POLITECNICO DI TORINO

Master of Science in Data Science and Engineering

Master's degree thesis

Declarative Data Pipelines: implementing a logical model through automated code generation

Internal Supervisor:
prof. Paolo Garza

External Supervisor:
dott. David Greco

Candidate:
Matteo Donadio

Academic Year 2023/2024

Acknowledgements

Ringrazio il professor Paolo Garza per aver accettato di essere il relatore interno in questo progetto di tesi e per avermi seguito minuziosamente in questi mesi di lavoro e Agile Lab per avermi dato l'opportunità di svolgere un progetto di tesi così stimolante e di attualissima rilevanza. In particolare ringrazio David Greco, che mi ha seguito in qualità di tutor esterno con costanza e puntualità durante tutto il percorso, trasmettendomi passione e dedizione per questo lavoro.

Questi anni mi hanno dato tanto, mi hanno affascinato, mi hanno permesso di imparare, di sbagliare e di riuscire, ma sopratutto di non accontentarmi e dare sempre il massimo. Per cui ringrazio tutti gli sguardi che ho avuto modo di incontrare durante questo periodo e chi in un modo o nell'altro ha contribuito a far parte di questa avventura. Inifine il ringraziamento più speciale va alla mia famiglia per avermi dato la grande opportunità di studiare e alle persone che mi stanno accanto, che hanno avuto la voglia di condividere insieme un pezzetto di vita e che sono riuscite a dare valore al mio percorso.

Contents

Chapter 1

Introduction

1.1 About this work

The design and operation of data pipelines that deal with the extraction, transformation, and storage of large data sets are crucial in the field of data engineering. This thesis, developed in collaboration with Agile Lab S.R.L, introduces a logic model aimed at establishing a clear and standardized approach to data pipeline architecture, providing a structured framework for defining entities, their interrelationships, and the operational rules essential for building effective and reliable data pipelines. To bridge the gap between theoretical models and practical implementation, a tool that automates the generation of executable code for data pipelines, designed to work independently of specific data management tools, has also been implemented. It takes advantage of a declarative programming approach, allowing it to generate Python code for Apache Airflow, while maintaining the flexibility to adapt to other technologies as needed. Abstracting the complexities of configuration, it allows data engineers to focus on specifying goals and pipeline logic, significantly improving development efficiency and reducing the likelihood of errors. The usefulness of this model and its accompanying tool is demonstrated through a real-world

use case involving building a COVID-19 data analytics pipeline. This example highlights the tool's ability to adhere to the logical model and efficiently translate high-level design specifications into operational workflows, highlighting the tool's ability to enforce model-imposed constraints such as acyclicity, non-concurrency, and idempotency, ensuring the robustness and scalability of the pipeline.

1.1.1 Goals

This thesis aims to design and implement a logical model for data pipelines using a declarative approach, ensuring that these pipelines are efficiently automated and adaptable to various technologies. This model will manifest itself through a tool that automates the transformation of high-level declarative specifications into executable code.
Moreover, the overall goal is to standardize and simplify the pipeline creation process, allowing data engineers to focus more on strategic data processing goals and less on the technical specifics of pipeline implementation.
This work highlights progress in automated data pipeline generation, illustrating the benefits of a model-based approach that is not tied to a specific data management platform, and through the flexibility and adaptability of this approach ensure that it can evolve with technological advances in the field of data engineering, making it a versatile solution in developing complex data pipelines.

1.1.2 Thesis structure

The thesis is structured in the following chapters:

- *Chapter 1*: introduction to the work.
- *Chapter 2*: presentation of the infrastructure behind big data, the defi-

nition of data pipelines and the current tools used in building them.

- *Chapter 3*: detailed description of the logical model designed for building the data pipeline, explaining the entity structure, relationships and rules that ensure effective operation of the pipeline.

- *Chapter 4*: description of the development details of the tool that translates the declarative specifications into executable code, implementing the key constraints of the model.

- *Chapter 5*: evaluation of the performance and usefulness of the tool, providing quantitative analyses based on its application in different scenarios.

- *Chapter 6*: discussion of the implications of this work and potential improvements.

Chapter 2

Big Data Pipelines

2.1 Introduction to Big Data

The contemporary digital landscape brings the proliferation of social networks, connected devices and online activites that determine an era of unpreceded data generation. This chapter would be an introduction into the vast expanse of Big Data—a term encapsulating the diverse, intricate, and transformative world of information reshaping industries, economies, and societies worldwide.

Big Data are something huge that is impossible for traditional systems to process them and can be retrieved from different sources in structured, unstructured and semi-structured forms.

The particular and challenging characteristics that are intrinsic to them do not allow to work directly upon relational database management systems (RDBMS). So that, a large variety of new solutions have adopted. For instance, Hadoop[6] is a well known open source distributing data processing systems.

Big Data can be described by the following characteristics:

- *Volume*: The first immediate consequence of Big Data is the large quantity of data that every day have been produced from a lot of different sources, from social networks to IoT devices. Furthermore, organizations hold huge amount of log data, but do not have the capacity to process them. Building appropriate frameworks that allow to process them is the main attraction for many future purposes.

- *Velocity*: The term velocity refers to the increasing speed with which data is generated, processed, and analyzed in real-time scenarios. The constant need to ingest, process and derive insight from data that comes from to IoT devices streaming, must be managed in a fast and innovative manner.

- *Variety*: The continuous generation of data causes variety, and it is mean that is impossible considering all data correct. Data comes from different sources, in structured and unstructured formats, and it is not always possible to put them directly into the same dataset.

- *Veracity*: Data can be inconsistent with each other, causing unpredictable changes in structure and quality. In order to deal with this different dynamic nature, new data pipelines and flexible architectures are needed to ensure reliability and consistent analysis.

- *Value*: Value is the most important aspect in Big Data. It is important to define clear and specific objectives to extract insights from them. This require costly IT infrastructures that can only be implemented with a return from the investment.

Given the various facets that Big Data presents, their processing requires important technologies. There is no solution that is provided for every use case and that requires and has to be created and made in an effective manner

according to company demands. A big data solution must be developed and maintained in accordance with company demands so that it meets the needs of the company. A stable big data solution can be constructed and maintained in such a way that it can be used for the requested problem.

2.2 Big Data Architecture

Big data architecture would be a definite solution to deal with an enormous amount of data. It defines components, layers and methods for communication, making the ingestion, processing and storing data possible, forming a cohesive framework:

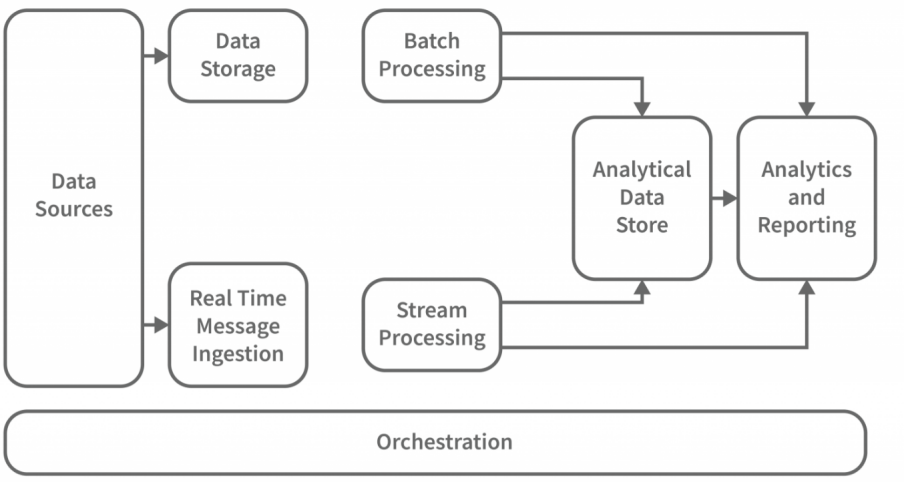

Figure 2.1: *Architecture of Big Data*

- *Data Sources* : Data sources can be open or third-party, represents the starting point in the architecture, playing a significant role. There is data stored in file stores that are distributed in nature and that can hold a variety of format-based big files. It is also possible to store large numbers of different format-based big files in the data lake.

- *Data Storage* : Data is stored in various distributed or local files that can hold a variety of format-based big files.
- *Batch Processing* : Data is divided in chunks which are split in different categories and prepared for the analysis with filters and aggregations.
- *Stream Processing* : The former consists in processing data by applying some preprocessing in order to prepare for analysis.
- *Analytical Data Store* : Data warehouse technologies are analytical data stores, based on HBase or any other NoSQL data warehouse technology. The data can be presented with the interactive use of a hive database, which can provide metadata abstraction in the data store.
- *Reporting and Analysis* : The insights captured by data can be represented in some reporting tools, that produces graphs, analysis and comments that can be useful for decisions at business level.
- *Orchestration* : The repetitive tasks are computed by some workflows, that convert source data, trasform them and load processed data into an analytical data store, or put data straight into a report or dashboard.

2.3 Data engineering lifecycle

As depicted in Figure 2.2, data engineering lifecycle can be seen as a subgroup of the entire data lifecycle, which underlies other fields such as data analytics, rather than the application of machine learning algorithms by ML engineers.

Let us now delve deeper into some of the processes discussed above, underlining the most important stages at the beginning of the life cycle of

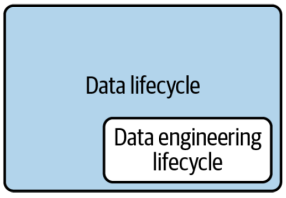

Figure 2.2: *Data engineering lifecycle is a subset of the full data lifecycle*

the data which will then lead to the detection of insights or the application of complex machine learning algorithms. The Figure 2.3 divides the data engineering lifecycle into 5 stages:

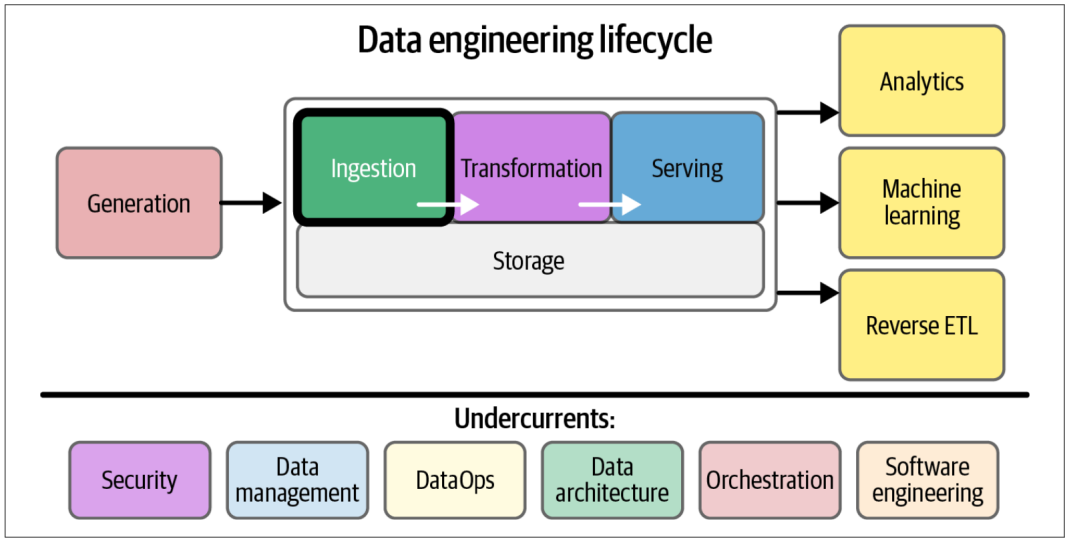

Figure 2.3: *Data engineering lifecycle*

- *Generation* : It is essential to have a good working understanding of the way source systems, which represents the beginning of the cycle. It can be a traditional source system with several application servers supported by a database or an IoT system that receives signals from smart devices and saves them in message queues. In addition to understanding the characteristics of the data source, the architecture built to manage the masses of big data must be able to manage the speed at which the data

is transmitted, and therefore the rate at which the data is generated and the memory occupied for now are essential elements to answer these types of questions. Furthermore, the consistency or presence of duplicates or missing data should not be underestimated, as well as the type of inferred schema and the frequency with which data is pulled from the source.

- *Storage* : Storage often occurrs in various steps of the entire data lifecycle, representing key and complicated stage of it. Dealing with storage means analyzing some key aspects that determine the quality of the final product, such as compatibility with required write and read speeds, the way in which storage works, that is, understanding whether to prioritize long term storage or frequent and fast reads, rather than the adopted schema, schemaless (schema on read) or fixed schema (schema on write). Finally, the choice of which storage system to use is not fixed, but each storage technology has its tradeoffs, which depend on various factors such as the data volume, format, size and frequency at which the data is ingested.

- *Ingestion* : From the data source we move on to collecting the same, and we enter the ingestion phase from the source system. These two phases can causes a ripple effect across the data lifecycle, often creating the bottleneck, since source systems are managed externally and ingestion can stop working at random times. The ingestion phase must respond to several factors that arise, such as the destination of the data at the end of the process, the frequency of access to the data from the source and its volume. But the greatest attention must be paid to two fundamental concepts: batch versus streaming.

  Batch has historically been the most popular method for moving data, particularly in analytics and ML, however we expect to see streaming trend to overtake it in the coming years. The question that must be

asked is what specific benefits do I gain from using streaming instead of batch, analyzing the differences in the form of cost, maintenance, time and opportunity cost.

- *Transformation* : The data transformation phase consists of changing data from its original form so that it becomes useful for reports, analysis, or ML. Making data useful for downstream use cases leads to extracting greater value from it, which would otherwise remain inert. In this way the data engineer can add value to business decisions, analyzing various phases of choice from the most basic, such as casting data to different data types, changing the schema or aggregating tables, to the most complex customizations to apply the business logic in a impactful way.

The final part of the lifecycle is interaction with stakeholders:

1. *Analytics*: includes published reports or dashboards, ad hoc analysis on the data. It can be further divided into BI, operational or embedded analytics.

2. *Machine Learning*: Includes the provision of data used for prediction or decision making purposes.

3. *Reverse ETL*: Involves feeding the result of the transformed data back into a source or another system for further use.

2.4 Data Integration

Suppose a company is able to retrieve the data it is interested in, but they reside in different data sources. It is necessary that information from all data sources is made easily available through a single platform, otherwise

without unified data single analysis it could lead to logging in with multiple accounts, or having the need to copy, format or clean them. One of the central points that must be taken into account is that the target system needs to have one common data model. This practice is rather challenging as source systems are rarely designed to be integrated, which means that additional adaptations and transformations are needed so that all the data can be represented in one common model. However, while this provides time and cost savings in the short-term, implementation can be hindered by numerous obstacles, such as the fact that the data comes from different sorts of sources such as videos, IoT devices, sensors, and cloud, and adapting the integration taking into account their variety, volume and speed can be a problem.
Integration can be done at different levels, based on the needs and size of the business and the resources available:

- *Manually* : an individual user will manually access the different source systems and interfaces directly, then cleans it up as needed, and combines it into one warehouse.

- *Middleware* : middelware applications can be used to provide reusable functionality and to reduce the work needed.

- *Application* : a single software application can locate, retrieve and integrate data, transmitting them from a source to an other and then creating a unified representation for the user.

- *Through uniform access* : data are left within the original sources and it will be created a front end that makes data appears consistent when accessed form different sources.

- *Using a common data storage* : data from different sources are loaded into a new common data storage for a unified view.

The ETL process has three distinct steps: extracting the data from the source system, transforming the data, and loading the data to the target data warehouse. The order of these steps depending on various factors that will be discussed later.

2.4.1 Extract

The first step of the ETL process involves the part of extracting data from different heterogeneous sources and converting them to a single suitable format for the transformation step, using a variety of frameworks that work with data of various formats, such as CSV, XML or JSON files. The design of the extraction phase should prioritize avoiding any detrimental impact on the source system's performance, response time, or any form of locking, since the rolling back, i.e. the process of undoing or reverting changes made to a database to return it to a previous state, might pose a challenge if corrupted data is directly transferred from the source to the data warehouse.

There are multiple ways to perform the extraction, the most common are:

- *Full extraction* : this method is commonly used when setting up initial data warehouse or there is a need to refresh the entire dataset. It involves extracting all the data from the source without applying any filters or conditions and keeping duplicate data.

- *Incremental extraction* : the load are used after a first initial load is performed at fixed intervals, recognizing which records have been changed and providing an extracting of such records instead of pulling the entire dataset again. This technique significantly reduces the amount of data transferred during extraction, making the process more efficient.

2.4.2 Transform

The goal of the transformation phase is to improve the quality of the source data, since the data were integrated from various sources, there are many unification measures needed. First, the data are cleaned by identifying and fixing (or removing) the existing problems in the data and prepares the data for integration, so as to solve the problem of so-called dirty data.
Some of the main tasks that are often performed during this step are data type conversion, splitting information, enriching the data by joining with other sources and deduplication.
When deciding what tool to use for the transformation, the options are either commercial ETL tools or coding manually with SQL stored procedures or other programming languages. ETL tools also have the advantage of taking care of the metadata creation, whereas when coding manually, this must be done by the programmer. In the next chapter we will dive into the description and usage of a common transformation tool called dbt.

2.4.3 Load

The load phase consists of loading the data from the staging area after the transformation phase into the target data warehouse. During this phase the data warehouse must be offline, and it is a key concept to find the right time interval to schedule loads without affecting users. To do this, a good solution might be to split the data into chunks and run smaller loads in parallel. A handling plan for files that undergo improper loading is also important, so that there are no inconsistencies between the contents of the fact and dimention tables.
Loading into the target data warehouse can be performed in different ways:

- *Initial Load* : this is typically the most common way to populate the

data warehouse table for the first time. Assuming there is the possibility to load the entire data warehouse in a single run, the correct procedure to maintain is to create the table from scratch for each load launched, after the entire data set has been divided into subloads.

- *Incremental Load* : incremental loading involves loading the changes that have occurred with respect to the previous run. To preserve periodic changes in the data warehouse, it is necessary to check if the primary key of the incoming record already exists in the database and update the contents, otherwise a new one must be created.

- *Full Refresh* : Full refresh is similar to initial load, with the difference that in this case data already exists in the data warehouse before incoming data is applied and it will be erased before applying the incoming data.

From a technical perspective, a full refresh is easier to implement becouse a proper strategy to adapt the changes like in updating is not required, but it is just needed to periodically replace the data warehouse tables.
From a technical perspective, performing a full refresh of data is simpler within a traditional data warehouse setting becouse a proper strategy to adapt the changes like in updating is not required, but it is just needed to periodically replace the data warehouse tables.
However, in the context of big data, where datasets are exceptionally large, this approach can become prohibitively challenging. The immutability of storage in big data environments means that reloading entire datasets repeatedly is not only impractical but also costly, so incremental updating is identified as the only viable method to maintain and manage data efficiently in such settings.

2.5 ETL and ELT

Traditionally, data warehouse solutions have employed ETL processes, where distinct systems handles data transformation and storage. The process consisted of performing the extraction and transformation processes from different data sources and then loading the data into the data warehouse.
Nowadays, the sequence of data extraction, transformation, and loading steps is flexible according to architectural and design choices, and one can still make a division into extract, transform and load (ETL), and extract, load, transform (ELT).
Differently from the traditional approach, in ELT processes data is extracted and loaded into the data warehouse prior to any transformation occurring within the data warehouse system itself. Some advantages of this sequence are that row data from the source system can be loaded to the target one faster, without requiring immediate transformation, facilitating quicker availability for advanced organizational analysis.
Furthermore, the order of steps is not the only difference, since in ETL the target store can be a data warehouse but also a data lake, that is the most used store since it allows both structured and unstructured data at massive scale. Additionally, staging raw data in the data warehouse decouples the transformation process, allowing alterations to the transformation logic without repeating the time-consuming extraction process. This decoupling also proves beneficial in scenarios where transformation errors halt execution. In ETL, such errors prevent any data from being loaded into the warehouse until fixed, while ELT allows for the loading of source data into staging tables, facilitating easier reruns of the transformation step post-error resolution. When using ETL, this would mean that no data is loaded to the data warehouse before the error is fixed. On the other hand, when using ELT, the source data is loaded to the staging tables, and the transformation step can then be more

easily rerun after the error has been fixed.
Given these reasons and the fact that the ELT approach is better able to support unstructured, real-time data, ELT has become a more popular method for data integration.

2.6 State of art of Data Engineering tools

Choosing and implementing practical ETL tools is crucial not only because they automate and simplify the complex work of processing Big Data, thus reducing the potential for human error, but also because they can have a significant impact on the scalability and efficiency of data systems. Effective ETL tools help organizations manage an increasing volume and variety of data, ensure data quality and consistency, and improve data security, but above all they play a critical role in integrating disparate data sources, which often represents one of the biggest challenges in data management. Therefore, selecting the right tools for ETL processes is not just a technical but a strategic decision that impacts the entire data lifecycle and, ultimately, business outcomes.
In the data engineering world there are various tools set more or less in detail for different tasks and operations, among which the following hold central importance both for their widespread use in real projects and for their utility.

2.6.1 Apache Airflow

Apache Airflow[13] is an open-source tool developed initially by Airbnb to manage complex computational workflows and data processing pipelines. Born out of the need to handle Airbnb's increasing data demands with a robust, scalable, and flexible system, it has grown into a widely adopted platform across various industries for programmatically authoring, scheduling, and monitoring workflows.

Figure 2.4: *Apache Airflow [13]*

Airflow is structured around several key components, as depicted in Figure 2.5 that enable it to manage complex workflows efficiently:

- *Scheduler*: it handles the triggering of scheduled tasks and ensures that tasks are started based on their scheduling and handles the logistics of task dependencies within the DAGs (Directed Acyclic Graphs). Essentially, the scheduler monitors all tasks and DAGs, decides when a task should run based on its dependencies and scheduling parameters, and then triggers the task execution.

- *Executor*: the Executor is the component responsible for executing the tasks that the scheduler deems ready to run. Airflow supports several types of executors: for instance, the LocalExecutor allows for parallel task execution on a single machine, while the CeleryExecutor can distribute tasks across a cluster of workers, ideal for production environments needing high scalability.

- *Worker*: they are the processes that actually execute the logic of the tasks. When the executor decides that a task should run, the worker performs the computation defined in the task.

- *Webserver*: it is a web application used to monitor and administer the Airflow environment, which is essential for operational monitoring and workflow maintenance. It provides a convenient and easy-to-use interface where users can view the status of DAGs, manually activate DAGs, check

completed tasks, view logs, and manage the configuration of their Airflow instance.

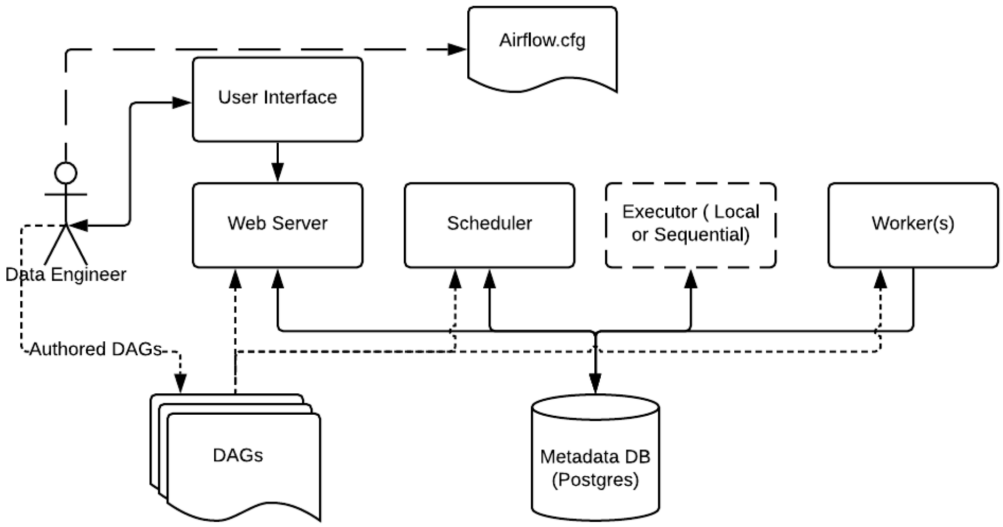

Figure 2.5: *Airflow basic architecture[2]*

Its simplicity of use, in which through special configuration files one can manage the installation of all necessary components, its high scalability and flexibility, and especially the ease of connecting Airflow to any other type of system to create more complex workflows makes it an optimal tool for etl pipelines processes.

2.6.2 Apache Spark

Apache Spark is an open-source distributed computing system designed to handle big data processing and analytics efficiently. Processing frameworks are critical components of Big Data systems, and for this reason, there are several on the market. Among these Hadoop's MapReduce[6] and Spark engine share several foundational principles, but Spark significantly outperforms MapReduce in terms of performance.

As processing framework, Spark is classified based on the type and condition of data they are designed to handle: it can process data in batches, streaming or both[21].

- *Spark Batch Processing Model*: A batch processing system gathers all data into a collections, which is then stored and processed at a later time. Spark's primary edge over MapReduce lies in its in-memory computation capability. Unlike MapReduce, which often writes intermediate data to disk, it only accesses disk to load the initial data into memory and to store the final results, so the intermediate data is processed in memory, significantly increasing performances. Furthermore, Spark's holistic optimization approach, which uses Directed Acyclic Graphs (DAGs) to pre-plan the entire set of tasks, contributes to its remarkable speed. To enable in-memory processing, Spark employs Resilient Distributed Datasets (RDDs). These read-only data structures reside in memory, ensuring fault tolerance without the need for continuous disk writes after each operation.

- *Spark Stream Processing Model*: Spark offers stream processing capabilities through the use of microbatches. In micro-batching, data streams are divided into very small batches, which are then processed sequentially by Spark's batch engine. While this method is effective, it may result in some performance differences compared to a dedicated stream processing framework.

Being one of the most used engines for data engineering, Spark has the following distinguishable features:

- *Speed*: it can be up to a hundred times faster than Hadoop for certain workloads and ten times faster than accessing data from disk. This is achieved through the use of Resilient Distributed Datasets (RDDs),

Figure 2.6: *Spark logo*

which allow data to be stored in memory, drastically reducing the time required for data processing tasks.

- *Usability*: Spark supports multiple programming languages, such as Python, enabling developers to write applications in languages they are most comfortable with.

- *In-Memory Computing*: spark's in-memory cluster computation capabilities enable it to execute iterative machine learning algorithms and interactive queries at lightning-fast speeds. By keeping data in the RAM of servers, Spark can quickly access and process data, enhancing performance for real-time data processing tasks.

- *Real-Time Stream Processing*: Spark Streaming facilitates real-time stream processing, making it a robust solution for handling live data streams.

Chapter 3

Logical Model

3.1 Technological landscape

The evolution of technologies useful in ETL processes spreads at an unprecedented rate, so that data engineers can effectively handle the most complex tasks with several helpful tools, which offer simplified solutions to handle the complexity of huge data sets.
However, despite this technological abundance, a significant challenge persists: *the strong alignment of each tool with a specific computational model.*
This specialization, while beneficial in some scenarios, has inadvertently led to a fragmented landscape in which the understanding and broader application of these tools is not formalized uniformly across different technology ecosystems, making challenging to apply a unified approach across different technologies.
Consider, for example, the landscape of data engineering tools where each is tailored to a specific computational paradigm. Dbt[4] is an excellent case in point, as it is works in conjunction with SQL computational model, therefore it only uses particular SQL engines, such as Snowflake or Spark, which only allowing you to manage structured data.

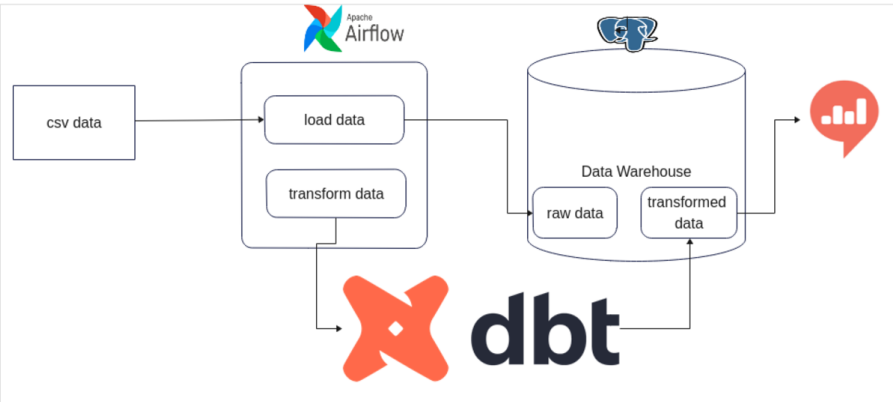

Figure 3.1: *Example of pipeline using Dbt*

Consider the scenario where a data engineer is tasked with developing a pipeline using dbt to process incoming binary data, such as images or machine-generated content, does not conform to the structured requirements necessitated by SQL-based tools. To integrate such data into a dbt-driven pipeline, one would typically need to transform this binary data into a format like Parquet, a process that requires auxiliary tools such as Apache Spark. This necessity to employ additional tools for data conversion illustrates a fundamental misalignment with dbt's capabilities, as it steps outside the expected relational data model framework.

This example underscores a critical limitation: dbt's dependency on SQL restricts its application to scenarios strictly involving structured data. It highlights the tool's inadequacy in directly managing diverse or unstructured data collections, which are increasingly prevalent in contemporary data-driven scenarios.

In response to these limitations, this thesis proposes a flexible, high-level logical model designed to abstract the concept of data collections and their relationships from any specific computational engine. This model aims to transcend the traditional boundaries set by specific data processing tools by providing a framework that can adapt to both structured and unstructured data seam-

lessly. This approach not only broadens the applicability of the model across various data types but also simplifies the integration of different data processing technologies without the need for intermediary transformation steps. By abstracting the handling of data collections at a higher level, the model ensures that the underlying computational specifics are decoupled from the data engineering logic, thereby enhancing flexibility and reducing dependency on any single technology or framework.

This versatility is particularly advantageous in modern data environments where the variety, velocity, and volume of data challenge the capabilities of traditional tools, focusing on decoupling the logical aspects of data engineering—such as data collection, transformation, and pipeline creation—from the technical specifics of task execution.

3.2 Objectives of the model

Against this backdrop, the proposed model tries to redefine the foundations of data engineering by establishing a unifying framework that is agnostic to the specificities of the technologies involved. The model's core objective is to distill and formalize the fundamental concepts inherent to data engineering—data collection, transformation, and pipeline construction—into a cohesive, technology-independent paradigm. This ambitious goal is not aimed at diminishing the value or utility of specialized tools but rather at providing a conceptual scaffold that ensures their capabilities can be harnessed more effectively and cohesively. By decoupling the logical planning and design of data processes from the intricacies of their technical execution, the model advocates for a more adaptable and resilient approach to data engineering. *It promises a paradigm shift from a fragmented landscape to a more integrated, principle-driven practice where the choice of tools and technologies is guided*

by strategic considerations rather than constrained by them. This initiative is not just about creating a theoretical construct; rather, it is aimed at building a practical and adaptable blueprint that can be adopted by any existing or future technology that will follow a more cohesive and standardized approach. A distinctive aspect of this proposed model is its focus on decoupling the logical aspects of data engineering from the technical nuances of task execution, making a clear separation between the high-level design of a data engineering task and the intricate details of how each component task is carried out. By establishing this distinction, the model promotes flexibility and adaptability, allowing data processes to be more technology-independent. The high-level logic of a job, under this paradigm, serves as a strategic blueprint, which can be implemented using different technologies, based on the specific requirements and constraints of the single project. So that, the actual construction of the pipeline remains reliant on the expertise and technical proficiency of the programmer, who will benefit from a standard structure within which data professionals can operate.

3.3 Implications of the model

The implications of implementing this model are profound and far-reaching for the domain of data engineering. On a practical level, it calls for a reevaluation of how tools and technologies are selected and applied, urging a move away from a reliance on specialized solutions towards a more versatile, principle-based approach. This shift not only enhances the adaptability of data engineering practices to the ever-evolving technological landscape but also fosters innovation by encouraging the exploration of new methodologies and the integration of emerging technologies.

Adopting a high-level logic approach inherently introduces a level of abstrac-

tion in tool implementation, leading to several critical implications:

- *Technological Neutrality* : By emphasizing the logical over the technical, the model fosters a foundational approach that remains effective regardless of the underlying technology. This neutrality ensures that the principles can be applied across a diverse range of computational models and data platforms.

- *Flexibility in Tool Implementation*: While the model provides a strategic blueprint for data engineering tasks, it inherently accepts that the specifics of tool implementation will vary. This variability is a consequence of the diverse capabilities and architectural nuances of different tools, as well as the unique requirements and constraints of individual projects.

- *Adaptability to Evolving Technologies*: The high-level logic framework is designed to be future-proof, accommodating the introduction of new tools and technologies. This adaptability is crucial in a field characterized by rapid technological advancements.

Furthermore, the model posits a significant transformation in the role of data engineers, who are envisioned as strategic architects of data solutions. In this capacity, they are empowered to design and execute data processes that are not only robust and scalable but also aligned with the strategic objectives of their organizations. By providing a standardized yet flexible framework, the model facilitates a deeper collaboration among data professionals, enabling them to leverage diverse tools and technologies more effectively. Ultimately, it aims to cultivate a more unified, efficient, and forward-looking field of data engineering, characterized by its ability to transcend technological barriers and drive meaningful innovation.

3.4 Core Entites

3.4.1 Data Collection

The aim of the model, as mentioned in the previous paragraph, is to find a standard for pipeline management that is capable of abstracting the most common way forward while meeting the constraints that a good data engineering project must satisfy.
Data Collection thus represents the basic element of modeling representation and should be seen as a high-level concept designed to go beyond the limitations of specific data types or storage methods. This core element recognizes the great diversity of data in the digital world, including relational, nonrelational, structured and unstructured forms, defining an open approach that aims to encompass the full range of potential data sources and formats in a unified framework.

The definition of a given collection can be split into two sub-entities:

- *Data Item*: A data item is conceptualized as the atomic unit of observation within the model, representing a singular record or observation, that could range from a binary blob or a JSON object to a row in a relational database table or even a standalone file.
  This opaque definition supports the integration of diverse data types into the data engineering process, ensuring that the model remains adaptable and inclusive of emerging data formats and structures.

- *Data Collection (DC)*: Building on the concept of data item, a data collection is defined as a collection of these items. For instance, it can be represented by a directory containing a bunch of files, or a table in a relational database. By treating a data collection as a directory of files, the model introduces a level of abstraction that simplifies the interaction

with complex datasets. This approach facilitates the management of data across different stages of the data engineering pipeline, from ingestion and storage to processing and analysis.

The model adoption of this opaque definition for data structure is a strategic choice designed to ensure that the data collection entity can seamlessly incorporate any kind of data. Thus, it opens a formal way of defining data containers that provides:

- *Enhancing Flexibility*: By abstracting away from the specifics of data formats, the model can effortlessly adapt to handle new and evolving types of data, ensuring that it remains relevant and effective in the face of technological advancements.

- *Standardizing Processing*: The high degree of abstraction and flexibility paves the way for the development of standardized processing techniques. These techniques can be applied universally across diverse datasets, fostering efficiency and coherence in data engineering practices.

3.4.2 Data Transformation

The transformation object represents a building block of the pipeline architecture that performs a specific operation on the data as it moves from source to destination. Each task, therefore, defines a granular piece of the data transformation process that is designed in a modular way, facilitating debugging and pipeline updating, since changes to one task do not necessarily impact the others, as long as the interfaces between tasks remain consistent. Moreover, single tasks can be arranged in sequences to perform complex transformations through a series of simple, ordered steps: this sequential arrangement ensures that data is transformed in a controlled and predictable manner, with each task building upon the outputs of its predecessors.

Formally, a *Transformation Object (TO)* can be defined as the embodiment of a single distinct activity within a data pipeline designed to perform a specific transformation on input data collections to produce a modified output data collection. It encapsulates not only the transformation logic, but also the metadata and rules required for its execution, ensuring that each TO is a modular and reusable component within the pipeline, capable of being independently configured and tested.
It can be represented as follows: given a set of n input data collections, a TO is a function that returns a single output data collection:

$$f_{TO} : (\mathrm{DC}_1, \ldots, \mathrm{DC}_n) \rightarrow \mathrm{DC}_{output}$$

The single output data collection that must be returned by the transformation object helps in enforcing data quality and integrity checks more effectively. With this approach, it becomes more straightforward to implement consistent validation, cleansing, and quality assurance practices across the pipeline.

3.4.3 Data Pipeline

A *Transformation Pipeline (TP)* is conceptualized as a sequential assembly of *Transformation Objects (TOs)*, each designed to perform discrete data manipulation tasks. This sequentiality is inherently mathematical, and can be designed as a composition of functions in which the output of one function (or TO) becomes the input of the next.
Let T_i be a specific TO within the set of all transformation tasks

$$T = \{T_1, T_2, \ldots, T_n\}$$

The definition of TP includes both a *denotational (declarative)* and a *operational* semantics.

The former focuses on the mathematical description and definition of data pipeline. Each transformation object T_i is a function that produces a specific output given a set of inputs data collections, then TP is an ordered TO sequence, where the output of the preceeding TO becomes the input of the next one.

Let the *Transformation Pipeline* be composed of n TO, it is represented as:

$$TP = T_n \circ T_{n-1} \circ \ldots \circ T_2 \circ T_1$$

where $\circ$ denotes the composition of functions, where T_1 is the first tranformation applied and T_n the last. In general, each T_i apply a transformation and "pass" the output to T_{i+1}.

However, since each transformation T_i can have more than one input data collection, the composition of function is not always direct as in the case of univariate functions. Indeed, the *partial composition* comes into play when we consider multiple inputs: for each input data collection of TO apply the partial composition of function keeping $n - 1$ data collections fixed

$$TP = T_n(\ldots(T_2(T_1(\mathrm{DC}_1, \ldots, \mathrm{DC}_1), \mathrm{DC}_2, \ldots), \ldots, \mathrm{DC}_n, \ldots)).$$

Beyond denotational semantics, operational semantics delves into the execution order of TOs, paying particular attention to dependencies among the transformations. From this perspective, it is crucial that TOs providing inputs to a specific TO are completed before the latter begins its processing.

This structure is crucial for understanding the pipeline's operational dynamics, emphasizing the acyclic requirement to avoid circular dependencies that can lead to processing loops or deadlocks, and for this reason, a graph formalization becomes necessary for a correct pipeline architecture.

According to Graph Theory[14], a *directed acyclic graph* G is defined as an unordered pair $G = (V, E)$, where:

- **V**: is the set of vertices.
- **E**: is the set of ordered pairs of vertices, known as edges. Each edge (u, v) is a directed edge from vertex u to vertex v.

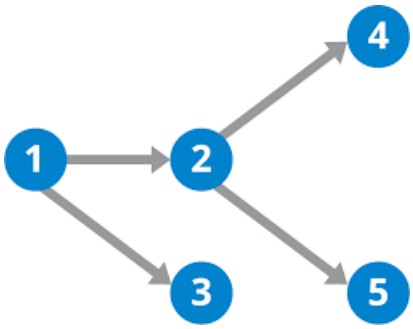

Figure 3.2: *Example of directed acyclic graph*

This approach assumes two fundamental properties:

- `Acyclicity` : The graph represents a valid pipeline if and only if it is acyclic. In particular, TP is acyclic if there are no paths starting from a node t_1 and returning to the same node t_1 following the direction of the arcs.

  Formally, there are no sequences of nodes $v_1, v_2, \ldots, v_k \in V$, with $k > 1$, such that $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k), (v_k, v_1) \in E$.

  In an ETL pipeline, the acyclicity constraint ensures that operations are executed in an order that does not create circular dependencies, preventing deadlocks and ensuring that each operation is executed only after the completion of all the operations on which it depends.

- `Idempotence` : a pipeline is idempotent if, when executed multiple times with the same set of inputs, it always produces the same final state,

regardless of any interruptions or errors that occur during execution.
Formally, for a pipeline P on an input set I, if $P(I)$ is the execution of P on I, then $P(I)$ is idempotent if, $\forall n \geq 1, P^n = P(I)$, where P^n denotes n consecutive executions of P. To ensure idempotency, each task in the pipeline must be designed so that it can be executed multiple times without undesirable side effects. This may include techniques such as version control for data, the use of transactions, or marking already completed tasks to avoid duplicate processing.

- `Non-Concurrency`: in a directed graph representing a data transformation pipeline, it is imperative that there are no multiple paths with data collections in common that allow concurrent writes to the same data collection. This principle ensures that no task within the pipeline attempts to write to a data collection in parallel with another task.
  Thus, for any data collection DC involved in the pipeline, there must not exist tasks t_i and t_j such that both tasks write to DC simultaneously. This non-concurrency constraint is crucial for maintaining data integrity and consistency within the pipeline. In practical terms, this means that each data collection can only be "active" or being written to by one task at a time, preventing race conditions and ensuring that data transformations are carried out in a controlled and orderly fashion.

In the presence of a failure during task execution, the pipeline must be able to stop and resume execution so that the end result is indistinguishable from an execution that has completed without interruption. This approach eliminates the need for a complex rollback mechanism and ensures that the pipeline is considered atomic, i.e. composed of a series of operations that are executed as an indivisible unit, meaning that either all operations are successfully completed, or none are applied. In case of pipeline failure, managing data recovery can prove to be a very complex process. To avoid having to run into this prob-

lem and make the model both simple and robust, we can implement a callback system for handling the data in case of failure. This approach aims to optimize the resilience and efficiency of the pipeline while reducing the complexity associated with data recovery.

The *callback* function is triggered to handle and clean up the partially transformed data, satisfying the idempotence constraint: the callback checks the output if there is data from previous processing and eliminates it if necessary. This ensures that no corrupt or incomplete data remains that could affect future pipeline executions. In the case of incremental execution, if a failure occurs, the callback function takes care of overwriting only the data that has been added or changed since the last transformation. This avoids the need to process the entire data set again.

By incorporating these concepts into the model, an increase in efficiency, flexibility, and resilience of the transformation pipeline can occur, while ensuring that data is handled safely and consistently even in the presence of errors or interruptions.

3.5 Model Limitations

This abstraction of the data collection entity within the logical model offers significant advantages in terms of flexibility and inclusivity, this approach is not without drawbacks: the model's high-level nature, designed to be technology-agnostic and broadly applicable, can inadvertently restrict its utility in scenarios that require deep integration with specific technologies or optimization for particular data formats.

Moreover, the trade-off between the high degree of openness and flexibility offered by a high-level model and the need for specialized, lower-level functionality is a critical consideration in the development of data engineering

tools. While the model's abstract nature fosters adaptability and inclusivity, enabling it to accommodate a wide range of data types and engineering scenarios, there is a clear need for a balanced approach. This approach would involve the integration of mechanisms or interfaces that allow for the incorporation of lower-level, data-specific optimizations and automations within the overarching high-level framework. Incorporating such mechanisms could mitigate the limitations of a purely high-level model, ensuring that data engineers have access to the best of both worlds: the flexibility to handle any data type in a standardized manner, and the ability to employ specialized functionalities and optimizations where necessary to enhance efficiency, performance, and automation in data engineering tasks.

Chapter 4

Tool Implementation

The upcoming chapter focuses on applying the logic model using a tool that generates the code structure of an ETL pipeline following the formalization and constraints described by the model in Chapter 3.
This tool is not merely an application of the model's principles but a bridge connecting the abstract realm of theory with the concrete demands of real-world data transformation tasks, so the development phase is driven by the recognition that while the logical model provides a robust foundation for understanding and structuring data transformation pipelines, its practical application requires a means through which users can easily translate these high-level concepts into operational realities.

The transition from model to implementation increases the probability for oversight and error, particularly when dealing with complex data sources, transformations, and storage mechanisms. Conceptual mistakes, such as inadvertently introducing cyclic dependencies, overlooking the idempotence of operations, or failing to prevent concurrent data mutations, can compromise the reliability and performance of the ETL process. Therefore, the tool addresses these challenges by offering guidance and validation at every step of the

development process, alerting programmers to potential issues and suggesting a code structure that align with the logical model's principles.

4.1 Declarative Programming

"In a nutshell, declarative programming consists of instructing a program on what needs to be done, instead of telling it how to do it."[18]

Instead of instructing the computer on how to achieve a certain task step by step (as in imperative programming[20]), declarative programming focuses on defining the desired outcome, allowing developers to express their intentions in a high-level, more abstract manner, leaving the specific details of execution to the underlying system or framework and presenting the following properties:

- *Expressiveness*: declarative programming allows developers to express their intentions in a clear and concise manner.
  By focusing on the outcome, the code often becomes more straightforward and easier to understand compared to imperative code that achieves the same result.

- *Abstraction of Control Flow*: in declarative programming, the control flow of the program is abstracted away. This means that developers do not need to write boilerplate code to control the execution order of operations, making the code more concise and focused on the domain problem.

- *Reduction of Side Effects*: declarative code typically minimizes side effects, which are changes in state that do not relate to the function's return value. Minimizing side effects leads to more predictable and testable code, as the outcome of a function or operation is dependent solely on its inputs.

The decision to employ a declarative approach in the thesis project stems from a strategic alignment with the high-level abstraction characteristic of data engineering pipelines, as detailed in the logical model. This choice was guided by the objective to provide programmers with a framework that simplifies adherence to the model's architectural principles, minimizing the need for direct engagement with the granular details of code syntax that typically accompany imperative programming methods.

As thoroughly explained in Section 4.3,the software utilizes a system where the actual executable code is generated automatically from an input YAML[1] file. This file serves as a declarative blueprint, outlining all necessary components and operations of the data engineering pipeline in a clear, concise manner. The YAML format was chosen for its ability to represent the desired outcomes and pipeline structure in a way that is both human-readable and machine-processable, offering an intuitive method for defining complex data transformations, data sources, and destinations without delving into the specifics of any programming language's syntax.

4.1.1 Advantages of the Declarative Programming

This type of programming offers some advantages for developers, like increased code *readability and simplicity*[5], since declarative code focuses on what the desired outcome should be instead of listing all the secondary details, and this results in a more easier way to express the concepts. Moreover, an *improved modularity and scalability*[5] that this type of approach brings makes it

[1]YAML, an acronym for "YAML Ain't Markup Language," is a data serialization format designed for human readability and interaction with programming languages, predominantly utilized in configuration files and data exchangeXML. `https://en.wikipedia.org/wiki/YAML`

easier to develop large software by separating the descriptive logic of software functionality from the control of it, that is, the set of technical mechanisms that are able to make the code working. This separation fosters the development of clean, modular code that is readily scalable, refactorable, and reusable.

In summary, the declarative approach for writing code is useful for the following reasons[5]:

- *Enhances Code Readability*: the code seems to be clear to understand, facilitating a comprehension of its functionalities.
- *Decrease the risk of errors*: since the complexity of implementation specific minimizes, also the number of bugs and accidental complication are reduced.
- *Improved Code Modularity*: the declarative approach helps to divide the processes into distincts and autonomous functionality units that allow developers to more efficiently organize the code.
- *Increased Code Reusability*: declarative programming's emphasis on higher-level abstractions leads to the creation of components that are easily repurposable across various scenarios.
- *Improved Scalability of Systems*: the declarative code typically lends itself to an easier parallelization and distribution over multiple processing units or nodes.

4.1.2 Disadvantages of the Declarative Programming

While declarative programming offers numerous benefits, such as enhanced readability and modularity, this kind of approach is not without its disadvantages:

- *Performance Reduction*: in some cases, declarative programming can lead to less efficient performance compared to imperative programming, since the control over the specific details is missing, and the underlying system or framework must interpret the declarative instructions and decide on the best course of action. This level of abstraction can sometimes result in less optimized performances.

- *Absence of Debugging*: the lack of explicit control flow and side effects, while beneficial for reducing bugs, can also make it harder to understand the path the program took to arrive at an incorrect state, so when a piece of code does not produce the desired outcome it can be difficult and disadvantageous to identify the source of the issue.

- *Size Restrictions*: for projects that require precise management of execution order, timing, or resource allocation, the declarative approach might feel restrictive since the level of abstraction can limit developers' ability to implement highly customized or optimized solutions.

4.2 Why Airflow?

The selection of *Apache Airflow*[13] as the orchestration tool for this project, detailed previously in Section 2.6, is rooted in its flexible and open architecture, which aligns perfectly with the principles and objectives set out in the project's logical model described in Chapter 3.
Airflow stands out as an orchestrator because of its ability to accommodate a wide variety of tools and technologies within its directed acyclic graphs (DAGs). This openness was a critical factor in the choice of the tool, as it ensures that the project is not tied to specific tools or data processing environments, and that is an essential feature for a project that aims to demonstrate a model that can be applied to different scenarios and data engineering tools.

Airflow's design philosophy is in line with the approach of flexibility and openness that this work seeks to pursue, in that it allows tasks and data flows to be defined in a way that abstracts the underlying complexities. The ability to maintain opacity in data collections and tasks within Airflow workflows means that the principles of the model can be seamlessly translated into practical and executable pipelines, ensuring that the freedom to use a specific technology, a central element of this work, is enhanced. The use of this software allows the project to benefit from an established, community-supported platform that improves the manageability and scalability of workflows. The choice of Airflow facilitates the practical demonstration of the workflow, showing its applicability in orchestrating complex data transformations and workflows by adhering to the high-level model with an abstract view of tasks and data collections.

4.3 Implementation Details

In this section, I will outline the technical methodology behind implementing our tool. It includes a concise explanation of the main technology choices, programming paradigms, and integration strategies that underlie the development of the tool.

4.3.1 Input Yaml File Structure

As previously introduced in section 4.1, this file is instrumental in translating the high-level, abstract concepts of our theoretical model into a structured, executable Airflow project. Below, a detailed structure and components of the input YAML file, chosen for its readability and ease of use, which allows programmers to define their data engineering workflows with precision and clarity and encapsulates the essence of the theoretical model while providing a practical means for its implementation.

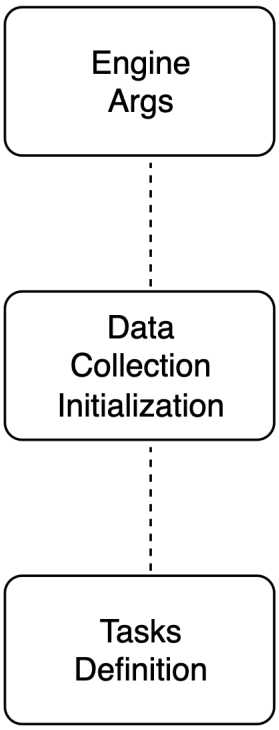

Figure 4.1: *Yaml File Strucure*

Figure 4.1 summarizes the structure of the file, which is divided into several key sections:

- *Engine Args*: it acts as a dictionary that allows the tool used, in this case Airflow, to set the basic parameters for a DAG [16]. It includes commonly shared settings, including designated owner and retry count, among others, eliminating the need to repeatedly define these common parameters for each new DAG, simplifying the process of creating multiple DAGs by automatically applying these universal settings.

```
default_args:
  owner: 'airflow'
  depends_on_past: False
  email_on_failure: False
```

```
    email_on_retry: False
    retries: 5
    retry_delay_min: 5
```

Listing 4.1: Default args example

- *Data Collections Initialization*: the input data collections of the first task must be instantiated separately. To be consistent with the theoretical model 3, the best way to represent a data collection 3.4.1 as an opaque object is to consider each of them to be a folder that contains files, a table in a database or whatever object storage, and which will then be represented by an array composed by the id of the data collection, which will correspond to the name of the instance that will be created by the tool, and a key-value object to indicate the metadata that characterizes it, such as the path where the files are located or other optional information.

  The simplest example it can be considered is represented in the following listing: assuming that the first task takes as input two data collections, A and B, it includes only the path where the files are located as a metadata description.

```
  data_collections: [
    ['data_collectionA': {path : './path/'}]
    ['data_collectionB': {path : './path/'}]
  ]
```

Listing 4.2: Data collections initialization example

- *Data Pipeline*: this entity is crucial in the model 3.4.3, and here it is represented as an object that contains dag parameters and task definitions.

- *Tasks*: The key *tasks* contains the declaration of each transformation object 3.4.2 and has as its value the specification of each *task*, which in turn is a key-value object with these features:
  - `id`: a unique identifier for the task, facilitating reference and management within the workflow.
  - `description`: a brief description of what the task is intended to accomplish, providing context for its purpose.
  - `input data collections`: specifies the names of input data collections for the task.
  - `output data collection`: defines the data collection object for the task's output. To adhere to the non-concurrency constraint of the model, the name of output data collection must be unique for each task.
  - `task function`: the Python function that will be generated by the code and that encapsulates the operational logic of the task.
  - `check output state`: a placeholder for a Python function that will be auto-generated that must be detailed by the programmer. This function will verify the output's state to ensure idempotence, as per the model's requirements.

```
id: task_1
description: 'First example task'
input_data_collections :
    ['data_collectionA','data_collectionB']
output_data_collection :
        ['data_collectionC', {path : './path/'}]
```

```
task_function: 'first_task_function'
check_output_state : 'output_state_function'
```

Listing 4.3: Single task example

4.3.2 Generator File

Once the input Yaml file is completed, the Python[19] code for the Airflow project structure is automatically generated, allowing a seamless transition from declarative specification to runnable code. The generator system is designed to serve as the foundational mechanism for constructing executable code tailored for Apache Airflow. At its core, this system leverages a generator script, tasked with transforming high-level declarative specifications into concrete Python code, facilitating a streamlined development process for complex data workflows.

As described in Figure 4.2, the code generation system is mainly composed of the *generator.py* file that transforms the input specified in the declarative file into executable code, which writes the generated code into two files:

- *dag.py*: it represents the direct output of the generator script, encapsulating the DAG definitions that Airflow will execute.

- *utils.py*: it contains the signatures of the functions critical for the tailored implementation of individual tasks within the workflow in the *dag.py* and for functions dedicated to ensuring the idempotence of the DAG.
  Given the generic and adaptable nature of this system, these function signatures are intentionally left blank, awaiting the programmer's specific implementations.

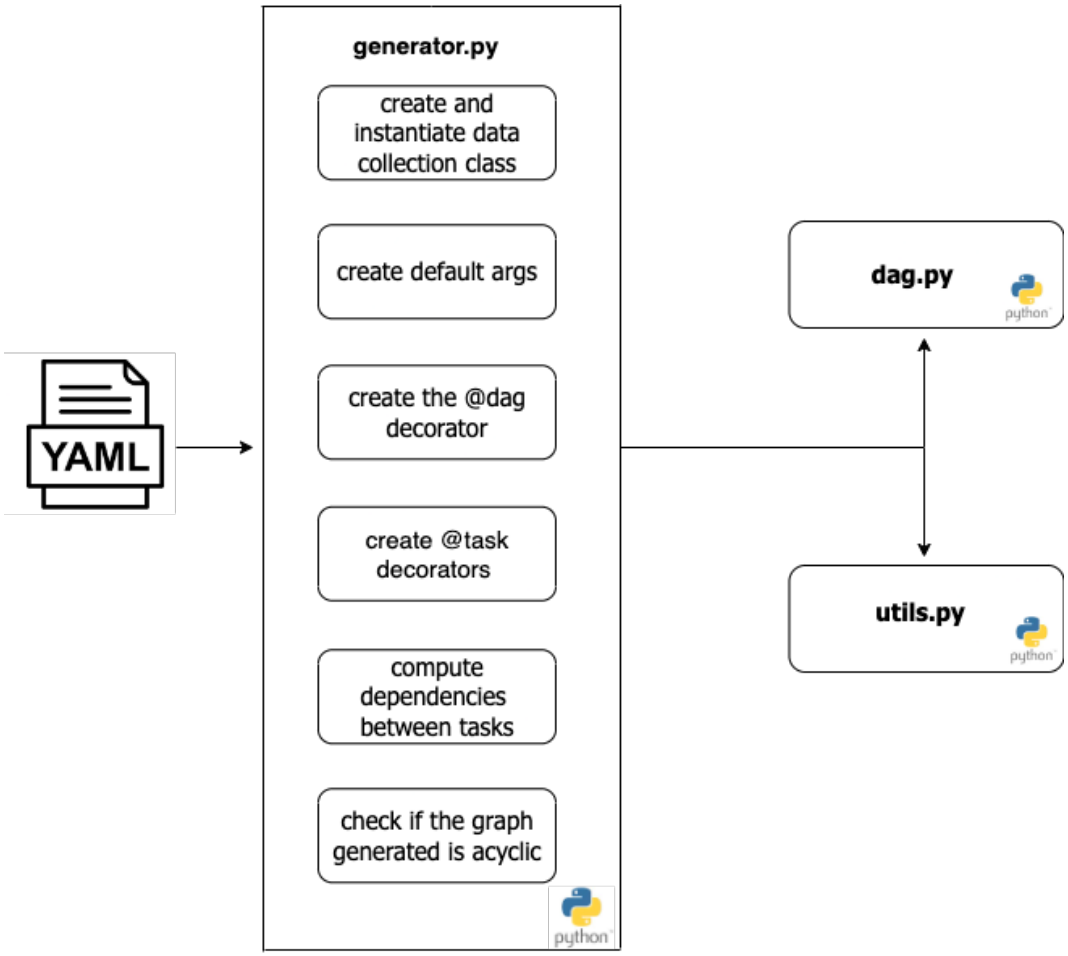

Figure 4.2: *Code Generator Structure*

The generator system is developed through several structured sections, each of which contributes to the generation of the above files:

- *Creation of a Data Collection Class*: the system creates a class that represents data collections, engineered to encapsulate essential attributes such as the name and metadata of the data collection, without any predefined methods, granting the programmer the freedom to implement methods that align with the specific demands of the project. This initial step involves parsing each data collection defined in the YAML file into instances of this class.

- *Creation of Default Args and Dag Decorator*: upon specifying parameters in the YAML file for default args and the dag, these configurations are directly translated into the generated code. As a programming choice, python decorators that are part of the Airflow Task Flow API are used as they improve the readability, maintainability, and efficiency of workflow development.

- *Creation of Tasks*: the generator system creates a task decorator function based on the input defined in the declarative file, which will call the callback explicitly coded within utils.py, to make the code more modular and manageable when tasks become complex, and allowing a clear separation of task logic from Airflow-specific declarations. Again, the use of decorators is motivated by the fact that this way tasks can be defined directly as Python functions, without the need to explicitly create Operator objects. This makes the code cleaner and easier to understand, especially for those familiar with Python.

- *Computing Dependencies between Tasks*: in keeping with the project's vision of placing data collections at the core of data pipelines, the generator system adopts a unique method for computing task dependencies baseing on the input and output data collections of the tasks. Rather than manually specifying dependencies, they are automatically inferred based on the input and output data collections of each task, ensuring a coherent and logical flow of data through the pipeline, with dependencies dynamically reflecting the actual data interactions. This automated dependency resolution significantly reduces the manual effort involved in pipeline configuration and helps prevent errors that could arise from manually managing task sequences. A more detailed algorithm for this step is specified in Algorithm 1.

- *Verifying Acyclicity of the Generated Graph*: by definition of DAG, the generated graph must be acyclic. In this regard, a check on acyclicity is done by the system, which will return an exception if positive. [2]

Algorithm 1 Compute Tasks Dependencies

```
1:  procedure FindDependencies(tasks)
2:      Initialize outputToTasks as an empty map
3:      Populate outputToTasks with task outputs as keys
4:      Initialize taskToUpstream to map each task to its dependencies
5:      for each task in tasks do
6:          for each input in task[input_data_collections] do
7:              if input in outputToTasks then
8:                  Map task[id] to tasks producing input in taskToUpstream
9:      dependencies ← list()
10:     for each task_id, upstreamTasks in taskToUpstream do
11:         if upstreamTasks is not empty then
12:             Sort and remove duplicates from upstreamTasks
13:             dependency ← format as a proper Airflow upstream
14:             Add dependency to dependencies
15:     return dependencies
```

By executing the command 4.3 in the terminal, users can initiate the code generation process:

```
~ python3 generator.py projectName pathYamlFile
```

Figure 4.3: *Command for the generation of Airflow Project*

The generated project will have the structure depicted in Figure 4.4:

[2]To verify the acyclicity of the generated graph, NetworkX library has been used. [7]

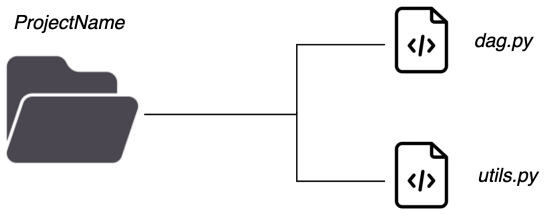

Figure 4.4: *Tree structure of the generated project*

In conclusion, the generator system presented in this chapter is a good starting point for merging high-level declarative logic with the practical requirements of workflow orchestration in Airflow. By automating the transition from YAML specifications to executable Python code, the system not only adheres to the principles of the logic model, but also significantly improves the development experience while respecting the modularity of the software. Ultimately, the generated code serves as a tangible manifestation of the logic model, embodying its ideals while providing a robust and flexible framework for executing complex data pipelines.

Chapter 5

Use Case

5.1 Project Overview

Once the general rules to follow for a correct setup of a given pipeline and the project structure that implements the theoretical model have been defined, in this chapter a complete application of the theoretical model and the generator tool, specifically focusing on simulating a data engineer's end-to-end pipeline using real datasets is provided.

The objective is to validate the practicality and effectiveness of both the model's guidelines and the automated capabilities of the generator tool in a real-world scenario, aiming to confirm that the theoretical principles laid out previously can be seamlessly translated into actionable, operational workflows within the framework of an Airflow-managed data pipeline.

By analyzing real datasets separately and then connecting them together in the form of tasks, we arrive at the simulation of a complete data engineering pipeline, from data entry to processing and output. This approach not only tests the adaptability and robustness of the model and tool, but also highlights their potential to facilitate correct and efficient pipeline development under real operating conditions.

In order to maintain a clear focus on evaluating the model and generator tool, the algorithms and data analysis components of the project are intentionally simplified, ensuring that the core objectives—assessing the utility of the model's rules and the structural efficacy of the generated Airflow project—are not overshadowed by complex data processing tasks, since the primary goal is *to illustrate how the model's guidelines and the automated project structure can support the development and management of effective data pipelines.*
Critical to the project's success is the use of Docker container[11] to configure and deploy Airflow[1], which ensures that the Airflow environment is consistent, reproducible, and isolated from external dependencies, aligning with the project's needs for reliability and control.

5.2 Project Structure

The structure of the project, depicated in Figure 5.1, is a direct result of inputs provided through the declarative YAML file, which specifies the data collections and tasks, and then the Python code is automatically generated by calling the *generator.py* file, which creates the structured project folder setup that Airflow can execute.

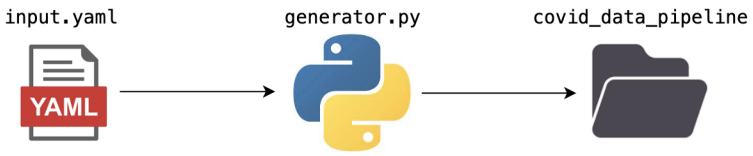

Figure 5.1: *Use Case Project Structure*

For this particular project, Apache Spark[10] was used to retrieve, trans-

[1]Setting up Airflow on a Docker container is particularly simple and fast with a Docker Container by following the specifications in the official documentation [9]

form and collect data, followed by storing the final data collection into a PostgreSQL[8] database, highlighting of the model's ability to integrate traditional database systems as endpoints for data storage, further emphasizing the model's adaptability. The choice to utilize both Apache Spark and PostgreSQL exemplifies the model's capacity to support diverse technological approaches in a single coherent workflow: Spark, renowned for its powerful data processing and transformation capabilities, is used to handle complex data operations efficiently, and PostgreSQL is utilized for its robust data storage capabilities, acting as the endpoint for storing transformed data.
This demonstrates the significant advantage of the model: its openness and lack of constraints regarding tool selection. This openness is essential for adapting the data pipeline to specific project requirements and for experimenting with different combinations of technologies to achieve optimal results and the abstraction allows for these tools to be interchanged or combined in myriad ways, fostering innovation and flexibility in data pipeline design.

5.2.1 Inital Data Collections

The use case topic is dedicated to the analysis and storage of COVID-19 related data. Utilizing two distinct yet complementary data collections, our pipeline is equipped to handle a rich array of pandemic-related information, providing a robust basis for our analytical tasks. These initial data collections are pivotal, as they set the stage for all subsequent data processing, analysis, and storage operations, enabling a detailed exploration of the pandemic from multiple perspectives.
In particular, the two initial data collections are the following:

1. *OWID COVID19*: The first dataset[1] offers a detailed and daily updated compilation of COVID-19 statistics, encompassing metrics like new daily figures for cases, deaths, and tests and it is particularly valuable for

its comprehensive nature and its breakdown of data by country, which provides a macroscopic view of the pandemic's evolution globally as well as nuanced, localized insights. A summary of a subset of features and data is reported in Table 5.1.

2. *COVID-19 Dataset*: The second dataset[15] aggregates extensive data on the coronavirus pandemic but focuses on the country-level breakdown of confirmed cases, deaths, active cases, and recoveries. Like the previous one, a subset of features and data is reported in Table 5.2.

Iso Code	Continent	Location	Date	Total Cases	Total Deaths
AFG	Asia	Afghanistan	2020-03-08	4.0	
AGO	Africa	Angola	2020-04-17	19.0	2.0
ALB	Europe	Albania	2020-04-12	433.0	178.0

Table 5.1: *OWID COVID19 sample data*

Country	Confirmed	Deaths	Recovered	New Cases	New Deaths
Afghanistan	36263	1269	25198	106	10
Albania	4880	144	2745	117	6
Algeria	27973	1163	18837	616	8

Table 5.2: *COVID-19 sample data*

5.2.2 Input file

```yaml
default_args:
  owner: 'matteo'
  depends_on_past: False
  email_on_failure: False
  email_on_retry: False
  retries: 5
  retry_delay_min: 5
data_collections: [['country_wise_covid',{'path': 'data/country_wise_covid/country_wise_latest.csv'}],
                   ['large_covid_data',{'path': 'data/large_covid_data/owid-covid-data.csv'}]]
data_pipeline:
  dag:
    dag_id: dag
    default_args: default_args
    schedule_interval: "None"
    description : 'Use Case dag'
    start_date : '2024,4,18'
  tasks:
    - id: task_1
      description: "First Use Case Task"
      input_data_collections : ['large_covid_data']
      output_data_collection : ['transformed_owid_covid_data', {'path': 'data/transformed_owid_covid_data'}]
      task_function: "transform_owid_data"
      check_output_state : "check_output_1"

    - id: task_2
      description: "Second Use Case Task"
      input_data_collections : ['transformed_owid_covid_data', 'country_wise_covid']
      output_data_collection : ['joined_data', {'path': 'data/joined_data'}]
      task_function: "join_data"
      check_output_state : "check_output_2"

    - id: task_3
      description: "Third Use Case Task"
      input_data_collections : ['joined_data']
      output_data_collection : ['best_recovery_countries', {'path': 'data/best_recovery_countries'}]
      task_function: "get_best_countries"
      check_output_state : "check_output_3"

    - id: task_4
      description: "Fourth Use Case Task"
      input_data_collections : ['best_recovery_countries']
      output_data_collection : ['db_data', {'path': 'data/db_data'}]
      task_function: "load_data_into_postgres"
      check_output_state : "check_output_4"
```

Figure 5.2: *Input Yaml file*

As previously detailed in Chapter 4, the YAML file serves the declarative backbone of the entire project, meticulously organizing and defining every aspect of the data pipeline, in such a way as to allow the developer to concentrate primarily on implementing the business logic rather than the orchestration of

the data flow.
Figure 5.2 reports the complete declarative file for the project, composed of four tasks that will be detailed in the next subsection. It contains the components necessary for a seamless and effective pipeline, including:

- *Initial Data Colletions*: the covid-related data collections described above.
- *Tasks*: it includes the input and output data collections for each task, and the function names that are placeholders that will be automatically generated by the tool, which programmer will subsequently flesh out with the necessary code to perform specific data transformations or analyses.

From the structuring definition of this file defined previously, Figure 5.2 does not have explicit dependencies between the tasks, simplifying the configuration process and reducing the potential for human error, as they will be deduced by the tool based on the input and output data collection present in each task.
This practical project case also wants to highlight the advantages deriving from the declarative approach, which stand out from the definition of the file in question, such as:

- *Reducing Complexity*: developers are spared the intricate details of pipeline orchestration, allowing them to focus on optimizing data processing logic and output.
- *Enhancing Productivity*: with the structural and flow dependencies handled automatically, developers can more quickly move from design to deployment, accelerating the overall project timeline.
- *Ensuring Consistency*: the declarative nature of the YAML file helps maintain consistency across the development cycle, ensuring that all

team members work with a clear, unified vision of the pipeline's architecture and objectives.

In summary, this structured approach significantly enhances the pipeline's manageability and adaptability, providing a clear pathway from conceptual design to practical implementation by detailing every element of the pipeline within this file.

5.2.3 Tasks

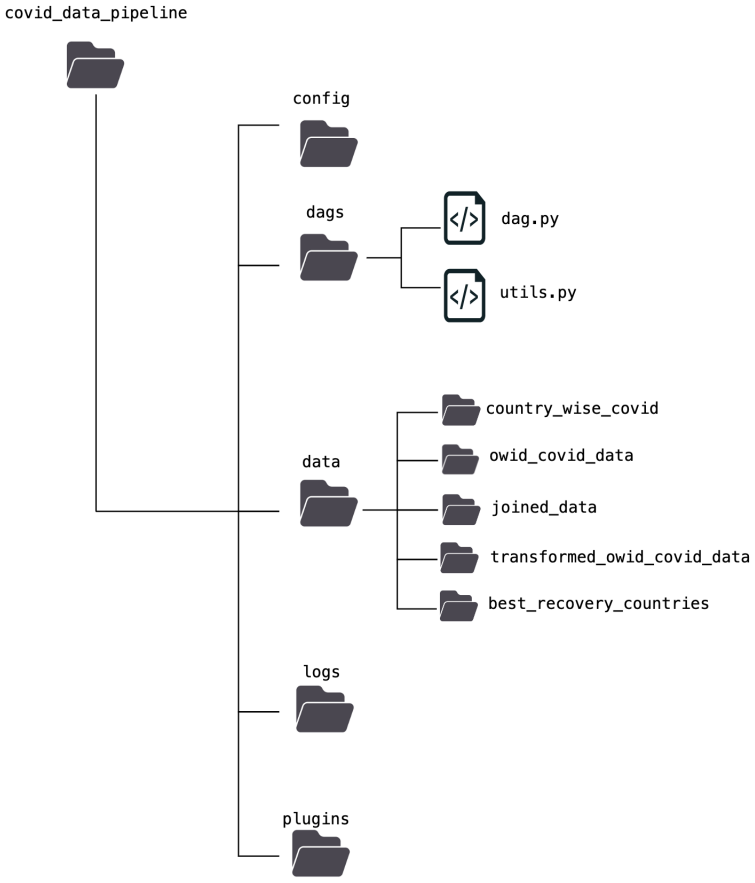

Figure 5.3: *Tree Structure of Use Case Project*

Once the declarative file has been defined, the generator file will generate the project structure, depicted in Figure 5.3, composed of the folders described

in Chapter 4 and the Python files that contain the dags and related functions for implementing the tasks and check the output state.
The project's structure, carefully outlined in the generated folders, is integral to managing the flow and storage of data across tasks. Moreover, each sub folder of *data* corresponds to a specific data collection involved in the pipeline, ensuring that outputs from one task are immediately accessible as inputs for the next, thereby maintaining data integrity and continuity. Notably, Figure 5.3 illustrates this data-centric organization clearly, showing how the tool's output is intrinsically oriented around data collections; for each output data collection, a distinct folder has been created, underscoring the tool's focus on data management and organization.

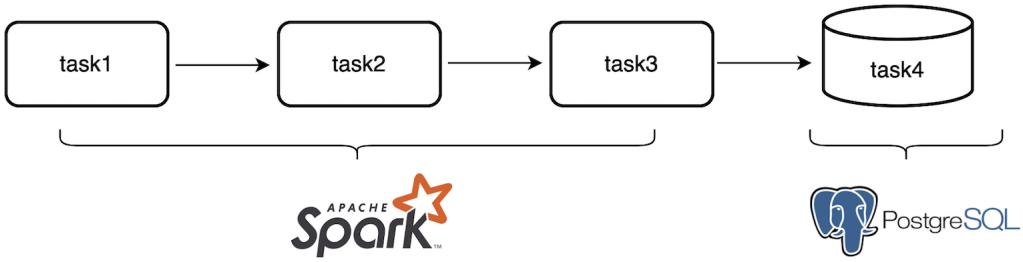

Figure 5.4: *Use Case Tasks*

Each task within the pipeline has a distinct role, with operations ranging from data cleaning to complex analyses and storage, utilizing Apache Spark[10] for processing and Python for interactions with a PostgreSQL[8] database.

The structure of the tasks is illustrated in 5.4. Each tasks plays a specific role in the data pipeline, in particular:

- *Task 1*: it concerns the analysis and cleaning of one of the two initial data collections: owid_covid_data. The analysis consists in streamlining the dataset by eliminating entries with missing values and performs the necessary aggregations to simplify the data structure, making it more

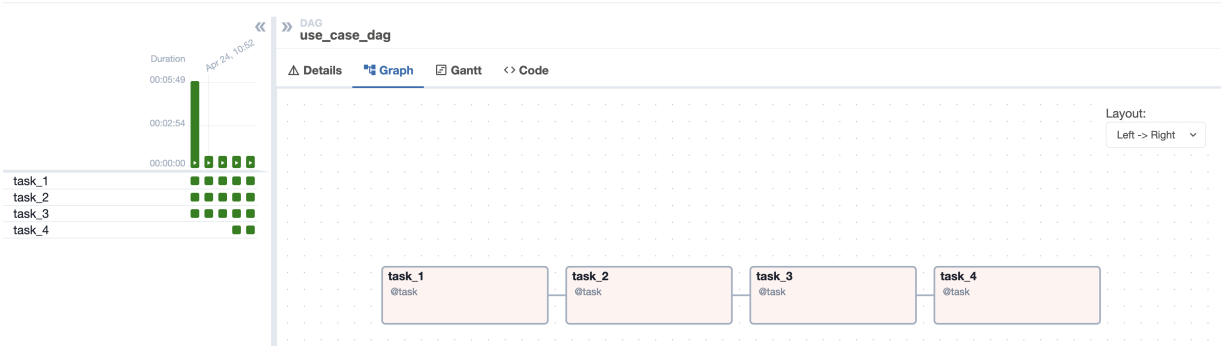

Figure 5.5: *Airflow Dag Successfully Completed*

manageable for downstream analysis.

- *Task 2*: it takes as input both the output data collection of Task1 and the other initialized data collection: country_wise_covid. It applies data transformations similar to those of the previous task, and then merges the two input data collections into a single one using a join, in such a way as to providing a unified dataset that encompasses comprehensive COVID-19 data across multiple dimensions.

- *Task 3*: from the unified data collection returned by Task2, Task3 is tasked with analyzing recovery rates, which involves calculating the ratio of recovered cases to confirmed cases for each country. This task emphasizes data reduction, focusing exclusively on recovery rates and related geographical information, thus preparing the data for targeted insights and decision-making, producing a dataset specifically tailored to recovery rate analysis, which is then directed to Task 4 for final storage.

- *Task 4*: the final task uses Python libraries to interface with PostgreSQL to store the output data collection as a table, ensuring the processed data is stored securely and systematically within a relational database, in order to preserve the integrity and availability of the data for future access and analysis.

5.2.4 Constraints Check

The generator tool embodies the principles of the theoretical model by rigorously adhering to the model's defined constraints. Prior to finalizing the code that will orchestrate the data pipeline in Apache Airflow, the tool meticulously checks to ensure that all constraints of the logical model —acyclicity, non-concurrency, and idempotency— are met. This proactive verification process not only reinforces the robustness of the pipeline but also ensures that the operational behavior of the system remains predictable and reliable under various execution scenarios.

The input file for this use case has no violations of these constraints, but if the tasks do not meet the theoretical guidelines, the tool will alert the programmer by throwing exceptions:

- *Acyclicity*: in Apache Airflow the data flow is represented as a Directed Acyclic Graph (DAG), so acyclicity takes on a twofold importance. During the DAG construction process, the generator tool systematically analyzes the relationships between tasks to ensure that the resulting graph remains acyclic and if the generator identifies that the task dependencies form a cycle, it raises a `CyclicGraphException`, an explicit error that informs the developer of the acyclic violation. This exception provides details about the tasks involved in the cycle, offering specific insights into the nature of the cyclic dependency.

- *Idempotence*: Idempotence ensures that repeated executions of a task under the same conditions produce identical results without causing unintended side effects. The generator tool facilitates adherence to this principle through its approach to code generation and task design by creating the signature of a specified function, as detailed in Chapter 4. Given the model's high level of abstraction, it is intentionally leaving the

implementation of the function's body to the programmer, allowing to tailor the function to meet the idempotence requirements of their specific use case.

In this project, the function `check_output_state` is designed to perform specific actions that confirm the idempotency of each task by iterating over the files and subfolders associated with a specific collection of output data, so as to identify any residual files or partially written data resulting from previously failed task executions.

When such files are detected, the function performs cleanup operations to remove or correct them. This ensures that the task environment is restored to a clean state before each execution, preventing residuals from previous executions from affecting the current task operation.

- *Non-Concurrency*: this requirement is encapsulated in the non-concurrency constraint of the logical model, which mandates that each output data collection should be associated with a single, unique task, ensuring that tasks do not simultaneously write to the same data collection. Consider a scenario where the non-concurrency constraint is inadvertently violated; for instance, if the output of a certain task presents more than one data collection in the declarative file, this setup could lead to simultaneous writes to the same data collections by multiple tasks, posing significant risks of data corruption or loss, and undermining the pipeline's reliability.

  For that reason, the tool will raise a `MultipleOutputDataCollectionsException`.

Chapter 6

Conclusions

6.1 Implications

The development of a logical model and tool for declarative data pipeline automation addresses critical industry challenges, notably the rapid obsolescence of data technology platforms. Current data engineering practices often involve significant investments in technologies that may quickly become outdated, leading to substantial economic and computational burdens during migration to new platforms, and the purpose of this work significantly alleviates these challenges by rendering data platforms more technologically agnostic. The implications of this work are profound, as it enables organizations to deploy data platforms that are insulated from the rapid advancements and changes in underlying technologies by abstracting the complexity of data pipeline creation through a high-level, declarative approach, minimizing dependency on any specific technology.

This approach aligns with the vision implemented by Agile Lab, which advocates for "shift-left" in data engineering. This concept involves tackling complex data processing issues with high levels of abstraction that remain resilient against shifts in technology, yielding substantial benefits such as reduced

lock-in and lower overall costs in technology evolution.

6.2 Future Works

The declarative approach detailed in this thesis suggests several avenues for further improvements of the tool and model:

- *Expansion for Multiple Workflows and Orchestrators*: future enhancements could enable the tool to support additional workflows and integrate with various orchestrators like Dagster[3], catering to diverse programming needs and environments, increasing the tool's versatility and its adoption across different data engineering contexts.

- *Automated Callbacks for Idempotence*: there is the possibility to automate the generation of code for managing idempotence, particularly through callbacks defined in the YAML file. Automating this aspect would reduce the manual coding required for ensuring idempotence, thereby enhancing the reliability and reproducibility of data pipelines.

- *Refinement of Data Collection Specifications*: expanding the declarative model to include detailed specifications of data collection types, such as distinguishing between file-based and database-oriented collections, could significantly enhance the tool's utility. By providing the tool with more detailed context about the nature of data collections, it could automatically generate more effective management and cleanup processes, further reducing the need for manual intervention.

- *Semantic Enrichment of the Declarative Model*: introducing more semantic depth to the declarative model could address even more complex data engineering scenarios, enhancing the tool's capability to handle diverse and intricate data operations with minimal configuration.

These potential enhancements not only extend the capabilities of the developed tool but also open new horizons for addressing the evolving needs of data engineering, further minimizing the gap between theoretical models and practical, scalable implementations in the field.

Bibliography

[1]

[2] Basic airflow architecture. https://airflow.apache.org/docs/apache-airflow/2.0.1/concepts.html.

[3] Dagster. https://dagster.io/.

[4] Dbt. https://www.getdbt.com.

[5] Declarative programming. https://www.studysmarter.co.uk/explanations/computer-science/computer-programming/declarative-programming/.

[6] Hadoop. https://hadoop.apache.org/.

[7] Networkx. https://networkx.org/.

[8] Postgresql. https://www.postgresql.org/.

[9] Running airflow in docker. https://airflow.apache.org/docs/apache-airflow/stable/howto/docker-compose/index.html.

[10] Spark. https://spark.apache.org/.

[11] What is a container? — app containerization — docker. https://www.docker.com/resources/what-container/.

[12] What is elt (extract, load, transform)? *IBM*, 2022.

[13] Apache Software Foundation. Apache airflow - programmatically author, schedule and monitor workflows. https://airflow.apache.org/.

[14] Wikipedia contributors. Graph theory — wikipedia the free encyclopedia, 2024.

[15] K. P. Devakumar. COVID-19 Virus Report Dataset, 2020.

[16] Murat Özcan. Airflow default args. https://medium.com/@muratozcann/using-default-args-in-airflow-make-your-workflows-more-efficien~:text=What%20is%20default_args%3F,retries%20when%20creating%20a%20DAG.

[17] Dom N. The data engineering lifecycle. 2022.

[18] Federico Pereiro. Declarative programming: Is it a real thing? https://www.toptal.com/software/declarative-programming.

[19] Python Software Foundation. Python programming language – official website, 2024.

[20] Benoit Ruiz. Declarative vs imperative. https://dev.to/ruizb/declarative-vs-imperative-4a7l.

[21] Eman Shaikh, Iman Ahmed Mohiuddin, Yasmeen Alufaisan, and Irum Nahvi. Apache spark: A big data processing engine. In *2019 2nd IEEE Middle East and North Africa COMMunications Conference (MENACOMM)*, Prince Mohammad University, November 2019. IEEE.