Big Data Management Delivery I: Landing Zone

Louis Van Langendonck
Pim Schoolkate

August 5, 2022

Instructions on how to use the application

To get data in the temporal zones, two scripts should be run, one for loading API data into the HDFS and the other for loading in the zipped data in the HDFS. The former is named `covid_api_to_temporal_hdfs.py` and the latter `zip_to_temporal_hdfs.py`. Both have default settings that might need to be adjusted depending on the settings of the user. Calling `python <script.py> -h` will explain what settings can be adjusted. In order to run these scripts, python 3 needs to be installed, as well as the `hdfs`, `paramiko` [3] and `pandas` [7] packages. Important to note is that this script must be run as administrator.

To move from the temporal zone to the persistent zone, the `to_persistent.py` file has to be run using the command prompt by changing directory to where the script is stored and running `python to_persistent.py`. The connection to the virtual machine and to MongoDB should all happen automatically, however if a different IP-address or port has to be used, default inputs can be overwritten in the command line. For more information on any of the inputs, please run `python to_persistent.py -h`. The last input considers whether or not to use a metadata file in the persistant zone to keep track of which data has already been loaded in to MongoDB. As this might yield errors if the necessary authorizations are not given on the local computer, consider turning this option off by running instead `python to_persistent.py -a False`. Please make sure that the Python interpreter used is up-to-date. It may be necessary to download a few packages to the environment like `PyMongo` [6], `Paramiko` [3], `pandas` [7], and `HdfsCli` [4].

1 Temporal Landing Zone

1.1 Architecture

The temporary landing zone consists of a directory within a **Hadoop Distributed File System (HDFS)**[2] on a virtual machine. HDFS was chosen as the technology for the temporal landing zone as it allows the data to be **stored in distributed fashion** and **supports all different types of files**, without any modifications, thus swiftly handling **source heterogeneity's**. In contrast, technologies such as Hbase or MongoDB both require some sort of data formatting, as with the former, data needs to be converted into key-value pairs, and with the latter, data needs to be converted into *.json* files.

Furthermore, HDFS **automatically produces replicas** (in this case 3) of the data, which provides a security to the data storage as well as better availability [1]. As HDFS **distributes data** randomly across the machines available, it is not needed to design a key which evenly distributes the data in this zone. Moreover, this implies good scalabilty as extra nodes can easily be added and thus provide improved storage and processing capabilities [1]. Another nice property of HFDS is its **rebalancing feature**, which helps storing data evenly across nodes, preventing one to have much more data compared to others.

Querying the data in the HDFS will always be done in **complete** scan fashion, as files are only queried to be send to the persistent zone and thus reducing the need to optimize the read

component of the storage.

Finally, by simple storing the original format of any of the source files, the method is flexible and resistant to changing data sources. Keeping in mind HDFS is just a file system supporting distributed storage, it is considered the simple and logical option for a temporary landing zone while still being more suited for storing big data compared to classical local file systems. Within the HDSF, the temporary landing zone can be found under */user/temporary*. Data arrives at this directory in two ways. Figure 1 shows this design visually.

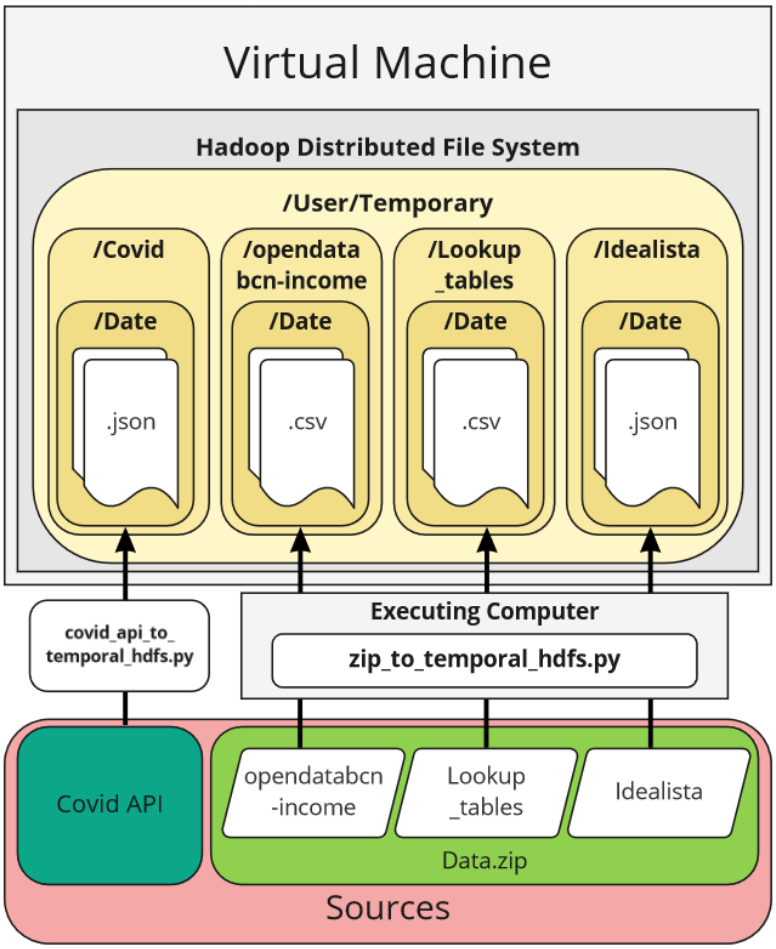

Figure 1: Visualization of the temporal landing zone.

First, data describing the daily Covid infections in Barcelona is directly downloaded into the sub-directory */user/temporary/covid* using an API. Each time the data is ingested, the data is placed in a folder with as a name, the date of that day. Ingesting on the first of April would thus look like */user/temporary/covid/04_01_2022/covid.json*. In this case, the data concerns information about Covid. However, any API that fetches data can insert data in the HDFS in a similar way, as should be possible with any Big Data Management system.

The other data sources all arrive the temporary landing zone through the computer which executes the script. That is, the files should be present on the executing machine and be transferred to the virtual machine. In this case, it concerns a *.zip* file which contains all the data, and thus a generalized pipeline was built which is able to insert any data from a *.zip* file into any HDFS system. Then, the files are placed in the HDFS under the sub-directory */user/temporary/idealista/04_01_2020/2020_01_02_idealista.json*, which in this case for Idealista on the first of April. Because the files are first downloaded from *learnsql* to a local machine, it seems that

two temporary landing zones are present. However, in actual implementations, this most likely not be the case. Further explanation of this is given in section 1.2.

1.2 Assumptions

The main assumption of the temporal landing zone is that the virtual machine is an actual distributed file system, with **multiple machines**. It would make little sense to go through the difficulties of putting all the files in an HDFS without it being distributed, as in the case of this project.

Tightly connected to this, is the assumption that the **data is big**. Currently, the data consists of mere megabytes, but in the case of big data, this number can easily surpass multiple gigabytes. If not for big data, it would not make much sense to use a HDFS, as the more filled the database is, the more uniform the distribution [1].

A third assumption considers how the data would be obtained in an actual implementation. Now, the *.zip* file is retrieved from *learnsql*, which is a closed learning environment. More realistically, the **data would be scraped from the cloud** (internet or database provided by the source) and **loaded immediately into the HDFS**, without local storing. This would imply only a single, and moreover distributed, temporal landing zone, therefore further strengthening the decisions made.

We also assume that the data in the **temporary landing zone does not need to be queried much**, and therefore does not need to be distributed perfectly uniform on the HDFS. As it assign the locations randomly, one might assume that the data ends up uniformly, however as the files are of different size, some machines will be allocated more data than the other [1].

Because this is the temporary landing zone and it is expected that within this zone no alterations to the data are made, the files are ingested in their original file format.

2 Persistent Landing Zone

2.1 Architecture

For the persistent landing zone, it is chosen to transfer the data into `MongoDB` [5]. This is executed by retrieving newly ingested data from the HDFS, transforming all file types to *.json* format and uploading each file to a corresponding mongoDB collection (either 'covid', 'idealista', 'lookup' or 'opendata'. A first reasoning behind choosing this documentation is **ease of implementation**, thus interfering very little with the data, as it is just the landing zone. As MongoDB expects the *.json* format as input, two of the five data sources (idealista & covid data) do **not need any conversion** whatsoever. The other two sources, both of *.csv* format, are **easily converted** to *.json* as any table can be considered as a single level *.json* file, with as key the column and as value the corresponding value. However, **a drawback** of this approach over, for example, a key-value approach, is **introducing some redundancy** in repeating the keys for every data row, thus substantially **increasing the volume of metadata.**

Secondly, as it is assumed that the data will have to be cleaned and transformed right out of the persistent zone, sufficient querying is needed. In MongoDB this is achieved using a relatively **straightforward querying language**. Moreover, the program allows for **building indexes on arbitrary keys**. Therefore, an approach to facilitate the transformations is by building indexes on often queried or selected keys. Even more, MongoDB **supports inter-query and intra-operator parallelism** [1], something that is further discussed later. However, even considering these details, compared to some other options, document stores are **not the best**

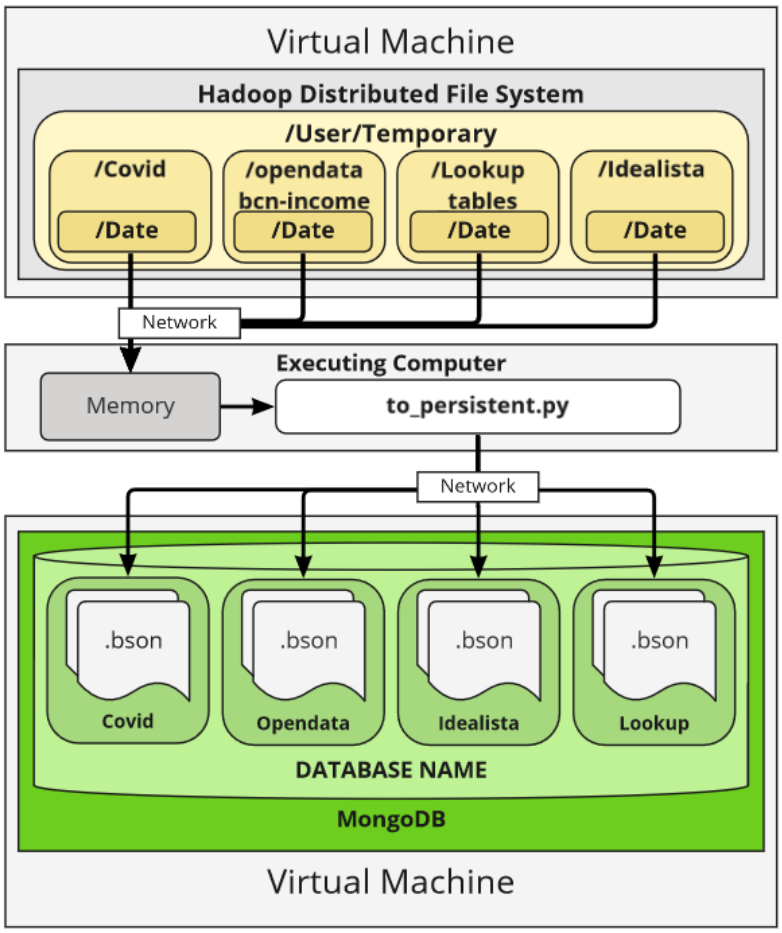

Figure 2: Visualization of the persistent landing zone.

options for fast querying. Moreover, it does not support joins, which will eventually have to happen in this case [1]. This means, other tools will be needed in future zones.

Thirdly, document stores like MongoDB are known to be **flexible** as *.json* documents are self-descriptive, allow nesting, are not of fixed length and do not require fixed keys [1]. Therefore, the semi-structured nature of MongoDB **accommodates for constant changes in the source**. This, in combination with the choice of HDFS for the temporal part, implies that the landing zone is completely compatible to the varying nature of big data.

A fourth argument for using document stores is the fact that MongoDB also offers **data distribution possibilities**, of which *sharding* is the main component. These are range-based or hash-based chunks as a result of horizontal fragmentation [1]. Generally this type of fragmentation leads to **improved data locality**. However, MongoDB **does not support vertical fragmentation**, which typically improves read-only purposes [1]. This is not a major issue though, as data from the landing zone only needs to be read when it is ingested in the formatted zone. That is, the landing zone is not a read intensive zone. Moreover, just like in HDFS, MongoDB store **three replicas of each shard**. As stated before, all of this allows for both inter-query and intra-operator parallelism and **increased security and availability** [1].

Lastly, in terms of performance and comparing to other possible methods, MongoDB excels at **insertion time**, which, from the perspective of *load first, model later*, is highly preferable.

However, MongoDB comes with a few **drawbacks**. One of which is the fact that because

the data is **stored without name** within each of the collection, any structure given by naming is lost. To for example not lose the date of the Idealista data contained in the document name, the pipeline adds to each *.json*, a new key-value containing the scrape date of the instance. Moreover, to know which files have already been transferred to MongoDB, the option of keeping track of transfers is added by creating a *transfer_metadata.json* metadata file. For every source it stores the list of files of a particular ingestion date that have been transferred into MongoDB, thus not transferring these again in the following iteration. As mentioned before, some other drawbacks are **the absence of join & vertical fragmentation support, the increase of metadata** and compared to some other options, the **sub-optimal query performance** [1]. One last assumption is that the speed at which the data arrives at the persistent zone is of such magnitude that it plays a major role in choosing a technology for this zone. Otherwise, a PostgreSQL nested JSON solution would also be a candidate for this zone.

2.2 Assumptions

Again, the two main assumptions are that **the data is actually big, and distribution actually happens on different machines**. Only in this situation, all decisions based on parallelism, data storage, etc. make sense. However, in contrast to the temporal landing zone, the persistent zone will **have to be queried** to transform it into the formatted zone. Another assumption that can be made is that in a more integrated implementation, the **python scripts** used for transforming the data from temporal to persistent, **should be stored and run in the virtual machine(s) themselve(s) and in parallel fashion**, instead of sending all data to a local computer and then sending it to MongoDB. This introduces a bottleneck and a lot of extra network transferring. Therefore, the current script is more a proof-of-concept than an actual practical implementation.

3 Conclusion

Using HDFS for the temporal landing zone and MongoDB for the persistent landing zone, results in a practical yet imperfect solution to the problem. More than anything, the Volume and Variety aspects of big data are accounted for, with flexible and distributed functionalities. The ingestion time of MongoDB also highly increases the ingestion part of the Velocity, prioritising *load first, model later* Moreover, the simplicity of implementation, by interfering little with the data, can be considered a strong aspect. However, some of the drawbacks include sub-optimal read and query performance of MongoDB.

References

[1] A. Abello and S. Nadal, *Lecture notes big data management*, February 2022.

[2] Apache Software Foundation, *Hadoop*.

[3] Jeff Forcier, *Paramiko*.

[4] Matthieu Monsch, *Hdfscli*.

[5] MongoDB, *Mongodb*.

[6] MongoDB Python Team, *Pymongo*.

[7] T. pandas development team, *pandas-dev/pandas: Pandas*, Feb. 2020.