Van-Nguyen Dinh, Hoang-Anh Nguyen

DATA PROCESS APPROACHES BY TRADITIONAL AND CLOUD SERVICES METHODOLOGIES

School of Technology
2022

VAASAN AMMATTIKORKEAKOULU
VAASA UNIVERSITY OF APPLIED SCIENCES
Bachelor of Information Technology

ABSTRACT

Author	Van-Nguyen Dinh, Hoang-Anh Nguyen
Title	Data Process Approach by Traditional and Cloud Services Methodologies
Year	2022
Language	English
Pages	81
Name of Supervisor	Timo Kankaanpää

The thesis examines the construction of data processing solutions for large data sets (over one million rows of data) utilising two distinct data processing pipeline architects. The purpose of this thesis was to develop and implement a process for extracting, transforming, and loading (ETL) raw data from a variety of different data sources into meaningful and useful information in a data warehouse based on enterprise resource planning (ERP) models. Action research was used in this thesis, as well as data mining and cleaning, as well as numerous methods of inspection for extraction. It compares two primary data storage services: traditional databases and cloud-based virtual computing instances with support for data orchestration. Testing techniques were used to analyse and certify the ETL specification.

The research accomplished the primary goals of data integration and modelling by using a variety of technologies on a data instruction basis. The upstream departments will utilise the output of the process in conjunction with their own use to address various difficulties.

This thesis provides deployment techniques, introduces libraries, and demonstrates how to use these libraries for data integration throughout the system's development. The deployment procedure was developed with the goal of resolving real-world difficulties. The result demonstrates both automated and non-automated ETL approaches.

Keywords	ETL, Data Integration, Data warehouse, Data orchestration, database

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

LIST OF CODE SNIPPETS

LIST OF ABBREVIATIONS

ETL	Extract, Transform, Load
ELT	Extract, Load, Transform
AWS	Amazon Web Services
S3	Simple Storage Service
EMR	Elastic MapReduce
EC2	Elastic Compute Cloud
UI	User Interfaces
DAG	Directed Acyclic Graph
ERP	Enterprise Resource Planning
.py	Python format file
.ipynb	IPython Notebook format file
SQL	Structured Query Language
CSV	Comma-separated values
IAM	Identity and access management
JSON	JavaScript Object Notation
HDFS	Hadoop distributed file systems
CPU	Central Processing Unit
.pem	Privacy-Enhanced Mail (key encryption services for email)

API	Application Programming Interface
ERD	Entity Relationship Diagram

1 INTRODUCTION

Data is the foundation of the modern corporate environment. Information is developed from scratch using data and is used to get insights and make decisions about a company's operations. Data has been utilised to provide answers to queries and to solve issues in the past. With the use of data analysis and data sciences, the insights are assisting in risk management, planning, and taking decisions, while also allowing for deeper analyses to be conducted.

This thesis is divided into seven chapters. The first section creates the framework for the investigation, which includes the data integration process, the reason for the research purpose, the methodological approach, and the overall scope of the project. The second section defines the methodology for the study. Chapter 2, which examines the literature review and technical tools, will provide an explanation of the technical words that are utilised throughout this thesis. It provides a short explanation of each word and outlines which aspects of the project each team member is responsible for, as well as the functionality of the services. Chapter 3 describes the techniques used in this project. It displays how we identified data with acceptable size and property requirements for the perspective of our project. Meanwhile, it explains how to do a data analysis and how to continue with data modelling using the Entity relationship diagram (ERD) and the methods necessary to finish it. The fourth chapter focuses on two distinct techniques that we implemented and provides basic overviews of each method, framework, and service. The following chapters 5 and 6 demonstrate how the output data may be used in real-world contexts with a variety of practical analysis and financial summaries for budget computation of cloud cluster services using Amazon Web Service (AWS). The last chapter, Chapter 7, discusses the role of implementing new advancements to increase the performance of ETL processes in optimization in real-life scenarios, as well as the problems occurring during the study.

1.1 Background

A data integration process combines data from several sources into a single, consistent data store, which is subsequently loaded into a data warehouse or other destination system. The acronym ETL stands for extract, transform, and load. Traditional activities, such as manual ETL coding and data purification are being phased out in favour of new technologies and self-service pipelines. Data engineers can dedicate more time to essential data strategy and pipeline enhancement activities using Snowflake's straightforward ETL and ELT options.

Furthermore, since no pre-transformations or pre-schemas are necessary when utilising Snowflake Data Cloud as a data lake and data warehouse, ETL may be effectively eliminated.

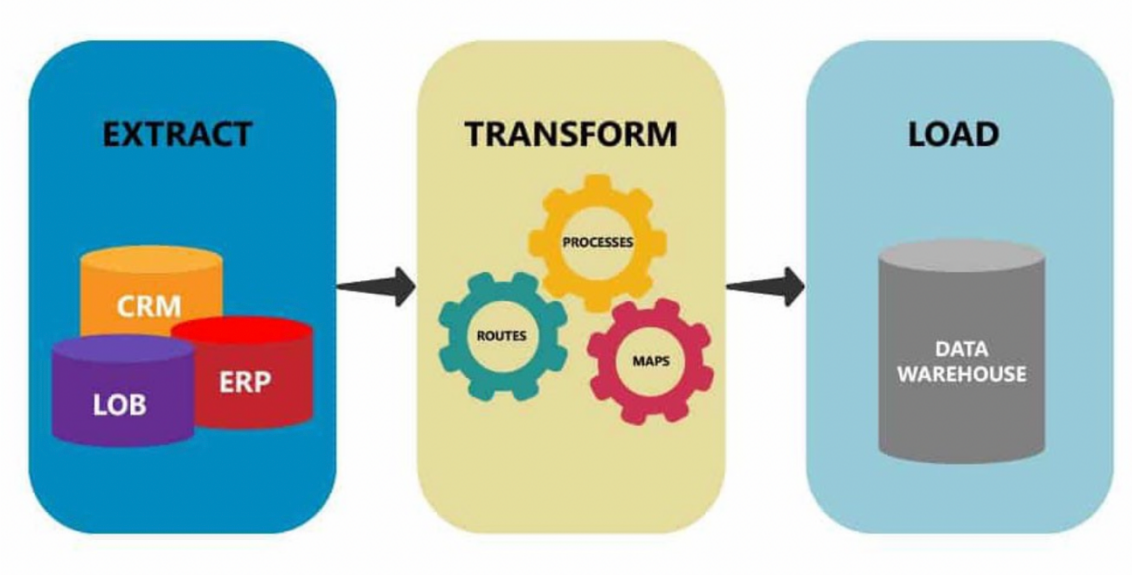

Figure 1. ETL process overview (Kasaraneni 2020, online)

Data orchestration is a relatively new field of study in computer engineering, especially as cloud computing and data storage technologies become more integrated. Data orchestration automates processes related to managing data, such as bringing data together from multiple sources, combining it, and preparing it for data analysis. It can also include tasks such as provisioning

resources and monitoring. While data management has been a point of contention in systems administration for decades, it has not always been as effective as it may be. One example is the concept of data curation, which entails assembling the most pertinent data for the most pertinent purpose. (Orlando 2021)

Orchestration solutions for data centres automate the setup of network services, computation, and storage for real, virtual, and hybrid networks. New applications can be deployed fast.

The following are some of the advantages of data centre orchestration systems: streamlined provisioning of private and public cloud resources.

There is less time to assess the difference between a business demand and when the infrastructure can supply the need. IT departments will have less time to provide a domain-specific environment. Data centre orchestration solutions provide a framework for controlling the data transferred between a business process and an application. They do the following tasks:

- Data service scheduling and synchronisation
- Large data collections may benefit from the use of a distributed data repository.
- Tracking and publishing APIs for automated metadata management changes.
- Policy enforcement is being updated, and notifications for corrupted data are being sent. Data services are being integrated with cloud services.

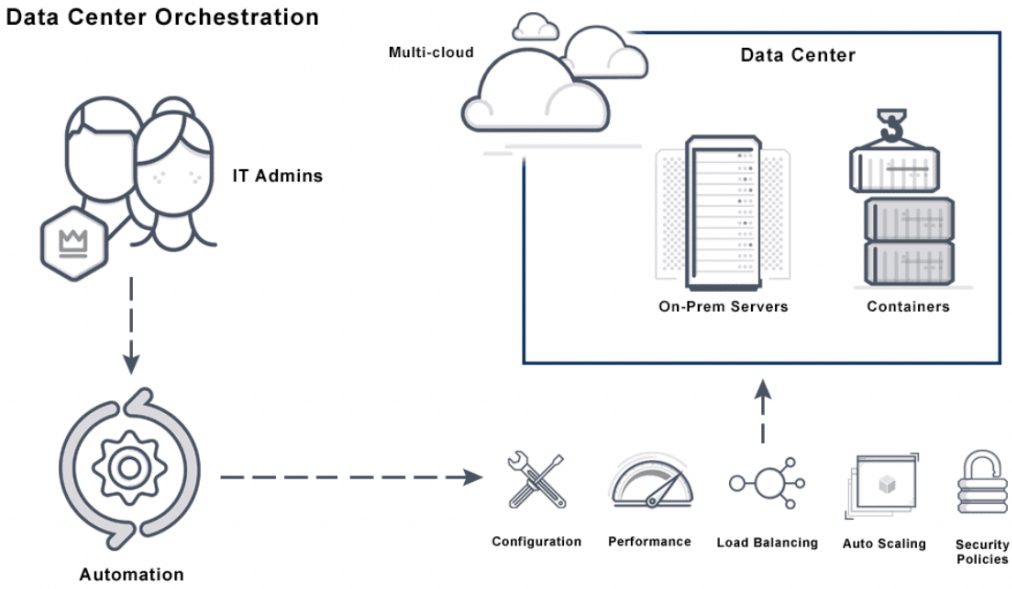

Figure 2. Data Centre Orchestration flow (Avi Networks 2022)

1.2 Motivation for the Project

The need for appropriate data has risen dramatically in recent years and will continue to rise across all businesses. In other words, as a result of technological improvements impacting information about marketing, consumer behaviour, prospects, and rivals, data has a significant effect on how organisations function. Indeed, data is at the core of some of the most well-known management principles and is a critical determinant in economic success. For example, by combining massive volumes of transactional data from many sources, primarily Enterprise Resource Planning (ERP) systems, a company may follow consumers' purchase histories and do worldwide operational research. The act of breaking down data silos so that data is not fragmented and can be accessed quickly is known as data orchestration. In principle, if a company managed its data well enough, it would not require data orchestration and could meet all of its data needs on its own. However, due to the rapid advancement of technology, managing data "well enough" is seldom feasible,

and many businesses opt for a big data strategy, which results in segmented data and fragmented systems.

The core goals of this project were achieved utilising a variety of technologies. The output of the output will be utilised by upstream departments to address issues. Throughout the construction of the system, this thesis examined deployment tactics, libraries, and how to use them for data integration. The deployment approach was designed to address real-world difficulties. The result showed both automatic and manual ETL approaches.

1.3 Focus and Constraints

This research focuses on developing ETL processes and enhancing current ETL and data orchestration technologies that connect cloud-based services. This thesis lacks a corporate case and does not cover the whole of a corporation's business system. The implementation scope is presumed to be acceptable for big data organisations, but may need to be tailored to the unique vocations in which businesses operate.

2 LITERATURE REVIEW

This chapter briefly describes the technological tools and terminologies used in this project's data processing.

2.1 Technical Tools

2.1.1 Python

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics that can be executed on a computer's desktop. This is due to the high level of built-in data structures, as well as the dynamic typing and dynamic binding, which make it very appealing for Rapid Application Development. It can also be used as a scripting or glue language to connect existing components together. Python's straightforward, simple, and easy-to-learn syntax emphasizes readability, which lowers the cost of program maintenance by a significant margin. As a result of Python's support for modules and packages, program modularity and code reuse are encouraged. Python's interpreter and extensive standard library are available for free in source or binary form for all major platforms, and they can be freely distributed under the GNU General Public License (Python, nd., online).

2.1.2 PostgreSQL

PostgreSQL is a powerful, open-source object-relational database system that makes extensive use of and extensions to the SQL language, as well as numerous features that allow it to safely store and scale even the most complex data loads. PostgreSQL's origins can be traced back to 1986 as part of the POSTGRES project at the University of California at Berkeley, and the core platform has been actively developed for more than 30 years (Postresql, nd., online).

2.1.3 Docker

Docker is an open-source containerization platform used for developing, deploying, and managing applications in lightweight virtualized environments called containers.

It is mainly used as a software development platform for developing distributed applications that work efficiently in different environments. By making the software system agnostic, developers do not have to worry about compatibility issues. Packaging applications into isolated environments (containers) also makes it easier to develop, deploy, maintain, and use applications.

2.1.4 Apache Spark

Apache Spark is a lightning-fast cluster computing technology, designed for fast computation. It is based on Hadoop MapReduce and it extends the MapReduce model to efficiently use it for more types of computations, which includes interactive queries and stream processing. The main feature of Spark is its in-memory cluster computing that increases the processing speed of an application (Tutorials Point, nd., online).

Spark is intended to be used for a variety of workloads, including batch applications, iterative algorithms, interactive queries, and streaming. Apart from consolidating all of these workloads into a single system, it also reduces the management burden associated with maintaining multiple tools.

2.1.5 Apache Airflow

Apache Airflow is a workflow automation and scheduling system that can be used to author and manage data pipelines. Airflow uses workflows made of directed acyclic graphs (DAGs) of tasks. A DAG is a construct of nodes and connectors (also called “edges”) where the connectors have direction, and it can be started at any arbitrary node to travel through all connectors. Each

connector is traversed once. Trees and network topologies are types of DAGs (Alooma, 2018, online).

Airflow workflows contain tasks whose output is used as an input for another task. As a result, the ETL process can be considered a type of DAG. It is not possible to loop back to a previous step because the output of each step is used as the input of the following step.

Defining workflows in code provides easier maintenance, testing, and versioning.

2.1.6 PySpark

Apache Spark is written in the Scala programming language. PySpark has been released in order to support the collaboration of Apache Spark and Python, it actually is a Python API for Spark. In addition, PySpark, helps users to interface with Resilient Distributed Datasets (RDDs) in Apache Spark and the Python programming language. This has been achieved by taking advantage of the Py4j library. PySpark LogoPy4J is a popular library that is integrated within PySpark and allows python to dynamically interface with JVM objects. PySpark features quite a few libraries for writing efficient programs (Databricks, 2022, online).

Figure 3. PySpark properties

2.1.7 Pytest

PyTest is a testing framework that allows users to write test codes using the Python programming language. It helps users to write simple and scalable test cases. It can be used in a wide range of testing API, UI, database, and so on. The test file of Pytest has a naming convention that it starts with test_ or ends with

_test keyword and every line of code should be inside a method that should have a name starting with test keyword. Also, each method should have a unique name.

2.1.8 SQL

Structured Query Language, commonly known as SQL, is a standard programming language for relational databases. SQL can be used to share and manage data, particularly data that is found in relational database management systems, which include data organized into tables. Additionally, multiple files, each of which contains a table of data, may be linked together by a common field. SQL allows users to query, update, and reorganize data, as well as create and modify the schema (structure) of a database system, as well as control access to the data stored in the database system.

2.1.9 Jupyter Notebook

The Jupyter Notebook is an open-source web application that can be used to create and share documents that contain live code, equations, visualisations, and text. Jupyter Notebook is maintained by the people at Project Jupyter (Real Python, nd., online).

Jupyter Notebooks is a fork of the IPython project, which previously had its own IPython Notebook project. Because it supports three programming languages in particular: Julia, Python, and R, the name "Jupyter" is derived from these three languages. Jupyter comes pre-installed with the IPython kernel, which enables users to write programs in the Python programming language.

2.1.10 Microsoft Power BI

Microsoft's Power BI is a business analytics service that allows users to analyze data. Its goal is to provide interactive visualizations and business intelligence capabilities through an interface that is simple enough for end-users to create their own reports and dashboards on their own computers.

2.2 Technical Terminologies

2.2.1 Data Processing

ETL is an abbreviation for Extract, Transform, and Load.

When comparing ETL and ELT, the most noticeable difference is the order in which the operations are performed. Rather than copying or exporting data from the source locations and loading it into a staging area for transformation, ELT transfers raw data directly to the target datastore, where it can be transformed as needed. Figure 4 below shows the difference between ETL and ELT processing.

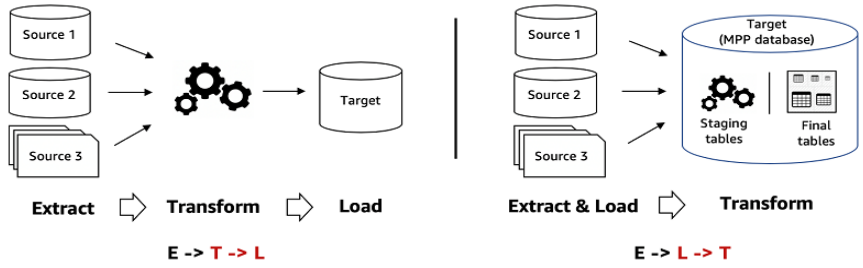

Figure 4. ETL and ELT processing

2.2.2 Data Source

A data source, in the context of computer science and computer applications, is the location where data that is being used comes from. In a database management system, the primary data source is the database, which can be located on a disk or a remote server. The data source for a computer program can be a file, a data sheet, a spreadsheet, an XML file, or even hard-coded data within the program (Techopedia, nd., online).

2.2.3 Data Wrangling

Data wrangling is the process of cleaning, organising, and enhancing unstructured data in order to make decision-making more efficient and effective. Data manipulation is becoming more prevalent in today's top organisations, according to the FBI. Increasingly diverse and unstructured data makes it necessary to spend more time selecting, cleaning, and organising data prior to performing a more comprehensive analysis. Business users have less time to wait for prepared data from technical personnel as a result of data guiding almost every business decision, which is a positive development (Geeks for Geeks, 2021, online).

2.2.4 Data Modelling

Data modelling is the act of visualising a whole information system or a subset of it in order to express the relationships between data elements and structures. The purpose of this section is to demonstrate the many forms of data that are utilised and stored inside the system, the connections between these data types, the various ways in which data may be grouped and structured, as well as the formats and properties of the data (IBM, 2020, online).

2.2.5 Database

A database is a systematic collection of data. It is accessible or may be kept on a computer system. It may be maintained using a Database Management System (DBMS), which is a kind of data management software. A database is a collection of connected data organised in a systematic manner (Geeks for Geeks, 2021, online).

In a database, data is organised into tables composed of rows and columns that are then indexed to allow for quick updating, expansion, and destruction of the information. Computer databases frequently contain data files that record transactions, such as money transfers from one bank account to another, sales and customer information, student fee information, and product information,

among other things. In the world of databases, there are many different types, ranging from the most widely used relational database to distributed databases, cloud databases, and NoSQL databases.

2.2.6 Data Lake

A Data Lake is a huge storage repository capable of storing organised, semi-structured, and unstructured data. It is a repository for any sort of data in its original format, with no restrictions on account or file size. It provides a large amount of data to boost analytic speed and native integration (Guru99, 2022, online).

A data lake can be thought of as a large container, similar to how lakes and rivers are thought of. A data lake is similar to a lake in that it contains structured data, unstructured data, machine-to-machine communication, and real-time logs that are constantly flowing through it.

2.2.7 Data Warehouse

A data warehouse, also known as an enterprise data warehouse (EDW), is a system that consolidates data from several sources into a single, centralised, consistent data storage for the purpose of supporting data analysis, data mining, artificial intelligence (AI), and machine learning. A data warehouse system allows an organisation to do complex analytics on massive amounts of historical data (petabytes and petabytes) in ways that a typical database cannot (IBM, 2020, online).

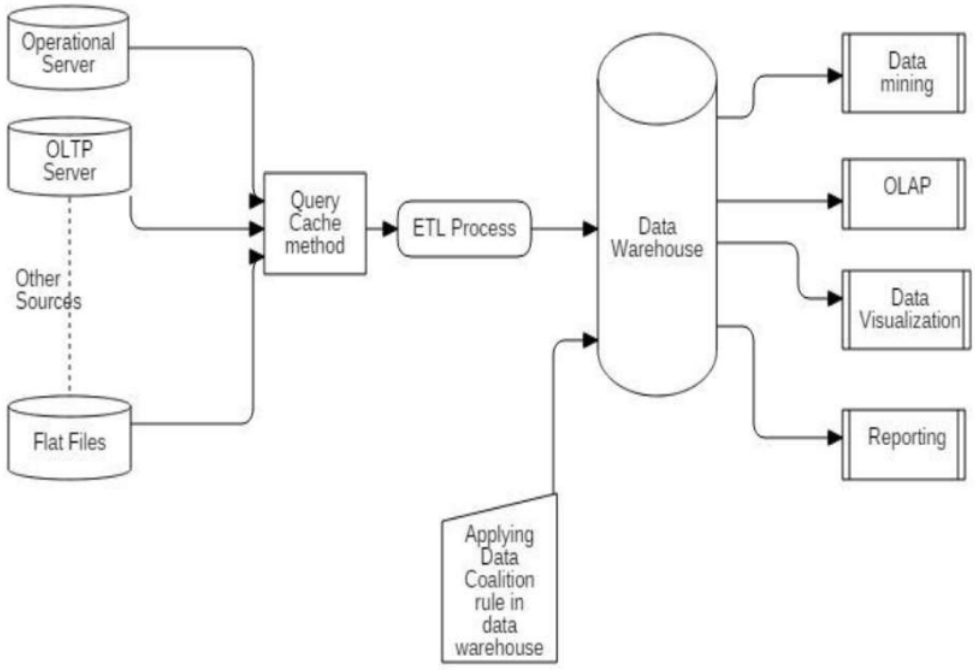

Figure 5. Data warehouse diagram

2.2.8 pgAmin

pgAmin a very popular open-source platform fully dedicated to PostgreSQL with graphical user interface administration tools to manage relational databases. Some features include a query tool for SQL statements and importing/exporting CSV files (Medium, 2020, online).

2.2.9 Docker Hub

Docker Hub is a service provided by Docker for finding and sharing container images with a team. It is the world's largest repository of container images with an array of content sources including container community developers, open-source projects, and independent software vendors building and distributing their code in containers (Docker, 2022, online).

2.2.10 Data Orchestration

Data orchestration refers to the technologies and processes that automatically transfer and process data throughout the enormous data ecosystem in the most effective and efficient manner possible (Escend, 2022, online). The terms big data orchestration and ETL or ELT orchestration are frequently heard in the industry. Organizing, executing, and monitoring one or more data pipelines in order to establish the ultimate capability: the ability to extract the greatest amount of value from data assets, are all fundamentally synonymous with the process of establishing the ultimate capability.

2.2.11 Star Schema

A star schema is a special form of a data model whose goal is not normalisation, but optimization for efficient read operations. The main field of application is a data warehouse and OLAP applications (The customize windows, 2021, online). The term "star schema" refers to the fact that the tables are organized in a star shape, with a fact table in the center and several dimension tables arranged around it. In most cases, a star scheme is denormalized. For performance reasons, it is acceptable to accept possible anomalies and an increased memory requirement. The snowflake scheme, which is related to the star scheme, has the potential to provide an improvement. Multi-level dimension tables, on the other hand, must be linked together using join queries in this case.

2.2.12 Snowflake Schema

The snowflake schema is a continuation of the star schema used in OLAP and data warehousing. With the star schema, the dimension tables are denormalized, which results in better processing speed at the expense of data integrity and storage space. In contrast, the snowflake schema refines the individual dimension tables by classifying or normalising them (The Customize Windows, 2021, online).

3 METHODOLOGY

3.1 Case Study

This project's case study is based on data from Kaggle, which provides information about airline delays and cancellations in the United States during Covid-19, which will take place in 2020. This case study was developed in response to real-world challenges. Therefore, the data should be big enough and verified by the public to avoid credential issues and be available for further research when we are not using any business practical cases. As a result, using this dataset allows us to experiment with our research projects.

COVID-19 has had a substantial influence on the global airline industry, resulting in extensive air service cuts till 2020. This data collection, which contains almost 11 million flights, will aid anybody interested in determining the impact of the virus on the domestic aviation industry in the United States. The Department of Transportation's Bureau of Transportation Statistics (BTS) analyzes domestic flights operated by major airlines for on-time performance. The data covers the period from January to June 2020 (and will be updated soon) and contains 11 million flight records from the top 10 US airlines, including information on on-time arrivals, delays, cancellations, and diversions.

The column description .txt file has a detailed explanation of each of the 47 data columns. U.S. DOT Bureau of Transportation Statistics gathered flight delays and cancellations data for this study.

3.2 Entity Relationship Diagram

The snowflake schema was used to construct the frame of the entity relationship in this project. As shown in Figure 6, all seven sub-datasets, which were the results of the data wrangling phase, were linked together. Each relationship describes the link between a fact table and one-dimension table, or between one dimension table and another dimension table, or between two dimension tables.

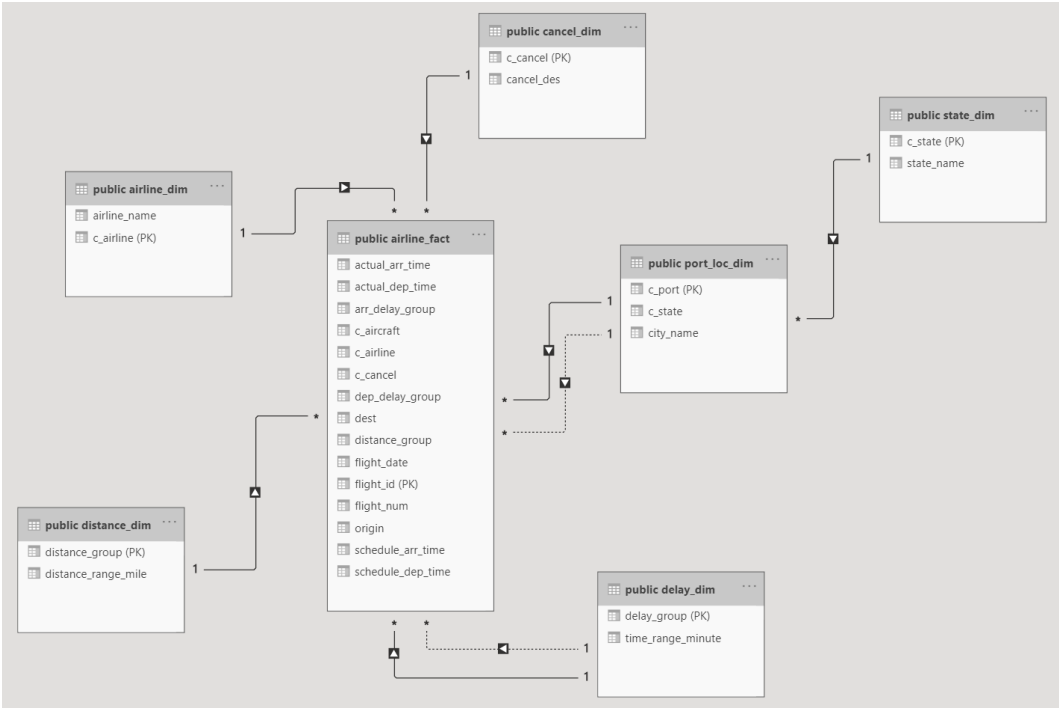

Figure 6. Entity relationship diagram

4 IMPLEMENTATION

The chapter describes two alternative techniques to data processing: Pure database-based ETL pipeline and Data orchestration in Building ELT pipelines in Data Lake clusters and AWS S3, respectively.

4.1 Pure Database-based ETL Pipeline Approach

4.1.1 Implementation Overview Structure

The core idea of this approach is based on the traditional ETL processing, and the datastore of the database, specifically the PostgreSQL database. Figure 7 below describes the ETL processing diagram.

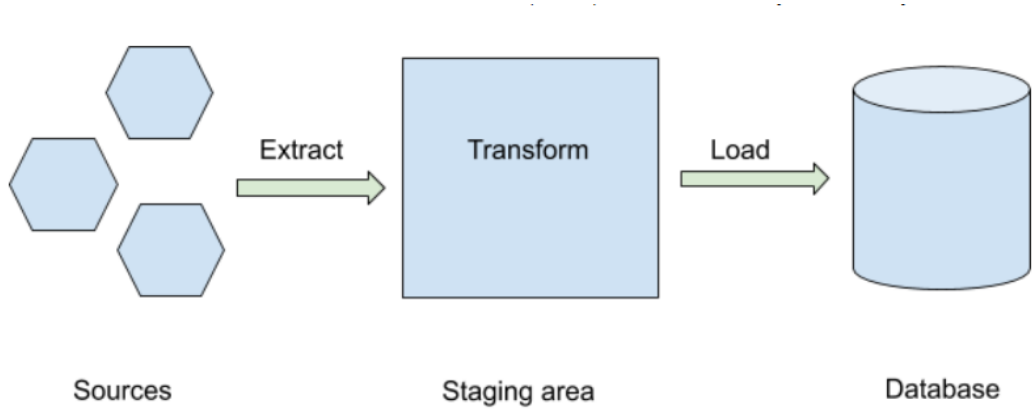

Figure 7. ETL processing diagram (Keboola, 2020, online)

The implementation can be divided into four main steps: Setting up the environment, data wrangling, data ingestion, and testing.

- Setting up the environment: use the existing Docker image for PySpark and customise the docker-compose.yml file for PostgreSQL.
- Data wrangling: clean the raw dataset and break it into smaller sub-datasets for data modelling.
- Data ingestion: connect to the PostgreSQL server; create the database and necessary tables into PostgreSQL; insert values from sub-datasets into PostgreSQL.

- Testing: Ensure that all records were completely uploaded.

4.1.2 Setting up the Environment

There are two ways of setting up the environment: installing PostgreSQL and PySpark locally or running them using Docker. Multiple steps and possible changes in operating systems may be required to install PostgreSQL and PySpark. By executing the Docker container, Docker can simplify the installation and ensure that the environment is consistent between team members. Users can quickly start a Docker container by either utilising an existing Docker image from Docker Hub or creating their own Docker file.

PostgreSQL: We constructed a docker-compose.yml file to run PostgreSQL with Docker, which allowed us to deploy, combine, and configure several docker-containers at once. The docker-compose.yml file for this project is shown in Code snippet 1 below.

```
version: '3.4'
services:
    postgresql_database:
        image: postgres:latest
        environment:
            - POSTGRES_USER=thesis
            - POSTGRES_PASSWORD=thesis2022
            - POSTGRES_DB=thesisDb
        ports:
            - "5432:5432"
        restart: always
        volumes:
            - database-data:/var/lib/postgresql/data/
    pgadmin:
```

```
    image: dpage/pgadmin4
    environment:
        - PGADMIN_DEFAULT_EMAIL=pgadmin4@pgadmin.org
        - PGADMIN_DEFAULT_PASSWORD=admin1234
    ports:
        - '5050:80'
    restart: always
    volumes:
        - pgadmin:/root/.pgadmin
```

Code Snippet 1. docker-compose.yml for PostgreSQL database

The properties of docker-compose.yml are as follows:

- version: the Docker Compose version
- services: the list of containers that will be generated.
- postgre_database, pgadmin: name of the service in this docker-compose file
- image: using a pre-built image to operate a service, while Docker Compose forks a container from that image location
- volumes: mounting disks in Docker
- ports: mapping the container's ports to the host computer

The environment is now ready to use, and users can get started by running the following command:

```
docker-compose up
```

The docker containers will be created and started with this command. Users can access the PostgreSQL server using pdAdmin at *http://localhost:5050* with the email address of *pgadmin4@pgadmin.org* and the password of *admin1234* when they are ready.

Figure 8. pgAmin server

PySpark: PySpark docker image is one of the Docker image definitions maintained by the Jupyter team in the Jupyter Docker Stacks. For this implementation, we used that docker image. This is demonstrated in the following sample, where container-name is the name of this container, and PWD is the current working directory.

```
docker run -it --name <container-name> -p 8888:8888 -v
${PWD}:/home/jovyan/work jupyter/pyspark-notebook
```

When the environment is complete, the user can connect to the PySpark server by using one of the URLs that have been provided. Figure 9 below depicts a sample of those URLs.

```
To access the server, open this file in a browser:
    file:///home/jovyan/.local/share/jupyter/runtime/jpserver-8-open.html
Or copy and paste one of these URLs:
    http://c710cc444857:8888/lab?token=eeb0f720f6cc71694a00e1091c68eb18c306ab179c32bcd7
 or http://127.0.0.1:8888/lab?token=eeb0f720f6cc71694a00e1091c68eb18c306ab179c32bcd7
```

Figure 9. URLs for accessing PySpark server

4.1.3 Data Wrangling

Not all of the raw datasets are clean and neat right from the start of the analysis process. There are occasions when they have missing values, incorrect data type, improper schema, or duplicate records. The user must employ a variety of various strategies in order to meet their objectives, which vary depending on the type of data and goals at hand. The entire process of cleaning, organizing, transforming raw data into the desired format is referred to as data wrangling.

Due to the fact that all records must be uploaded into the PostgreSQL database in this implementation, having a correct schema is extremely crucial and should be prioritized during the data wrangling and data modeling phases of the project. In the case study of this project, the snowflake schema was applied for flexible querying across more difficult dimensions and relationships.

As a result, from a single raw dataset, we have one fact sub-dataset and six-dimension sub-datasets, which correspond to seven tables in the PostgreSQL database, which we will cover in more detail in the following phase. The schemas of those sub-datasets are shown in the following figure.

```
root
 |-- c_port: string (nullable = true)
 |-- city_name: string (nullable = true)
 |-- c_state: string (nullable = true)
```

Figure 10. Schema of port_loc_dim sub-dataset

```
root
 |-- c_state: string (nullable = true)
 |-- state_name: string (nullable = true)
```

Figure 11. Schema of state_dim sub-dataset

```
root
 |-- c_airline: string (nullable = true)
 |-- airline_name: string (nullable = true)
```

Figure 12. Schema of airline_dim sub-dataset

```
root
 |-- distance_group: string (nullable = true)
 |-- distance_range_mile: string (nullable = true)
```

Figure 13. Schema of distance_dim sub-dataset

```
root
 |-- c_cancel: string (nullable = true)
 |-- cancel_des: string (nullable = true)
```

Figure 14. Schema of cancel_dim sub-dataset

```
root
 |-- delay_group: string (nullable = true)
 |-- time_range_minute: string (nullable = true)
```

Figure 15. Schema of delay_dim sub-dataset

```
root
 |-- flight_id: string (nullable = true)
 |-- flight_date: string (nullable = true)
 |-- c_airline: string (nullable = true)
 |-- flight_num: string (nullable = true)
 |-- c_aircraft: string (nullable = true)
 |-- origin: string (nullable = true)
 |-- dest: string (nullable = true)
 |-- schedule_dep_time: string (nullable = true)
 |-- actual_dep_time: string (nullable = true)
 |-- dep_delay_group: string (nullable = true)
 |-- schedule_arr_time: string (nullable = true)
 |-- actual_arr_time: string (nullable = true)
 |-- arr_delay_group: string (nullable = true)
 |-- distance_group: string (nullable = true)
 |-- c_cancel: string (nullable = true)
```

Figure 16. Schema of airline_fact sub-dataset

4.1.4 Data Ingestion

Data is becoming increasingly important in today's world. Enterprises must be able to access all of their data sources for analytics in order to make better decisions. The transportation of data from various sources to a storage medium where it can be accessed, used, and analysed is also critical, and this process is referred to as data ingestion.

As part of the data ingestion process into the PostgreSQL database, this project was divided into three main Python files: one file (sql_queries.py) that contains all SQL queries for manipulating and storing data in the database, one file (modify_tables.py) that creates the database and necessary tables in the PostgreSQL server, and the last file (etl.py) that contains all codes for performing data processing in the database.

The **sql_queries.py** file contains all SQL queries for deleting tables, creating new tables, and inserting values into those tables in the database. These actions can be applied to any of the seven tables depicted in the previous figure, as shown in an example of one table in the following snippet.

```
# drop tables

airline_dim_drop = 'DROP TABLE IF EXISTS airline_dim;'

# create tables

airline_dim_create = """

      CREATE TABLE IF NOT EXISTS airline_dim (

            c_airline varchar(2) primary key,

            airline_name varchar(100)

      )"""

# insert into tables

airline_dim_insert = """

      INSERT INTO airline_dim (

            c_airline,

            airline_name

      ) VALUES (%s, %s)"""
```

Code Snippet 2. SQL queries for airline_dim table in sql_queries.py file

At the end of the file, there are two lists that contain all of the string variables that are used for dropping and creating tables, as shown in code snippet 3 below.

```
# query list

create_tables_query = [airline_dim_create, cancel_dim_create,
delay_dim_create, distance_dim_create, port_loc_dim_create,
state_dim_create, airline_fact_create]

drop_tables_query = [airline_dim_drop, cancel_dim_drop,
delay_dim_drop, distance_dim_drop, port_loc_dim_drop,
state_dim_drop, airline_fact_drop]
```

Code Snippet 3. Query list in sql_queries.py file

In the **modify_tables.py** file, the project first establishes a connection to the PostgreSQL server, after which it deletes and creates databases and tables. Figures 17, 18, 19 will demonstrate how it works.

```python
import psycopg2
from sql_queries import create_tables_query, drop_tables_query
import config

def create_database():
    """
    Function: Create and connect to airline database in postgreSQL
    """

    # connect to postgres database
    conn = psycopg2.connect(database='postgres', host='127.0.0.1', port='5432', user=config.user, password=config.password)
    conn.set_session(autocommit=True)
    cur = conn.cursor()

    # create airline database
    cur.execute("DROP DATABASE IF EXISTS airline;")
    cur.execute("CREATE DATABASE airline WITH ENCODING 'utf8' TEMPLATE template0")

    # close connection to postgres database
    conn.close()

    # connect to airline database
    conn = psycopg2.connect(database='airline', host='127.0.0.1', port='5432', user=config.user, password=config.password)
    conn.set_session(autocommit=True)
    cur = conn.cursor()

    return cur, conn
```

Figure 17. Create database function in modify_tables.py file

```python
def drop_table(cur, conn):
    """
    Function: Drops each table using the queries in `drop_tables_query` list.
    Parameter:
        - cur: cursor of postgres server
        - conn: connection to postgres server
    """

    for query in drop_tables_query:
        cur.execute(query)
        conn.commit()

def create_table(cur, conn):
    """
    Function: Create each table using the queries in `create_tables_query` list
    Parameter:
        - cur: cursor of postgres server
        - conn: connection to postgres server
    """

    for query in create_tables_query:
        cur.execute(query)
        conn.commit()
```

Figure 18. Drop and create table functions in modify_tables.py file

```python
def main():
    """
    Drop (if exist) airline database
    Create airline database
    Drop all tables
    Create all tables
    Close connection
    """

    cur, conn = create_database()

    drop_table(cur, conn)
    create_table(cur, conn)

    conn.close()

if __name__ == '__main__':
    main()
```

Figure 19. Main function in modify_tables.py file

The **elt.py** file connects to the PostgreSQL database, reads all of the records from seven sub-datasets (which were created as a result of the data wrangling phase), and inserts those records into the tables in the database, completing the process. Figures 20, 21 below show how they work in this project.

```python
import pandas as pd
from sql_queries import *
import psycopg2
import config

def process_data(cur, conn):
    """
    Function: insert all records to the postgres database
    Parameter:
        - cur: cursor of postgres server
        - conn: connection to postgres server
    """

    # create the dataframe for each dataset
    airline_df = pd.read_csv('data/airline.csv')
    cancel_df = pd.read_csv('data/cancellation.csv')
    delay_df = pd.read_csv('data/delay_group.csv')
    distance_df = pd.read_csv('data/distance_group.csv')
    port_loc_df = pd.read_csv('data/port_loc.csv')
    state_df = pd.read_csv('data/states.csv')

    fact_df = pd.read_csv('data/fact.csv')

    # save value into postgress sever
    for _, row in airline_df.iterrows():
        cur.execute(airline_dim_insert, row)
        conn.commit()

    for _, row in cancel_df.iterrows():
        cur.execute(cancel_dim_insert, row)
        conn.commit()

    for _, row in delay_df.iterrows():
        cur.execute(delay_dim_insert, row)
        conn.commit()

    for _, row in distance_df.iterrows():
        cur.execute(distance_dim_insert, row)
        conn.commit()

    for _, row in port_loc_df.iterrows():
        cur.execute(port_loc_dim_insert, row)
        conn.commit()

    for _, row in state_df.iterrows():
        cur.execute(state_dim_insert, row)
        conn.commit()

    for _, row in fact_df.iterrows():
        cur.execute(airline_fact_insert, row)
        conn.commit()
```

Figure 20. Process data function in elt.py file

```
def main():
    """
    Connect to the postgres server
    Insert all records to postgres database
    Close connection
    """

    # connect to airline database
    conn = psycopg2.connect(database='airline', host='127.0.0.1', port='5432', user=config.user, password=config.password)
    cur = conn.cursor()

    process_data(cur, conn)

    conn.close()

if __name__ == '__main__':
    main()
```

Figure 21. Main function in etl.py file

4.1.5 Testing

Testing is an important part of any phase of the development lifecycle, regardless of the field. Users can ensure that the product operates in the correct manner and meets their expectations by conducting testing. Throughout this project, the Pytest framework was utilised to allow users to customise their assert statement code in a straightforward and efficient manner.

It is the logic of this project that the user accesses all sub-datasets that were created as a result of the data wrangling phase in order to obtain the actual total number of rows for each sub-dataset. The user then establishes a connection to the database in order to query the total number of rows for each corresponding table in the database and, finally, to compare those two sets of numbers together. This implementation allows users to determine whether or not all of the records were successfully uploaded into the PostgreSQL database. Figure 22 below illustrates an example of testing for one table in this project, and this method is applied in a similar manner for all of the other tables.

```python
import pytest
import psycopg2
import config as cfg
import pandas as pd

# connect to the postgres server
conn = psycopg2.connect(database='airline',
                        host='127.0.0.1',
                        port='5432',
                        user=cfg.user,
                        password=cfg.password)

cur = conn.cursor()

def test_port_loc_dim():
    """
    Function: check that all rows were completely uploaded to port_loc_dim table
    """

    real_rows = pd.read_csv('data/port_loc.csv').shape[0]
    cur.execute('SELECT count(*) FROM port_loc_dim;')
    rows = cur.fetchone()[0]

    assert rows == real_rows
```

Figure 22. Test case for port_loc_dim table

Users can run the test file by entering the following command into their terminal window:

```
pytest -v test.py
```

After running the test.py file, the following is the output displayed in Figure 23 below.

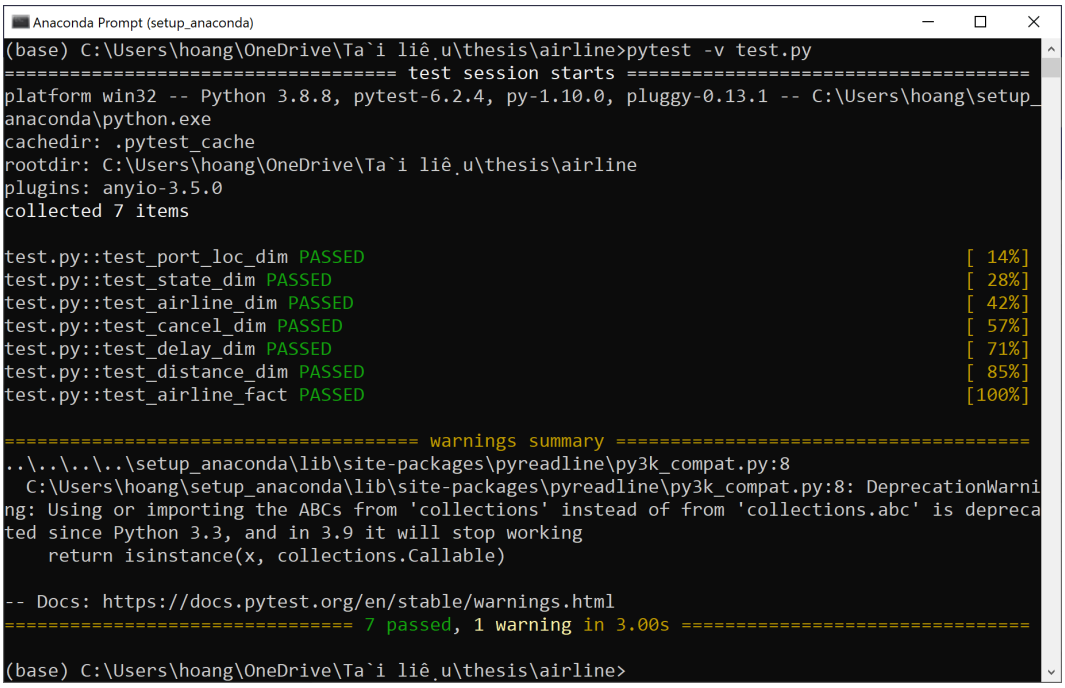

Figure 23. All test cases are passed in the terminal

4.2 Data Orchestration in the Context of Developing ELT pipelines in Data Lake Clusters and AWS S3

4.2.1 Data Orchestration and Apache Airflow

The data pipeline is being used to power an existing web tool that allows users to examine home prices based on their proximity to subway stations. The average property price of properties within a one-kilometre radius of a certain subway station may be seen by users. As a result, the new data pipeline illustrated in this repository is used to enrich the application and to combine sentiment scoring and theme analysis in order to provide users with a better sense of the overall mood as well as an indicator of the type of milieu that exists in a particular location.

The repository also includes a Python-based web scraper that utilises, for example, BeautifulSoup, to get geo-specific home prices from property

websites around London, but this will not be discussed in depth here. (Kindl, n.d., online)

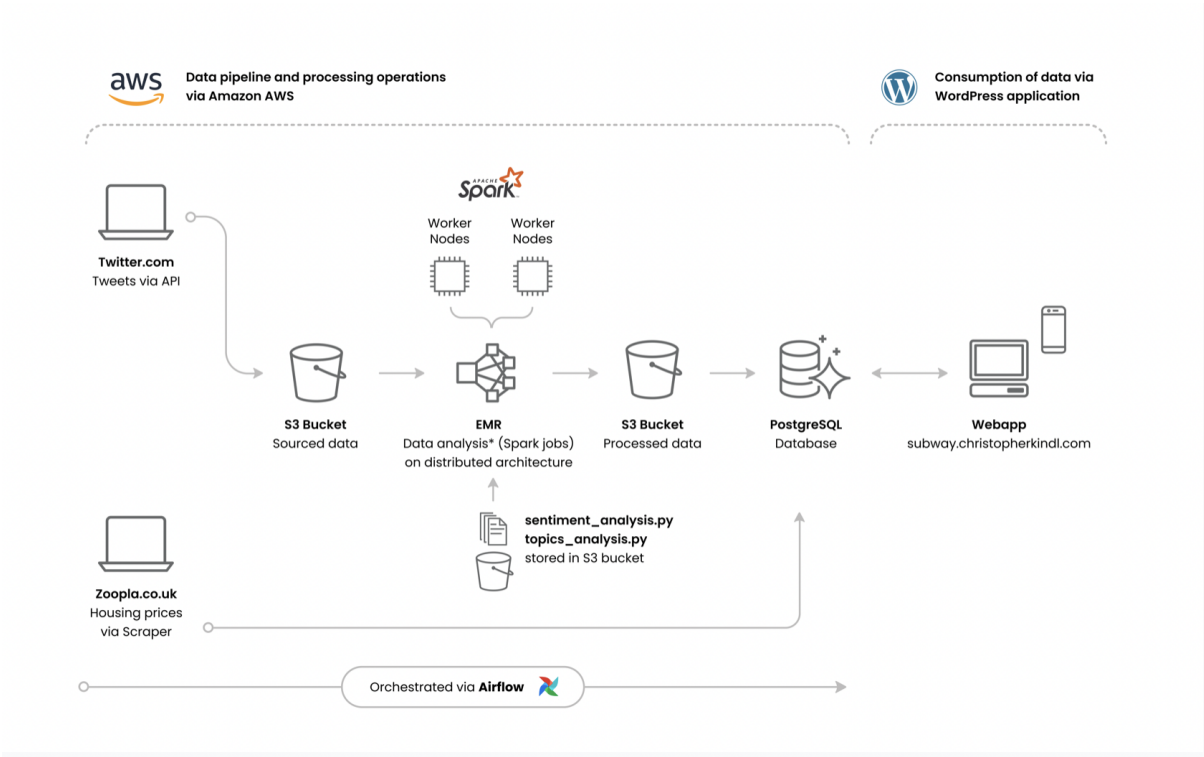

Figure 24. Twitter data pipeline building via AWS which is orchestrated via Airflow (Kindl, n.d., online)

A typical Airflow DAG is composed of multiple tasks that collect, modify, or process data in a variety of ways. Due to the MWAA's restricted workload capability, it is not recommended to execute intensive data processing tasks. Amazon SageMaker is used to execute machine learning tasks, while Amazon EMR is used to do complex data analysis in a distributed manner. In this example, we utilised Apache Spark and the Python API PySpark to analyse data on an Amazon EMR cluster.

We may either write new functions (for example, requesting data through the Twitter API) or make use of predefined modules, which are often used to activate external actions (e.g. data analysis in Spark on Amazon EMR). (Kindl, n.d., online)

4.2.2 Airflow on Docker

First, docker-compose.yml file is fetched

```
curl -LfO 'https://airflow.apache.org/docs/apache-
airflow/2.0.1/docker-compose.yaml'
```

Meta info:

- airflow-schedule: The scheduler monitors all tasks and DAGs, then triggers the task instances once their dependencies are complete.
- airflow-webserver: web server at *http://localhost:8080*
- airflow-worker: The worker that executes the tasks given by the scheduler.
- airflow-init: The initialization service.
- flower: The flower app for monitoring the environment. It is available at *http://localhost:8080*.
- postgres: Database
- redis: The redis - broker that forwards messages from scheduler to worker.

Some directories in the container are mounted, which means that their contents are synchronised between the computer and the container.

./dags - for DAG files.

./logs - contains logs from task execution and scheduler.

./plugins - for custom plugins.

4.2.3 AWS Roles, Keys, and Security Groups

a. AWS EC2 Keypair

We constructed the Key Pair in the EC2 session in order to link an SSH key pair securely to the cluster we were utilising.

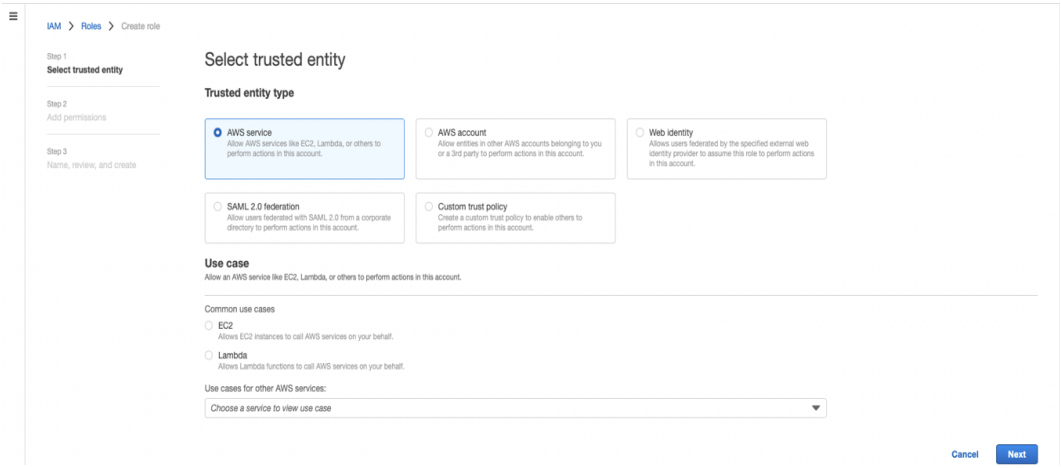

Figure 25. I AM role creation generators

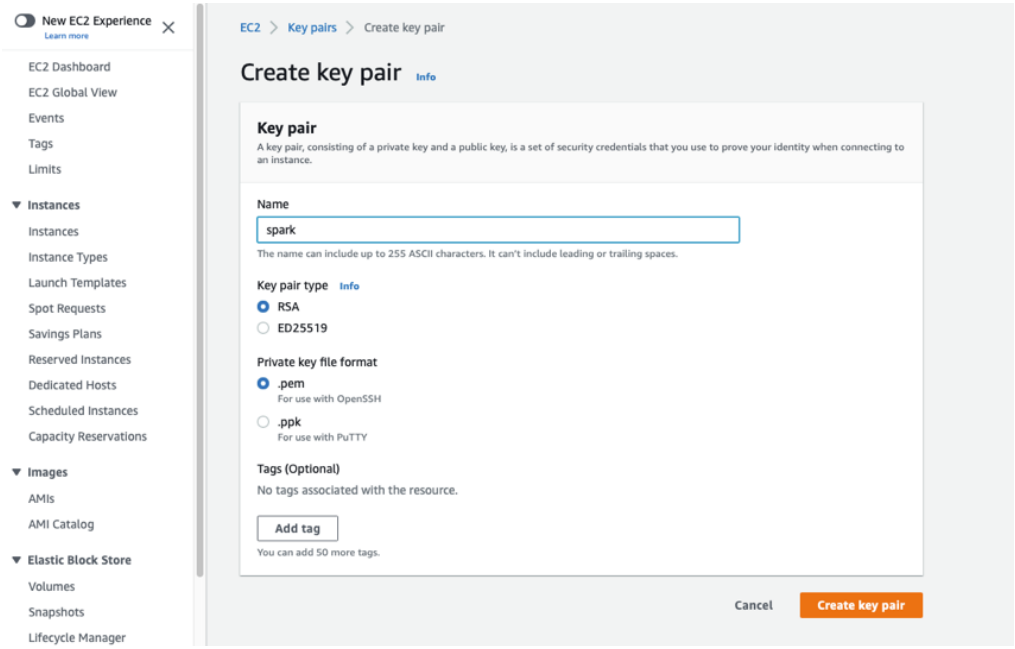

Figure 26. EC2 Key pair creation

After clicking Create key pair, we were presented with a.pem file that we could save to our local PC.

b. IAM user

This step is required to create an IAM user with Administrator permissions who will be able to do all AWS account-related activities only through the command line interface. After creating an IAM user with the necessary rights, we used its Access key (long-term credentials) to configure the AWS CLI on our local PC.

An AWS IAM user was created and copy the Access key generated in the previous step copied.

The AWS Identity and Access Management (IAM) service allows to authorise users / applications (such as the AWS Command Line Interface) to access AWS services and resources on the AWS Cloud.

An Access Key was created by establishing an Access Key ID first, followed by a Secret Access Key. An IAM user was created by navigating to the IAM Dashboard.

To determine the state of the configuration from AWS to our environment, the aws configure list command was used.

```
(base) alexnguyendinh@Alexs-MacBook-Pro ~ % aws configure list
      Name                    Value             Type    Location
      ----                    -----             ----    --------
   profile                <not set>             None    None
access_key     ****************CBE3 shared-credentials-file
secret_key     ****************dwVB shared-credentials-file
    region                us-west-2      config-file    ~/.aws/config
(base) alexnguyendinh@Alexs-MacBook-Pro ~ % cat ~/.aws/credentials
[default]
aws_access_key_id = [redacted]
aws_secret_access_key = [redacted]
```

Figure 27. Configuration check for AWS in Shell

4.2.4 AWS S3 Bucket

Before any data can be uploaded to Amazon S3, an S3 bucket in one of the AWS regions must first be created (for example, photos, videos, and documents). Buckets are used to store items on Amazon S3. An unlimited number of items may be saved in a bucket, and the account can have a total of up to 100 buckets at any time. Buckets and items are managed in Amazon S3, which are considered AWS resources, using APIs provided by Amazon S3. In order to build a bucket and upload items, the Amazon S3 API may be used. The same operation may be accomplished with the Amazon S3 interface. It is the console that communicates with Amazon S3 APIs in order to send queries to the cloud storage service. (Amazon, n.d., online)

To create an S3 bucket from the SHELL, the following command was used to create a bucket named flight-thesis-covid19 with region west US 2 with an allowance for access in public in default and the result in figure 28.

```
aws s3api create-bucket --bucket flight-thesis-covid19 --region
us-west-2 --create-bucket-configuration LocationConstraint=us-
west-2
```

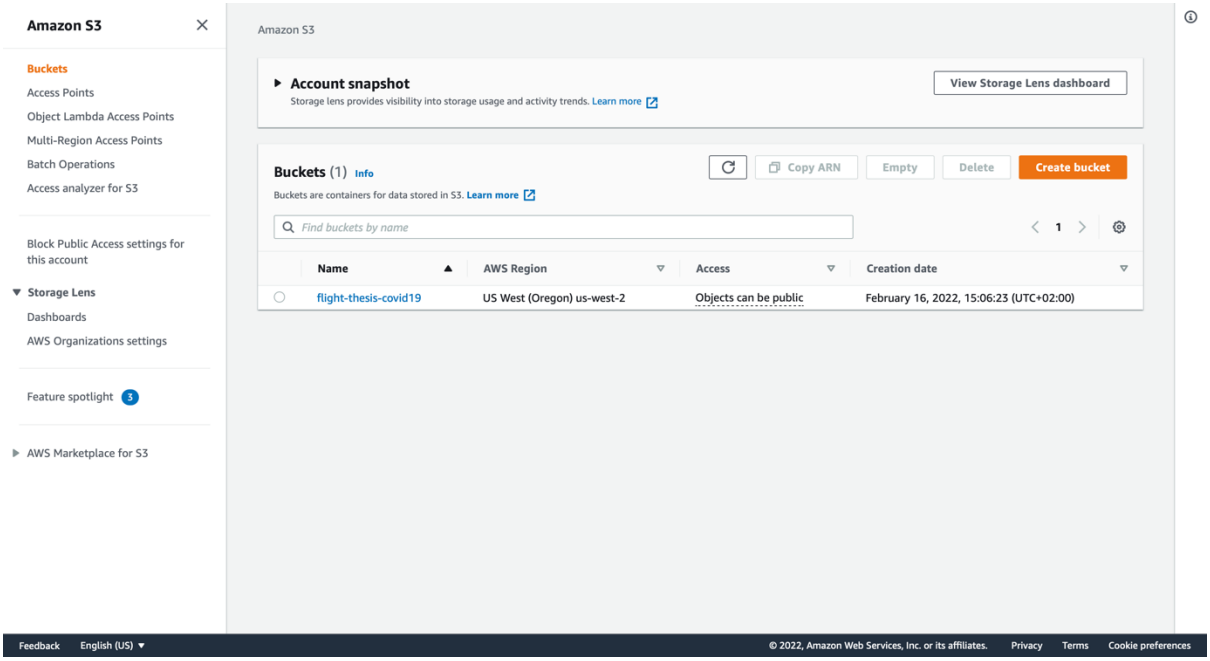

Figure 28. S3 bucket on the S3 console

We had to make sure that the Access session of the bucket should be “Objects can be public” process.

4.2.5 DAG File

a. DAG Initialisation

One concept to grasp (it may not be clear at first) is that this Airflow Python script is essentially simply a configuration file that specifies the DAG's structure in code. The real tasks described here will be executed in a context distinct from that of this script. Due to the fact that various jobs execute on separate workers at different times, this script cannot be used to communicate across tasks. Take note that we have a more complex function called XComs for this reason.

People often mistake the DAG definition file for a location where they can do real data processing but this is not the case. The objective of this script is to define a DAG object. It must be rapid to assess (seconds, not minutes), since the scheduler will run it frequently to reflect any changes.

We defined the DAG in the DAG.py file and proceeded with the module definition in this project. The DAG library from airflow ensures that the DAG is correctly initialised. Otherwise, we relied heavily on the S3 and EMR libraries to construct clusters and transmit data.

We initialised the DummyOperator to be used for the Start execution and End execution in Tree Graph view in Web UI to simplify management in a visual way.

```
# import libraries

# These libraries for S3 modify

from airflow import DAG

from airflow.hooks.S3_hook import S3Hook

from airflow.operators.dummy_operator import DummyOperator

from airflow.operators.python_operator import PythonOperator

# Libraries for EMR creation

from      airflow.contrib.operators.emr_create_job_flow_operator
import (

    EmrCreateJobFlowOperator,

)

#libraries for step of EMR and terminate the job

from   airflow.contrib.operators.emr_add_steps_operator   import
EmrAddStepsOperator

from       airflow.contrib.sensors.emr_step_sensor       import
EmrStepSensor

from   airflow.contrib.operators.emr_terminate_job_flow_operator
import (

    EmrTerminateJobFlowOperator,

)
```

Code snippet 4. Module definition

The task continued with the DAG initialization; we needed a DAG object to hold our tasks, so we offered the DAG id, which serves as a unique identifier for the DAG in this section. Also included is the newly created default argument dictionary, which is set to a one-day scheduling interval and the newly built default argument dictionary. As we were going to build up a DAG and a number of jobs, we had the option of explicitly supplying a set of inputs to the constructor of each job, which saved us from having to hand in the same information again. It is possible to define a dictionary of default parameters that may be used in the task creation process.

We established the following properties: The project's owners, start date specifies the date of the first operation, depending on the history, the preceding task instance was not required to succeed (except if it is the first run for that task), it would be preferable to disable catchup if DAG is not constructed to manage its catchup (i.e., it is not restricted to the interval, but rather to Now, for example.). We scheduled the system to run just once for testing and to save money.

```
from datetime import datetime,timedelta
### Define dag
default_args = {
    'owner': 'Alex',
    'start_date': datetime(2022, 2, 15),
    'depends_on_past': False,
    'catchup': False
}

dag = DAG('Airflow_and_EMR',
          default_args=default_args,
```

```
          description='Load and transform data in EMR with
Airflow',
          schedule_interval='@once'
        )
```

Code snippet 5. Define the DAG

b. Tasks

As soon as operator objects are instantiated, tasks are created for them. A task is an object that is formed as a result of the use of an operator. The task's unique identifier is determined by the first task id that is entered.

The purpose of this project was to create a DAG flow to show and simplify the process of establishing an ETL pipeline for the majority of users. It is claimed that the graphic – the Tree graph portion of the Airflow UI located at localhost address and port 8080 – depicts a logical flow for data engineers to monitor and control in a more ideal situation.

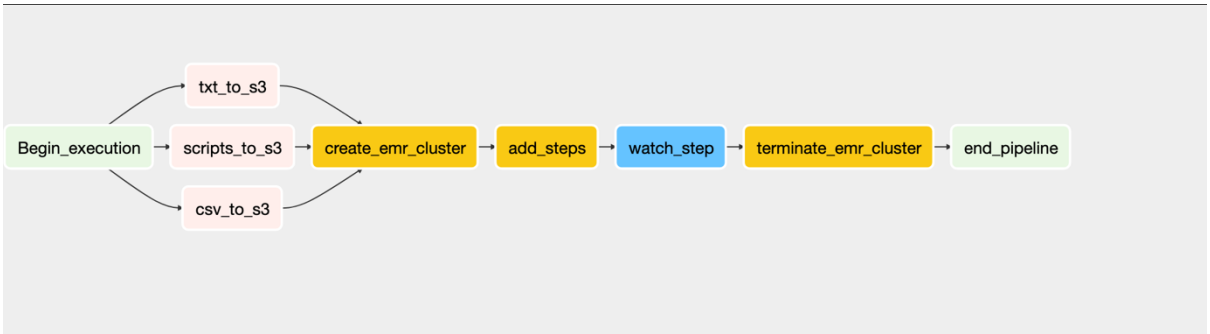

Figure 29. DAG diagram, Tree graph in Airflow web UI

We began the cycle with a start_operator task and then migrated all local files to S3. In practice, raw data may be sent not just through files but also using a web API, for example, to get the data and transmit it to cloud computing services for processing.

As specified in 4.2.5.a module declaration about DummyOperator function, we used that method to start and stop the process. The following tasks csv_to_s3, txt_to_s3, and script_to_s3, all utilize the PythonOperator function. This function is mostly used to execute Python format code, which is defined in the same file as Python callable for the purpose of directing to the executed functions (here is the _local_to_s3).

```
start_operator = DummyOperator(
    task_id ="Begin_execution",
    dag=dag
)
csv_to_s3 = PythonOperator(
    dag=dag,
    task_id = "csv_to_s3",
    python_callable = _local_to_s3,
    op_kwargs = {"filename":local_csv,"key":s3_csv,}
)
```

Code snippet 6. Task generators for starting the process and transfer .csv file from local to S3 bucket.

We addressed the issue of creating clusters on EMR services after completing the work of copying local files to an S3 bucket. The EMR clusters are virtual computers that include Hadoop, Python, and Spark environments to aid with data conversion to an appropriate data platform. The cluster task must execute the EmrCreateJobFlowOperator() function and leave the connection ids with AWS and EMR at their default settings.

```
create_emr_cluster =EmrCreateJobFlowOperator(
    task_id="create_emr_cluster",
    job_flow_overrides = JOB_FLOW_OVERRIDES,
    aws_conn_id ="aws_default",
```

```
    emr_conn_id="emr_default",
    dag =dag,
)
```

Code Snippet 7. Create EMR cluster task

We processed the next tasks of the EMR cluster with EmrAddStepsOperator function(). The function is coming up with access to the EMR cluster JSON format file as a dictionary format of Python.

- job_flow_id (Optional[str]) -- id of the JobFlow to add steps to. (templated)
- job_flow_name (Optional[str]) -- name of the JobFlow to add steps to. Use as an alternative to passing job_flow_id. will search for id of JobFlow with matching name in one of the states in param cluster_states. Exactly one cluster like this should exist or will fail. (templated)
- cluster_states (Optional[List[str]]) -- Acceptable cluster states when searching for JobFlow id by job_flow_name. (templated)
- aws_conn_id (str) -- aws connection to uses
- steps (Optional[Union[List[dict], str]]) -- boto3 style steps or reference to a steps file (must be '.json') to be added to the jobflow. (templated)
- xcoms are explicitly “pushed” and “pulled” to/from their storage using the xcom_push and xcom_pull methods on Task Instances. Many operators will auto-push their results into an XCom key called return_value if the do_xcom_push argument is set to True (as it is by default), and @task functions do this as well.

```
#Adder step
step_adder = EmrAddStepsOperator(
    task_id="add_steps",
```

```
    job_flow_id="{{
task_instance.xcom_pull(task_ids='create_emr_cluster',
key='return_value') }}",

    aws_conn_id="aws_default",

    steps=SPARK_STEPS,

    # these params are used to fill the paramterized values in
SPARK_STEPS json

    params={

        "BUCKET_NAME": BUCKET_NAME,

        "s3_csv": s3_csv,

        "s3_text": s3_text,

        "s3_script": s3_script,

        "s3_clean": s3_clean,

    },

    dag=dag,

)
```

Code snippet 8. Step Adder task generator

As a result of the Step Adder function, which has been integrated into Airflow and is linked to the task generator, the system is more likely to stay on track and under control. It will be shown on the EMR services interface as well as on the AWS console's web page.

```
step_checker = EmrStepSensor(

    task_id="watch_step",

    job_flow_id="{{
task_instance.xcom_pull('create_emr_cluster',
key='return_value') }}",

    step_id="{{    task_instance.xcom_pull(task_ids='add_steps',
key='return_value')["

    + str(last_step)

    + "] }}",

    aws_conn_id="aws_default",
```

```
    dag=dag,

)
```

Code snippet 9. Step checker task

To follow if each step ran successfully or not, we added an EMR step sensor task on the flow to terminate the process unless there is no conflict of data processing.

```
terminate_emr_cluster = EmrTerminateJobFlowOperator(

    task_id="terminate_emr_cluster",

    job_flow_id="{{
task_instance.xcom_pull(task_ids='create_emr_cluster',
key='return_value') }}",

    aws_conn_id="aws_default",

    dag=dag,

)
```

Code snippet 10. Cluster termination task

Because this project is cost-consuming and needs the control of a single employee, we wanted an automatic cluster termination when the project is completed. Traditional thinking is that this is an experiment for the ETL pipeline that is overseen by a single person with the goal of testing the pipeline. Given the complete pack subscription for fixed-use services, we may not have to be concerned about termination work in the long run, if this is the case (we can use several different services at the same time, and normally a company will pay monthly, quarterly, by 6-month, or annually).

Finally, to bring the DAG flow to a close, a dummy function with an End job is employed.

This is the architect that we created for the tree diagram at the beginning of the DAG file, which is located at the end of the file. The sign a >> b represents the flow of the pipeline from a to b, indicating that we must complete job a before

moving on to task b, or else the whole flow would be halted due to the unexpected mistake. Concurrent execution of the jobs wrapped in square brackets may be achieved by using the syntax [task1, task2,...].

```
start_operator >> [csv_to_s3,scripts_to_s3,txt_to_s3] >> create_emr_cluster

create_emr_cluster >> step_adder >> step_checker >> terminate_emr_cluster

terminate_emr_cluster >> end_operator
```

Code snippet 11: Tree graph establishment

c. Spark Steps Definition

A JSON-formatted file is required in order to build the EMR cluster's steps. To summarise, we moved files from an S3 bucket to HDFS and then processed and scripted them in Python. /flight will serve as the destination for all data. In order to accept the inputs from /flight and return the output to /output, we established an "ETL pipelines" step after shifting the files to the HDFS directory. The AWS S3 clean data location is defined by the S3 clean configuration variable, therefore copy the data from the cluster HDFS location /output to this location.

xcom is used to acquire the EMR cluster id, as stated in the job flow id= "an example of a job.

To establish an EMR cluster, task-ids="create" key="return value") was used. Additionally, we took the most recent step id from xcom and stored it in our EmrStepSensor. Every time a step is taken, the step sensor will check to see whether the one before it was completed, skipped, or cancelled. To be more specific, we consider at this code snippet:

```
[
    {
        "Name":"Move raw data to S3 to HDFS",
```

```
        "ActionOnFailure":"CANCEL_AND_WAIT",
        "HadoopJarStep":{
            "Jar":"command-runner.jar",
            "Args":[
                "s3-dist-cp",
                "--src=s3://{{ params.BUCKET_NAME }}/data_files",
                "--dest=/flight",
            ],
        },
    },
    {
        "Name":"ETL pipelines",
        "ActionOnFailure": "CANCEL_AND_WAIT",
        "HadoopJarStep": {
            "Jar": "command-runner.jar",
            "Args": [
                "spark-submit",
                "--deploy-mode",
                "client",
                "s3://{{ params.BUCKET_NAME }}/{{ params.s3_script }}",
            ],
        },
    },
    {
        "Name": "Move clean data from HDFS to S3",
        "ActionOnFailure": "CANCEL_AND_WAIT",
        "HadoopJarStep": {
            "Jar": "command-runner.jar",
```

```
            "Args": [
                "s3-dist-cp",
                "--src=/output",
                "--dest=s3://{{     params.BUCKET_NAME     }}/{{
params.s3_clean }}",
            ],
        },
    },
]
```

Code snippet 12. Spark step JSON

4.2.6 Cluster Definition

In the DAG.py file, we modified the EMR cluster under the JSON format (dictionary key-value) properties:

- Name: set the name for the job flow
- ReleaseLabel: EMR version
- Applications: The environment which is installed to the cluster, here we used HDFS and Spark.
- Configurations: declare the environment configuration in EMR cluster, like spark-env, directories to python and java
- Instances initialization: using one Master node and two Core nodes On-Demand usage and type m5.xlarge nodes with x86_84 architecture, 4 virtual CPUs and 16GB memories, EBS only storage.

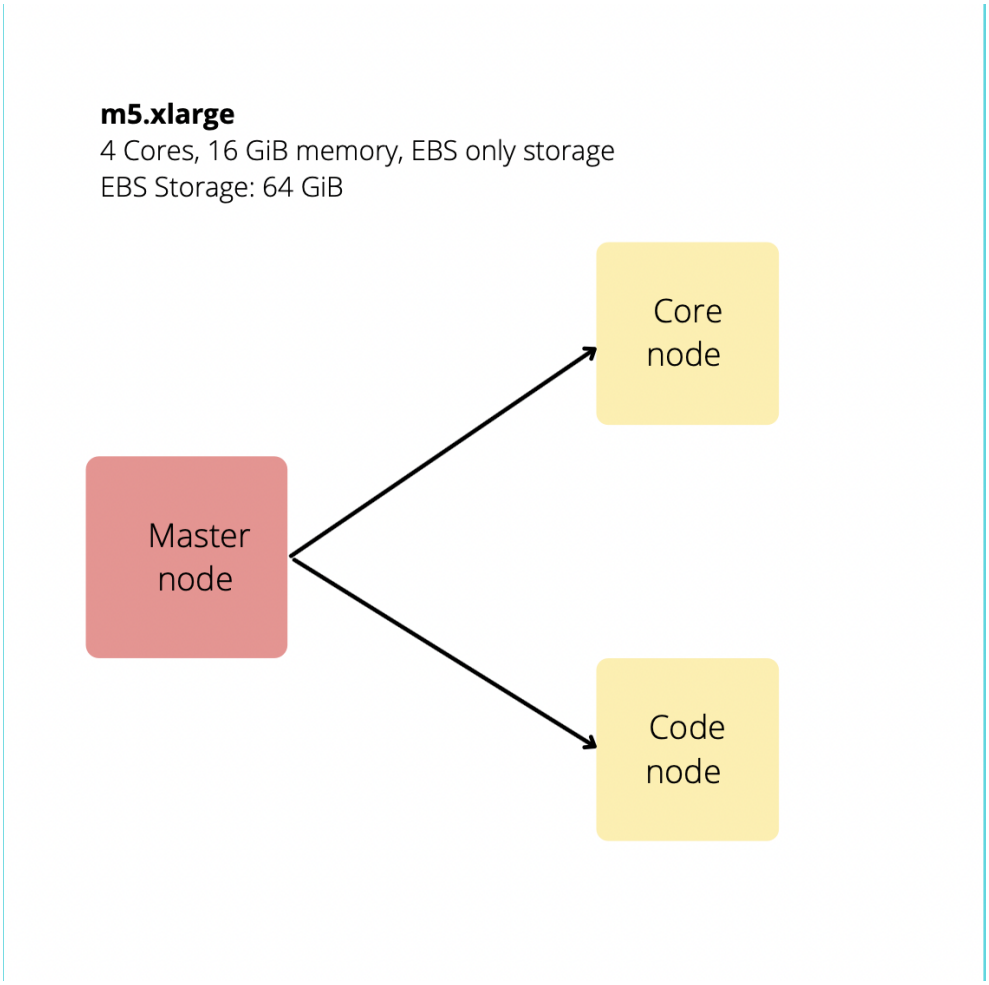

Figure 30. Computing instances used in EMR cluster

We directed the cluster to EC2 key pair, which is generated at 4.2.3.a and set up to configuration of the EMR cluster. Then, we listed the Job Flow role and Service role respectively to EMR_EC2_DefaultRole, EMR_DefaultRole. We created a log storage to record the activities of the flow and it has abilities to check if an error occurs. It helps the Testing task can be implemented. Finally, the cluster was made visible to all authorised users.

```
JOB_FLOW_OVERRIDES={
    "Name":"Flight-cancelation-during-covid19",
    "ReleaseLabel":"emr-5.29.0",
    # In this EMR cluster, we need HDFS and Spark
    "Applications":[{"Name":"Hadoop"},{"Name":"Spark"}],
```

```
    "Configurations":[
        {
        "Classification":"spark-env",
        "Configurations":[
        {
            "Classification":"export",
"Properties":{"PYSPARK_PYTHON":"/usr/bin/python3","JAVA_HOME":
"/usr/lib/jvm/java-1.8.0"},
                }
            ],
        }
    ],
    "Instances":{
        "InstanceGroups":[
            {
                "Name":"Master node",
                "Market":"ON_DEMAND",
                "InstanceRole":"MASTER",
                "InstanceType": "m5.xlarge",
                "InstanceCount":1,
            },
            {
                "Name": "Core Nodes",
                "Market": "ON_DEMAND",
                "InstanceRole": "CORE",
                "InstanceType": "m5.xlarge",
                "InstanceCount": 2,
            },
```

```
        ],
        # This lets us programmatically terminate the cluster
        "KeepJobFlowAliveWhenNoSteps":True,
        "TerminationProtected":False,
        "Ec2KeyName": "spark",
    },
    "JobFlowRole":"EMR_EC2_DefaultRole",
    "ServiceRole":"EMR_DefaultRole",
    "LogUri":"s3://flight-thesis-covid19/job/",
    "VisibleToAllUsers":True
}
```

Code snippet 13. EMR cluster definition

4.2.7 Data processing script

As indicated in 4.2.5.b, we created a simple spark script file entitled data etl.py that received input from /movie and output to /output, both of which were stored in the newly created S3 bucket. This script illustrates how schema works with aggregated and modified fact and dimension tables created using Spark, a module that enables the rapid deployment of enormous amounts of data at a cheaper cost than standard approaches involving database contact.

Include the standard-use modules Spark and Spark SQL at the beginning of the program. Spark Session is required to produce the file and ensure that it works properly. Following that, we constructed new dimension and fact tables from the raw data, depending on our needs, and made an attempt to filter out extraneous entries.

There are two distinct methods for creating and retrieving data from dimension tables. The first one uses a text file, while the second one uses the original raw table in a CSV file. Because the majority of the data lines are contained in a text file, we must use the textFile method to read the file into Spark, and then

change the string element using the list comprehension and string methods. We build the Spark data frame once we wrangle all of the data.

Here is a code snippet for a dimension table is airline table.

```
def create_airline_table(path, spark):
        """
        Function: Generate and create airline table code
        param:
            - path: txt file
        output: airline df
        """
        f = sc.textFile(path)
        content = f.collect()
        content = [x.strip() for x in content]
        #strip(): removes any leading (spaces at the beginning) and trailing (spaces at the end)
        #characters (space is the default leading character to remove)


        airline = content[10:20]
        splitted_airline = [c.split(":") for c in airline]
        c_airline = [x[0].replace("'","").strip() for x in splitted_airline]
        airline_name = [x[1].replace("'","").strip() for x in splitted_airline]
        airline_df = spark.createDataFrame(zip(c_airline, airline_name), schema=['c_airline', 'airline_name'])
        return airline_df
```

Code snippet 14. Wrangle data from text file and create airline dimension table.

4.2.8 Run Airflow on Docker and Airflow UI

Docker Compose is a command-line tool for creating and managing multi-container applications. Compose enables us to declare services in a YAML file and spin up or down the whole system with a single command.

Compose's primary feature is that it allows you to define your application stack in a single file, place it at the root of your project repo (where it is now version controlled), and simply enable people to contribute to your project. A malicious user may just clone your repository and execute the compose application. Indeed, you may see that a number of projects on GitHub/GitLab are now doing this function.

Using docker-compose up, we were able to start all the services in the container, including the web server and databases, in our project's root directory at figure 31.

```
(base) alexnguyendinh@Alexs-MacBook-Pro ~ % cd testing/spark_submit_airflow
(base) alexnguyendinh@Alexs-MacBook-Pro spark_submit_airflow % docker-compose -f docker-compose-LocalExecutor.yml up -d
Recreating spark_submit_airflow-postgres-1 ... done
Recreating spark_submit_airflow-webserver-1 ... done
```

Figure 31. Turn on Airflow container with docker-compose up

We went to the local host address at port 8080 to inspect the Airflow UI. The perfect and working Airflow DAG does not have any errors which were displayed by the red line on top of the UI.(Figure 32)

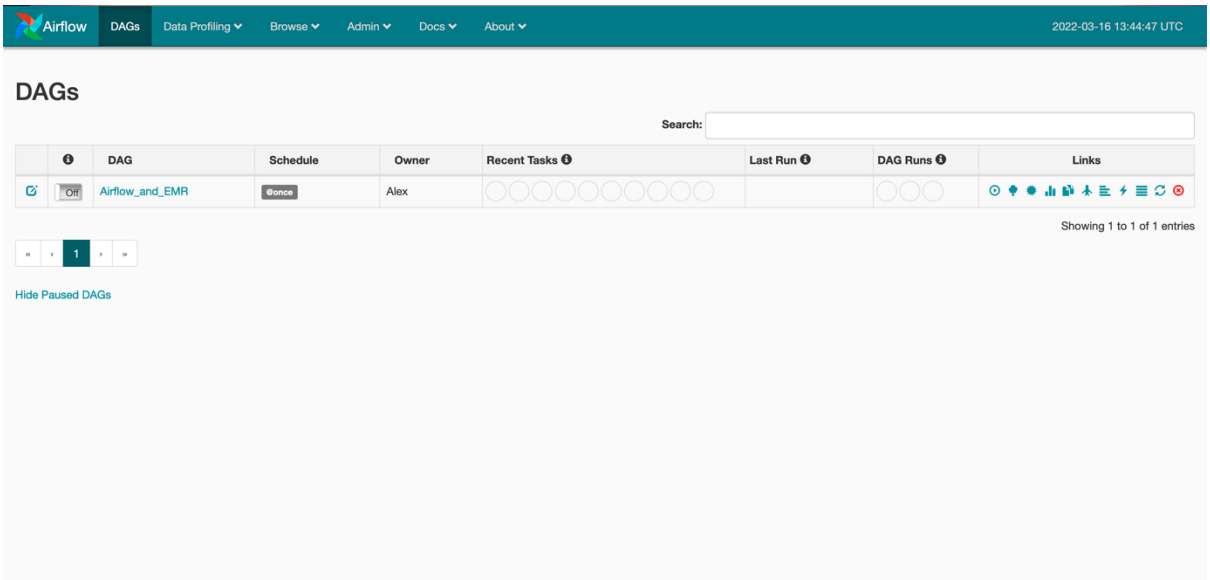

Figure 32. Airflow Web UI

To run the DAG, we turned ON the DAG and clicked to trigger to officially run it.(Figure 33)

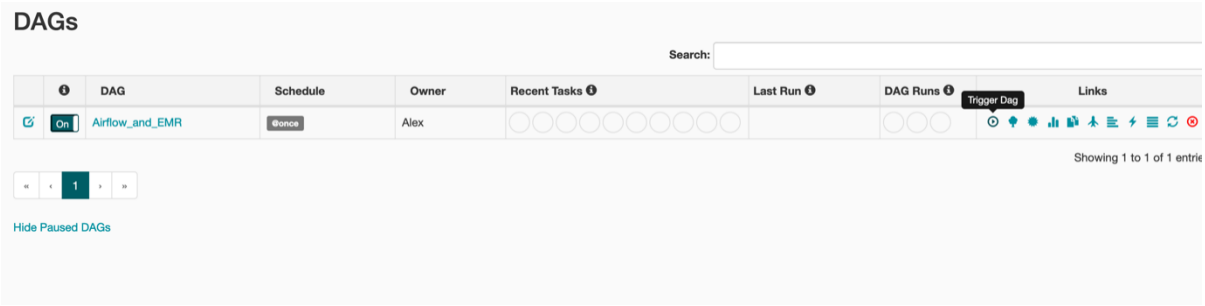

Figure 33. Turn ON and trigger DAG on Airflow web UI at localhost:8080

Following that, we can see Successful Tasks highlighted in pine green, Failed Tasks highlighted in red, Running Tasks highlighted in light green, Waiting Tasks highlighted in cyan, and Upstream Failed highlighted in orange under Recent Tasks (Figure 34). We can easily see the numbers of tasks assigned to each status and have the option to click inside the tasks to examine it in further detail.

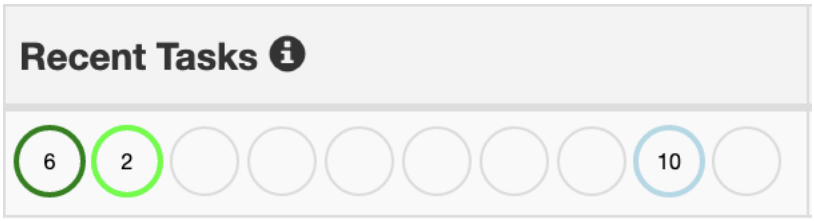

Figure 34. Recent Tasks description

As a representation of "a specific run of a Task," Airflow Task Instances in figure 35 are classified as "having a DAG, a task, and a point of time." Depending on the condition of the Airflow Task Instance, a follow-up loop is shown. The following are some examples of these states: running, succeeding, failing, skipping, and so on. To sum it up, a DAG identifies the parameters of a job. There are both tasks and DAGs written in Python. When it comes to DAG Runs and an execution date, an Airflow Task Instance is different. Airflow Task Instances are "instantiated" executables. Whenever a DAG is activated, a significant number of DAG Runs are created by the scheduler. As on this date, the DAG Runs will be produced by comparing the above-mentioned with the above-mentioned sample code (Arora et al.)

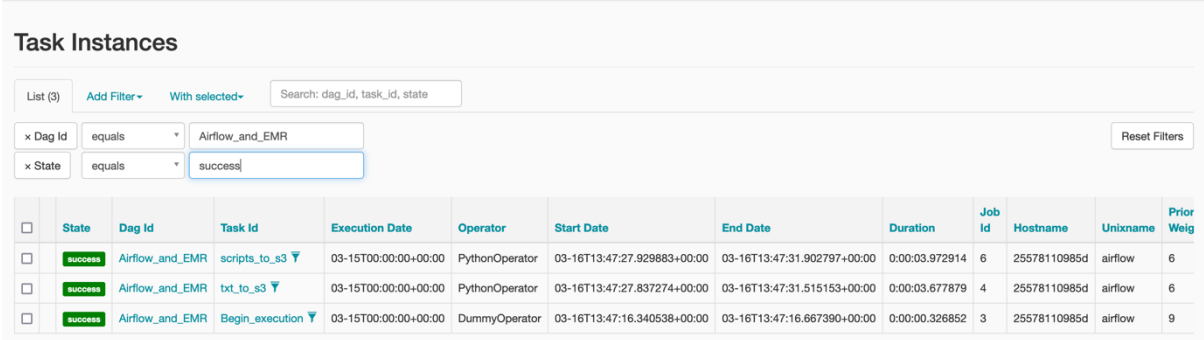

Figure 35. Task Instance

We may analyse the DAG flow by going via a tree view line map, a tree diagram, and a log view for overall tasks.

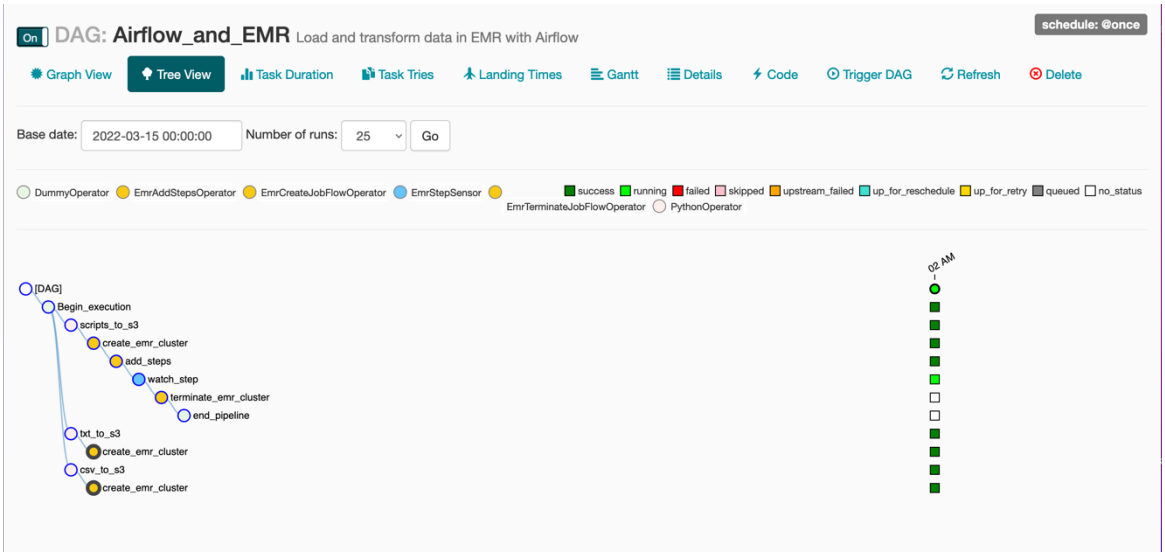

Figure 36. Tree view for running DAG

We've now switched to the AWS EMR interface to analyse the cluster. It takes a few minutes from the time you start creating cluster jobs to the time they start working and are visible in the Cluster listing view. When the Green circle status with the status 'Running' appears, we can navigate around the cluster and reach the 'Step' sections to view the Spark step we established in 4.5.2.c to observe and manage the steps. We may also view the log file to check which tasks are running when anything may stop them. We can see in the below two figures 37 and 38.

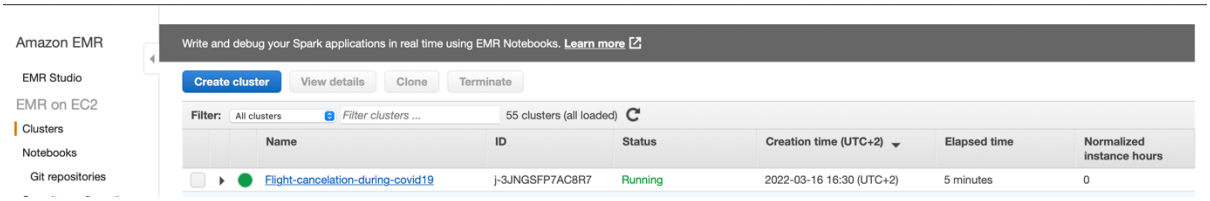

Figure 37. Running cluster

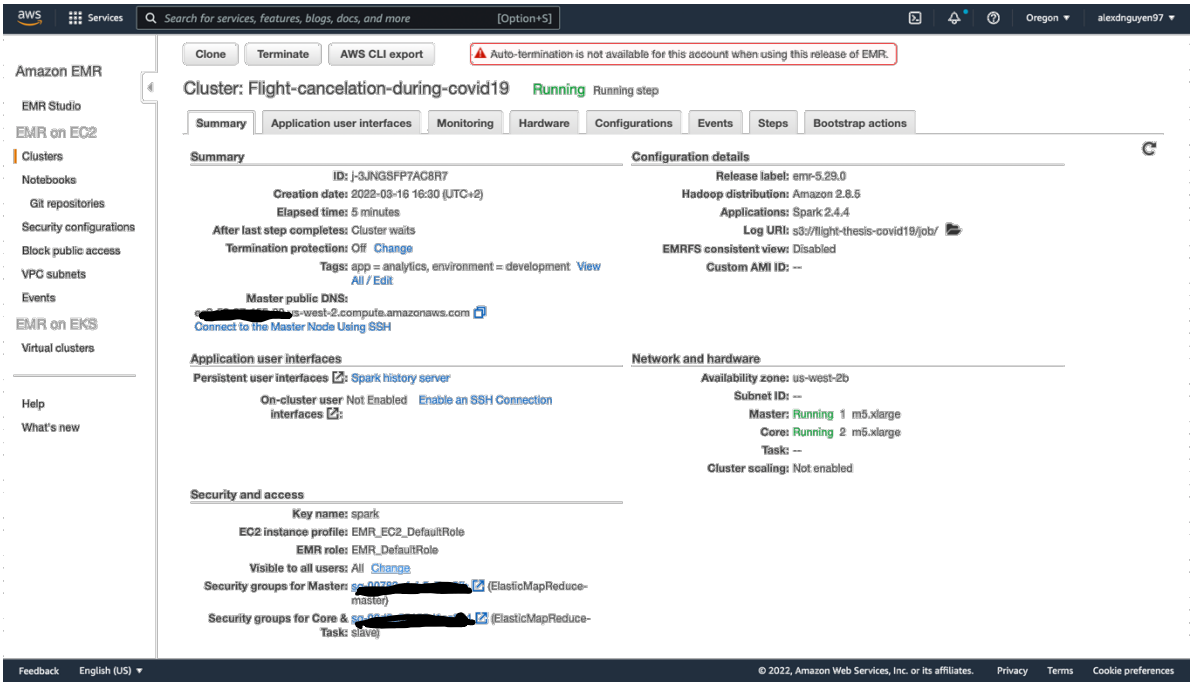

Figure 38. EMR cluster console site

After everything went well, we thoroughly checked to verify whether the cluster and EC2 instances were stopped automatically. Because the expense of not shutting off a service might go into thousands of euros. Finally, we shut down the Docker container using the command docker-compose down to turn off and erase everything on docker in order to save server resources.

```
(base) alexnguyendinh@Alexs-MacBook-Pro spark_submit_airflow % docker-compose -f docker-compose-LocalExecutor.yml down
Stopping spark_submit_airflow_webserver_1 ... done
Stopping spark_submit_airflow_postgres_1  ... done
Removing spark_submit_airflow_webserver_1 ... done
Removing spark_submit_airflow_postgres_1  ... done
Removing network spark_submit_airflow_default
(base) alexnguyendinh@Alexs-MacBook-Pro spark_submit_airflow %
```

Figure 39. Stop all Airflow service when everything completed

4.3 Testing and Quality Check

The S3FS library was used to read S3 parquet files and turn them into Pandas data frames before they could be retrieved.

We used a created function for quality checking and printing out the columns and quantities of values for each dimension and fact tables from /output. Figures 40 and 41 show that the quality tests have been successfully completed and the appropriate results have been issued.

```
airline_df = pd.read_parquet("s3://flight-thesis-covid19/output/airline_dim.parquet/")
cancel_df = pd.read_parquet("s3://flight-thesis-covid19/output/cancel_dim.parquet/")
delay_group_df = pd.read_parquet("s3://flight-thesis-covid19/output/delay_group_dim.parquet/")
dist_group_df = pd.read_parquet(path="s3://flight-thesis-covid19/output/dist_group_dim.parquet/")
port_loc_df = pd.read_parquet(path="s3://flight-thesis-covid19/output/port_loc_dim.parquet/")
state_df = pd.read_parquet(path="s3://flight-thesis-covid19/output/states_dim.parquet/")
fact = pd.read_parquet(path="s3://flight-thesis-covid19/output/fact_dim.parquet/")
```

```
# Source/Count checks to ensure completeness
def quality_check(df, description):
    """
    Function: Data quality check
    args:
        df: data frame
        description: description of table
    output: print status of dataframe
    """

    row,column = df.shape
    if row == 0:
        print("Data quality check failed for {} with zero records".format(description))
    else:
        print("Data quality check passed for {} with {} records and {} columns".format(description, row, column))
    return 0
```

```
quality_check(airline_df, "Airline table")
quality_check(cancel_df, "Cancelation table")
quality_check(delay_group_df, "Delay flight by group table")
quality_check(dist_group_df, "Distribution by group table")
quality_check(port_loc_df, "Port location table")
quality_check(state_df, "States table")
quality_check(fact, "Fact table")
```

Figure 40. Read data from s3 and declare a quality check function

```
Data quality check passed for Airline table with 10 records and 2 columns
Data quality check passed for Cancelation table with 5 records and 2 columns
Data quality check passed for Delay flight by group table with 189 records and 2 columns
Data quality check passed for Distribution by group table with 26 records and 2 columns
Data quality check passed for Port location table with 375 records and 3 columns
Data quality check passed for States table with 52 records and 2 columns
Data quality check passed for Fact table with 2458513 records and 14 columns
```

Figure 41. Quality check passed

4.4 Results

4.4.1 Airflow check

We began by going through the Airflow services available via the web UI. The finished and successful DAG would be added to the original 'Recent tasks' list, along with the nine successful pine green tasks (Figure 42).

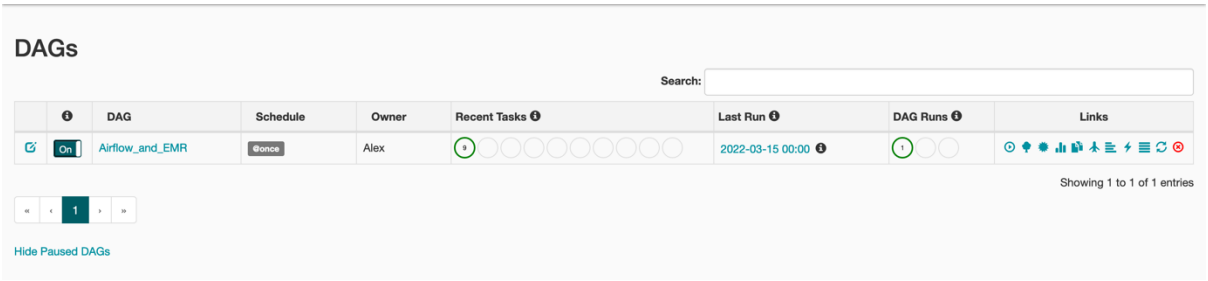

Figure 42. Completed successful task (all 9 tasks in pine green)

The tree line map would also become green in Figure 43.

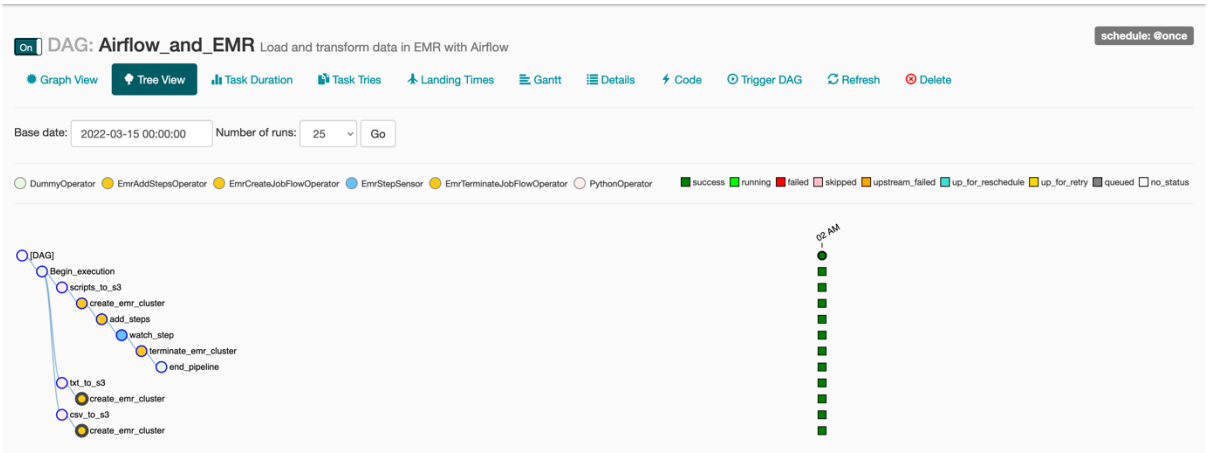

Figure 43: Tree view of completed DAG flow

Graph view diagram either (Figure 44).

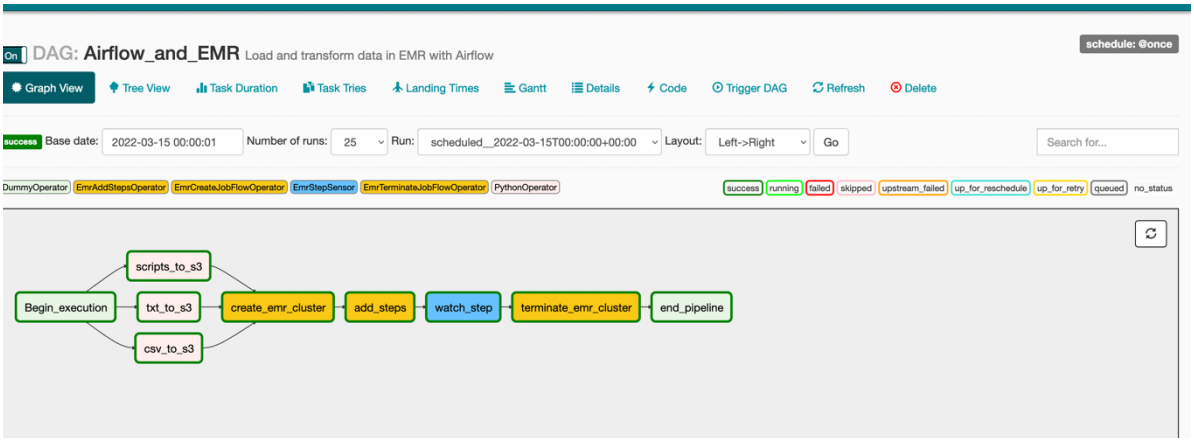

Figure 44. Graph view of completed DAG flow

4.4.2 AWS checking

After inspecting the Airflow web UI, we travelled to the cloud to see the outcome using the EMR and S3 consoles. After inspecting the Airflow web UI, we travelled to the cloud to see the outcome using the EMR and S3 consoles.

The procedures are complete when the status is 'completed' and the cluster is automatically terminated after executing.

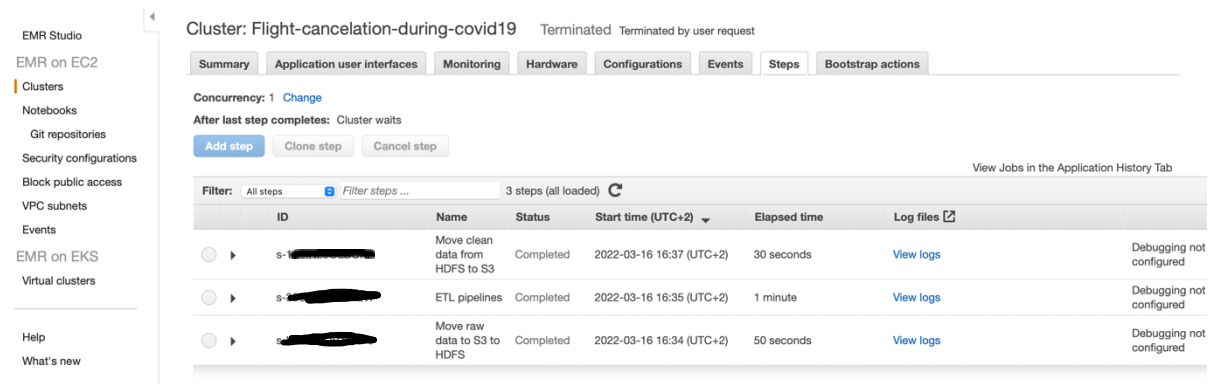

Figure 45. Steps flow completed in EMR consoles and auto terminated cluster

Transferring to S3, we built the four folders for data input and output, as well as the /job directory for storing the log file for examination. I went over and verified the following four figures.

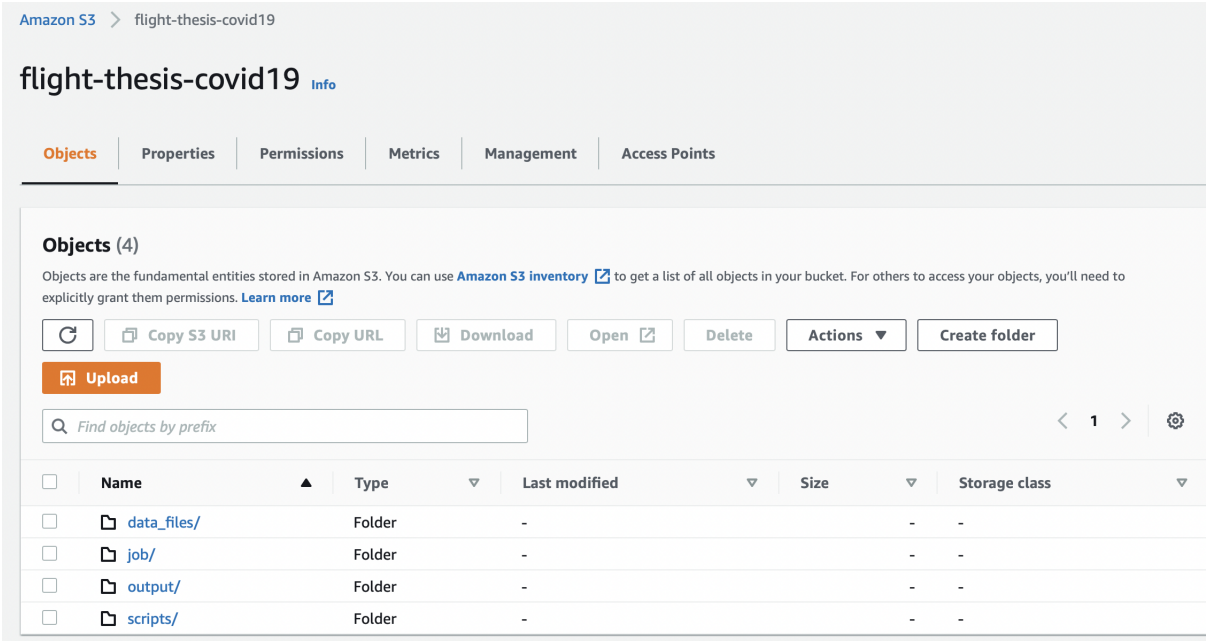

Figure 46. S3 bucket list folders

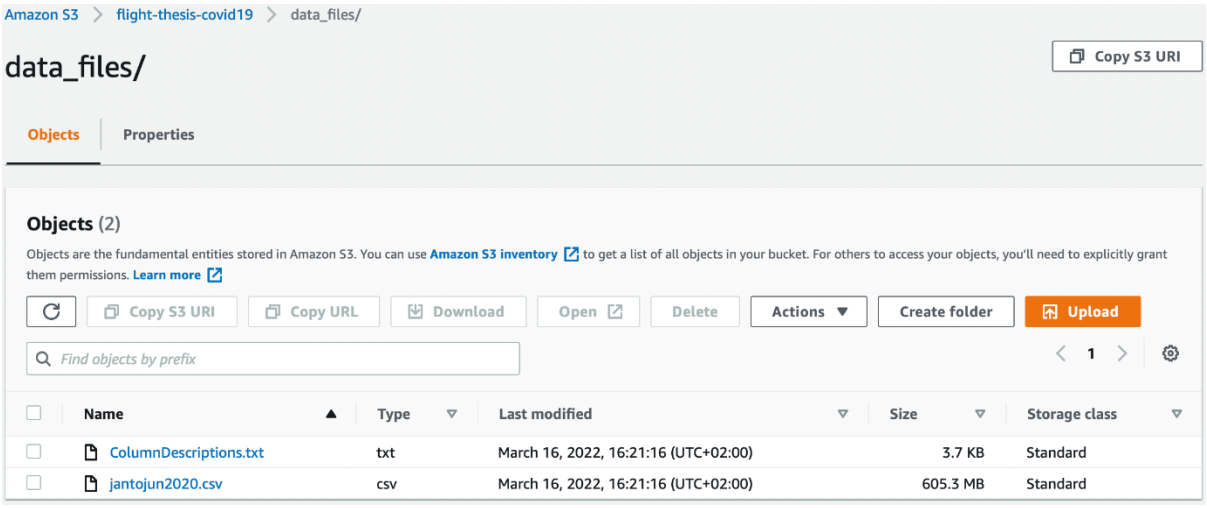

Figure 47. Raw data files on S3

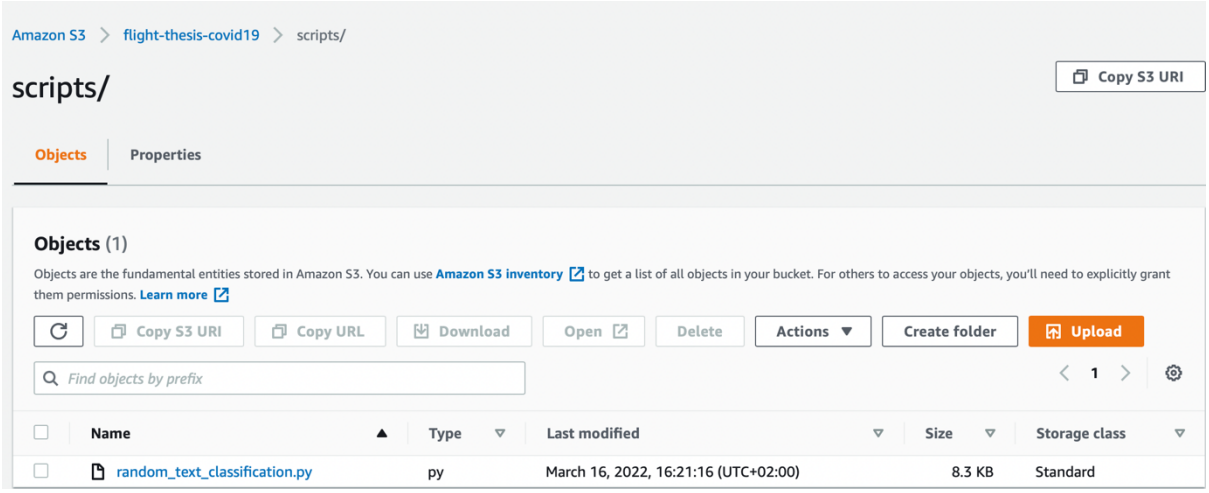

Figure 48. Naïve script on S3

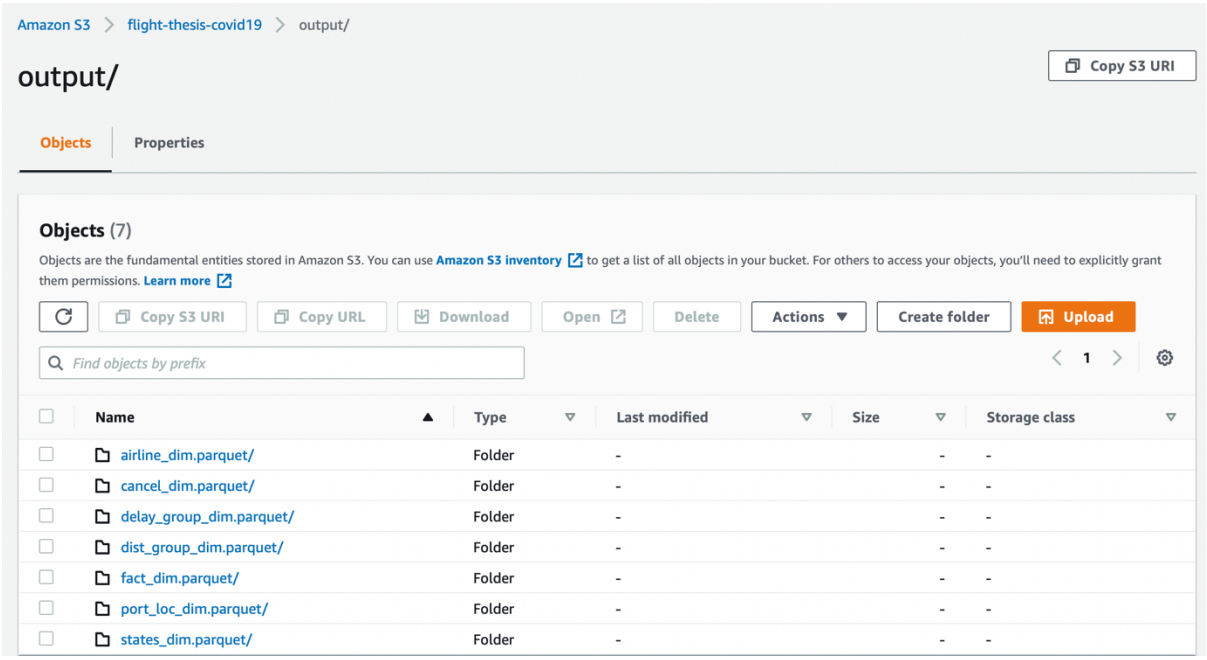

Figure 49. Output parquet files on S3

5 FINANCIAL ASSUMPTION

5.1 Amazon EMR Clusters

This calculation was based on AWS Pricing calculation ("AWS Pricing Calculator" n.d., online) and pricing based on US-West-2 pricing.

Assume you are running an Amazon EMR application on Amazon EC2, with one m5.xlarge EC2 instance serving as the master node and two m5.xlarge EC2 instances serving as core nodes. You will be billed for both the EMR and EC2 nodes. If you operate for one month at 100% usage and utilise on-demand pricing for EC2, your costs will be as follows:

5.1.1 Master Node

EMR charges = 1 instance(s) x 0.048 USD hourly x (100 / 100 Utilised/Month) x 730 hours in a month = 35.0400 USD (EMR master node cost)

EMR master node cost (monthly): 35.04 USD

5.1.2 Core Node

EMR charges = 2 instance(s) x 0.048 USD hourly x (100 / 100 Utilised/Month) x 730 hours in a month = 70.0800 USD (EMR core node cost)

EMR core node cost (monthly): 70.08 USD

5.2 Amazon EC2 Instances

EC2 charges = 3 instances x 0.204 USD x 730 hours in a month = 446.76 USD (monthly on Demand cost)

Amazon EC2 On-Demand instances (monthly): 446.76 USD

5.3 S3 Bucket

5.3.1 S3 Standard

a. Unit Conversion

S3 Standard storage: 20 TB per month x 1024 GB in a TB = 20480 GB per month

Data returned by S3 Select: 5 TB per month x 1024 GB in a TB = 5120 GB per month

b. Pricing Calculations

Tiered price for: 20480 GB

20480 GB x 0.0230000000 USD = 471.04 USD

Total tier cost = 471.0400 USD (S3 Standard storage cost)

10,000 PUT requests for S3 Storage x 0.000005 USD per request = 0.05 USD (S3 Standard PUT requests cost)

50,000 GET requests in a month x 0.0000004 USD per request = 0.02 USD (S3 Standard GET requests cost)

5,120 GB x 0.0007 USD = 3.584 USD (S3 select returned cost)

5 GB x 0.002 USD = 0.01 USD (S3 select scanned cost)

471.04 USD + 0.02 USD + 0.05 USD + 3.584 USD + 0.01 USD = 474.70 USD (Total S3 Standard Storage, data requests, S3 select cost)

S3 Standard cost (monthly): 474.70 USD

5.3.2 Data Transfers

a. Unit Conversion

Inbound:

Internet: 5 TB per month x 1024 GB in a TB = 5120 GB per month

Outbound:

Internet: 5 TB per month x 1024 GB in a TB = 5120 GB per month

b. Pricing Calculations

Inbound:

Internet: 5120 GB x 0 USD per GB = 0.00 USD

Outbound:

Internet: 5120 GB x 0.09 USD per GB = 460.80 USD

Data Transfer cost (monthly): 460.80 USD

S3 charges = 474.70 USD + 460.80 USD = 935.50 USD

5.4 Total charges

Total charges =35.04 USD + 35.04 USD + 446.76 USD + 935.50 USD = 1,496.78 USD (Total 12 months cost: 17,961.36 USD, Includes upfront cost)

6 APPLICATION

The objective of data processing is to provide an approach to integrating all different data sources into a single location that can be used easily for querying, extracting, analysing, visualising, and training models.

Nowadays, end-users can gain insights from data by connecting the datastore and visualising the charts using either technical libraries or code-free platforms, depending on their needs. There are many different types of charts that can be used to help the audience understand the information that the dataset is attempting to convey. Those charts enable the audience or enterprise to make a more informed choice. This process is referred to as Data-Driven Decision Making.

Microsoft Power BI was used as a platform in this project to help with the analysis of the case study's dataset. The charts created from the dataset in the datastore are depicted in the figures 50, 51, 52 below.

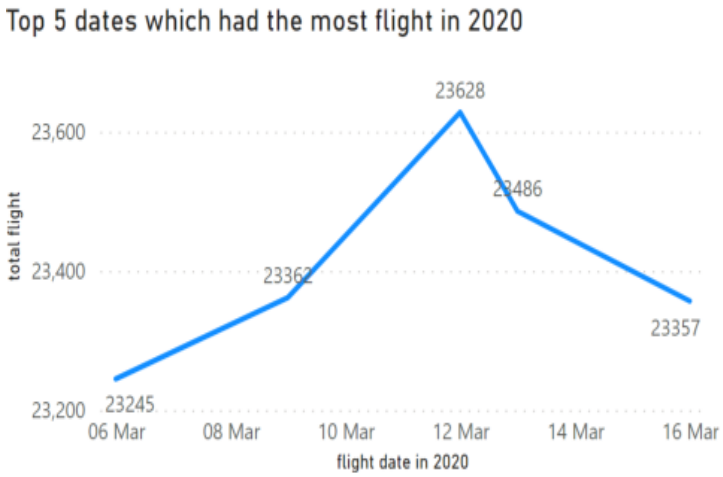

Figure 50. The most flight date in 2020

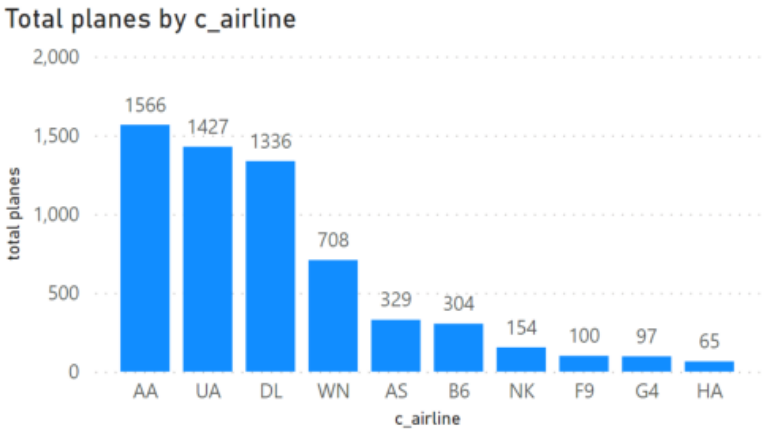

Figure 51. Total planes per airline in US

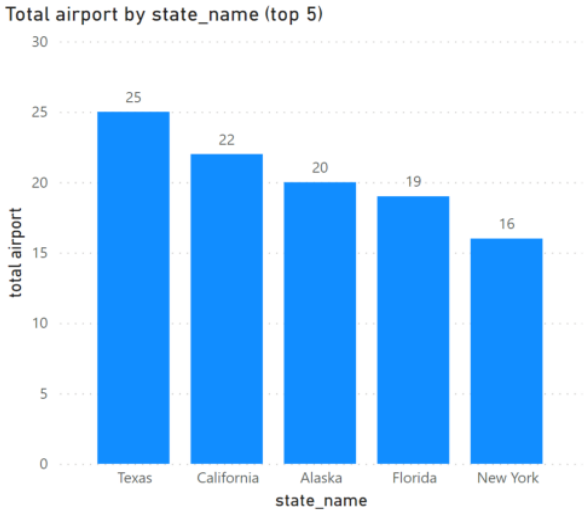

Figure 52. Total airports per state in US

7 CONCLUSIONS

Extracting massive amounts of data from diverse platforms and loading it into a data warehouse is the goal of the ETL process. The importance and benefits of efficient integration models were established in this thesis. This makes it easier for companies to manage their data and gives them a quick access to all of their data in one location. Having all of a company's historical data in one place allows them to make better strategic decisions, discover trends, and predict the future. With more data, it is possible to come up with more accurate findings and generate more sales opportunities.

Various technologies were used to achieve the essential goals of data integration and modelling. The process result will be utilised by upstream departments to solve a range of issues. This thesis supplied deployment strategies, introduced libraries, and demonstrated how to use these libraries for data integration. Useful for solving real-world issues. The findings showed both automated and non-automated ETL methods.

After a successful deployment, ETL processes now execute on a regular basis on servers. The goal of the thesis was achieved, and the ETL scripts were put into use. For big data analytics development, this effort increases the requirement for fast, precise information and a substantial amount of data obtained, while also reducing inaccurate information and the possibility of human error in manual procedures.

REFERENCES

Alooma Team. 2018. What is Apache Airflow? Accessed 18 March 2022https://www.alooma.com/answers/what-is-apache-airflow

Apache Spark - Introduction. n.d. Accessed 18 March 2022 https://www.tutorialspoint.com/apache_spark/apache_spark_introduction.htm

Arora, Yash, et al. Apache Airflow Task Instances: A Complete 101 Guide. Accessed 23 Feb. 2022. https://hevodata.com/learn/airflow-task-instances/#3.

AWS Pricing Calculator. n.d. Accessed 18 March 2022 https://calculator.aws/#/.

Amazon. Creating a bucket - Amazon Simple Storage Service. n.d. Accessed 18 March 2022 https://docs.aws.amazon.com/AmazonS3/latest/userguide/create-bucket-overview.html

DAG Runs. Airflow Documentation., https://airflow.apache.org/docs/apache-airflow/stable/dag-run.html. Accessed 15 Mar. 2022.

Taylor, David. 2022. What is Data Lake? It's Architecture: Data Lake Tutorial. https://www.guru99.com/data-lake-architecture.html [Accessed 18 March 2022].

Davila, M. 2020. How to use pgAdmin. A brief overview. medium.com. Available at from: https://medium.com/@malexmad/how-to-use-pgadmin-a9addc7ff46c [Accessed 18 March 2022].

Dinh, A. n.d. Github. github.com. Available at from: https://github.com/alexdinh1997/Airlines_covid19_ETL_pipelines_with_Airflow_AWS [Accessed 30 March 2022].

Docker Hub Quickstart | Docker Documentation. 2022. docs.docker.com. Available at from: https://docs.docker.com/docker-hub/ [Accessed 18 March 2022].

Driscoll, Mike . n.d. *Jupyter Notebook: An Introduction – Real Python*. realpython.com. Available at from: https://realpython.com/jupyter-notebook-introduction/ [Accessed 18 March 2022].

Docker documentation. “Use Docker Compose | Docker Documentation.” Accessed 16 Feb. 2022. https://docs.docker.com/get-started/08_using_compose/.

Education, IC. 2020. *What is a Data Warehouse?* www.ibm.com. Available at from: https://www.ibm.com/cloud/learn/data-warehouse [Accessed 18 March 2022].

Education, IC. 2020. *What is Data Modeling?* www.ibm.com. Available at from: https://www.ibm.com/cloud/learn/data-modeling [Accessed 18 March 2022].

Education, IC. 2020. *What is ETL (Extract, Transform, Load)?* www.ibm.com. Available at from: https://www.ibm.com/cloud/learn/etl [Accessed 18 March 2022].

ETL process overview: design, challenges and automation. 2020. www.keboola.com. Available at from: https://www.keboola.com/blog/etl-process-overview [Accessed 18 March 2022].

Ghosh, Abhishek. 2021. What is a Snowflake Schema? Accessed 18 March 2022https://thecustomizewindows.com/2021/11/what-is-a-snowflake-schema/

Knapp, Sean. 2022. *What is Data Orchestration? | Ascend.io*. www.ascend.io. Available at from: https://www.ascend.io/blog/data-orchestration/ [Accessed 18 March 2022].

Kindl, C. n.d. *twitter-data-pipeline-using-airflow-and-apache-spark,christopherkindl*. githubhelp.com. Available at from: https://githubhelp.com/christopherkindl/twitter-data-pipeline-using-airflow-and-apache-spark [Accessed 18 March 2022].

Nguyen, A. n.d. *GitHub*. github.com. Available at from: https://github.com/AnhKun/airline_flight_delay_cancellation_US [Accessed 30 March 2022].

Orlando, Lynn. “What Is Data Orchestration? A Guide to Handling Modern Data | WEKA.” *WEKA*, www.weka.io, 9 Aug. 2021,

PostgreSQL: About. n.d. www.postgresql.org. Available at from: https://www.postgresql.org/about/ [Accessed 1 April 2022].

Running Airflow in Docker; Airflow Documentation, airflow.apache.org, https://airflow.apache.org/docs/apache-airflow/stable/start/docker.html. Accessed 15 Mar. 2022.

What is a Container? - Docker. 2022. www.docker.com. Available at from: https://www.docker.com/resources/what-container [Accessed 18 March 2022].

What is a Data Source? - Definition from Techopedia. n.d. www.techopedia.com. Available at from: https://www.techopedia.com/definition/30323/data-source [Accessed 18 March 2022].

What is Data Center Orchestration? Definition & FAQs | Avi Networks. 2022. avinetworks.com. Available at from: https://avinetworks.com/glossary/data-center-orchestration/ [Accessed 18 March 2022].

What is Database ? - GeeksforGeeks. 2021. www.geeksforgeeks.org. Available at from: https://www.geeksforgeeks.org/what-is-database/ [Accessed 18 March 2022].

What is PySpark? - Databricks. 2022. databricks.com. Available at from: https://databricks.com/glossary/pyspark [Accessed 18 March 2022].

What is Python? Executive Summary. n.d. www.python.org. Available at from: https://www.python.org/doc/essays/blurb/ [Accessed 1 April 2022].