**Take home test**

**Data Processing and Programming**

Code

A code was built to run the ET(L) process of the given dataset. It can be found on pipeline.py.

This code consists of 3 parts:

* Extract  
  The parquet file is loaded into pandas and a name is given to the index
* Transform

This is the main part of the code. The strategy was to build 3 separate tables: parts, holes, and unreachable flags. As it will be seen in the ERD, these are the proposed tables:

* + Parts: contains all the parts of the dataset
  + Holes: contains only the holes of each part. Each hole has its features, such as center, length, directions, etc. Additionally, this table contains a column called part\_index, so that each hole can be associated with the respective part
  + Unreachable\_flags: contains, for each hole of the “Holes” table, the two required flags: *has\_unreachable\_hole\_warning* and *has\_unreachable\_hole\_error*.

The Transform process starts by taking the input dataset with the parts, looks at their “holes“ column, and extracts all the features of each hole. Since a part can have multiple holes, this has to be done for each one of those holes, so that we end up, for each part, with a dictionary with the following format: {'feature\_1': [value\_1, value\_2, ...], ...}

Where each the features are center\_x, center\_y, center\_z, direction\_x, direction\_y, direction\_z, end1\_closed, end1\_reachable, end2\_closed, end2\_reachable, facet\_count, length, and radius.

And each one of the values in the list corresponds to the feature of the respective hole.

For example, if a given part has 2 holes, its dictionary could look something like

{‘center\_x': [10, 15], ‘center\_y’: [4, 0], ‘center\_z’: [2, 12], ...}

This would mean that this part has one hole with center (10,4,2) and another with center (15,0,12). The rest of the features would appear in the dictionary, but this toy example simplifies the explanation.

Then, these dictionaries are transformed into the “holes” table, so that each row corresponds to a hole and each column to an extracted feature of the respective hole. Additionally, two columns are added: *hole\_index*, which is a unique identifier of the hole; and *part\_index*, which can be used to associate a hole with its respective part of the “parts” table.

Finally, a final table called “unreachable\_flags” is built. This table has 3 columns: *hole\_index* (which can be used to identify the hole), *has\_unreachable\_hole\_warning,* and *has\_unreachable\_hole\_error*, as required. This is straightforward once we have the “holes” table with the required features properly extracted for each hole.

* Load

Depending on the used dataset, this part can be used to load the 3 tables. To exemplify its use, the code is already written to load the datasets into a local postgres database.

To run the pipeline, we can run the following command:

*python pipeline.py --filename [filename] --write[optional]*

Where [filename] is be the filename (path) of the parquet file and --write is an optional flag. If this flag is given, the Load phase will be executed to load the 3 tables into the default database used for the exercise. If this flag is not written, the tables won’t be loaded into the default database, but will be instead saved locally as csv files.

The reasoning behind the 3 tables is because a part and a hole can be understood as individual entities. They both have features and characteristics that define themselves. That is why I decided to create a table for parts and a table for holes. Then, I could have created the *has\_unreachable\_hole\_warning* and *has\_unreachable\_hole\_error* columns in the “holes” table. However, as those definitions are based on parameters and are business-specific, I decided to have them in another table.

Optional: load tables into local database

The default database used to exemplify the Load phase is a local postgres database. It must be first initialized with Docker by typing:

*docker-compose up -d*

This will create a docker container with the Postgres database on port 5432 and the GUI pgadmin on port 8080. Pgadmin was included simply for visualization purposes.

After running the docker container, the local database is up and pipeline.py can be executed with the --write flag. When this is executed, we can go into pgadmin on localhost:8080, login with the chosen username and password (admin@admin.com and root), then open the database by adding a server with hostname pgdatabase on port 5432, username root and password root.

We will see the 3 tables correctly created:

A screenshot of a computer

Description automatically generated

The parts table looks like:

A screenshot of a computer

Description automatically generated

The holes table looks like:

A screenshot of a computer

Description automatically generated

The unreachable\_flags looks like:

A screenshot of a computer

Description automatically generated

Just to validate that the tables were correctly transformed and loaded, we can run a simple query to validate how many holes have unreachable hole warning and error:

A screenshot of a computer

Description automatically generated

So, 3020 (out of 135.015) holes have warning and 72 holes have error

The following files are given:

* pipeline.py: ETL process
* docker-compose.yml: docker configuration file to run locally the postgres database (optional)
* requirements.txt

Architecture

As explained above, the proposed architecture is as follows:

A diagram of a diagram

Description automatically generated

The database where the tables will be loaded depend on the required database. In the example, I created a local postgres database in Docker to exemplify this process.

The ERD diagram is:

A screenshot of a computer

Description automatically generated

What else to do?

The initial requirement would be fulfilled with the proposed architecture and ETL code. There are some additional things that could be done to make the process more reliable and complete. Some of these options would be:

* In this case I only checked the “holes” column of the original table. However, this dataset has many other columns that are also saved in a not convenient format such as json (or dict). Take for example job\_run\_time, latheability, machining\_directions, among others. Saving those variables in a relational database in json format is not ideal, so it would be a good idea to process those columns and split them into isolated values, just like it was done with the *holes* column.
* Do some preprocessing of the dataset and look for duplicate rows, missing values, formatting, etc. It is important to have consistency in the type of data and values of each column, so that the format is correct, and the values are stored cleanly.
* The conditions for a hole to be flagged as unreachable warning or error depends on some parameters (2, 10, and 40 in this case). If these parameters are not standard and can change from time to time, it would be a good idea to have a “parameters table” instead of hardcoding those values in the code. If we do that, by simply changing those values in the parameter table, the ETL process will keep working without even knowing that those values are now different.
* Assuming that this process would be needed to run recurrently to include more parts and holes, it would be needed to modify the code so that it can append new parts/holes or update them (if already present)
* Following the previous idea, if there is a possibility that a part/hole will be updated in the future with a new file, it would be a good idea to have an additional table with the historical data of each part/hole. If, for example, a part is modified, we would have the current part in the “parts” table, but we could have a snapshot of the respective part in another table with its modification date. By doing this, it would be very easy to analyze how entities have changed through time.
* It would be very important to modify the code so that it has error handling. The current code assumes that the input parquet file is stable in terms of column names and format. This could change and, if this happens, the code will probably fail. To have a more robust ETL process, it would be needed to do some error handling.
* Alerts. When putting the code into production, it would be a good idea to raise alerts to the stakeholders and users when/if the process fails. This could be a mail or a notification via Slack, for example. This would alarm the owner of the process that something went wrong, and some debugging and fixing is needed.

How to put the code into production?

This code must be put into production so it can be run automatically. To do that, the first requirement would be to define the database where the tables will be saved. After this definition, it would be needed to modify the code so that it receives the connection parameters. Afterwards, it is important to know if the load phase requires a simple “drop and replace”, or if it should be set as “append”.

I would follow a strategy based on the following items:

1. Containerization: I would recommend using a containerization tool such as Docker, so that everything that is needed is properly included and the code easily runs in the production environment.
2. Version Control: the code would be uploaded to version control such as Git. This helps in tracking changes, collaborating, and rolling back to previous versions if needed.
3. Configuration files and credentials: since the tables will be uploaded to a database, it will be needed to define credentials and some parameters. It will be better to have these credentials and parameters in a different file or environment variable so that they are easier to change. In terms of credentials, instead of having them hardcoded in the code file (which is a bad idea in terms of security), it would be a good idea to have them in a tool such as GCP Secret Manager.
4. Unit tests: unit tests must be written so that we are sure that the code is properly working when a new version is deployed.
5. Error handling and logging: it was already discussed in the previous section.
6. The code could be put into production with a tool such as Dagster. This tool would allow the code to run periodically based on how often it should be executed.
7. Documentation: it is very important to properly document the process so that it is easy to maintain and debug.

**Problem solving**

Question 1

We have seen that Data Warehouses have evolved and become more complex over the recent years. One thing we have noted is that the volume and variety of our data consumers is growing due to the presence of reverse-ETL, operational use-cases on analytics data and the development of ML pipelines.

When all these things co-exists “under the same roof” problems such as degrading performance in the database might surface. What other problems might this situation cause? How would you tackle this problem and which tools or frameworks would you use?

Other problems:

* Degrading performance  
  The more workload from different stakeholders and data consumers can produce a decrease in performance of the database, as more people are trying to access the data at the same time.
* Increased complexity  
  More data consumers imply that the database must be able to respond to more and more complex tasks. This increases the efforts needed to maintain the databases.
* Data Quality  
  The different use cases and data consumers could require different (and sometimes colliding) data quality requirements.
* Scalability  
  More data implies more challenges in terms of how to scale the database, especially if the database is in-premise and has fixed configurations.
* Resource priorization  
  Processes and workloads could “compete” in terms of resources.
* Security  
  As there are more data consumers, there is a higher risk of data leakage and data security threats. This translates in more complex access control policies.

How to tackle the problems:

* It is very important to work with optimized queries and have the **correct indices** in the tables. In my current role we had that problem while doing a query for one of our algorithms. When we checked the queries and indices, we realized that we needed to include a new index. As soon as we did that, the algorithm showed 10x decrease in time.  
  A basic, but very powerful tool is the EXPLAIN command in SQL.
* It could be also good idea to partitiontables to improve query performance.
* A thorough data governance strategy and documentation is critical
* Automatic monitoring and an alarm system
* Unit tests and validation checks to monitor consistency and robustness. Monitoring solutions.
* Scaling: it could be viable to consider cloud-based solutions, such as GCP o AWS that provide auto-scaling. By doing this, we can step up or step down our infrastructure according to the needs. Solutions such as BigQuery, Snowflake, etc.
* Tools such as Dagster or Kubernetes for resource management so that more critical tasks get the required resources.
* Caching common queries so that they are quicker to consult. Redis can be used for this
* Clear communication channels, such as Slack, Google Chat, or Teams.

Question 2

Another big challenge Data Engineering teams are currently facing is alignment with Data Producers. Although backend developers are highly knowledgeable on transactional data, for some reason there is a considerable gap between the transactional and analytical systems.

What problems might a poor alignment with data producers could cause? How would you tackle this problem and which tools or frameworks would you use?

Poor alignment with Data Producers could cause the following problems:

* Lags and lack of data availability  
  Data could be available for analytical systems with some delay that can be unacceptable. Some insights and models require data with minimal lag.
* Bugs, re-processes, and slow development time  
  Time could be needed to spend resolving conflicts, transforming data, and going back and forth.
* Data inconsistence  
  Data can be easily non consistent in terms of format, type of data, naming, conventions and, specially, data quality.
* Low business understanding  
  If there is no clear communication and alignment, it could be very hard to data engineers to understand the data that is being produced. On the other hand, the data producers can produce better data (in terms of what the analytical systems require) if there is a clear business understanding. Inaccurate transformations or aggrupations.
* Errors when extracting data  
  Very inefficient methods and processes to extract the data that can easily cause delays, re processes and strain on the databases and systems.

How to tackle the problem:

* Clear communication channels and ticket reporting (Jira, Slack, Teams)
* Data Governance: define ownerships with name and last name, including definitions.
* Documentation and dictionaries: have a central repository with metadata, explanations, definitions, etc.
* Work together: don’t consider the Data Engineering team and the Data Producers team as completely isolated teams. It is a good idea to have sessions where both teams get together and discuss what the final user is doing with the data.  
  It is also a good idea to get together to discuss the requirements of the analytical team that will be using the dataset.
* ERDs, Maps, and diagrams to show the data flow from origin to consumption, and its structure.
* Standardize data formats and definitions.
* Explore the possibility of real-time data streaming (Kafka, for example)
* Periodic and automatic data quality checks to ensure the data is aligned with the defined standards.
* Standardized ways to access data via APIs (OpenAPI)
* Use tools such as DBT