Objective

Operationalize a data stack. The components would have the following utilities –

1. ELT (data loader)
2. Orchestrator for transforms
3. Data Store
4. Dashboards/ Visualizer

Here is a simplistic diagram for project data flow, setting up each of them will require multiple AWS services for instantiation

Data Source

Salesforce Opportunity daily slice will be converted to sales coverage trend metrics

Execution Plan

These are the details of execution that will be carried out for this project:

Local/ Sandbox/ Test

1. Salesforce source access
    a. Get access to Salesforce Sandbox Environment, setup API access credentials
2. Install ELT Service Airbyte (local mac)
3. Configure Salesforce connection in Airbyte
4. Create Dockerized container environment for python code runs, and code deployment
5. Docker push to ECR

Production/ AWS Cloud

1. Salesforce Prod access
    a. Get access to Salesforce Sandbox Environment, setup API access credentials
2. Install ELT Service Airbyte on AWS EC2
3. Instantiate a Redshift cluster on AWS
4. Instantiate Airflow instance (managed Airflow) - AWS MWAA
5. Docker push to ECR for Airflow jobs/ Redshift transforms
6. Use Tableau to read/ consume Redshift transforms on Salesforce opportunity trend metrics

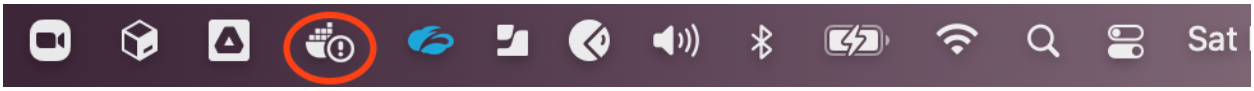
Pre-requisites

1. Local Machine

   All workings detailed are performed on a Macbook M1 machine. If you are working on Linux machines, or Macbook Intel chip machines, the commands and settings can be replicated verbatim. For Windows machines, the local system workings/ commands may not work as mentioned here.

2. Docker
   - Context -
     - Docker Desktop is an easy-to-install application for your Mac, Linux, or Windows environment that enables you to build and share containerized applications and microservices. It provides a simple interface that enables you to manage your containers, applications, and images directly from your machine without having to use the CLI to perform core actions.
     - We will be running Python code in the Docker containers. The python code will be used for data transforms on Redshift. Using Docker containers allows us to replicate airflow host settings through exposing environment variables.
   - Signup/ Access –
     - start at https://hub.docker.com/
     - signup with email
     - opt for single-user free tier $0 *(to begin with)*
   - Download & Install –
     - https://www.docker.com/products/docker-desktop/
   - Docker Desktop should be running when trying to test/ execute commands locally.

     TEST/ Check#1: On Mac Menu, you should be able to see the Docker icon, IF the docker desktop is active/ running

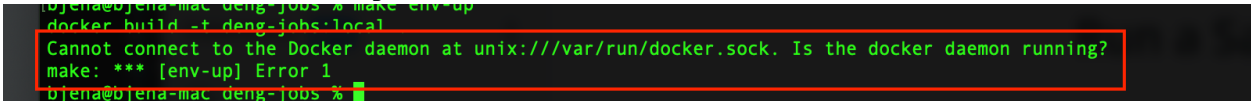


     TEST/ Check#2: On Mac Terminal : run to see all running containers

     `docker ps --all`

     if this command runs fine (i.e. no error exceptions, even with no running containers) – it indicates docker is ON

This is what we see in terminal for Docker dependent commands, if Docker isn't running



- When pushed to production environment, the docker images are pushed into AWS – ECR (Elastic Container Repository) for workflow read and pipeline execution

2. Python & Poetry
    - Install Python, using a consistent package manager (like brew, conda, pip , etc)
    - Although latest python versions are advised, we will use Python 3.9 / 3.10 since I find these versions most stable and lesser tweaks to deal with package dependencies
        - With higher python versions managing python dependencies becomes a routine and this is where using consistent package manager, or potery like dependency managers becomes key
    - Consistent use of package manager is key, and so is maintaining package version dependencies
        - Recommend to use poetry for version and package dependency management
        - Poetry is a tool for **dependency management and packaging** in Python. It allows you to declare the libraries your project depends on and it will manage (install/update) them for you. Poetry offers a lockfile to ensure repeatable installs, and can build your project for distribution.
        - Create the project using poetry i.e., start from here https://python-poetry.org/docs/basic-usage/
            - Initialize poetry – it results in a **`pyproject.toml`** file
            - To add any package say pandas use `$ poetry add pandas`
            - this poetry add commands reflects in entry in `pyproject.toml` and poetry.lck file is the one which controls the registry versions of the installed packages for the project

3. Email & Distribution Alias
    - We will need a mailbox for instantiating Airflow. Ideally we need 2 email ids. The first one is a full inbox to be used by airflow for its smtp configuration. This mailbox mimics any human user, and has the same features as a regular mailbox. (Note: singup for a gmail account, and ensure to enable IMAP settings). The second is a creation of a group email alias, the group email has the formerly created mailbox as its member. The idea is to add as many users as members of the group, and the airflow email account is a member of the distribution.

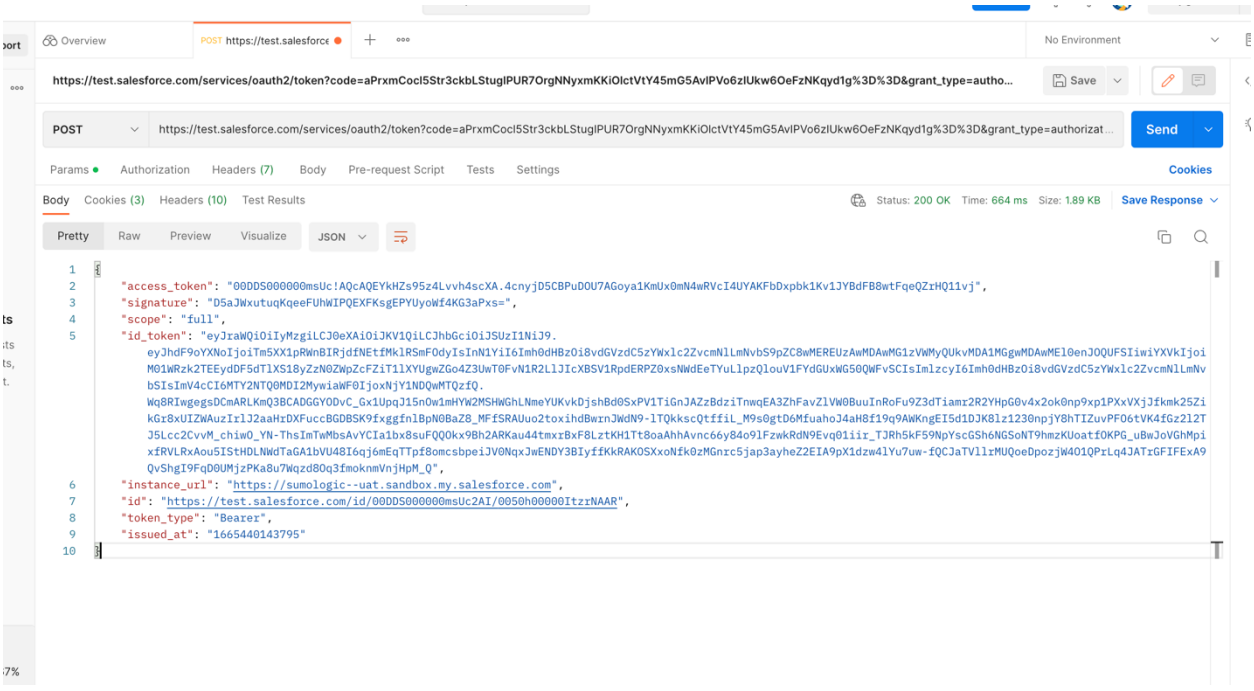
Implementation Log

1. Salesforce Access (Sandbox environment)

We get Salesforce credentials as username and password. However for programmatic ingestion, we will need api access methods, which follows OAuth2 authentication. Here are the steps -

a. Get access to Salesforce (username & password)

b. Salesforce Sandbox environment domain eg: https://test.salesforce.com
c. You will also need the CLIENT_ID for the Salesforce credentials (which looks something like `3MVG96LA2t1yu9WK_2g3tejYpVbOYWaH0dj8gu0OAo7Tv.RHqpRWTitDOgLl5gDy6..ZsBZ.WQXte1XntAaoH`
d. Can create a connected app in Salesforce for all sfdc interactions
    i. details outlined here
        1. https://docs.datawatch.com/swarm/desktop/Generating_a_Client_ID_and_ClientSecret_Key_for_Salesforce_Connections.htm
    ii. Also refer these:
        1. https://community.boomi.com/s/article/howtoconnecttosalesforcerestapiwithoauth20
        2. https://medium.com/@bpmmendis94/obtain-access-refresh-tokens-from-salesforce-rest-api-a324fe4ccd9b
e. So summarily: inputs for SFDC access are username, password, and CLIENT_ID (from sfdc- admin or connected app setup)
    i. Using these credentials we will need CLIENT_ID, CLIENT_SECRET and REFRESH_TOKEN for use in subsequent steps
    ii. I had to use postman to post the api methods and code, and finally landed on the refresh token screen as this (P.S. you can use any rest api handler, I used postman by signup with a free account on https://www.postman.com/)



So, the credentials I had generated were for CLIENT_ID, SECRET_KEY and REFRESH_TOKEN.

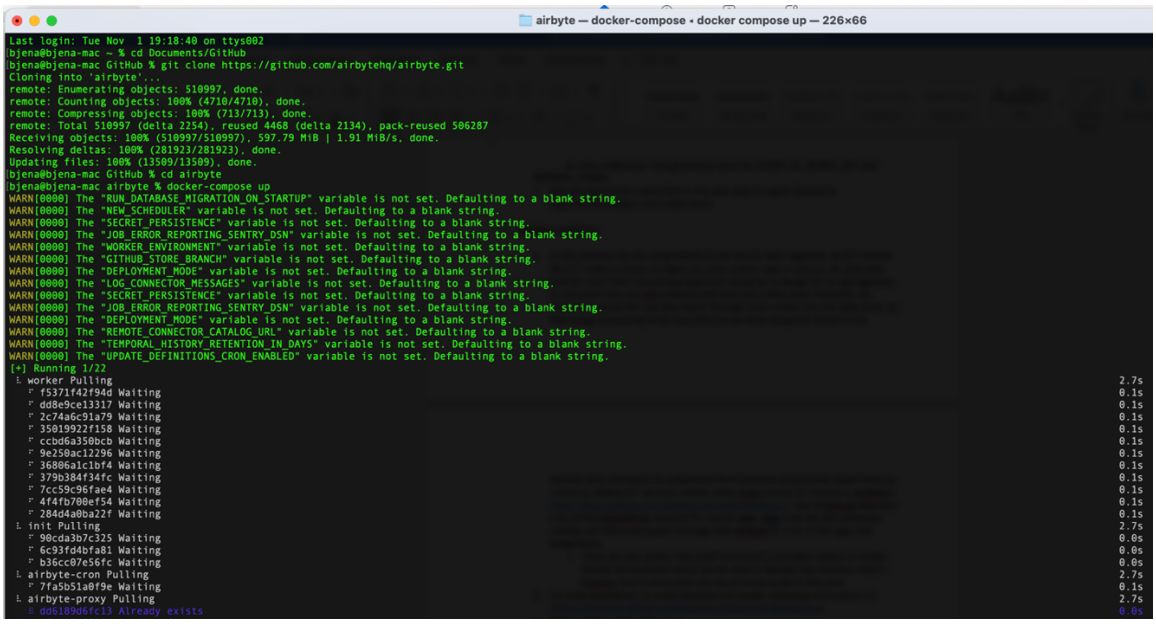
f. We will need these credentials in the next step to ingest Salesforce Opportunities object into a data store

2. Airbyte Installation

The installation is for local macbook as well as EC2 machines. Security and networking aspects associated with EC2 machines is beyond the scope of this project, but a necessary aspect for the system to be operational. Typically bastions are configured to secure access to EC2, along with dedicated keys along with user name, password, IP authentication methods.

a. An ELT service like ELT makes it faster to ingest any such custom apps / sources. An alternate (and bit more time consuming) approach would be to design for an api ingestion to consume from any api endpoint and load into a data store. However, my intent is to pump the raw data faster through some means into the data store, so that using/ processing of the raw data can be done using SQL (which is my comfort skill, and easier to understand from business stakeholder aspect too), so I chose to deploy ELT services. Similar other oopn-source ELT service is meltano (https://docs.meltano.com/getting-started/installation). Use of Airbyte abstracts a lot of the heavylifting required for custom apis. Also if we see the connector catalog, we find pretty good coverage with Airbyte for a lot of the apps and integrations.
    i. There are also quite a few licensed ELT providers where in environment setup can be done in few minutes. Stitch[1], Fivetran [2]are 2 names that are top of mind recalls in this area.
b. For local installation, its nicely detailed and simple, following instructions on https://airbytehq.github.io/deploying-airbyte/local-deployment
c. Here are the commands to use:

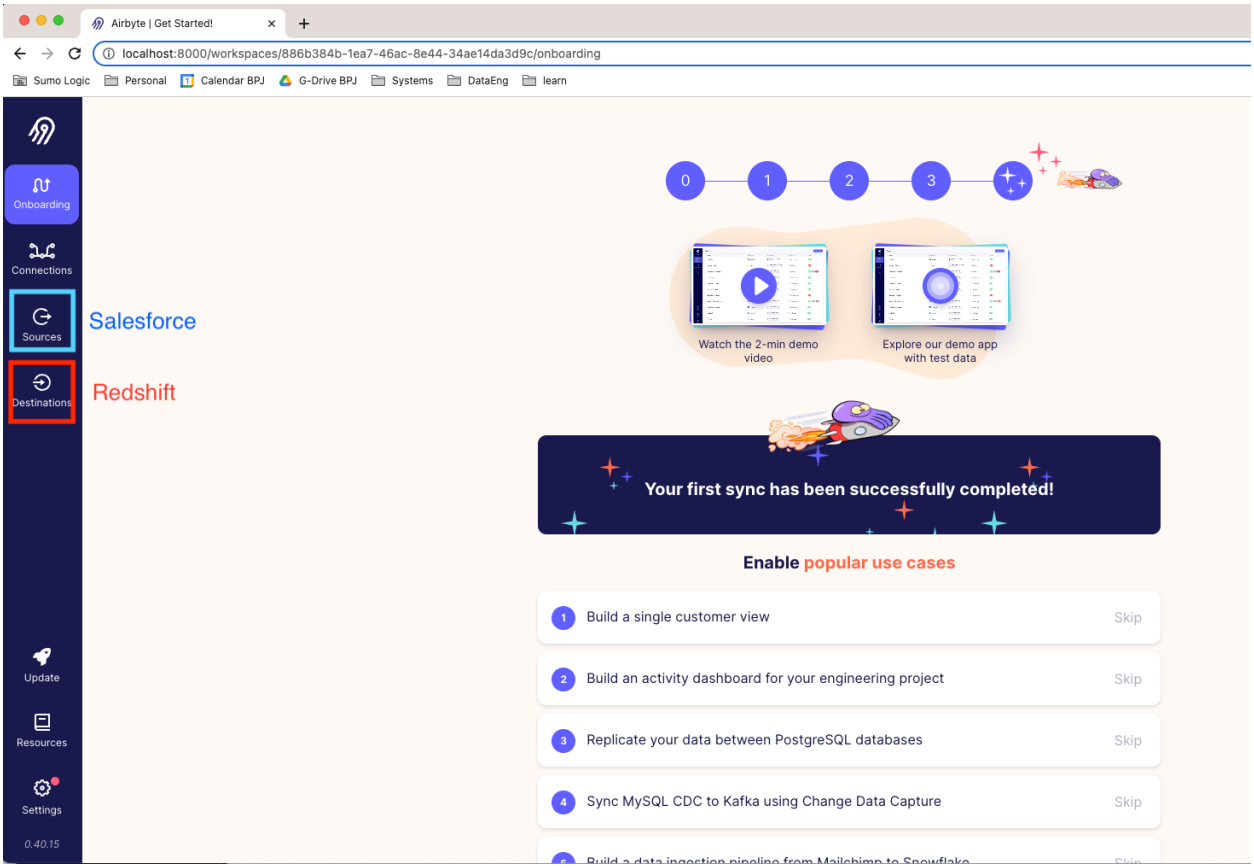
```
git clone https://github.com/airbytehq/airbyte.git
cd airbyte
docker-compose up
```



[1] https://www.stitchdata.com/
[2] https://www.fivetran.com/

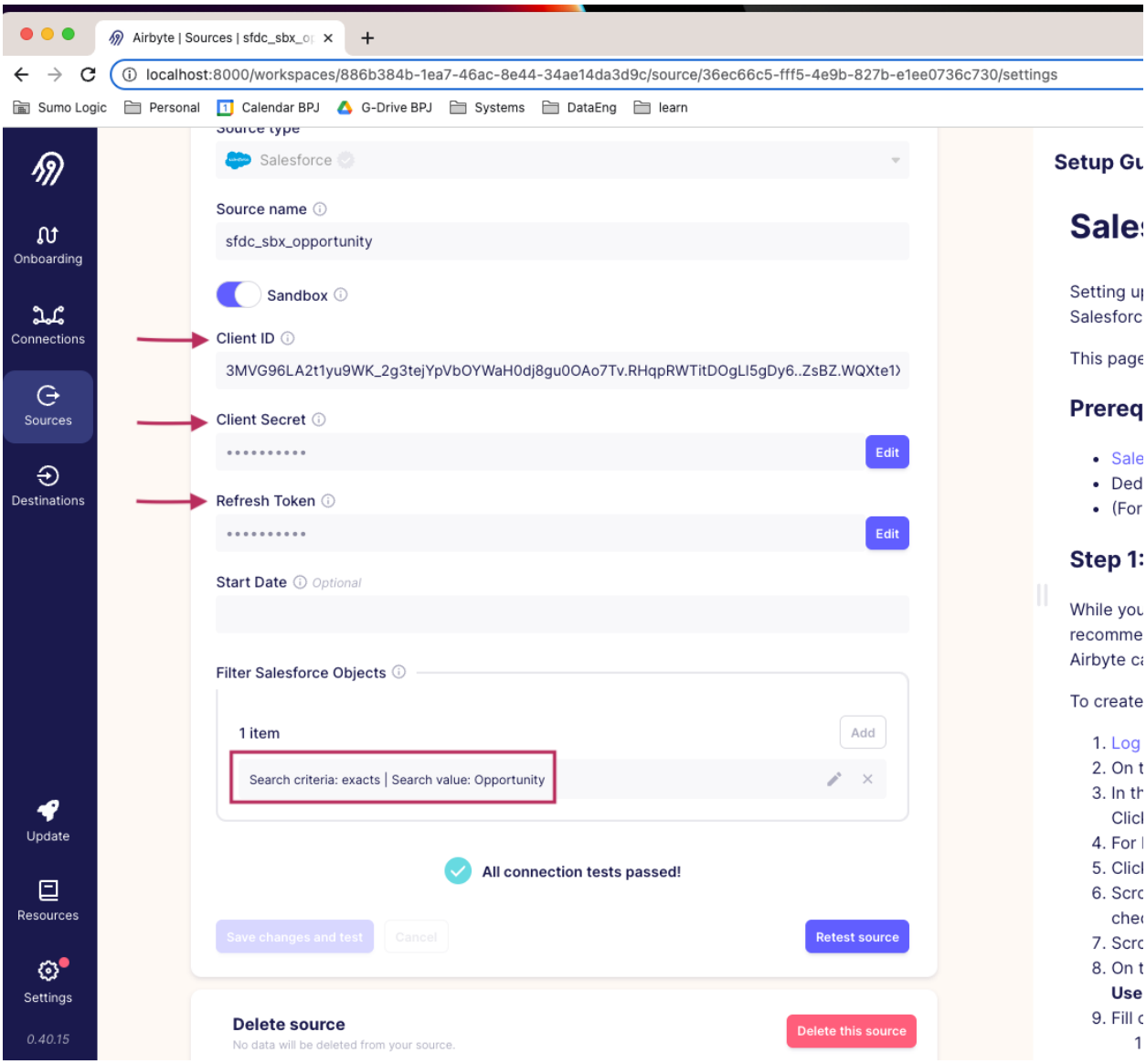
d. For EC2 installation i.e. to productionalize Airbyte, the steps are listed https://airbytehq.github.io/deploying-airbyte/on-aws-ec2 . Beyond bringing up the application, providing SSH keys for secure access, securing the EC2 host behind a bastion, having RDS (Postgres/MySQL) datastore as the application backend, and enabling cloudwatch logs for application monitoring are some of the additional steps that need to be done in the AWS environment.

e. Airbyte can be accessed at http://localhost:8000
In a nutshell, The links marked as “Sources” and “Destinations” are linked/ made operational through “Connection”



3. Add Salesforce Source to Airbyte

a. For trend analysis, I will be extracting **`Opportunity`** object alone. The credentials generated from #1 are provided on a “New Source”. The credentials are provided here…



4. Redshift Instance

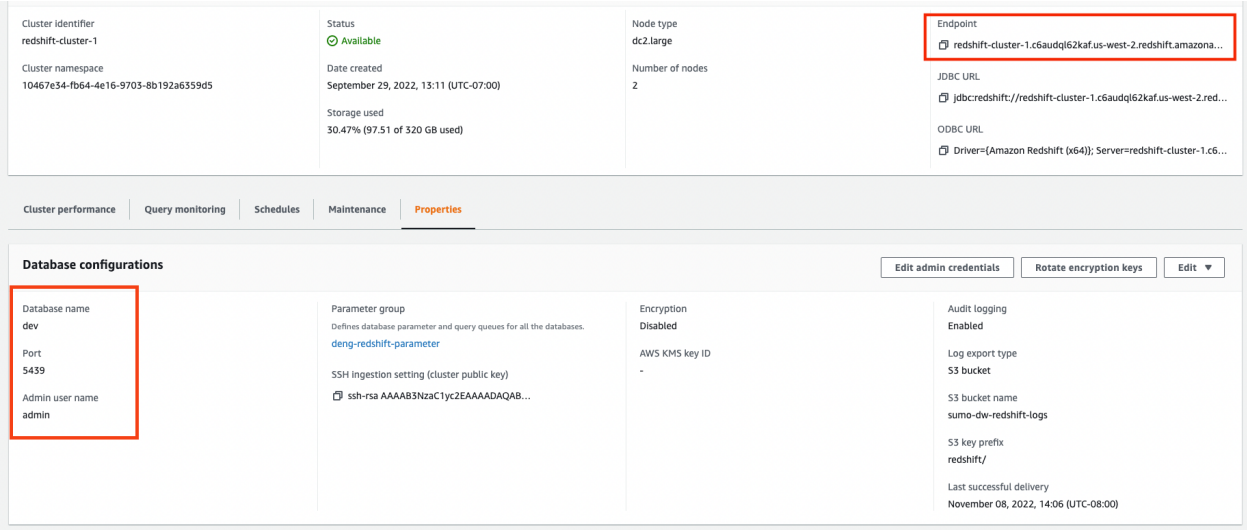
   a. The core of this data stack is the analytic data store. It can be called Data Mart or Data Warehouse depending on the scope of use cases it supports. For this task, it will host and process only Opportunity object from Salesforce (so can be referred to as a Marketing Data Mart).
   b. Access to AWS, IAM role provisioning, firewall rules, MFA enforcement and having rights on the account are necessary aspects which I wont be going into as much detail. Instead I will be focusing on the Redshift instantiation following the instructions https://docs.aws.amazon.com/redshift/latest/gsg/bring-own-data.html
   c. While the instance is being instantiated and initialized, need to have a bit futuristic vision of data access w.r.t. roles and permissions on the database object organizing.

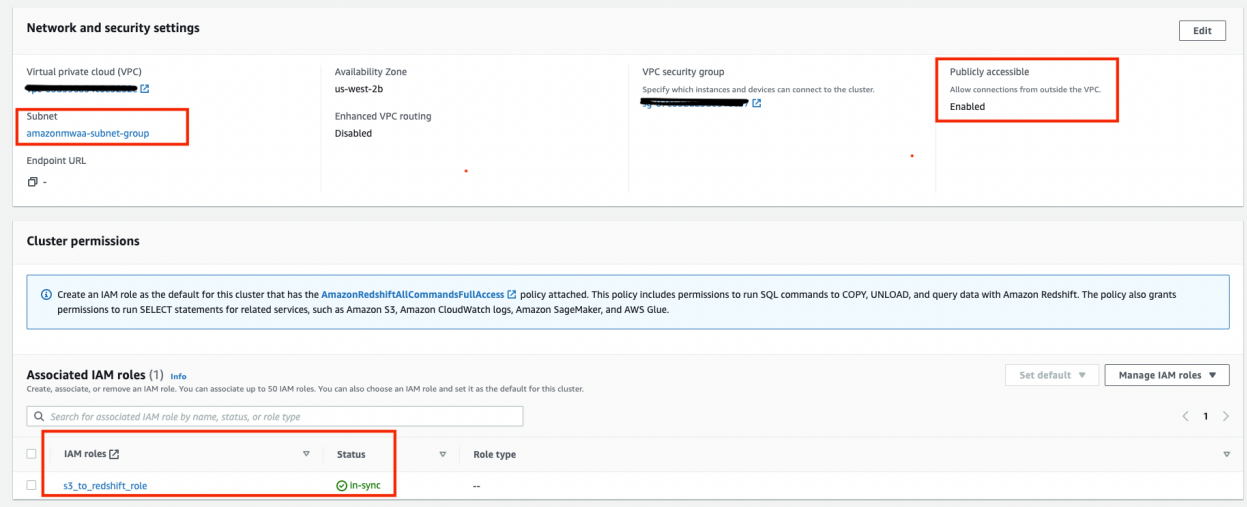
d. So the redshift instance will be created resulting in super-admin user which will be used to create admin user for all subsequent role grants and user administration.
e. For redshift user creation the command would be as:

```
create user xyz in group read_only_users password 'abc';
```

f. For creating a Redshift cluster,
    i. signin to AWS,
    ii. type Redshift in search bar,
    iii. create cluster
    iv. provide cluster configuration details – identifier, node type, nodes
        1. this is result of a planning/sizing exercise
        2. chose dc2.large and 2 nodes for this assignment
    v. provide these values:
        1. database name: `db_prod`
        2. database port: `5439`
        3. admin user: `admin`
        4. admin password: `awsuser`

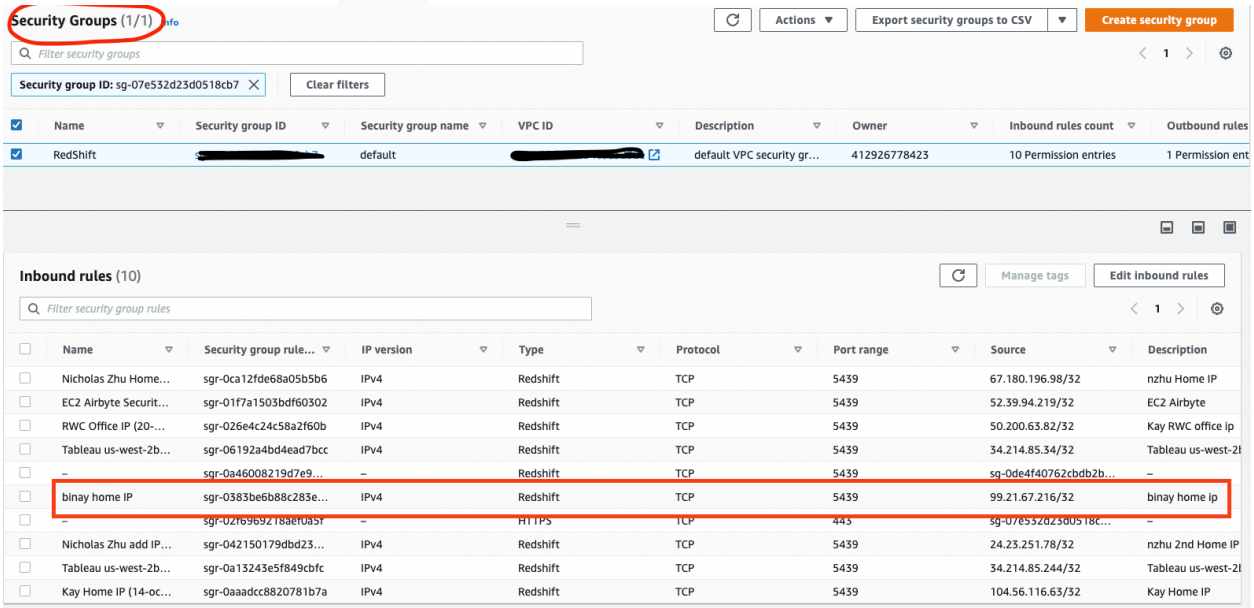
This is how ready-to-use Redshift instance looks like





vi. I have ignored detailed around IAM role creation and managing access. One key aspect is configuring rules with security groups around inbound SQL clients. So the idea is, Redshift instance is not a public data store. We need to control who can access what.

1. Redshift roles and groups controls the user's permissions and access levels within Redshift
2. Modify security group rules to allow inbound traffic only from Redshift recognized users. Here's how inbound SQL clients can be limited only to the IP's bound in the security group of the Redshift cluster

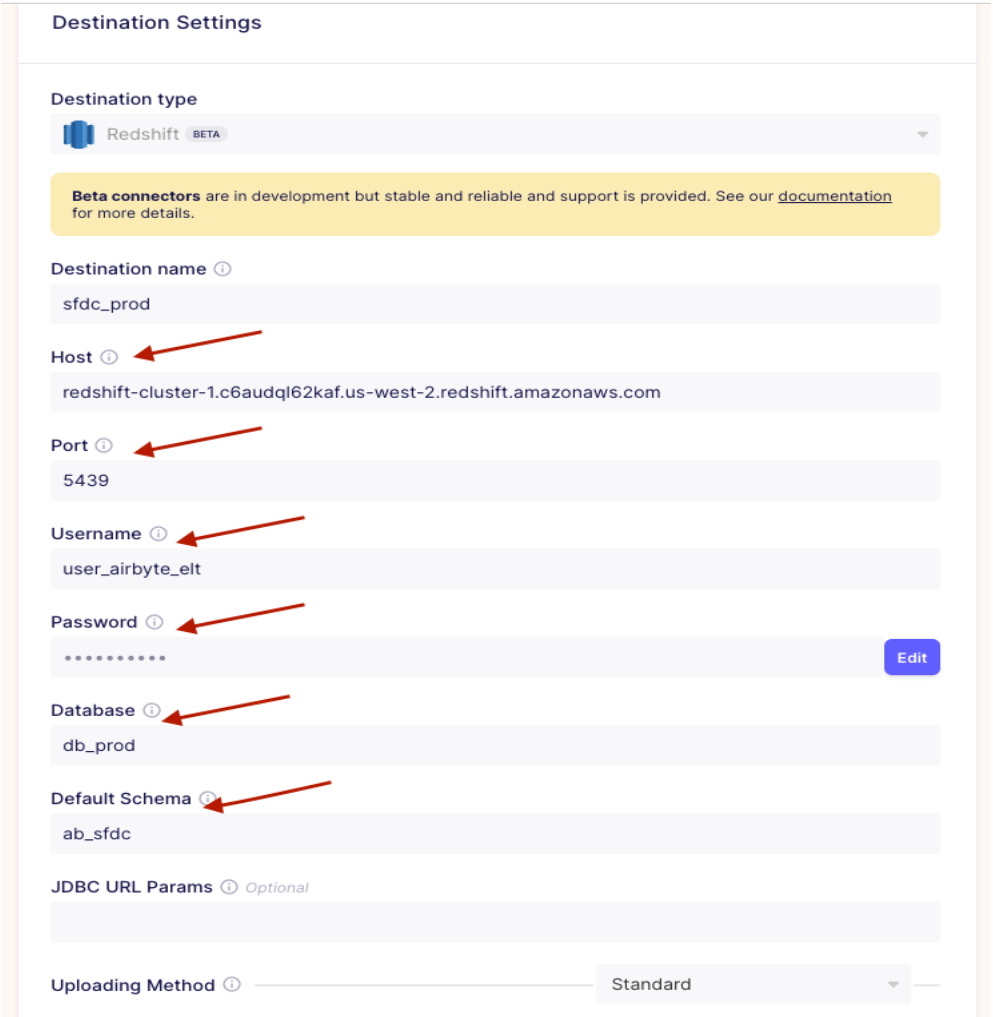


vii. Since it is an analytic data store, we need monitors with Redshift queries , disk usage, CPU, etc. CloudWatch allows for these configurations. However, I will skip the monitoring piece on Redshift to pivot to project scope.

g. To access the Redshift cluster,
    i. can use clients like DBeaver, Aginity, SQL Workbench, Datagrip, etc with the following credentials:
        1. Hostname
        2. Port
        3. Username
        4. Password
        5. Database
    ii. For each new user, the inbound rules need to be updated along with access provision
h. Create a Redshift user credentials (service user) to be used in Airbyte data load . So we want to load the Salesforce object to Redshift.
    i. Create a separate schema for Airbyte access with the created user.
    ii. I have named the schema **`ab_sfdc`** , destination of sfdc raw data in Redshift

5. Airbyte Redshift Destination

The credentials created in #4 above are used to setup the Airbyte Redshift destination.

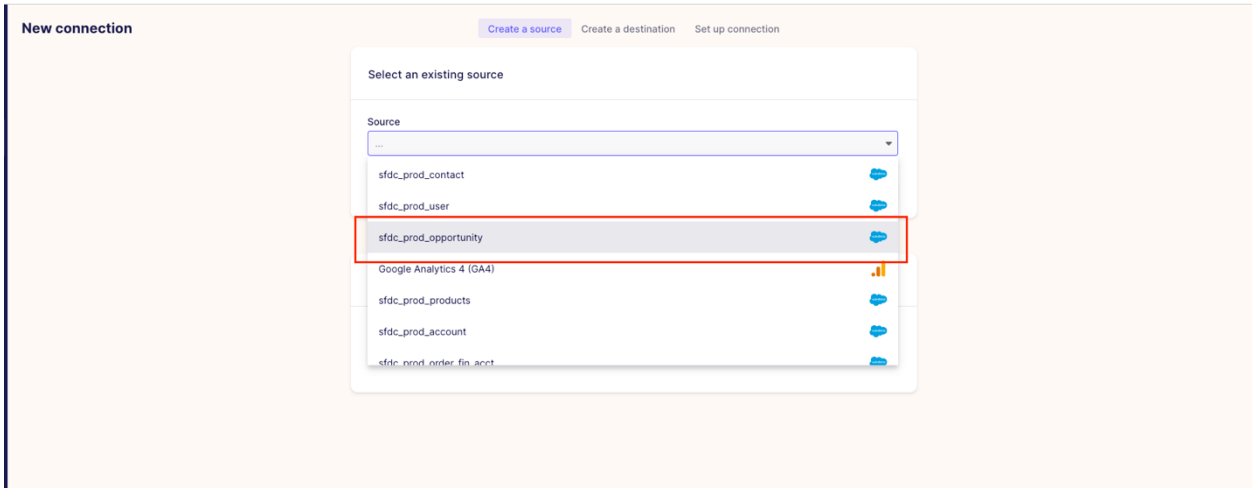


NOTE: Its important to have users isolated from each other. So in Redshift we create a separate schema for each source, and a dedicated user for each schema. So in this example `user_airbyte_elt` is for sfdc source, all object of which get loaded into `ab_sfdc` schema in Redshift. If this detail isn't practiced, there's a risk of one source malfunction impacting others which isn't ideal.

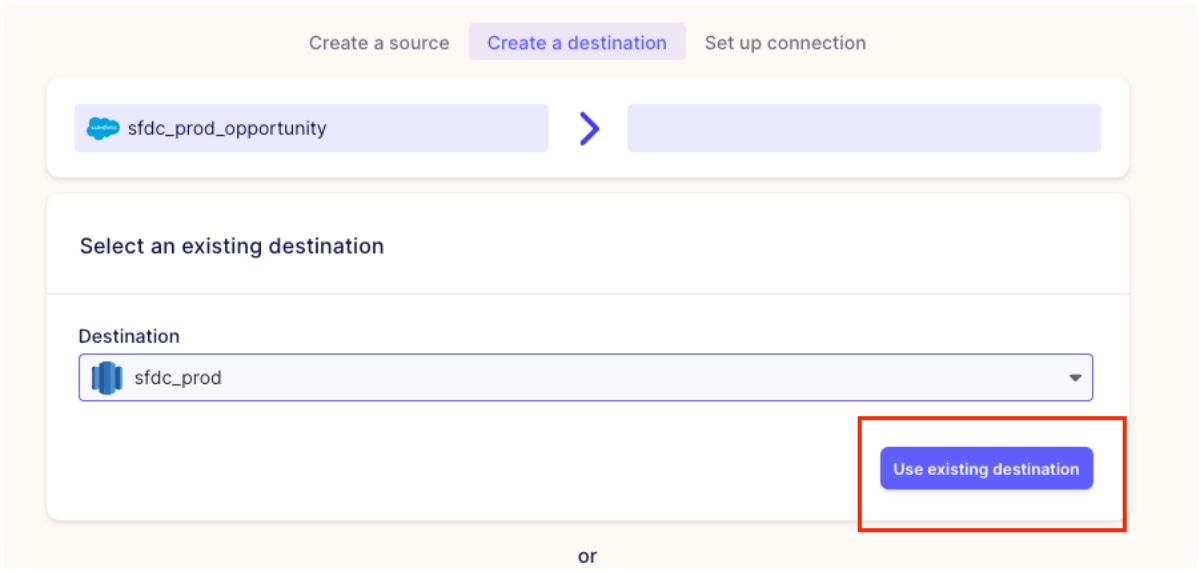
6. Airbyte connection: Salesforce - Redshift

So, in #3 we had created a source in Airbyte (a Salesforce source object which parses for "Opportunity" object), and in #5 we added a destination in Redshift. The "Connection" in Airbyte is what connects either of these.

Use the source created in #3,

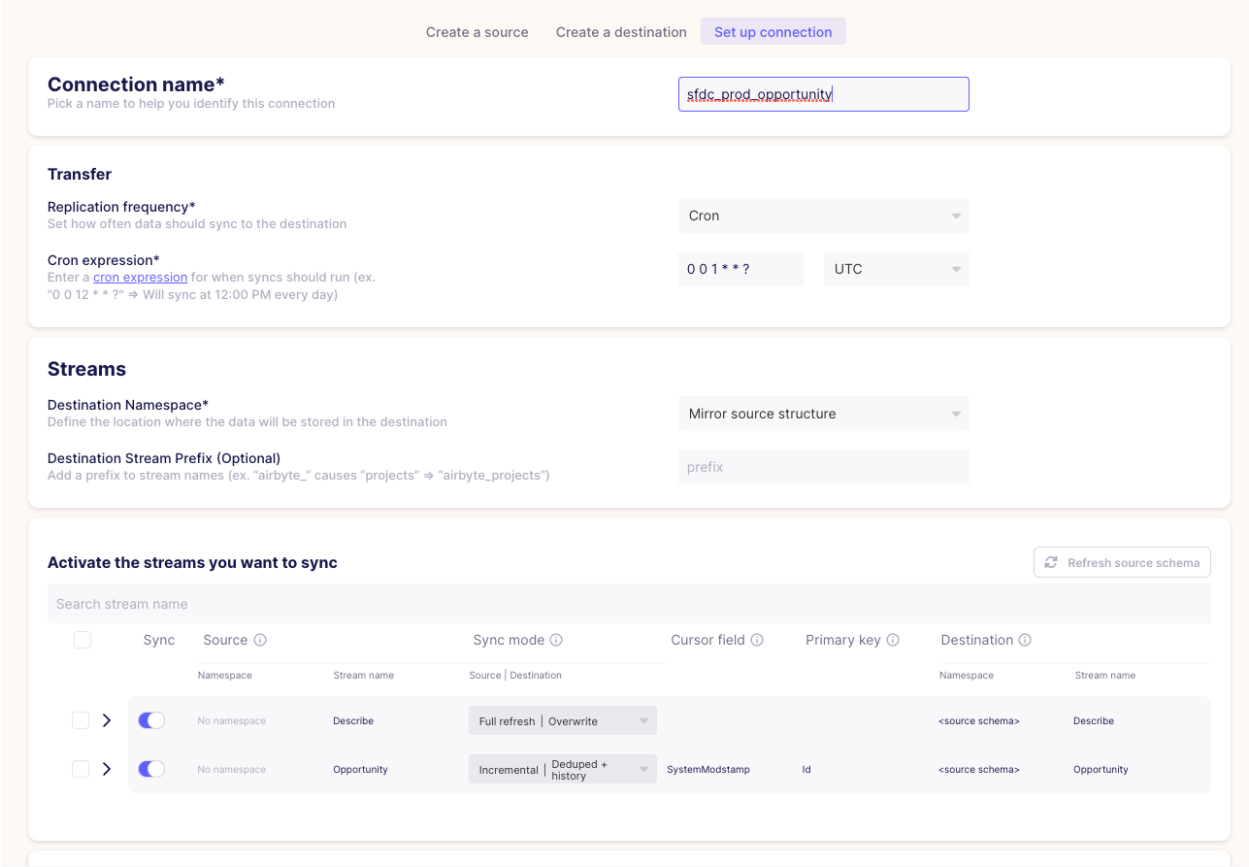


Use the Destination in #5,



Finally, name of the connection, and schedule it. I have done a *cron* schedule at UTC 1 daily.

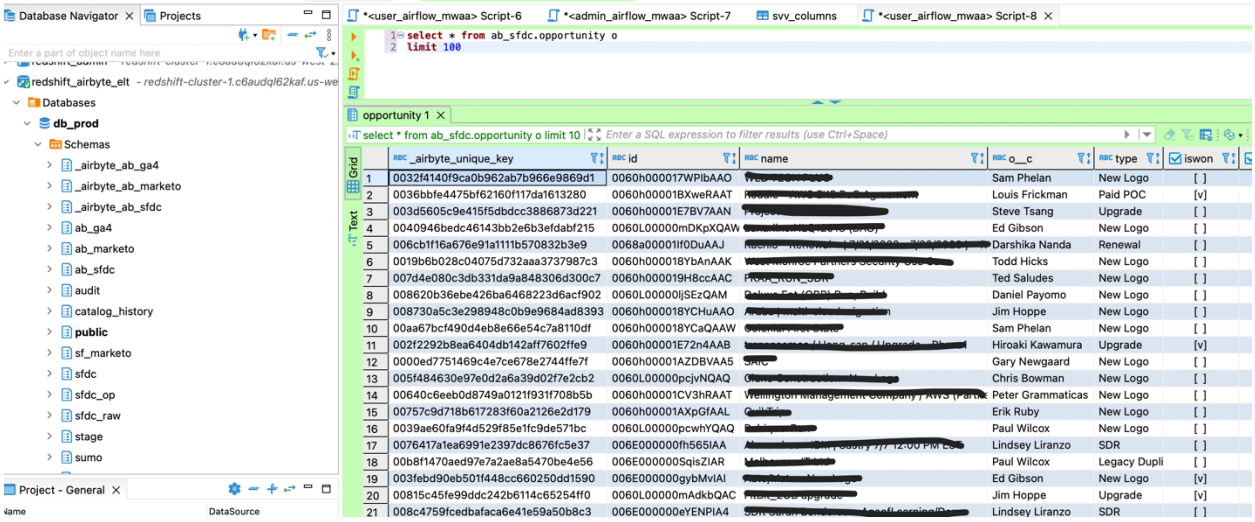
On this screen we can use if the loads needs be overwrite, incremental loads, etc. Typically I chose overwrites to begin with (so that full history is available), and then subsequent schedules are incremental dedupe and append loads.



Once connection is setup, sync the connection (it will eventually run on the configured schedule, this is just force running once manually, to see everything is fine)

7. Access the created/updated table in Redshift

Assuming a scheduled run of the connection is fine, we would expect the `opportunity` table in `ab_sfdc` schema in Redshift. I use DBeaver client to connect to Redshift.



8. Redshift transforms

So we have the raw/ source sfdc opportunity object loaded into `ab_sfdc.opportunity`. Opportunity in real world is a business expansion entity, it can relate to a team or a department in a company.
Requirement:

- trend of business opportunities dollar value at various slices as type, geo, industry segment, etc.
- weekly trend generated on 1st, 8th, 15, 22nd and 29th day of each month (and not Mondays/ or Sundays)
- we have a plan / forecast/ estimate for the dollar values, and would like to have a comparative view of the metrics

To implement these requirements,

- we keep a daily snapshot of the `opportunity` table , (let's say its pipeline)
- use a `schedule` table which has list of dates i.e., 1,8,15,22 and 29
- use a `plan_closed_won` table for forecast/ plan of leads.

The query to produce weekly trend of opportunity dollar values is https://github.com/bpjena/ms-dsc/blob/master/dsc680_project/redshift/sfdc_opportunity_trend.sql .

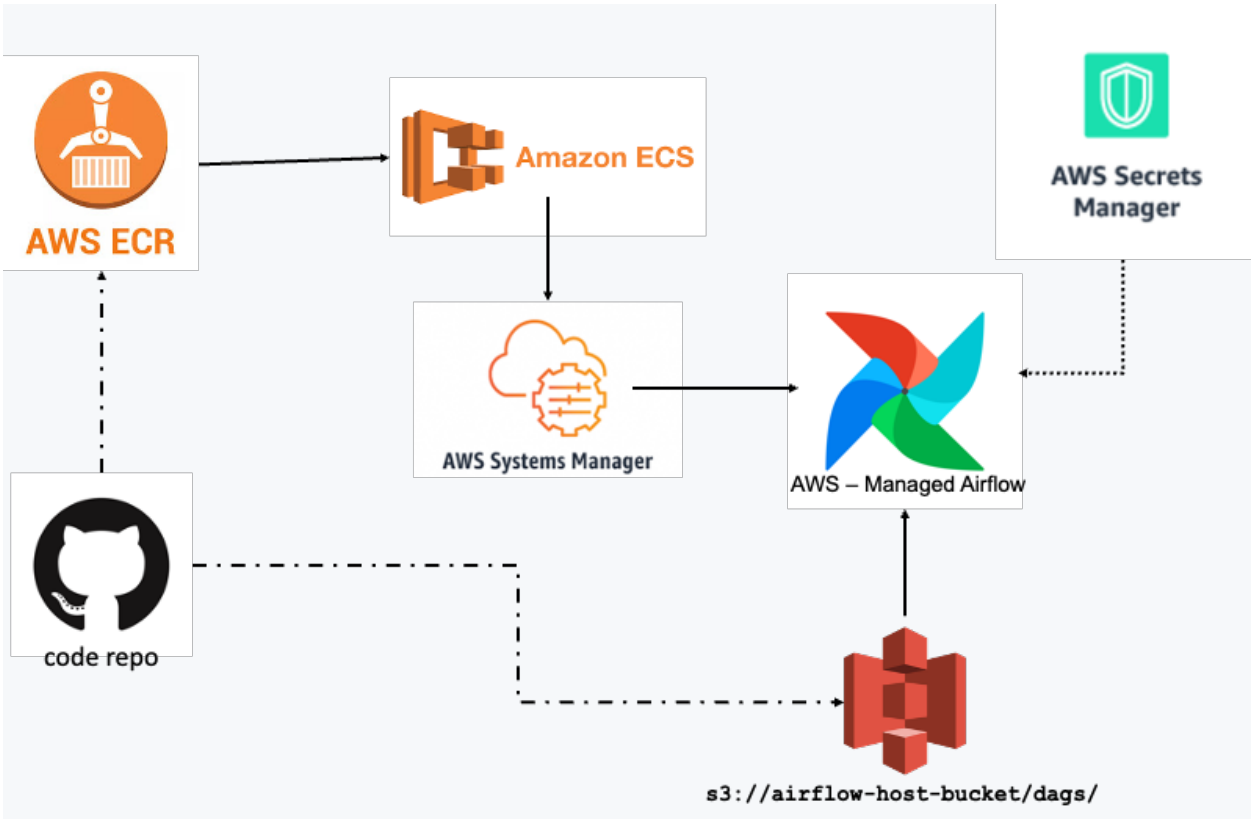
With the current week being represented as ref_week = 0, final dataset produces a futuristic 52-week view and similarly a historical 52-week view of the opportunity dollar values. Simply executing the query produces a dataset using DBeaver. However, to make this a data pipeline, will need to automate this dataset creation.

9. Airflow jobs

The intention is to be able to automate the redshift transformation activity through a scheduler. While I could have chosen a simple cron scheduler on any EC2 machine as a working solution, I choose to perform a mature deployment of open-source orchestration platform like airflow to not only be able to perform this transform automation. But also use airflow features for any data related automation like ingestion jobs, checking data quality of datasets, etc

I will be using python redshift connector as the core library to script a python job which will be scheduled using airflow. Code and related instructions to run this local machine is detailed https://github.com/bpjena/ms-dsc/tree/master/dsc680_project . Core idea is to run code in Docker containers, and ensure the replication of container when code runs by Airflow. This method has inherent scaling benefits, since scheduling and code execution environments are decoupled and each can scale as needed.

While there are many custom Airflow installations in use, due to the open-source nature of this utility, I've chosen Managed Airflow from AWS as my airflow version of use, since a lot of airflow components and hence its maintenance/ infra is abstracted through this managed service from AWS. The code detailed here is for executing the python jobs in local docker containers. When airflow runs these, the local docker container is replicated by ECS Operator usage from Airflow (elastic container service). Here is how the deployment of the code looks like from Airflow job flow aspect.

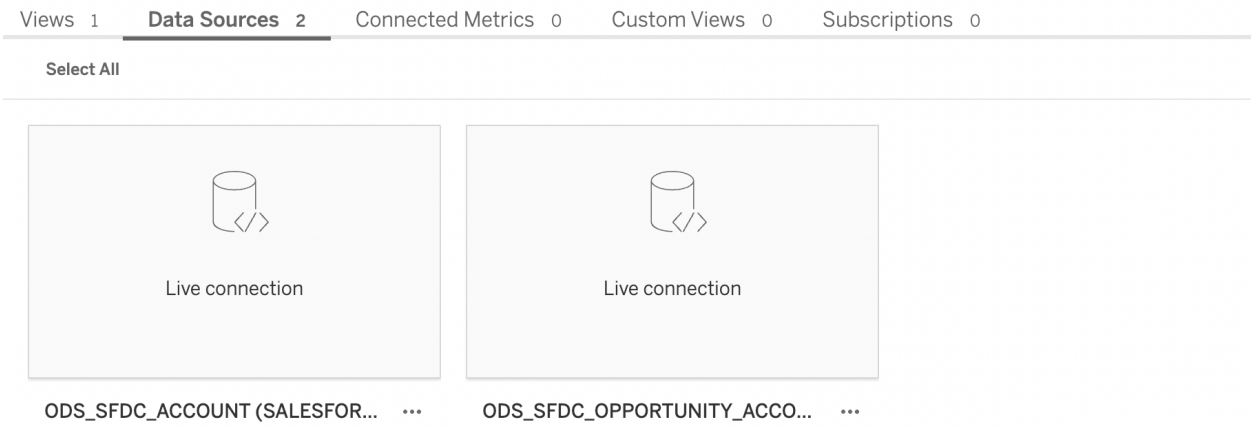


In order to initialize the Airflow instance, refer the steps https://docs.aws.amazon.com/mwaa/latest/userguide/get-started.html

Code push/ deployment as an activity is about pushing the code into AWS ECR (elastic container registry) and copying the dag file into S3 bucket (`dags` folder in S3 maps to the Airflow jobs or DAGs).

10. Tableau Setup

Tableau Redshift connection credentials has read privilege on the transformed dataset created from the job. Create separate credentials to be used for Tableau connection to Redshift.



Here is a screenshot of the graphical plots for the transformed dataset when rendered through Tableau.

11. Local python code run / setup

Please refer the instructions in the https://github.com/bpjena/ms-dsc/tree/master/dsc680_project/README.md for local setup. To run the python code locally will need `.env` file within which has the Redshift access credentials or any security params that we want to pass as environment variables. AWS credentials can also be stored here in the `.env` file for local code runs and tests. Commands to run the job in local machine is detailed in .local file. So the job run in local docker container has the following execution logs.