structured data sets.

HBase has built-in support for keeping multiple versions of each data record, making it a good choice for use cases where data changes over time. MongoDB does not have built-in support for versioning, so you would need to implement it yourself. HBase supports a wide range of file formats, including Avro, which is what you have chosen for the temporal zone. MongoDB supports JSON natively, but you would need to convert CSV files to JSON before storing them in MongoDB. HBase is designed to scale horizontally by adding more nodes to the cluster, making it a good choice for distributed data management backbones where the volume of data can grow over time. MongoDB also supports horizontal scaling, but it may not be as efficient as HBase when dealing with large data sets. HBase provides strong consistency guarantees, which means that all nodes in the cluster see the same data at the same time. MongoDB provides eventual consistency by default, which means that data changes may not be immediately visible to all nodes in the cluster.

To conclude, HBase seems the best fit for the use case. Especially its ability to handle the structured data from Avro would make the best overall fit for the whole of the landing zone.

Code Assumptions

- When a new source appears in the temporal zone, the code adapts and nothing needs to be changed
- We have created two column families: data (contains values) and metadata (contains Avro schema)

- The schema is implied when reading .avro files in fastavro.
- For versioning we have used the timestamp at which each file was processed. To avoid using potentially prohibited characters we replaced whitespaces and colons with underscores.
- Our row key identifier has the following format *fileName_Timestamp*. To avoid overwriting, each column qualifier is followed by _X where X represents the number of the row of the file being processed. So, if the file being processed has 50 rows and all rows have a value for Year, the column qualifiers for Year will be Year_1, Year_2, Year_3… Year_50. The _ is used as a separator that can be used in the future for retrieving data. We considered this to be the best solution to avoid overwriting. Another solution could have been to change the row key identifier to *fileName_rowNum_Timestamp* where *rowNum* represents the row inside of the file. This way we would avoid overwriting the column values. However, this solution seemed less clean and potentially harder to maintain.
- We used batch to perform uploads of records in batches of 1000. This was done to optimise the process as uploading one record at a time is time and resource consuming.
- Ideally, this code should be run in the same machine as HDFS and Base because that way we avoid sending data over the network. The data moved within the same machine will be much quicker than moving data between machines over the network.
- After processing each file from the temporal zone and moving it to the persistent zone, we delete the file in temporal.
- The HDFS connection is closed automatically once the program ends.

Conclusion

Various design choices were examined and their use was justified according to the use cases of the project. The project was concretised by producing code for every phase of the pipeline. The programs and their relationship between each other and the other parts of the project are showcased in Figure 1.