Two heads are better than one: Pairing work on building our first Data Engineering Pipeline in 3 Steps

Currently I am taking a Data Science Bootcamp. As a student coming from business background without coding or data analyzing experience before, it can be scaring for me even just seeing at the title: **Building Up the Data Engineering Pipeline.**
But till I met my classmate Stephanie, we both somehow lost the pace in the class and end up meeting each other at the tutor checkpoint. We decided to work in pairing model as we believe two heads are better than one. And at the end, we made it work as advanced students do. This is the reason why I write this blog and would like to share you how you can build your own **Data Engineering Pipeline just in 3 ETL steps:**

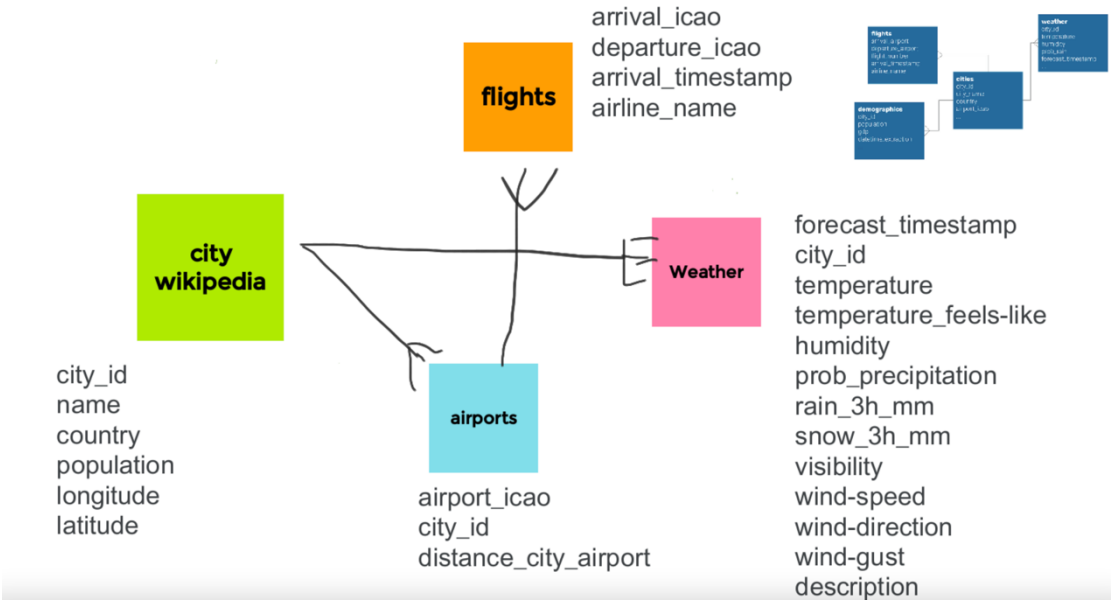
- **Ingest and Process Data**
- **Store and Create Data Pipelines**
- **Connecting Data on the Cloud**

Analytical Business Cases:

Our task is to help the company Goos - the startup developing an e-scooter-sharing system, to collect Data that will help the company to relocate the scooter-movements from A to B in big airport cities. The more flights arrive in the airport, the more frequently e-scooters are used. **The company needs to know in advance how to arrange forklifts to deploy e-scooters to the point of use.** Of course, the company also has other corresponding policies, for example, rebate user money to incentivize users to return to the indicated e-scooter points, etc.. But we cannot expect all users will do so. They are

Let's get started:

Are we able to analyze the use of e-scooters at the airport? It all depends. Depends on the data , data and data. We need to know the arrival of flights. Weather factors also affect the frequency of e-scooter use, if it rains and snows, users will prefer not to use an e-scooter. So, we first drafted some of the schema we needed with pen and paper. For example, the following chart.



1 Ingesting and Cleaning Data

Having a rough bucket of data, we split up to collect data. Our next thought is this. Through web scraping from Wiki to get the city data including population and geography information etc. Through API end to request Flights/Weather/Airport data in python.

There are many ways to retrieve information about cities from. We simply grab data from Wikipedia. The global community takes care to frequently update and curate the data, so we just need to care about grabbing the right numbers or names.

1.1 Collecting Data- Cites and Geography Airports:

Here is an example what methods and the function we used to request data in Python.

Wikipedia city scraping

```
def scrape_wiki_cities() -> DataFrame:
    from bs4 import BeautifulSoup
    import requests

    #get html code
    doc_url = 'https://en.wikipedia.org/wiki/List_of_cities_in_the_European_Union_by_population_within_city_limits'
    response = requests.get(doc_url)
    if response.status_code != 200:
        raise Exception(f'wikipedia returned code {response.status_code} for url = {doc_url}')
    soup = BeautifulSoup(response.content, 'html.parser')
    table = soup.select('table.wikitable > tbody > tr')

    # prettify the names and take only selected ones
    header = [h.text.strip().replace(' ', '_').lower() for h in table[0].select('th')][1:-2]
    cities = [[cell.text.strip() for cell in city.select('td')[1:-2]] for city in table[1:]]

    import pandas as pd
    return pd.DataFrame(data=cities, columns=header)

cities = scrape_wiki_cities()
```

```
from pandas import DataFrame
def cleanup_cities(df : DataFrame):
    import pandas as pd
    df.loc[:, 'officialpopulation'] = df['officialpopulation'].str.replace(',', '').astype(int)
    df.loc[:, 'date'] = pd.to_datetime(df['date'])

cleanup_cities(cities)
cities.info()
```

1.2 Collecting Active Data- Weather , Airports & Flights:

Since weather and flight data are dynamic and vary from time to time. So we need data that can be updated at any time. Thanks to the API provider, we can read the real-time weather and flight data directly, and there are many resources on the web. As a student, it is best to experiment with the Freemium, the disadvantage is the limit of requests. So, for commercial purposes, you can pay for the usage directly.

```
def get_weather(weather_arguments : dict) -> dict:
    import requests
    import json
    #check that only string arguments are present
    if not all(isinstance(val, str) for val in weather_arguments.values()):
        raise Exception('all arguments must be string')

    #preparing the request url
    weather_api = "http://api.openweathermap.org/data/2.5/forecast?"
    api_arguments = repr(weather_arguments).replace("': '", '=').replace("', '", '&')[2:-2]
    weather_request = weather_api + api_arguments

    response = requests.get(weather_request)
    if response.status_code != 200:
        raise Exception(f'openweather returned code {response.status_code} for url = {weather_request}')
    return response.json()

# weather_arguments defined at the top of the page
weather_json = get_weather(weather_arguments)
```

```
# handle nan values
from pandas import DataFrame
def cleanup_weather(df : DataFrame):
    import pandas as pd
    if 'rain_3h_mm' in df:
        df.loc[:, 'rain_3h_mm'] = df['rain_3h_mm'].fillna(0)
    if 'snow_3h_mm' in df:
        df.loc[:, 'snow_3h_mm'] = df['snow_3h_mm'].fillna(0)
    df.loc[:, 'date'] = pd.to_datetime(df['date'])

cleanup_weather(weather_df)
weather_df.info()
```

```
weather_df.describe()
```

	temp_celcius	temp_feels_like_celcius	humidity_percent	clouds_percent	wind_speed_meter_sec	wind_direction_degree	wind_gust_meter_sec	pop_percent
count	1.00	1.00	1.0	1.0	1.00	1.0	1.00	1.00
mean	25.34	25.08	44.0	0.0	2.12	242.0	4.83	0.02
std	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
min	25.34	25.08	44.0	0.0	2.12	242.0	4.83	0.02
25%	25.34	25.08	44.0	0.0	2.12	242.0	4.83	0.02
50%	25.34	25.08	44.0	0.0	2.12	242.0	4.83	0.02
75%	25.34	25.08	44.0	0.0	2.12	242.0	4.83	0.02
max	25.34	25.08	44.0	0.0	2.12	242.0	4.83	0.02

```
test_cities
```

	city	date	temp_celcius	temp_feels_like_celcius	humidity_percent	weather_description	clouds_percent	wind_speed_meter_sec	wind_direction_degree	wind_gust_meter_sec	pop_percent	rain_3h_mm	pod
0	Kraków	2022-06-08 15:00:00	20.74	20.83	75	light rain	75	1.51	67	2.07	0.61	0.47	d
1	Kraków	2022-06-08 18:00:00	19.74	19.88	81	light rain	82	2.25	64	4.19	0.91	2.17	d
2	Kraków	2022-06-08 21:00:00	17.41	17.48	87	light rain	81	0.89	84	1.09	0.72	0.58	n
3	Kraków	2022-06-09 00:00:00	14.90	14.90	94	light rain	89	0.89	99	1.00	0.59	0.13	n
4	Kraków	2022-06-09 03:00:00	14.31	14.25	94	light rain	82	0.73	96	0.88	0.36	0.19	d
...	...	...	...	...	...	...	...	...	...	...	...	...	...
715	Vila Nova de Gaia	2022-06-13 00:00:00	20.18	20.32	79	clear sky	4	1.61	147	1.97	0.00	0.00	n
716	Vila Nova de Gaia	2022-06-13 03:00:00	19.80	19.85	77	clear sky	0	1.23	121	1.48	0.00	0.00	n
717	Vila Nova de Gaia	2022-06-13 06:00:00	19.99	20.06	77	clear sky	0	0.89	153	1.35	0.00	0.00	d
718	Vila Nova de Gaia	2022-06-13 09:00:00	24.61	24.75	62	clear sky	0	0.90	339	1.85	0.00	0.00	d
719	Vila Nova de Gaia	2022-06-13 12:00:00	26.98	27.85	57	clear sky	3	3.15	283	2.94	0.00	0.00	d

720 rows × 13 columns

```
def flightdata2(icao):
    url = f"https://aerodatabox.p.rapidapi.com/flights/airports/icao/{icao}/2022-06-10T00:23/2022-06-10T11:00"

    querystring = {"withLeg":"true","direction":"Arrival"}

    headers = {
        "X-RapidAPI-Key": "yourkey",
        "X-RapidAPI-Host": "aerodatabox.p.rapidapi.com"
    }
    ICAOcode = requests.request("GET", url, headers=headers, params=querystring)
    ICAOcodedf=pd.json_normalize(ICAOcode.json())
    flights=pd.json_normalize(ICAOcode.json()['arrivals'])
    flights2=flights.drop(flights[flights['isCargo']==True].index,).copy()
    flights2=flights2.drop(flights2[flights2['status']=='Arrived'].index,)
    flights2=flights2.drop(flights2[flights2['status']=='CanceledUncertain'].index,)
    flights2=flights2.drop(flights2[flights2['status']=='Canceled'].index,)
    flights2.drop(
        ['codeshareStatus',
         'callSign',
         'departure.airport.iata',
         'departure.airport.name',
         'departure.scheduledTimeLocal',
         'departure.actualTimeLocal',
         'departure.scheduledTimeUtc',
         'departure.actualTimeUtc',
         'departure.terminal',
         'departure.checkInDesk',
         'departure.gate',
         'departure.quality',
         'departure.runwayTimeLocal',
         'departure.runwayTimeUtc',
         'departure.runway',
         'arrival.runwayTimeLocal',
         'arrival.runwayTimeUtc',
         'arrival.runway',
         'aircraft.reg',
         'aircraft.modeS',
         'arrival.actualTimeUtc',
         'arrival.scheduledTimeUtc',
         'arrival.terminal',
         'arrival.quality',
         'aircraft.model',
         'status',
         'isCargo'
        ],axis=1,inplace=True)

    flights2.rename(columns = {
        'departure.airport.icao':'departure_airport_icao',
        'arrival.scheduledTimeLocal':'arrival_scheduledTimeLocal',
        'arrival.actualTimeLocal':'arrival_actualTimeLocal',
        'arrival.scheduledTimeUtc':'arrival_scheduledTimeUtc',
        'arrival.actualTimeUtc':'arrival_actualTimeUtc',
        'arrival.terminal':'arrival_terminal',
        'arrival.baggageBelt':'arrival_baggageBelt',
        'arrival.quality':'arrival_quality',
        'aircraft.model':'aircraft_model',
        'arrival.baggageBelt':'arrival_baggageBelt',
        'airline.name':'airline_name'},
                    inplace = True)
    flights2=flights2.assign(arrival_icao= icao)
    return(flights2)
```

```
flightdata2('EDDB')
```

	number	departure_airport_icao	arrival_scheduledTimeLocal	arrival_actualTimeLocal	airline_name	arrival_icao
13	FR 2919	LDZD	2022-06-10 08:30+02:00	NaN	Ryanair	EDDB
19	U2 5182	LIML	2022-06-10 08:45+02:00	NaN	easyJet	EDDB
25	DY 1102	ENGM	2022-06-10 09:05+02:00	NaN	Norwegian Air Shuttle	EDDB
27	EW 8001	EDDS	2022-06-10 09:30+02:00	NaN	Eurowings	EDDB
28	EW 8059	EDDK	2022-06-10 09:40+02:00	NaN	Eurowings	EDDB
42	U2 5630	NaN	2022-06-10 09:45+02:00	NaN	easyJet	EDDB
63	U2 5512	NaN	2022-06-10 10:30+02:00	NaN	easyJet	EDDB
64	U2 5762	LFMN	2022-06-10 10:45+02:00	NaN	easyJet	EDDB
66	U2 6819	EGPF	2022-06-10 10:05+02:00	NaN	easyJet	EDDB

```
!pip install pymysql
```

1.3 Cleaning data

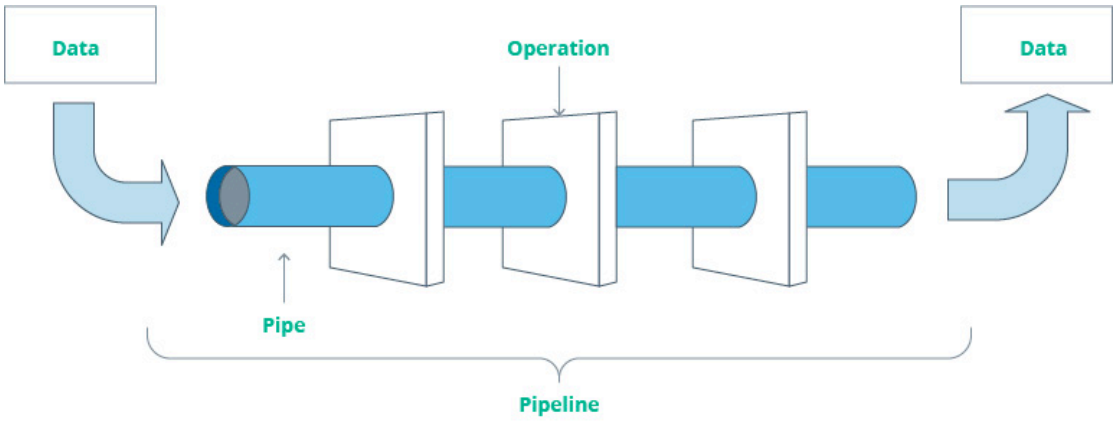
After a week of working day and night, the data was cleaned up. Now we need to save the data to SQL for testing. Data cleanup is a hard but rewarding job. I won't go into details. But here are two Python libraries in particular to mention working this project.

1) Using JSON to endcoing data. The data extracted from the Internet is quite messy, I personally prefer to view the data list in the form of DataFrame. One library that helps particularly well in python is JSON. To use it you just need to import directly: import Json.

2) Using the Datetime module in Python to format the time. This package provides classes that can manipulate dates and times in a variety of ways. While supporting datetime mathematical operations, the implementation focuses more on how its properties can be more efficiently parsed for formatting output and data manipulation. Our project needs to know exactly what time the plane arrived.

```
import pandas as pd
from datetime import datetime, date, timedelta
import json
from pytz import timezone
```

2 Store and Create Data Pipelines



2.1 Connecting data to MySQL and create table to test it locally.

After the data is manipulated. We can use the data we collect and allocate all in SQL to create tables. We type the commands to create database as shown in the below.

```
1 •  create database yourprojectname;
2 •  use yourprojectname;
3 •  select * from tablename;
```

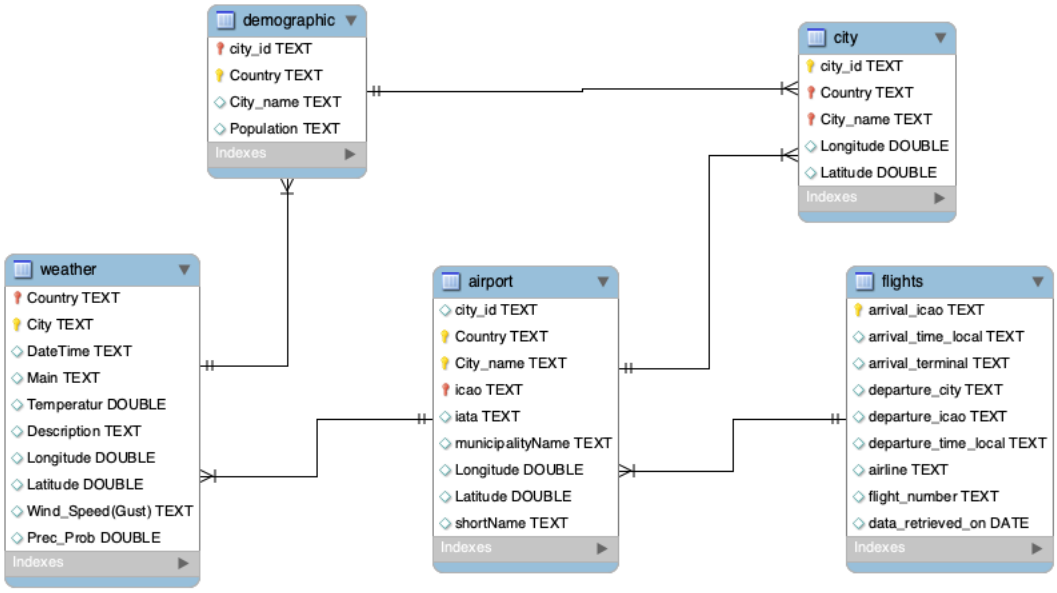
SQL has two main advantages: it allows you to access many records at once, and group, filter or aggregate them. It also has different types and can be grouped to form relations. Most programming languages let you do that, but SQL was the first, which is why it's been so influential.
Before we can push all the date to SQL we need to connect Python with SQL.
First import **pymysql** and use your SQL profile info like below to tell Python connecting it.
After that you shall be able to see your tables by refreshing it on SQL end.

```
schema="yourschema"
host="awshostsaddress"
user="admin"
password="password."
port=3306
con = f'mysql+pymysql://{user}:{password}@{host}:{port}/{schema}'
```

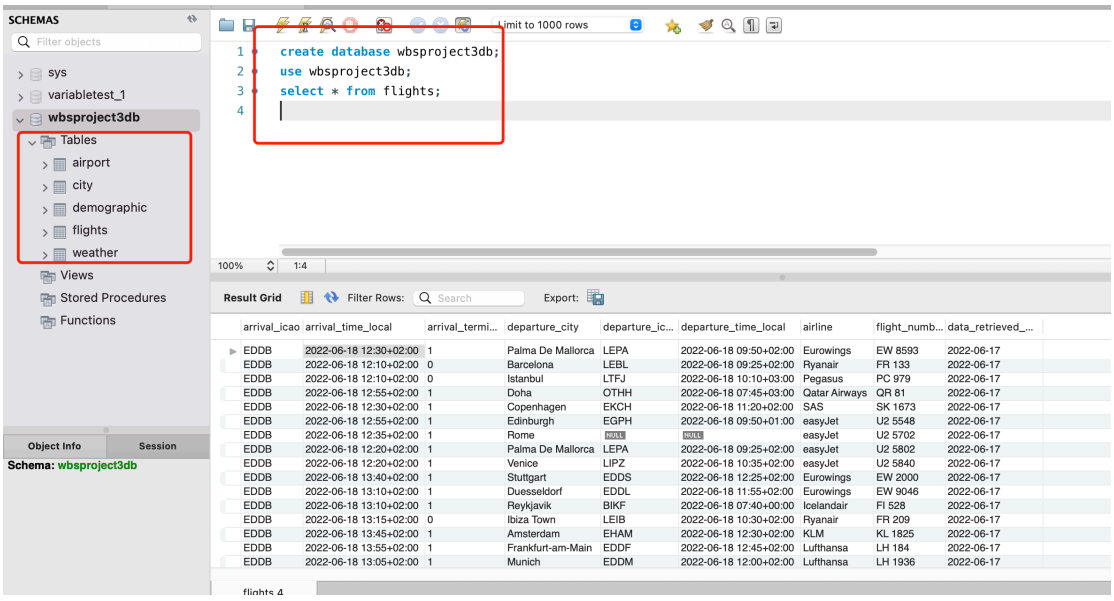
```
airport.to_sql('airport',
               if_exists='append',
               con=con,
               index=False)
```

```
50
```

2.2 Create an SQL data model, crafting the relationships between tables.



2,3 Playing around and test the tables if they are related



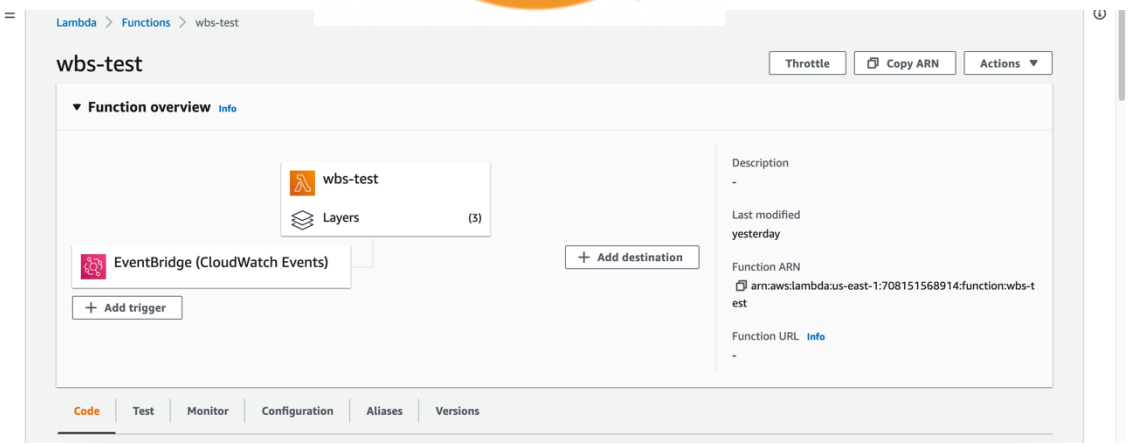
3 Testing running data pipeline on airflow

After we test the tables are working well in local computer. We want to transfer them into cloud. The reasons why we want to run all the data on third party(cloud) is simply due to cost of building up data processing environment too cost effective.

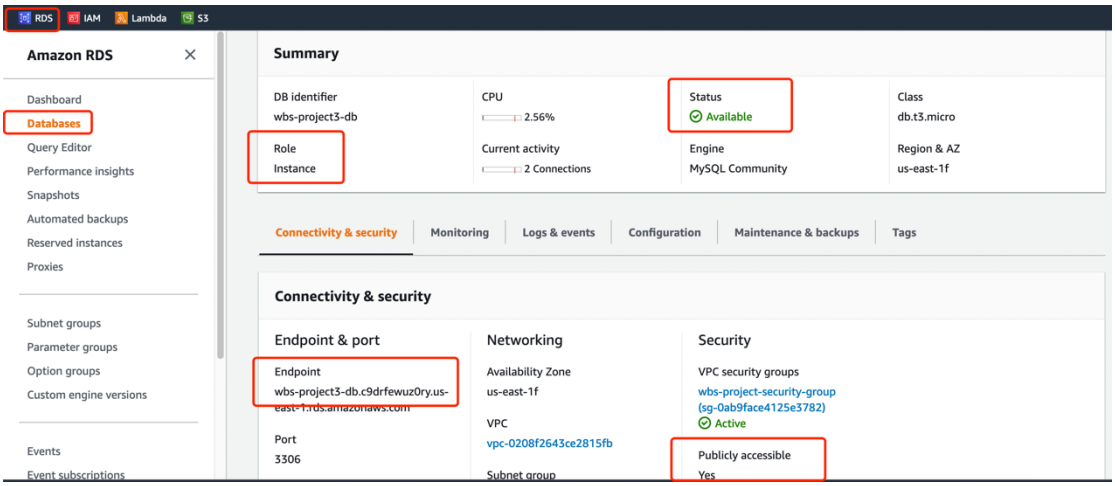
Companies can process data in their own data centre. We can imagine racks of servers, ready to be used, that the company has to buy. We also need a room to store them, and if we move offices, we have to transport servers without losing service. The electrical bill and maintenance would be at the company's cost. Moreover, data processing tasks can be more or less intense, and don't happen continuously.

In the cloud, we rent servers, and the rent is cheap. We don't need a room to store them, and we use the resources we need, at the time we need them. Many companies moved to the cloud as a way of cost optimization. Also, the closer the server is to the user, the less latency they will experience when using our application. To serve a global customer base, we need servers all over the world. That is also realistic solution for the start-up company - Goose e- scooter company.

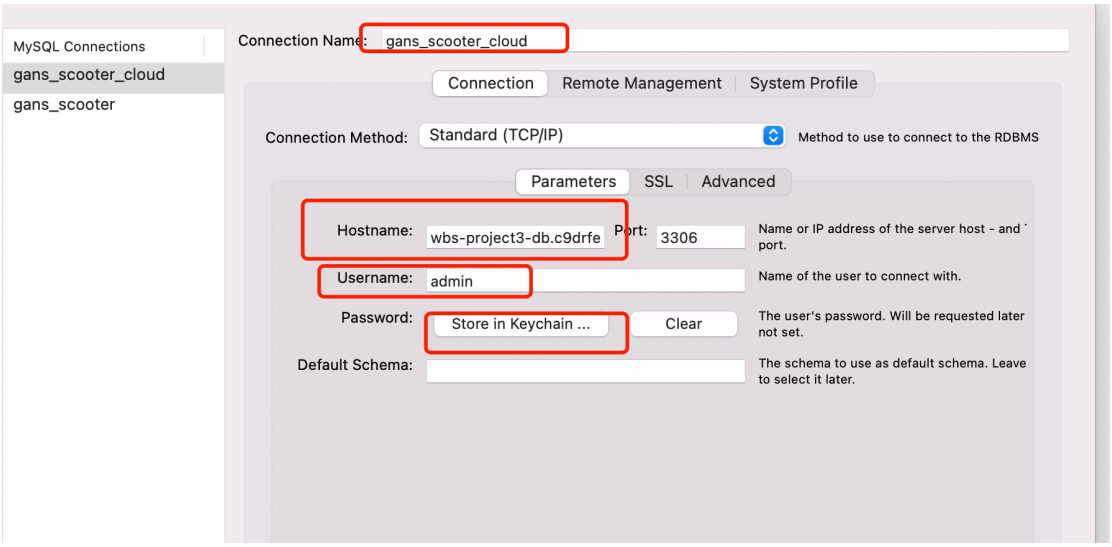
The three big players, in decreasing order of market share, are Amazon Web Services. In this project we will use the AWS Free Tier.



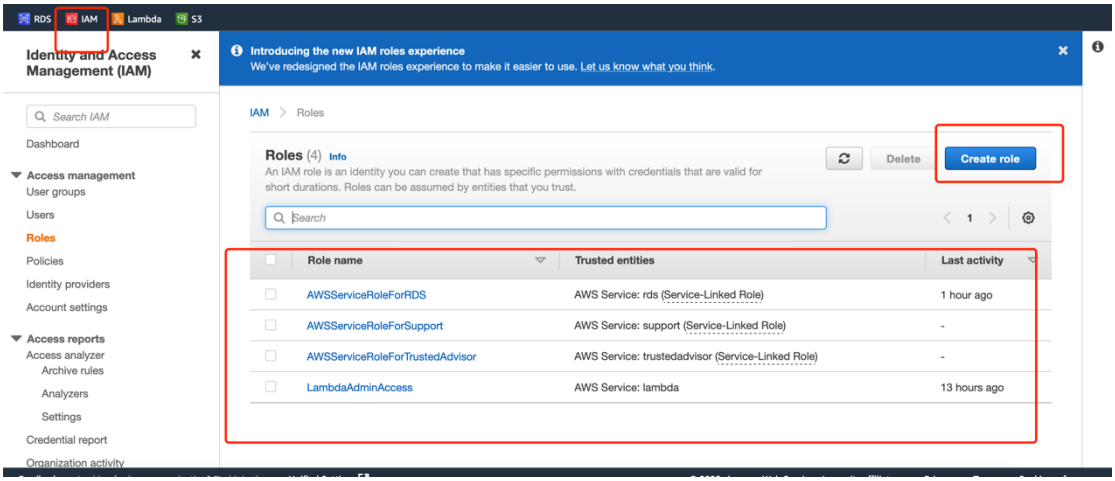
Let first set up the RDS. Just type RDS in the search bar and you will see it and fill up setting up columns and wait till the RDS active is. then it can be using.



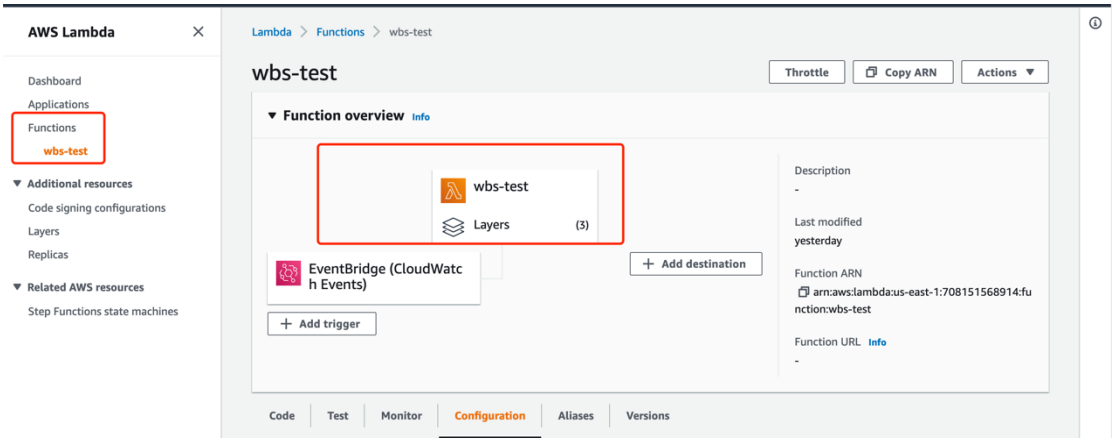
After RDS is active, we can go to SQL workbench to create instance connecting with our cloud RDS in AWS. Make sure copy the Endpoint link into the Hostname column and use the RDS login name from AWS for SQL to access cloud RDS.



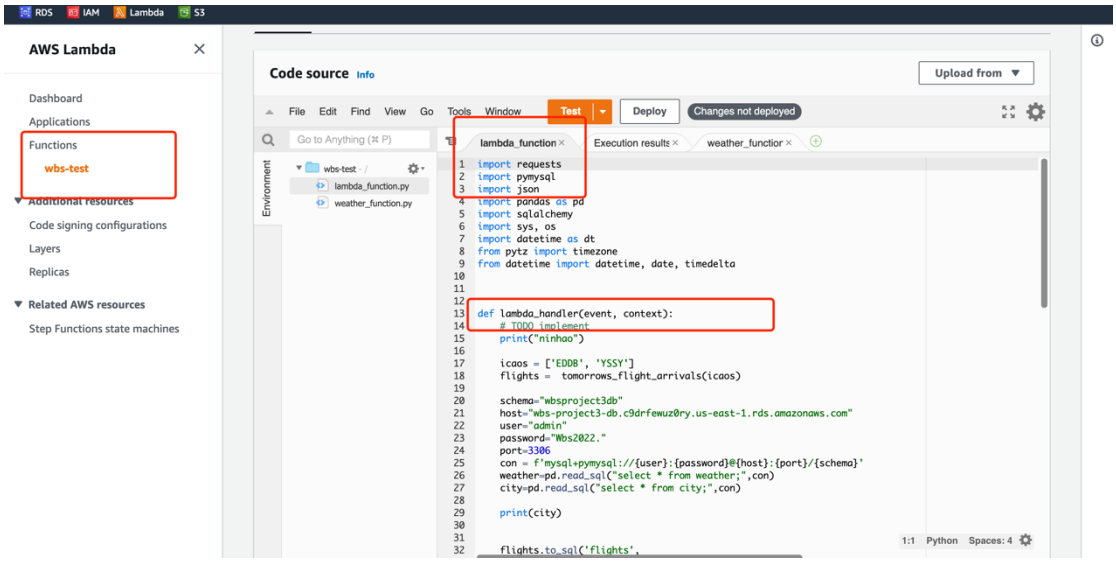
After having SQL cloud instance settled down. we need to set up IMA **-** AWS Identity and Access Management (IAM) is a web service that helps you securely control access to AWS resources. You use IAM to control who is authenticated (signed in) and authorized (has permissions) to use resources.



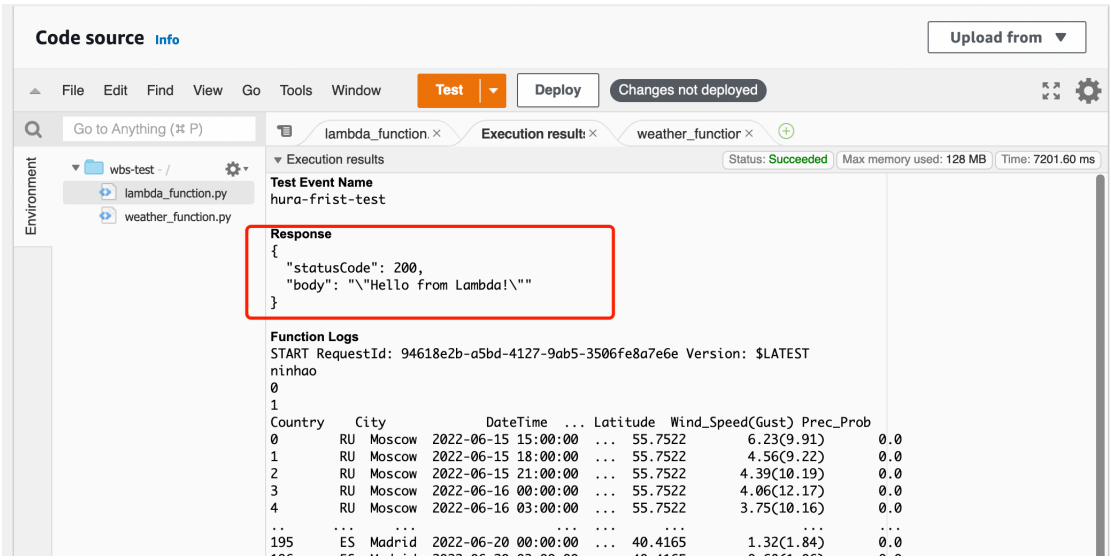
Now, it comes the exacting moment. Let's create our fist Lambda function. Go to the Lambda page and select function, Select functions in the left sidebar, then create function. Lambda will create a role for itself with basic permissions to write to CloudWatch logs, but we can add more later. After creating the function, we see the main configuration page.



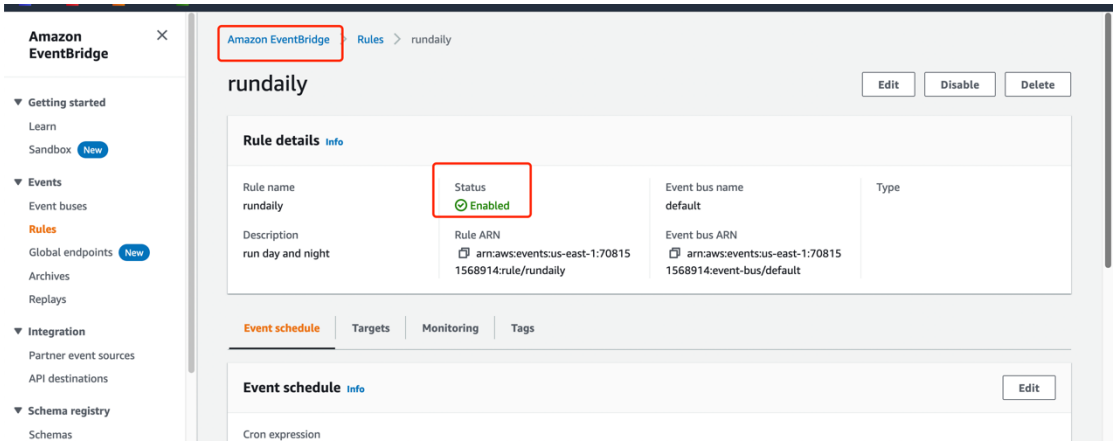
Scroll down the page, we see a code editor with a sample Hello World function. Here we can insert our Python function. Just copy the function from Python and past inside here. But remember to call the date frame inside the handler. After insert the function then deploy it and test it.

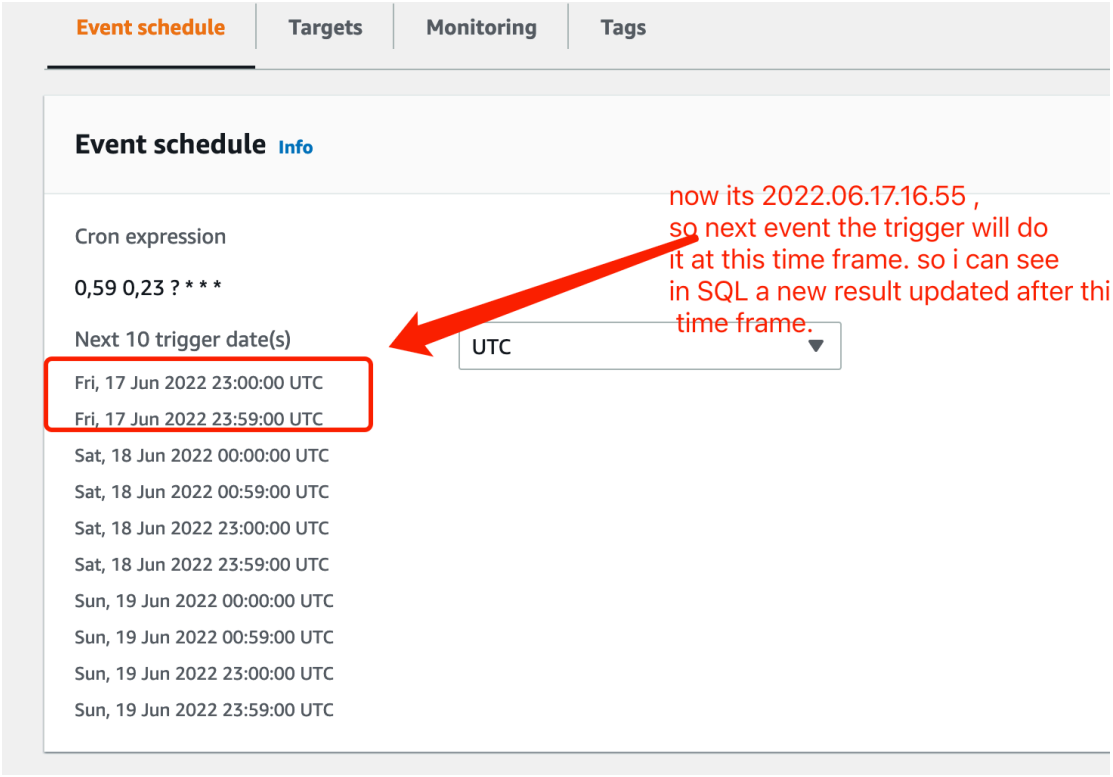


If the test result shows 200. It means the codes are working well.

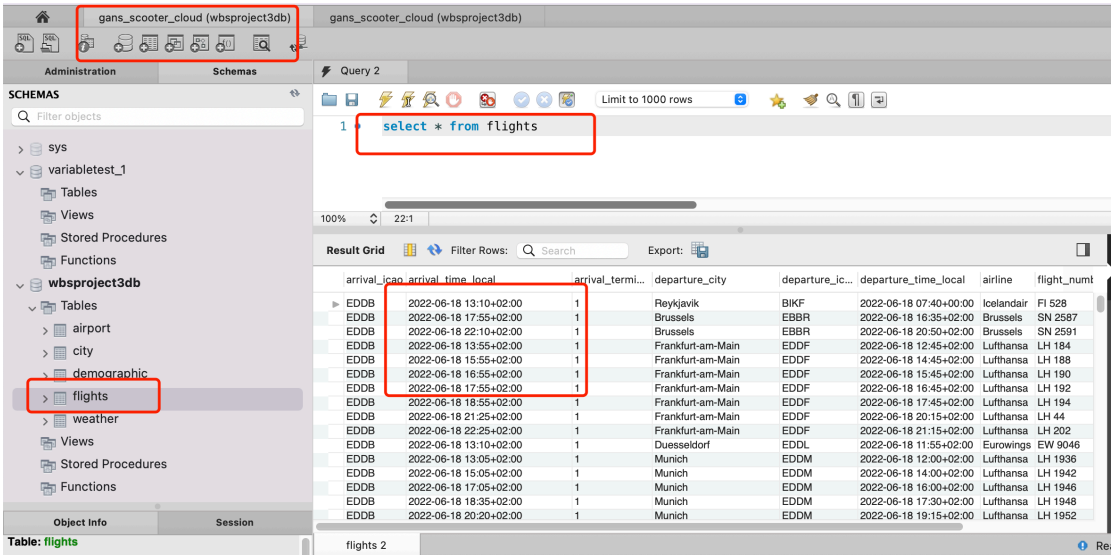


We are almost done. In our business case, we want to get flights and weather data daily. So instead of manually pulling updated information. We want to use a trigger to push data to our RDS in the time frame as we wanted. In our project we just need to get daily flights and daily weather. So we will set up trigger to updated one or two times in a day. As you amy see in the below picture. The trigger will let you know the updating schedule.

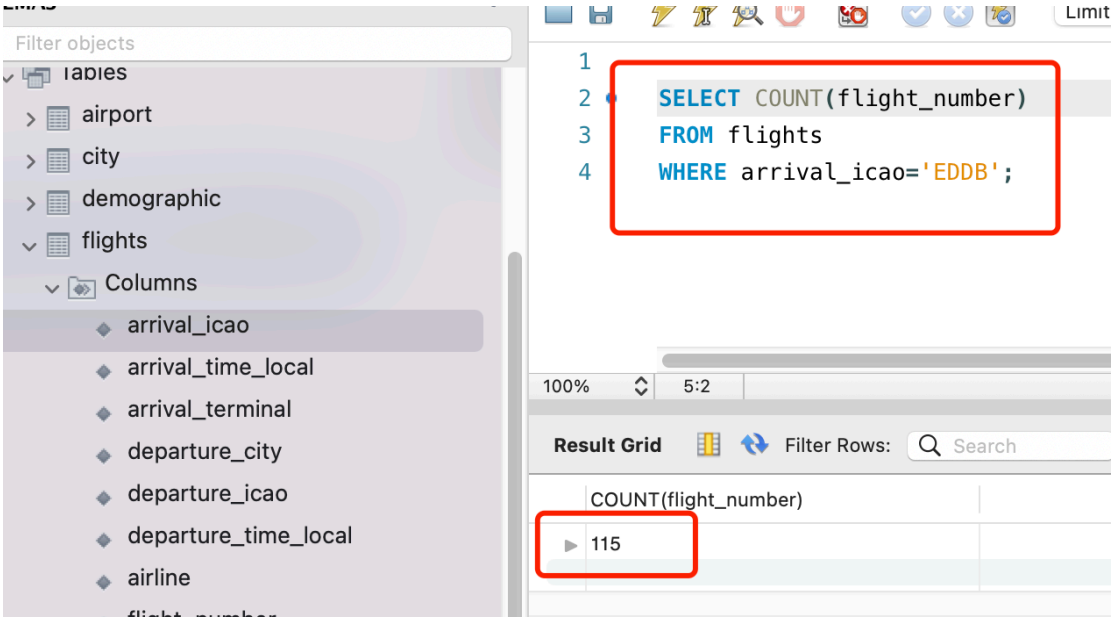




Now we can check up on the SQL workbench see if data is correctly updated. As you can see the date was pushed from this morning regarding the arriving flights for next day.



At this point, we can do the analysis now. We can already check how many flights are arriving today in the airport ICAO EDDB is. And start the rest data exploring and finding some useful data to report to your counterparts or marketing department.



Conclusion

I learned a lot from the team during this project. In particular, I learned to understand the usage of functions and the terminology needed for some of the data normalization processes. Building up data pipeline is mainly to understand the whole structure, steps and tools. For me the challenging part is scraping and extracting the data from online or APIs as well as normalizing the data and finding its usability for later analysis. As for our data analysis, data is valuable to us.
I hope this blog will help you to build your data engineering pipeline easily too.

good luck! :)

Any feedback and suggestions are always welcomed. Please reach me under twitter: @mspengde
Thank you in advance!