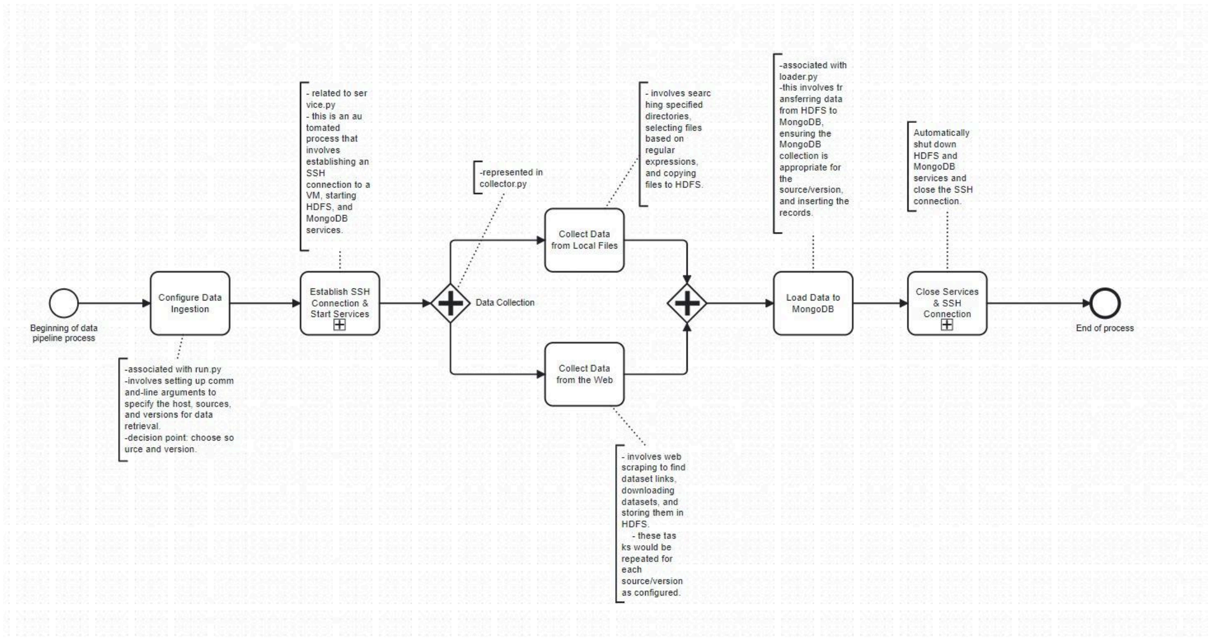
David Candela, Olivia Momeu

Landing Zone

Introduction

The aim of this project was to create a data pipeline, similarly to what could be encountered in a Data Science project, for data ingestion from heterogeneous sources. After setting some business requirements, we proceeded to develop a solution that could answer those requirements as efficiently as possible. Such tasks could be encountered in a real-life scenario in DataOps/MLOps positions, being a perfect exercise for Data Science students. In this particular case, we focused on the Data Management backbone, which is usually a common component for the entire organization in which the goal is to ingest and store external data into the system and expose a cleaned and centralized repository. Our solution aimed to contour a so-called Landing Zone, made out of Temporal Landing and Persistent Landing connected through a Data Persistence Loader. At a high level, the Temporal Landing stores data in its raw format in our storage system of choice, that in this case was HDFS, while the Persistent Landing stores data in the format that is useful for future data operations, in our case, MongoDB. The Data Persistence Loader bridges the gap between the two and acts as means of moving data between one and the other. Details will be presented in the following sections.

Process Diagram


Beginning of data pipeline process
Configure Data Ingestion
-associated with run.py -involves setting up comm and-line arguments to specify the host, sources, and versions for data retrieval. -decision point: choose so urce and version.
Establish SSH Connection & Start Services
- related to ser vice.py - this is an au tomated process that involves establishing an SSH connection to a VM, starting HDFS, and MongoDB services.
Data Collection
-represented in collector.py
Collect Data from Local Files
- involves searc hing specified directories, selecting files based on regular expressions, and copying files to HDFS.
Collect Data from the Web
- involves web scraping to find dataset links, downloading datasets, and storing them in HDFS. - these tas ks would be repeated for each source/version as configured.
Load Data to MongoDB
-associated with loader.py -this involves tr ansferring data from HDFS to MongoDB, ensuring the MongoDB collection is appropriate for the source/version, and inserting the records.
Close Services & SSH Connection
Automatically shut down HDFS and MongoDB services and close the SSH connection.
End of process

Data Collector

As we have to start with collecting from local files and later be able to expand to collect from sources through the internet, a system that would be able to at least accommodate these two different methods is to have an implementation for each of them with a common interface to easily switch between them. This design strategy would allow later on adding new collectors as necessary (if they adapt to the common interface).

The common factors to consider for the common interface are that for each source there can be multiple versions (each year with annual data, the last time it was modified with continuously updated data, etc) and the steps to collect one version or another from the same source are almost the same (changing the file path or the URL). As the steps from collecting datasets from one local source to another (and the same for ones collected through the internet) won't change drastically (which directories to look for the files and where to collect the URL for the datasets) we can make a class for local datasets and another for datasets through the internet and abstract the rest of the details to a metadata file with information like which implementation to use, which directories to search or what URL to use to retrieve the data.

For collecting from local files we added an attribute in the metadata to store where to search the files (a set of directories) and two regular expressions to decide which files and what part of the file name indicates the version. Then we make a copy of the files in HDFS as our temporal landing zone.

For collecting from the internet, we implemented a simple web scraping system with regular expressions to retrieve the links for all versions of the dataset from a web page instead of having the user add them manually whenever there is a new one. We then request the web for the dataset and stream the response directly to HDFS's temporal landing.

Data Loader

Given a file in the HDFS temporal landing, the data loader uploads the data into the MongoDB instance held on the VM without applying any data transformation. We make use of the multiple functions from pandas to load both files in CSV format and in JSON. We then connect to the MongoDB server running in the VM from our code, ensure the corresponding collection (named `{source}/{version}` to prevent different datasets overwriting one another) is empty and store all records from the file there.

Data storage systems

HDFS

The decision to use HDFS for temporal landing has been made due to the following advantages:

1. Scalability: HDFS is designed to be highly scalable. It can store and distribute large data sets across hundreds or thousands of servers. This makes it a good choice for temporal landing, where data volumes can be large and unpredictable.

2. Fault Tolerance: HDFS is designed to be resilient to failures. Data is automatically replicated across different nodes in the cluster, which means that if one node fails, the data is still available. This is crucial for temporal landing, where data loss can be costly.

3. Processing Speed: HDFS is optimized for batch processing, which is often the case in temporal landing scenarios. Data is stored in large blocks and distributed across the cluster, allowing for fast, parallel processing of large datasets.

4. Real-Time Data Ingestion: HDFS can handle streaming data, allowing us to store data as soon as it arrives. This is particularly beneficial in scenarios where data is being generated continuously and in different formats.

5. Format-Agnostic: HDFS is format-agnostic, meaning it can store data regardless of its format, be it JSON, CSV, or any other. This flexibility is crucial when dealing with data from various sources that may not always have the same format.

MongoDB

We considered several reasons for choosing MongoDB as the persistent landing for our data:

1. Schema-less: MongoDB is a NoSQL database, which means it does not require a fixed schema and can handle a variety of different data types. This is particularly useful when dealing with data that may change over time or when the structure of the data is not known in advance.

2. Scalability: MongoDB is designed to be scalable and can handle large amounts of data. It supports sharding, which allows you to distribute data across multiple machines.

3. Performance: MongoDB is known for its high performance. It uses a binary data format called BSON (similar to JSON) that allows for fast data parsing.

4. Flexibility: MongoDB allows for flexible querying and indexing. We can query and index the data in many different ways, which can be beneficial when dealing with complex data structures.

5. Replication: MongoDB supports built-in replication, providing high availability and data redundancy.

6. Integration: MongoDB provides drivers for many programming languages, making it easy to integrate with various applications.

In our case, each collection in MongoDB represents a version of the source, and each source has separate collections. This setup leverages MongoDB's flexibility and schema-less nature, allowing us to easily add new versions or sources as our data evolves.