Assignment documentation

# Setup

The assignments have been solved in Databricks, using Python and Spark 3.5.0, for big data work. The Medallion architecture was employed for this kind of task, meaning there are 3 layers in which the data is flowing: bronze layer, silver layer and gold layer. In bronze layer, there’s raw, unprocessed data. In silver layer, the data is cleaned and transformed accordingly. In gold layer, the data is aggregated and ready for consumption.

# Entity Resolution

## Data Exploration

The first step was to understand the data. Exploring its schema and the actual data, I’ve noticed key columns that might define a company unique: “company\_name”, “main\_country\_code”, “main\_country”, “main\_region”, “main\_city\_district”, “main\_city”, “main\_postcode”, “main\_street”, “main\_street\_number”, “main\_latitude”, “main\_longitude” and “last\_updated\_at”. In theory, main\_latitude and main\_longitude attributes should be enough, since it tells the exact location of a company (or any other place), but because the data is missing, the other attributes should make up for it. The last\_updated\_at column is also taken into account, because we want to have the most recent information, which should be most reliable.

Another interesting column was “locations”. It had all the columns mentioned above, aggregated by commas. To further make up for the missing data, locations has been split and used its components to replace missing entries. Because not all these factors were present in the locations columns, I’ve only used the first 3 elements, main\_country\_code, main\_country, main\_region, and last 2 elements, main\_latitude, main\_longitude.

## Silver Layer

In silver layer, the right data types were assigned, according to their columns. For instance the year\_founded, num\_locations and employee\_count were converted from string to integer. Columns like created\_at and last\_updated\_at were cast to timestamp and revenue, main\_latitude, main\_longitude to double.

## Gold Layer

In this final layer, the deduplication logic was applied. For that, row\_number function was used, along with proper partitions (“company\_name”, “main\_country\_code”, “main\_country”, “main\_region”, “main\_city\_district”, “main\_city”, “main\_postcode”, “main\_street”, “main\_street\_number”, “main\_latitude”, “main\_longitude”) and ordered by “last\_updated\_at”, descending, which lead to the removal of approximately 5000 duplicates.

# Product Deduplication

## Data Exploration

Same as for Entity Resolution challenge, first step was to understand the data. It was not as straightforward as Entity Resolution, because there were more relevant attributes to take into consideration. After exploration, I considered that ”root\_domain”, “product\_title”, “product\_name”, “product\_identifier”, along with product specifics columns like “materials”, “intended\_industries”, “price”, “color”, etc. were proper columns to take into account for deduplication.

## Silver Layer

In this layer, the columns that hold array type of values are ordered, so the deduplication logic doesn’t treat them as different factors. For instance, price value of [{"amount":639,"currency":"USD","type":"max"},{"amount":449,"currency":"USD","type":"min"}] would be different than [{"amount":449,"currency":"USD","type":"min"},{"amount":639,"currency":"USD","type":"max"}], which will result in two different products, so ordering the elements within an array is necessary.

## Gold Layer

Here, the same function row\_number was applied, but slightly different. As partitions, root\_domain, product\_title, product\_name and product\_identifier were used. The rest of the specifics were used in the order by clause. I used the size function to retrieve the number of elements of a column, because we want uniqueness, but we also want to keep as much information as possible, so the product that has the most information (number of elements within an array), will be kept. This approach led to removal of around 2000 duplicates.

# Suggestions of improvements

As mentioned in the challenge, Veridion is working with billions of records, so to improve the code:

1. partitions could be applied to the tables (e.g. date of ingestion or date of creation) for faster ingestion of large amounts of data
2. indexing common used columns (product name for products or company name for entities) for faster retrieval of data
3. enforcing schemas for tables to ensure data integrity
4. unit tests to ensure deduplication logic is working as expected