Data Integration

Contents

[Overview 2](#_Toc43053692)

[1 – Pre-Processing 2](#_Toc43053693)

[2 – Normalization 2](#_Toc43053694)

[3 – Extraction 3](#_Toc43053695)

[4 – Integration 3](#_Toc43053696)

[5 – Product Matching 3](#_Toc43053697)

[5.1 Dynamic columns/Updates 3](#_Toc43053698)

[5.2 No upsert datastore 4](#_Toc43053699)

# Overview

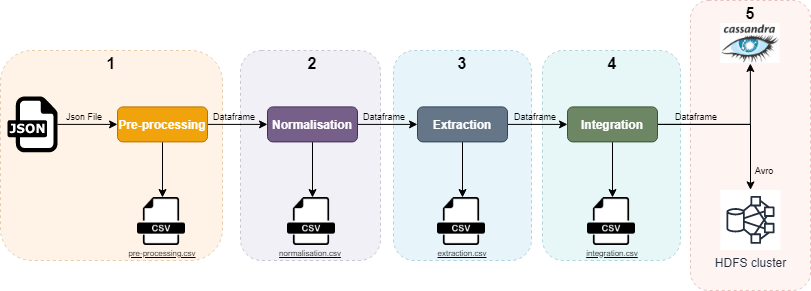
This project is an exercise to take in a JSON file as input, and submit it through a dataflow whereby each step is part of the integration process necessary for the input to match the intended data structure of the target database.

The implemented solution for the data ingestion pipeline was built using Spark and Scala, and is available on Github. The solution runs a Spark program in local mode, writing out CSV files according to the guideline for each step of the pipeline. 4 CSV files are stored for each step of the dataflow, in the output folder of the project:

1. pre-processing.csv
2. normalisation.csv
3. extraction.csv
4. integration.csv

The final step, product matching, is only theoretical and not implemented in the code. However options and solutions for it are discussed in more detail at [5 – Product Matching](#_5_–_Product).

The following diagram gives an overview of the entire process:

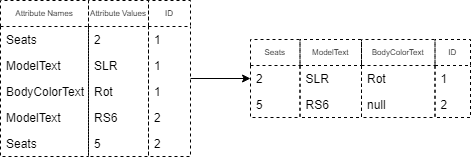


The project uses Maven for dependency management and project lifecycle. More instructions for building or running the project can be seen in the README file.

# 1 – Pre-Processing

The first part of the pipeline reads the json file present in the resources folder (supplier\_car.json), creating a dataframe from it. Following that, to achieve the wanted granularity of one row per product ID the dataframe is pivoted by grouping by the ID column before normalization, so that each attribute turns into its own column.

The following diagram shows an overview of the result of the function:



# 2 – Normalization

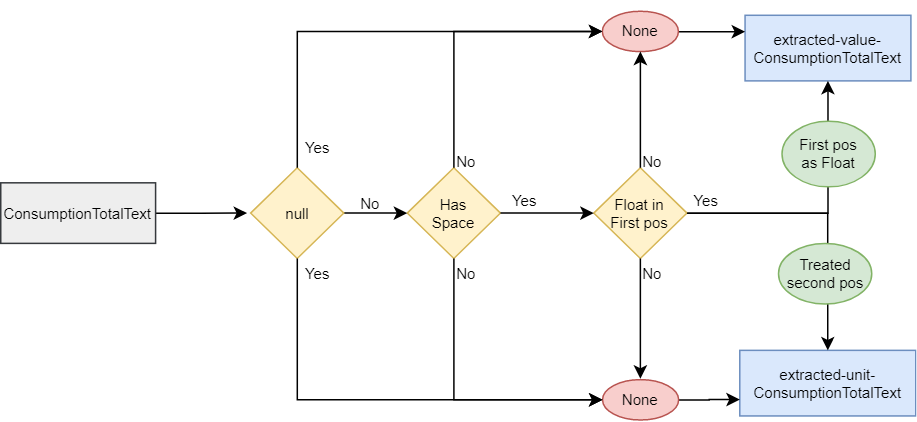
After processing the json file into a dataframe, the ‘BodyColorText’ and ‘MakeText’ columns are normalized. The first is normalized according to a hardcoded mapping of colors and translations in Scala, while the second is normalized according to a json mapping file (‘brand\_mapping.json’), with any brand not in that file being normalized according to a default function that treats the initial String.

The following table shows an example of the conversion of colors:

|  |  |  |
| --- | --- | --- |
| Initial Color | Treated Color | Final Color |
| Grün | grun | Green |
| null | null | None |
| grun mét. | grun mét | Green |
| Rosa | rosa | Other |

# 3 – Extraction

With the intended columns normalized, we retrieve the ‘extracted-value-ConsumptionTotalText’ and ‘extracted-unit-ConsumptionTotalText’ from the initial ‘ConsumptionTotalText’ attribute. The following flow exemplifies how the extraction function works:



# 4 – Integration

The final step of the process is a function that extracts and renames the intended columns for the final CSV, so that we only have the following columns:

|  |  |
| --- | --- |
| Initial Column | Final Column |
| BodyColorText | color |
| MakeText | make |
| ModelText | model |
| TypeName | model\_variant |
| City | city |

# 5 – Product Matching

Since the final database is not defined for this final task, I thought it best to go over two very different possible approaches depending on the final database.

## 5.1 Dynamic columns/Updates

If the final base is a NoSQL database like Cassandra and HBase, we have the possibility to use concepts like wide rows or dynamic columns to add info to already present rows in the final table as new cells.

As an example, when receiving a new attribute ‘sold’ to the product and using the above NoSQL stores we can dynamically create a cell for the ‘sold’ attribute by quite simply just upserting the new data into the table.

This way the product matching implementation can be agnostic to if the final product is present in the table or not. By just upserting incoming info we can enrich existing products if we happen to have new attributes or add new products if they are not existent.

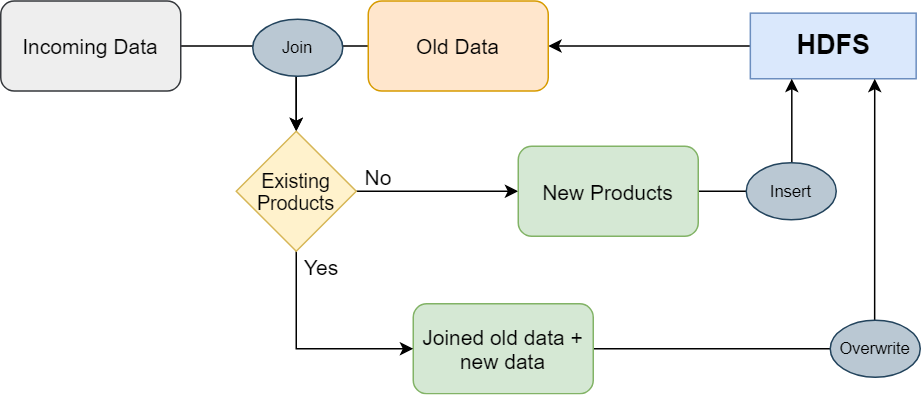
## 5.2 No upsert datastore

Thinking of the worst-case scenario, the final datastore could be rigid with no delete/update functionality. In this scenario, I will consider HDFS as the final datastore, and detail the simplest but also most expensive way to keep the database coherent.

Since HDFS files are immutable, we do not have the possibility to delete or update data. Therefore, we would need to completely re-write the final files in the case of an update with new attributes. To do this, we would need to join the incoming data with the final data to detect new info. Then this new info would need to be joined with the stored info and written over the previous files to not have duplicate info.

To handle an evolving table schema where new attributes are expected, we can add Avro versioning to the table so that old data is still readable without needing to re-write it, but new data can be stored with a newer schema.

The following diagram shows an overview of this process:



All this complexity could be reduced in a variety of ways, for example:

* Add partitions, so that the re-writes and joins are smaller
* Add cache for already stored products, so that the initial join and get is only done if we need to rewrite
* Come up with a data model where duplicates are accepted (“updated-time” column for example, to filter for outdated info)