Python Database Utilities Documentation
Kenneth Burchfiel

Table of Contents

Introduction

This document provides an overview of how I sourced data for and constructed a SQLite database; how I set up database accounts for online database providers; and how I connected to, then exported my SQLite database tables to those online databases.

This project was completed as part of an independent study project at Columbia Business School, where I am currently a 2-year MBA sttudent. Professor Daniel Guetta, my advisor, provided helpful instructions and guidance. For instance, he came up with the idea to connect to and test out the 7 different databases covered in this documentation.

1. SQLite Database Setup

One of the main goals for this project was to upload various data types (e.g. .csv files, .json data, image files, and photo files) to multiple online database hosts, such as Amazon Web Services, Google Cloud Platform, and Azure. Although I could have set up those databases manually using the command prompt, I predicted that creating a SQLite database on my computer, then exporting that database's tables to the online hosts, would save time overall. Therefore, before I began uploading data to my online databases, I first created a local copy of my database.

1.1. Data sources for my SQLite database

1.1.1. Flights table

The source for my flights table was the T-100 Segment (All Carriers) table from the Bureau of Transportation Statistics, stored in .csv form. I chose this table because it provides information on both airline routes and the aircraft flown on that route, which would make for interesting analyses. As reported by the BTS:

> "This table combines domestic and international T-100 segment data reported by U.S. and foreign air carriers, and contains non-stop segment data by aircraft type and service class for transported passengers, freight and mail, available capacity, scheduled departures, departures performed, aircraft hours, and load factor. For a

uniform end date for the combined databases, the last 3 months U.S. carrier domestic data released in T-100 Domestic Segment (U.S. Carriers Only) are not included. Flights with both origin and destination in a foreign country are not included."

The table was available via this link, and fields to download could be selected via this link.

I then downloaded this file and stored it on my computer as 21503323_T_T100_SEGMENT_ALL_CARRIER.csv, the standard file name. I chose to analyze data from 2018 (since I wanted to choose the most recent year that did not have COVID-related impacts).

This file contains 'AircraftType', 'AircraftConfig', and 'AircraftGroup' information. Tables for interpreting these codes were downloaded from the following BTS links:

AircraftType lookup table
AircraftConfig lookup table
AircraftGroup table:

In order to make it possible to plot these routes on a map, I then sourced airport coordinates from the Global Airport Database, (which is licensed under the MIT license.

1.1.2. Photos table

I sourced public-domain photos from NASA and Flickr. Most Flickr photos were created by the US Government, but I also found public-domain images from other Flickr users as well.

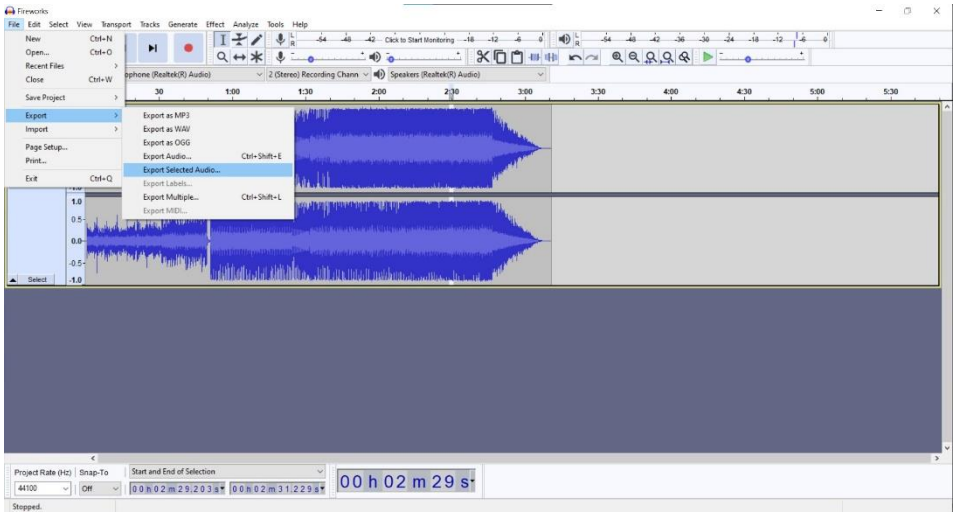
1.1.3. Steps table

I sourced my steps data by downloading all of my Fitbit data from my online Fitbit dashboard. This data contained monthly .json files showing my step counts. I then used a Python script that I created (fitbit_steps_reader.ipynb) to convert these .json files into one massive .json file containing over 500,000 lines. (fitbit_steps_reader.ipynb also shows how to convert this file to a DataFrame that contains both daily and rolling 7-day step counts, and how to translate this data into

line and bar charts. I plan to share this .ipynb file as a separate repository in the future.)

1.1.4. Music table

I found music available under a CC0 (public domain) license at https://freepd.com/. Since I like electronic music, I downloaded all of the tracks within the electronic music section. Of these tracks, 'Fireworks' by Alexander Nakarada was my favorite. I then went into Audacity and created 64 short samples of this song; information about these samples would constitute the entries in my music table.



Converting the 'Fireworks' track into individual samples

1.2. Media storage decisions

Initially, I planned to store my image and music files directly within my databases in BLOB format. However, I found an interesting discussion on Stack Overflow regarding media storage options. Although various opinions were presented within this discussion, I ultimately decided that keeping the files within a separate storage option (such as a Google Cloud Storage bucket) made more sense. Since I planned to create 7 different databases with the same content, a central media storage location would prevent my storage costs from increasing linearly with my database count.

Therefore, I created a storage bucket within a Google Cloud Platform project, then stored my music and images there. The upload speed was around 23.5-24 Mbps, but I'm not sure whether the bottleneck was my internet provider or Google itself.

Within Google Cloud Platform, I then added a new permissions category with 'allUsers' as the principal and 'Storage Legacy Object Reader' as the role. This allowed others to view the files on my database using public URLs.

Once these files were uploaded, I stored the URLs to my photos and music files within my photos and music databases, respectively, rather than the binary media data themselves. As Jose V. noted on Serverfault.com, Google Cloud Storage URLs can be obtained as follows:

> https://storage.googleapis.com/{bucket.name}/{blob.name}

1.3. Incorporating these tables into my SQLite database

I used Pandas and SQLAlchemy to import these four data sources into a SQLite database. (Although I could have used the command prompt to create the database instead, I preferred to set up the database entirely within Python in order to reduce the amount of manual work involved.) Converting tables to Pandas files, then exporting these tables to SQLite using the to_sql Pandas function made building the database much easier. The final size of my SQLite database amounted to 203 megabytes.

The code used to export the tables to SQLite is stored within this repository as sqlite_database_builder.ipynb.

2. Exporting SQLite Database to Online Database Providers

2.1. Introduction

Now that I had my SQLite database, I needed to find a way to upload its 'flights', 'steps', 'music', and 'photos' tables to the following online database providers:

1. Amazon Web Services (AWS)
2. Google Cloud Platform (GCP)
3. Microsoft Azure
4. Snowflake
5. Airtable
6. Heroku
7. Databricks

Google Cloud Platform, AWS, and Azure were the only 3 companies within the 'Leaders' section of Gartner's July 2021 "Magic Quadrant for Cloud Infrastructure and Platform Services". Although I had heard of these 3 providers, I had little to no familiarity with Databricks, Heroku, Snowflake, and Airtable, the other four databases that Professor Guetta wished for me to explore.

First, I had to choose which version of SQL to use for the online databases. I went with PostgreSQL because I prefer its permissive open-source license to the copyleft/commercial licensing options for MySQL. Therefore, I installed the Windows x86-64-bit version of PostgreSQL onto my computer using this link, and also added PostgreSQL to my environment path.

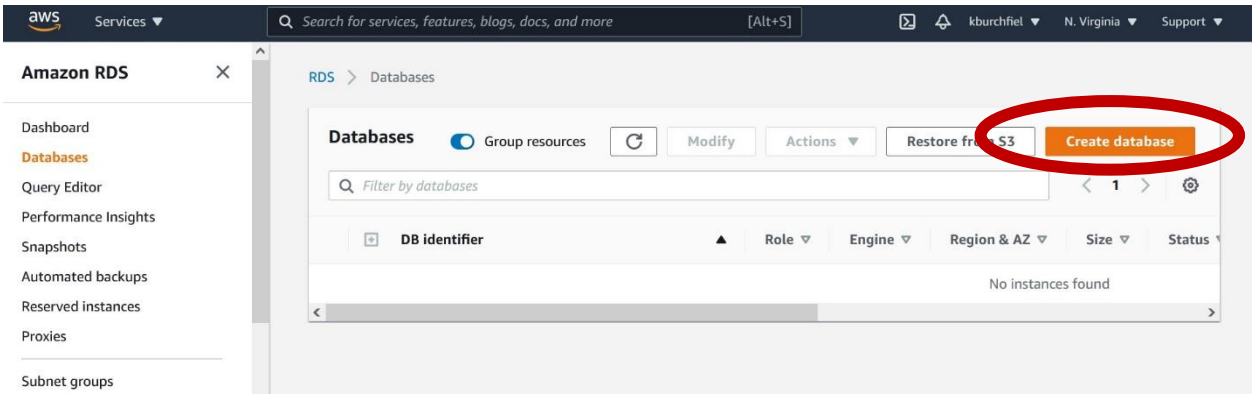
Next, I needed to determine how to upload my SQLite database into a PostgreSQL database. I initially experimented with pgloader (see appendix), but ultimately decided that learning and applying SQLAlchemy, a powerful Python library, would be a better way to set up my Amazon database. After all, SQLAlchemy works not only with PostgreSQL but with MySQL, Oracle, and Microsoft SQL server as well (in addition to SQLite): https://docs.sqlalchemy.org/en/14/dialects/ Furthermore, it integrates easily with Pandas, which can make the process of setting up new tables much easier.

The following sections focus on how to create a new database within each online provider. For the actual script that I used to connect to and populate each database, visit my database_uploader.ipynb file within the project repository.

2.2. Amazon Database Setup Process

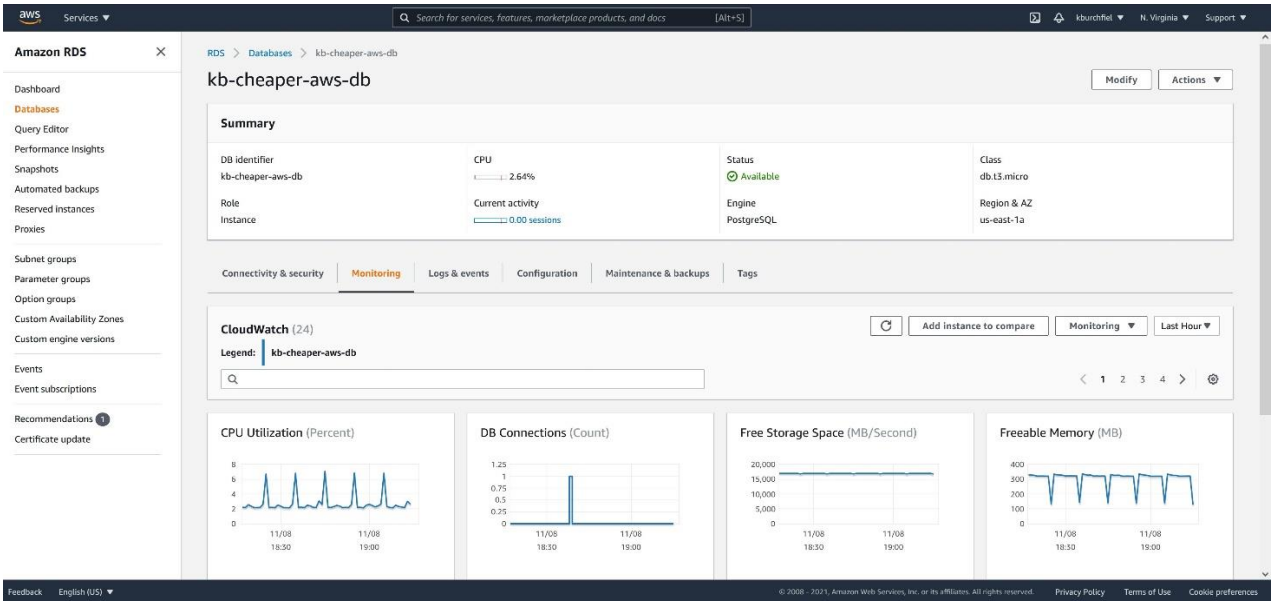
In creating my AWS database, I relied on a guide put together by Professor Mattan Griffel at Columbia Business School as part of his Intro to Databases class. However, since this guide is not publicly available, I recommend using Amazon's "Create and Connect to a PostgreSQL Database" and "Connecting to a DB instance running the PostgreSQL database engine" guides.

To set up my AWS database, I went to https://console.aws.amazon.com/rds/home, clicked on the 'Databases' tab, and then selected 'Create database.'



The first version of my AWS database incurred a cost of $180 for a single month. Therefore, I ended up switching that database to a cheaper option. The AWS pricing calculator was a helpful resource.

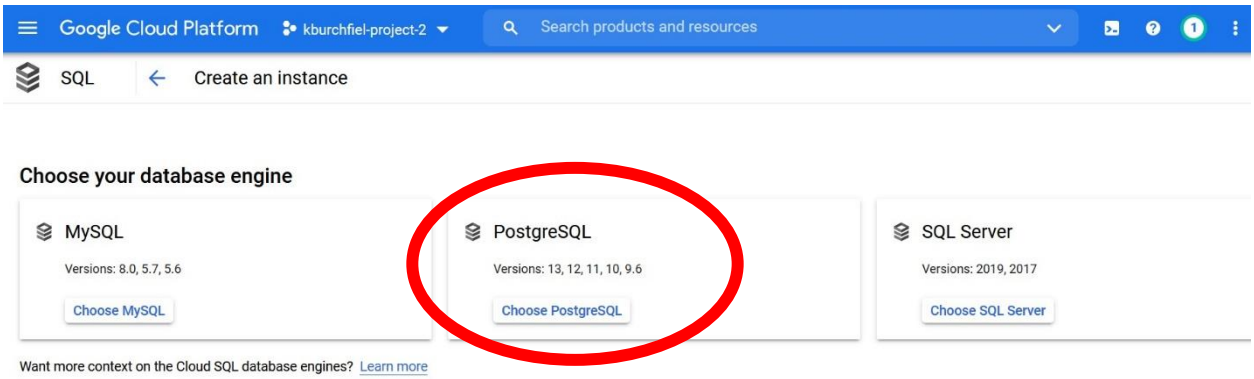
I ended up creating a new database with a db.t3.micro computing option (2 vCPUs, 1 gigabyte of RAM, and a 2,085 Mbps network connection) and only 20 GB of storage. The new estimated monthly cost was $15.44, but AWS warned that this didn't "include costs for backup storage, IOs (if applicable), or data transfer." It's possible that I overlooked a cheaper setup option.



A look at the UI for my AWS database

2.3. Google Cloud SQL Database Setup Process

Next, I turned my attention to setting up a Google Cloud database. The documentation links available on this page should help you get your own Google Cloud PostgreSQL database set up. In particular, I recommend looking into the "Set up your project" and "Create a Cloud SQL instance" sections of the Quickstart for Cloud SQL for PostgreSQL page.



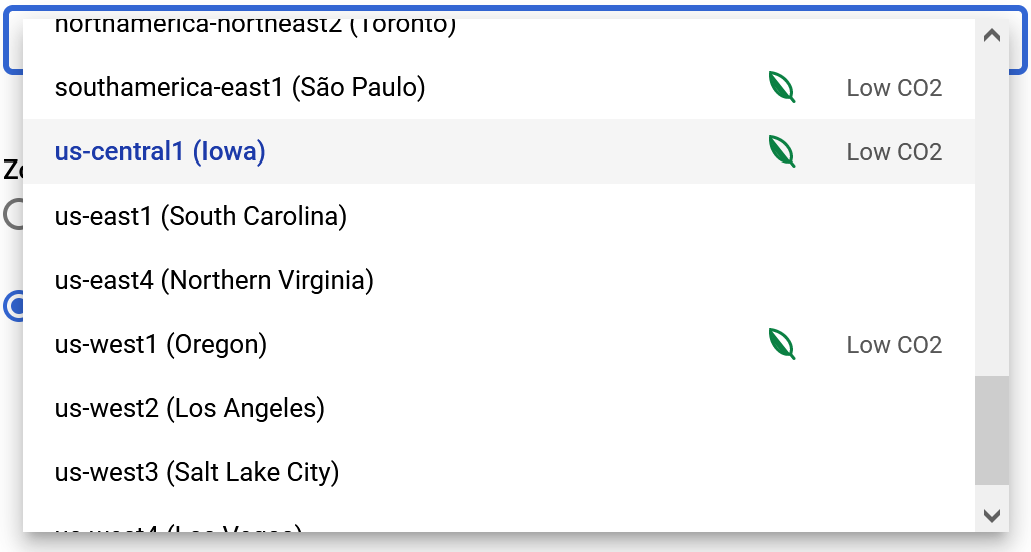
Setting up a Google Cloud Platform PostgreSQL instance

When creating my database, I chose to mix things up and select Iowa (us-central1) as the data region. Google said that this was a low-CO2 option, which couldn't hurt.

Choose region and zonal availability

For better performance, keep your data close to the services that need it. Region is permanent, while zone can be changed any time.

Region



I opted for a low-powered database to save costs. Here is the copied and pasted 'Summary' section of my database:

Summary
Region
    us-central1 (Iowa)
DB Version
    PostgreSQL 13
vCPUs
    1 vCPU
Memory
    628.74 MB
Storage
    10 GB
Network throughput (MB/s)
    125 of 125
Disk throughput (MB/s)
    Read: 1.2 of 125.0

Write: 1.2 of 37.9

IOPS

Read: 7.5 of 3,000

Write: 15 of 15,000

Connections

Public IP

Backup

Automated

Availability

Single zone

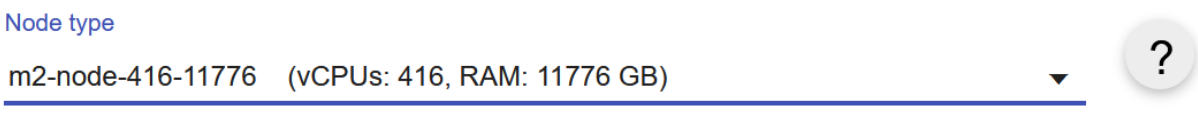
Point-in-time recovery

Enabled

I was then able to connect to my database within Python. The "Create a TCP connection by using SQLAlchemy" documentation page helped explain the connection process. These two articles helped explain the connection process, as did this Stack Overflow page. It turned out that the correct database name to use was simply 'postgres'.

In order to make it possible to connect to my database from any IP address, I entered 0.0.0.0/0 as an authorized network in the Connections page within Google Cloud Platform (as suggested by this page at mongodb.com.)
I then received the message: "You have added 0.0.0.0/0 as an allowed network. This prefix will allow any IPv4 client to pass the network firewall and make login attempts to your instance, including clients you did not intend to allow. Clients still need valid credentials to successfully log in to your instance."

Cloud SQL pricing details can be found here. On the pricing calculator page, I was very impressed by this node type:

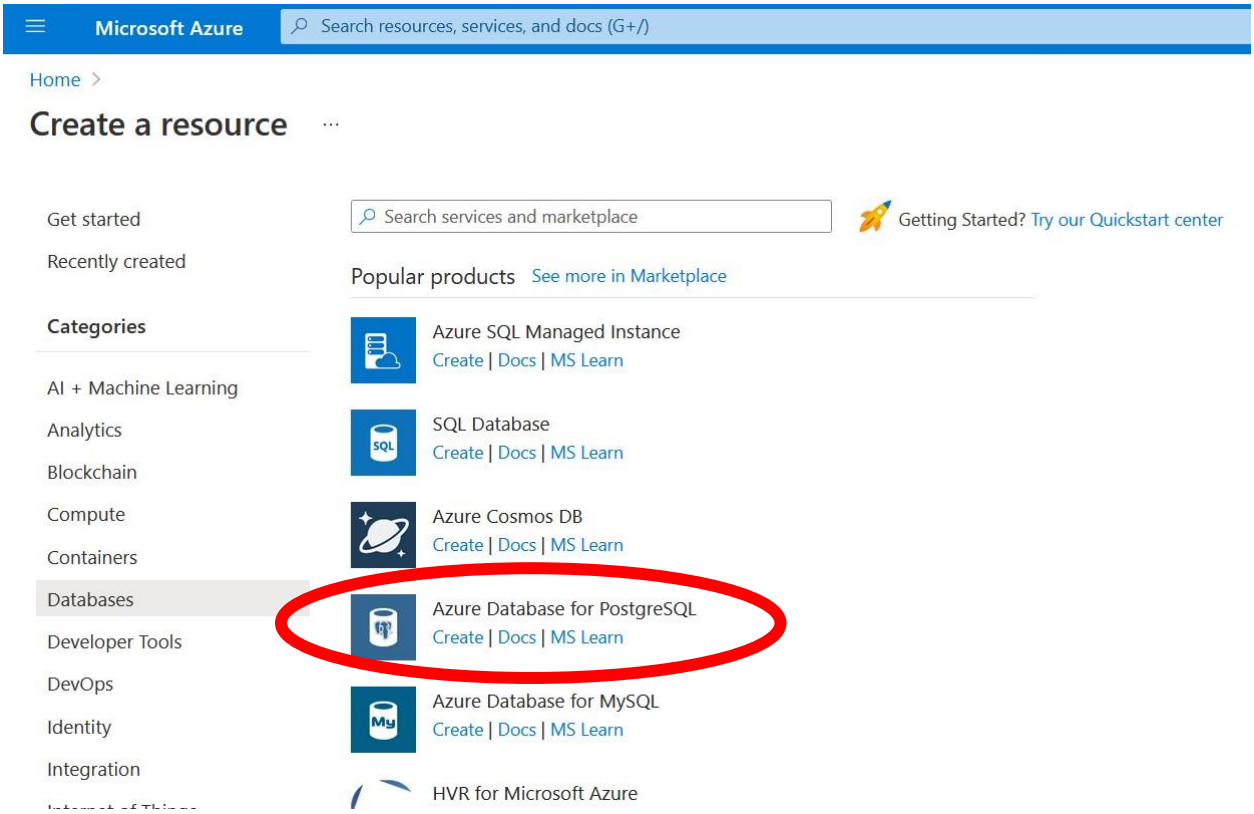


That's a lot of RAM! And certainly more than I needed for this project.

2.4. Microsoft Azure Setup Process

I next set up a Microsoft Azure database. I was happy to see that Azure Database for PostgreSQL was one of the free services options that Azure offers. It was also nice to receive $200 in free credits, even though I didn't plan to incur too many expenses with my database to begin with. A helpful guide to setting up a PostgreSQL database within Azure is available via this quickstart page.

After you create an account (which for me involved logging in with my regular Microsoft account), you can go to https://portal.azure.com/#home; select 'Create a Resource'; and choose the "Azure Database for PostgreSQL" option.



When configuring my database, I chose the 'Development' option in order to save money, but this option was still projected to cost $28 a month.

I then added a firewall rule to allow all IP addresses (from 0.0.0.0 to 255.255.255.255 to

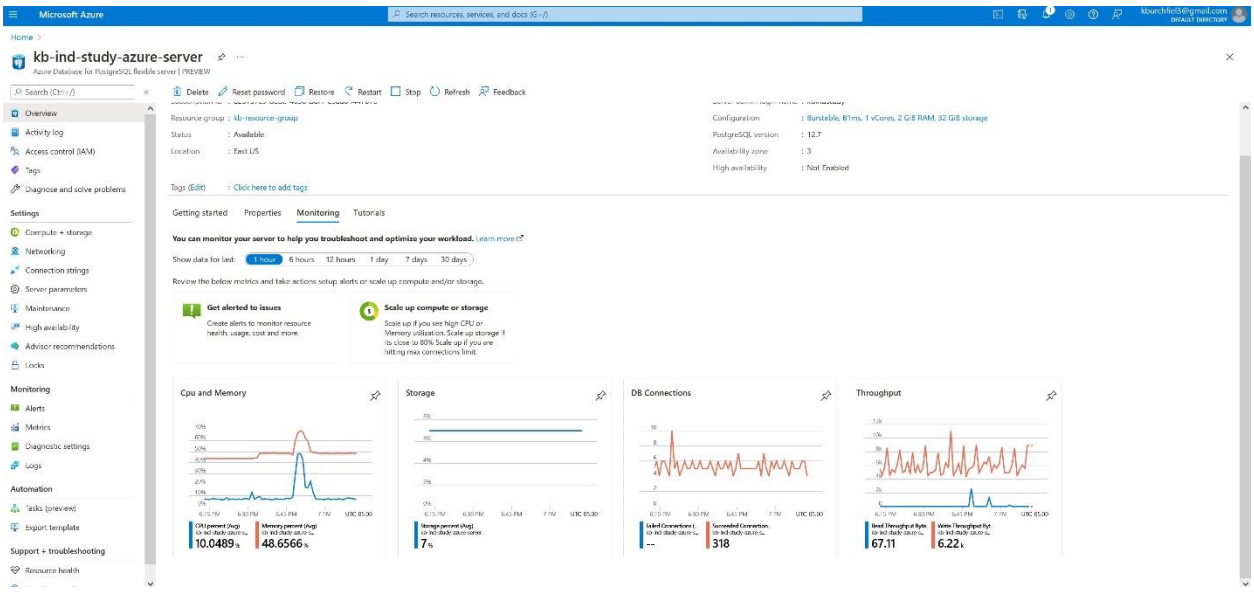
make the server easier to access.)

These were the details that I entered:
Flexible server (Preview)
Microsoft - preview
BasicsRequired
Networking
Tags
Review + create
Product details
Azure Database for PostgreSQL
by Microsoft
Terms of use | Privacy policy
Estimated cost per month
28.50 USD
Basics
Subscription
Azure subscription 1
Resource group
kb-resource-group
Server name
kb-ind-study-azure-server
Server admin login name
kbindstudy
Location
East US
Availability zone
No preference
High availability (Zone redundant)
Not Enabled
PostgreSQL version
12
Compute + storage
Burstable, B1ms, 1 vCores, 2 GiB RAM, 32 GiB storage
Backup retention period (in days)
7 day(s)
Networking

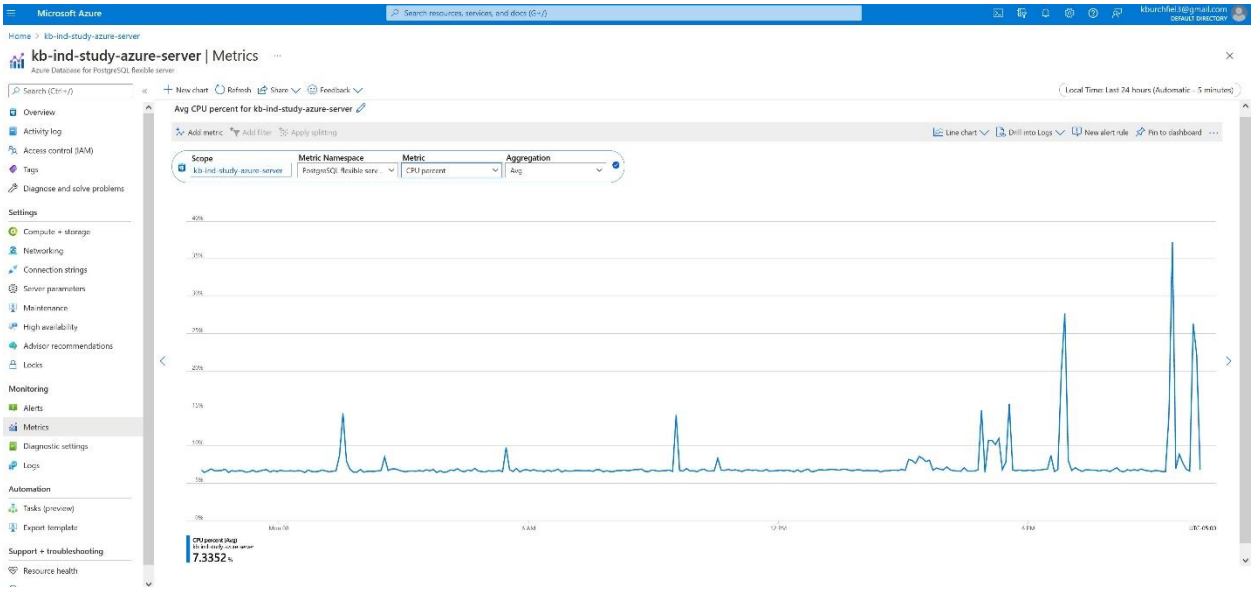
Connectivity method
Public access (allowed IP addresses)
Allow public access from any Azure service within Azure to this server
Yes
Firewall rules
1

The ‘Connection Strings’ page within the Azure portal proved helpful for connecting to my database. For instance, it provided the following connection string:
PostgreSQL URL: postgres://kbindstudy:{your_password}@kb-ind-study-azure-server.postgres.database.azure.com/postgres?sslmode=require

I was able to use this connection string to connect to my database once I changed postgres:// to postgresql://, as SQLAlchemy no longer supports the ‘postgres’ dialect.



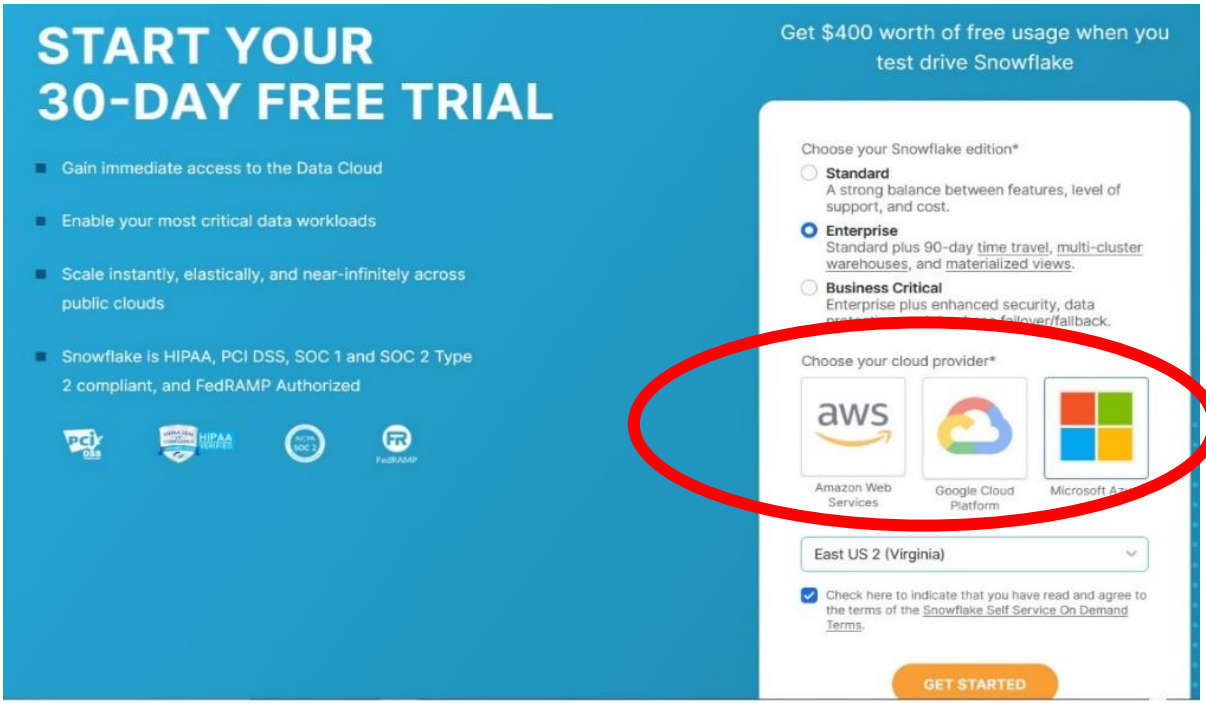
A look at Azure’s UI



Azure's Metrics page

2.5. Snowflake Setup Process

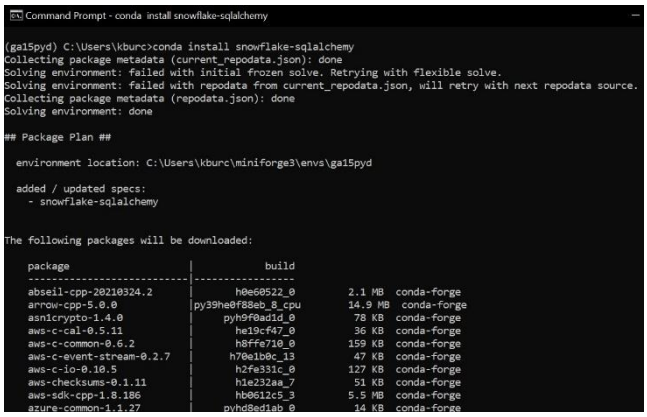
I found it interesting that, when setting up a database with Snowflake, I could select either AWS, Google, or Microsoft as my cloud provider.



In addition, I was surprised that I wasn't asked to specify a specific SQL type (e.g. PostgreSQL) when creating my database. I also did not have to provide any credit card information to launch my Snowflake account, which I also appreciated. (I'm guessing this was due to the 30-day free trial feature.)

Snowflake allows tables to be created manually; within the page for a given database, I could press the 'Create' button to build a table with a series of columns. However, I preferred to build my database using a direct upload of my SQLite database, since it would require less manual input.

Since I had connected to my AWS, GCP, and Azure databases using SQLAlchemy, I wished to do the same for my Snowflake database; I learned how to do so using this guide. I like to install packages using conda-forge (since I run Python using Miniforge, thus letting me avoid Anaconda usage fees), and was able to find the Snowflake SQLAlchemy package there.



Installing Snowflake's SQLAlchemy package using conda-forge and Miniforge

In order to create a connection to my Snowflake database, I needed to enter my account locator, which could be found as the first part of my URL. For instance, given that the URL to my Snowflake portal was https://rv85777.east-us-2.azure.snowflakecomputing.com/console#/data/views?databaseName=KB_SNOWFLAKE_DB, my account identifier was 'rv85777.east-us-2.azure.'
I was also able to find my current account by entering select current_account() into a new

worksheet (as shown in this documentation page).

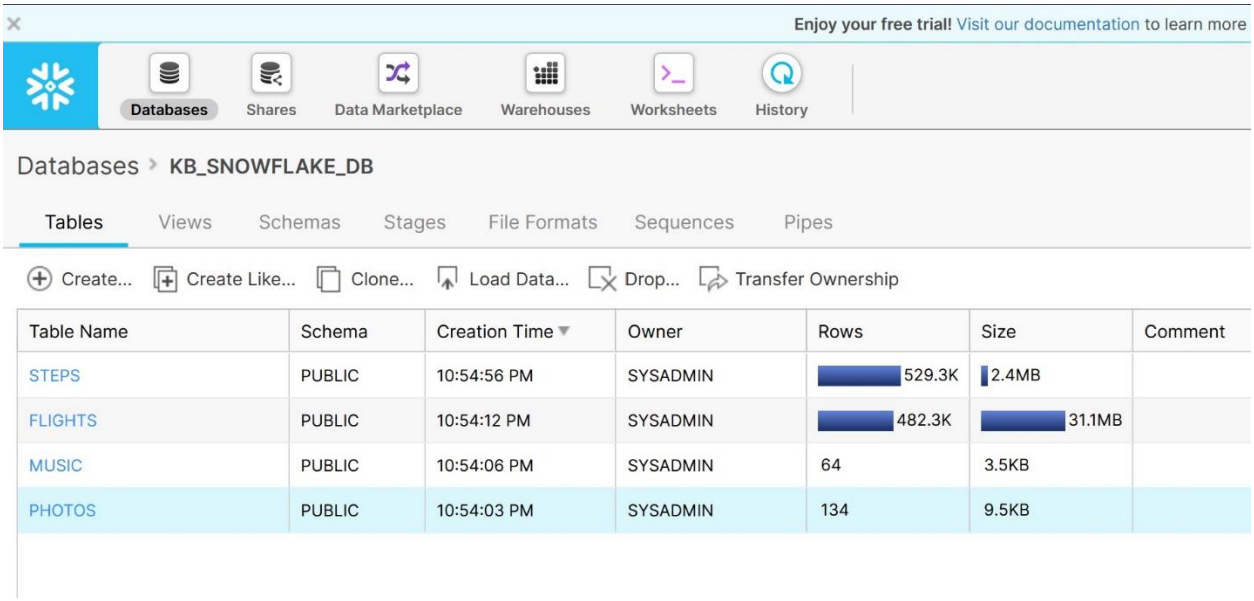
The account identifiers page also shows the corresponding Snowflake Region ID for each Region ID: https://docs.snowflake.com/en/user-guide/admin-account-identifier.html

I had trouble getting my username and password to work successfully within SQLAlchemy, but finally solved the issue by changing to a simpler password. My password was about 50 characters long and included a number of special characters; I'm guessing that one of those characters might have caused an issue with the string.

I thought I would then be able to create databases in the same way as I had with AWS, GCP, and Azure, but when I tried to run my code that copied SQLite tables into Snowflake, I received a 'NotImplementedErorr' saying that "Snowflake does not support indexes." (This documentation provides a bit more information.)
To solve this issue, I set to_sql's index parameter to False and imported pd_writer from snowflake.connector.pandas.tools so that it could be used as the 'method' argument in the to_sql function call. (See Snowflake's "Using Pandas DataFrames with the Python Connector" page for more information.)

Once I dealt with the lack of index support by selecting 'indexes = False' within the to_sql documentation, I then received an 'invalid identifier' error that appears to be connected to my use of lowercase table column names, as indicated by this post on community.snowflake.com and this GitHub issue.
To solve this problem, I used map() to convert all of my tables' column names to uppercase as suggested by Stack Overflow user CodingInCircles here.

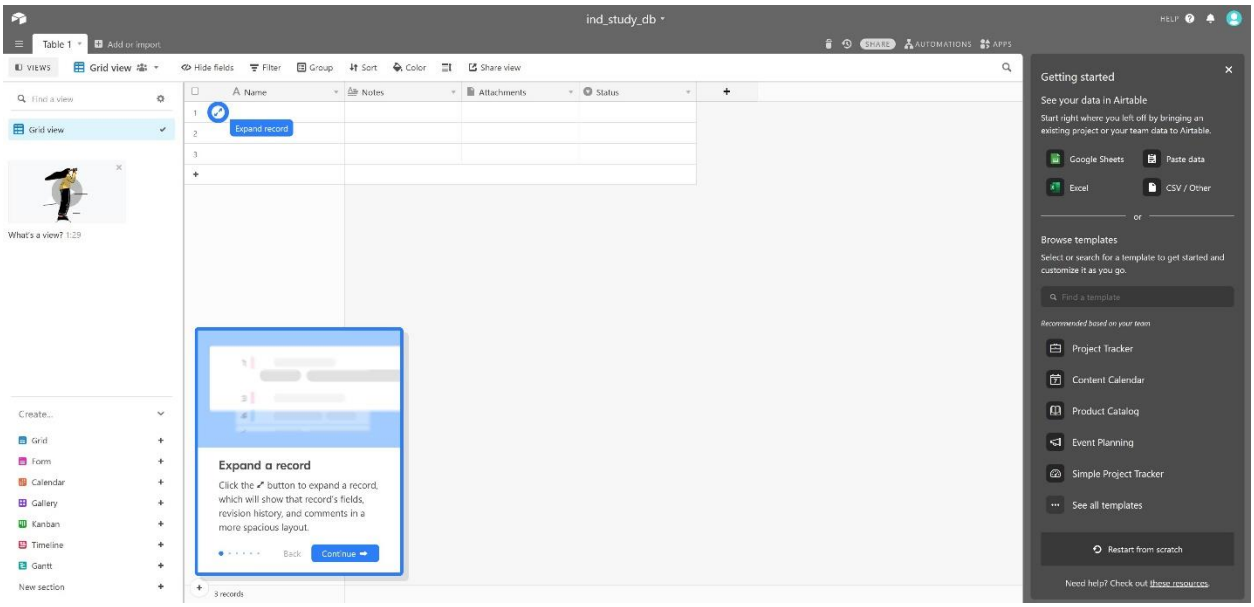
Ultimately, I had to jump through some additional hoops to get my SQLAlchemy workflow to function for Snowflake, so I found the setup process to be somewhat less straightforward than for GCP, AWS, and Azure.



Snowflake's UI, showing my 4 tables along with their row counts and sizes. Interestingly, the sum of the sizes of all four tables is much smaller than the 203-megabyte size of my SQLite database, which leads me to wonder whether Snowflake is applying some sort of compression algorithm on my data.

2.6: Connecting to Airtable

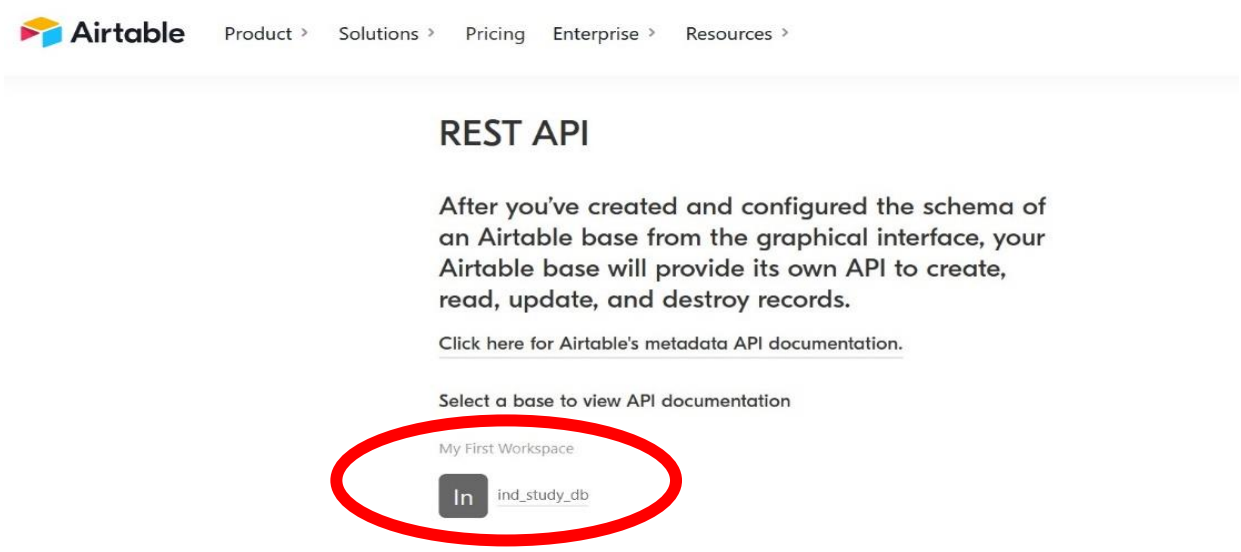
After I created a free Airtable account, I was then invited to create a new 'base.' This base resembled a spreadsheet, although I found the 'Kanban' and 'Gantt' options interesting as well. The 'Project Tracker' and 'Event Planning' options could also prove helpful.



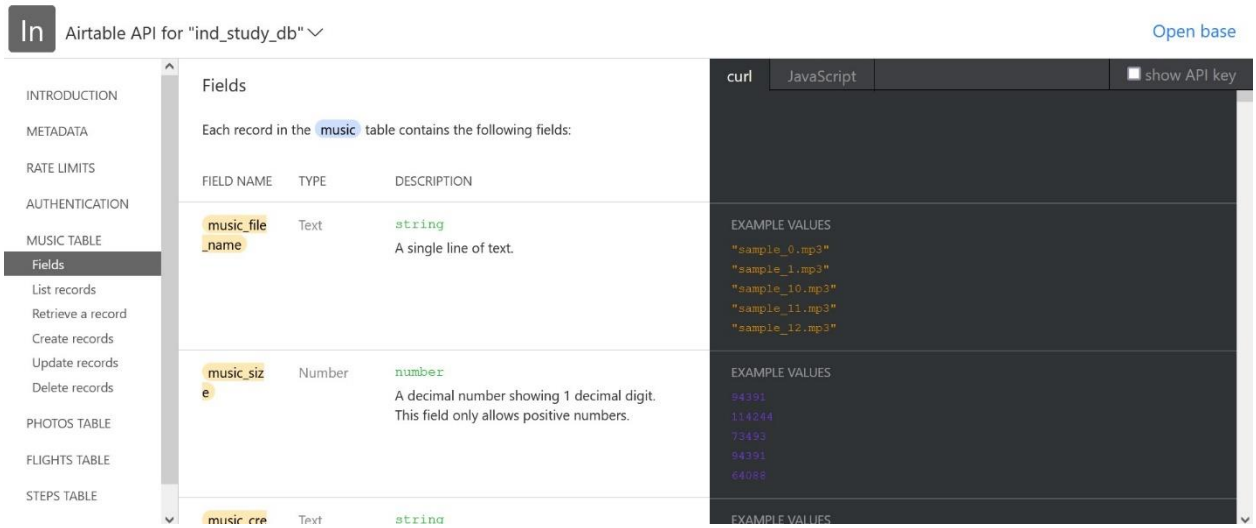
Beginning the Airtable setup process. Note the 'Create' options on the bottom left.

While I had the opportunity to import data from Google Sheets and Excel, I preferred to set up a database through using Python.

I found a Python connector available through CData, but I would have to pay for it. I then learned that Airtable has its own API. Once you have a workspace set up, you can access that API by going to the Rest API page and clicking on the link to that workspace. Meta documentation for Airtable's API is available here.



Clicking on ind_study_db (circled in red) allowed me to access the custom API for my workspace.



The custom API page for my workspace. The tables, field names, and example values automatically update to match the values in my tables.

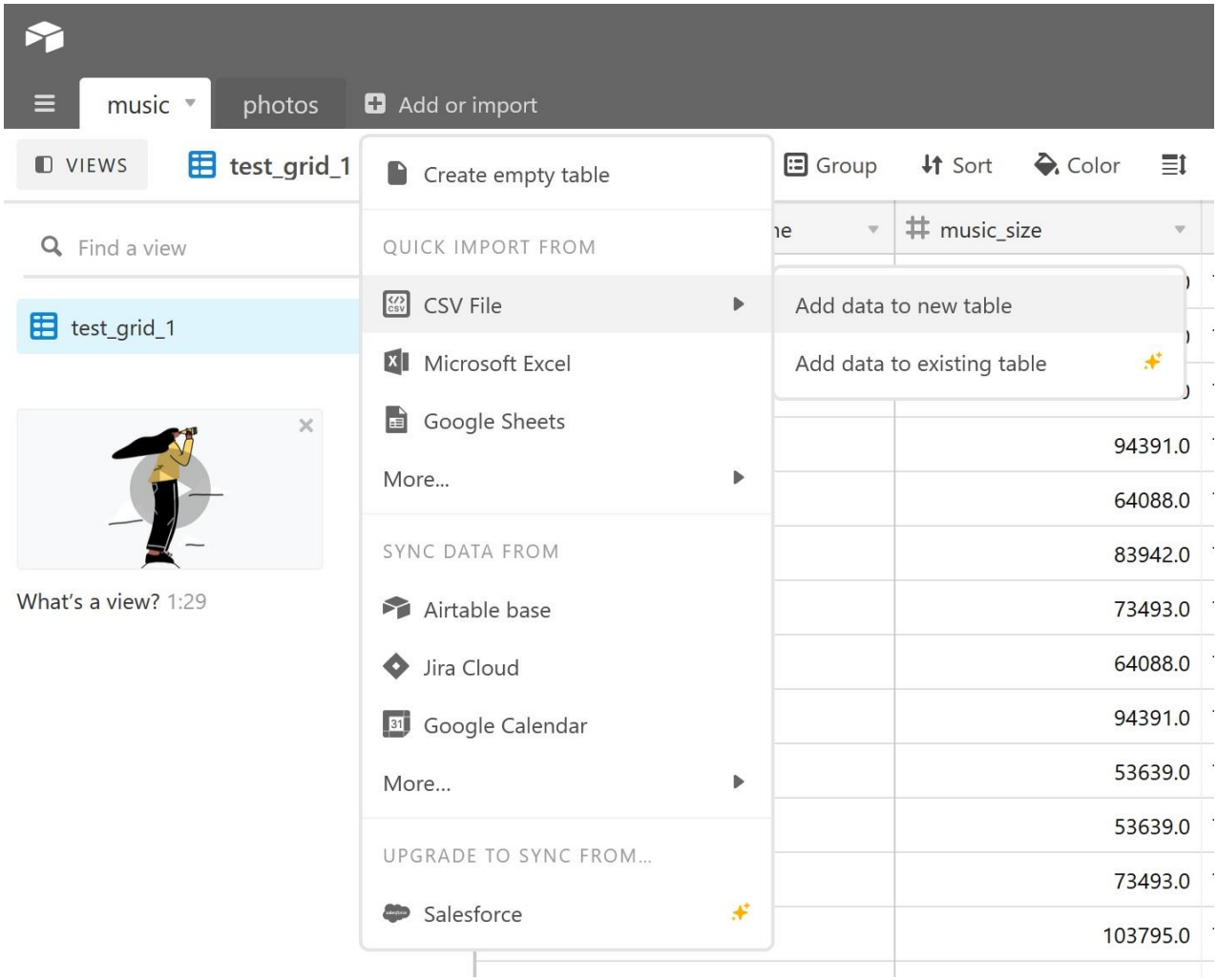
The API, according to the custom documentation page created for my ind_study_db workspace, "provides an easy way to integrate your ind_study_db data in Airtable with any external system. The API closely follows REST semantics, uses JSON to encode objects, and relies on standard HTTP codes to signal operation outcomes."

To use the API, I generated and downloaded an API key on my account page, then found the ID for my base. The API page noted airtable.py and pyAirtable as two methods for connecting to Airtable using Python. Of the two, pyAirtable was updated more recently and had more stars on GitHub, so I went with that one.

Ultimately, I was able to upload both my music and photos tables to Airtable using the API; however, some manual work was necessary to prepare the tables for upload. (See the database_uploader.ipynb script for details.)

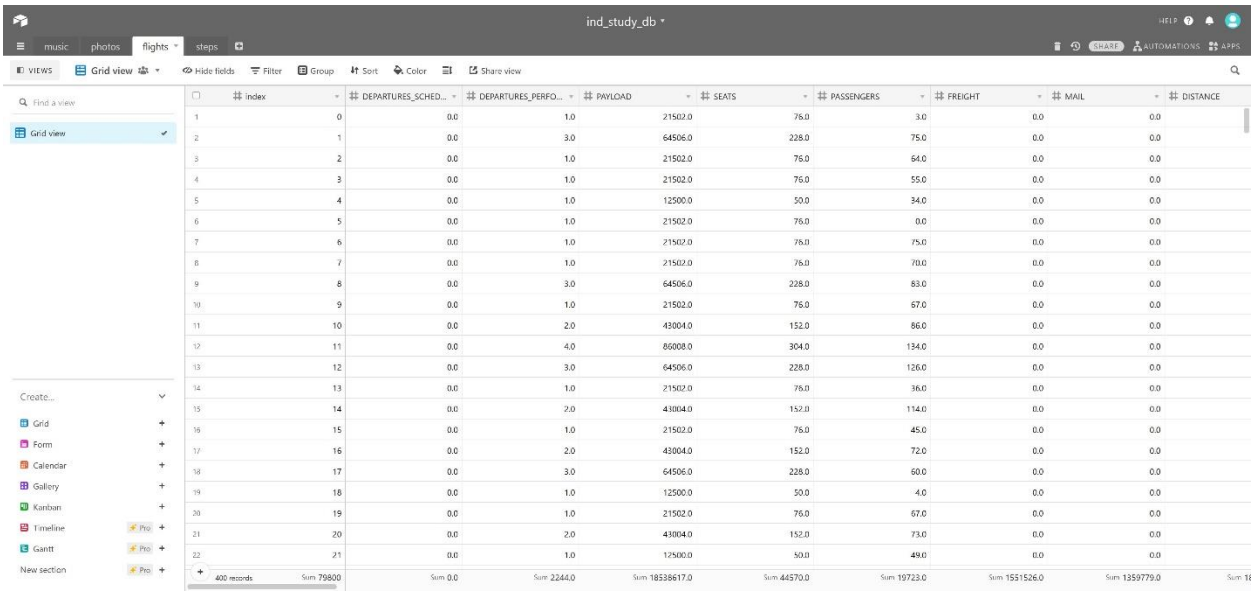
The API limits the rate of batch updates to 50 records per second (e.g. 5 requests of 10 records each), which meant that it would have taken 2.7 hours to upload 500,000 records to Airtable via the API. Therefore, I decided to use Airtable's .csv upload feature to import my flights and steps tables, as both contain hundreds of thousands of rows. However, when I tried to do so, I received a message saying that "the accepted file size is less than 5 MB." In addition, I found that even the CSV import app from the 'Pro plan' could only support 25,000 rows—far fewer than the 482,273 rows in my flights table and the 529,256 rows in my steps table. Furthermore, a given table supports only 50,000 rows. Airtable's explanation for this, which can be found via the same link, is that "Airtable is meant primarily for cases where users interact with data through the interface, and we find that once you get over 50-100K records, it really requires a more powerful querying language to meaningfully interact with the data."

I therefore decided to create 400-row versions of my flights and steps tables that would fit not only within the 50,000-row-per-table limit, but within the limit of 1,200 total records per base. I saved these versions as .csv files that I could then import into Airtable. (Otherwise, I would have to manually set up the columns for each of these tables, which would take a while in the case of the 62-column flights table.)



Airtable facilitates manual .csv import via its web user interface.

Ultimately, it looks like Airtable is better suited as a product management tool than as a serious replacement to a database host like AWS, GCP, or Azure.



Airtable User Interface. Note that Airtable allows you to view the individual cells within tables.

2.6.1. Airtable vs. Google Sheets

Although it's nice to be able to view rows in the tables that I create within Airtable, I can also do that via Google Sheets, which supports up to 5,000,000 cells within a single worksheet.

I have found that, with Google Sheets, once I get my service account set up, it's very easy to import Google Sheets data to Pandas by using gspread's get_all_records function, which returns an object that can then be converted to a DataFrame. (See gspread's "Examples of gspread Usage" page for more details.) In addition, it's easy to export DataFrames to Google Sheets using gspread_dataframe's set_with_dataframe function (https://pythonhosted.org/gspread-dataframe/). Therefore, if I wanted to host a database in an easily viewable format, I would choose Google Sheets over Airtable.

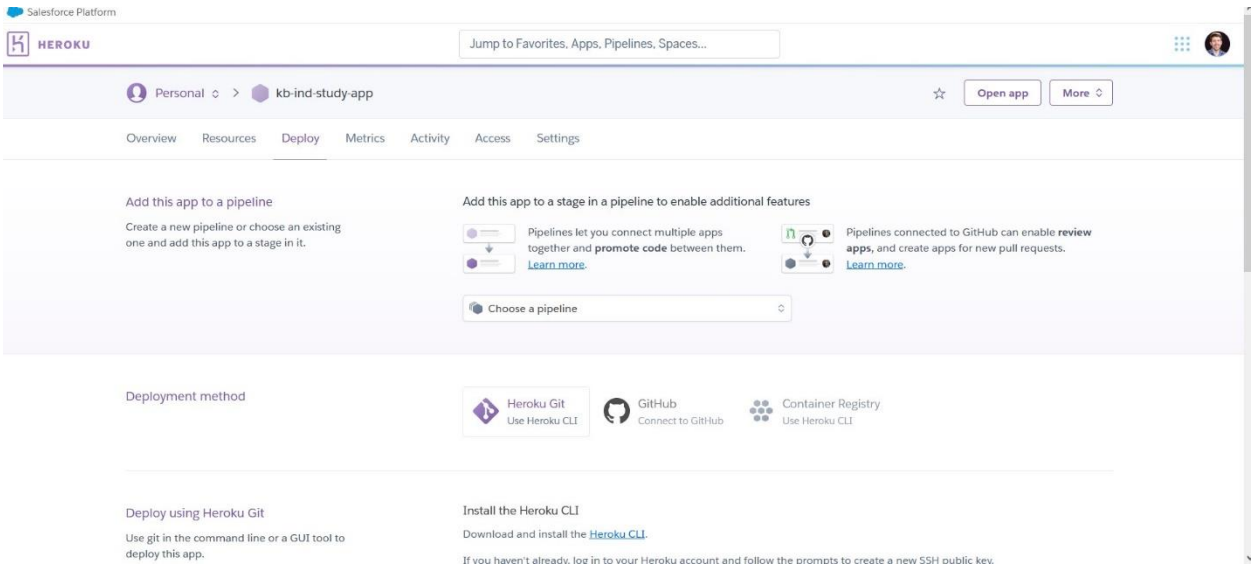
I plan to create a separate GitHub repository showing how to import data from a SQLite

database into Google Sheets; once I do so, it will be available under the name db_gs_import.ipynb.

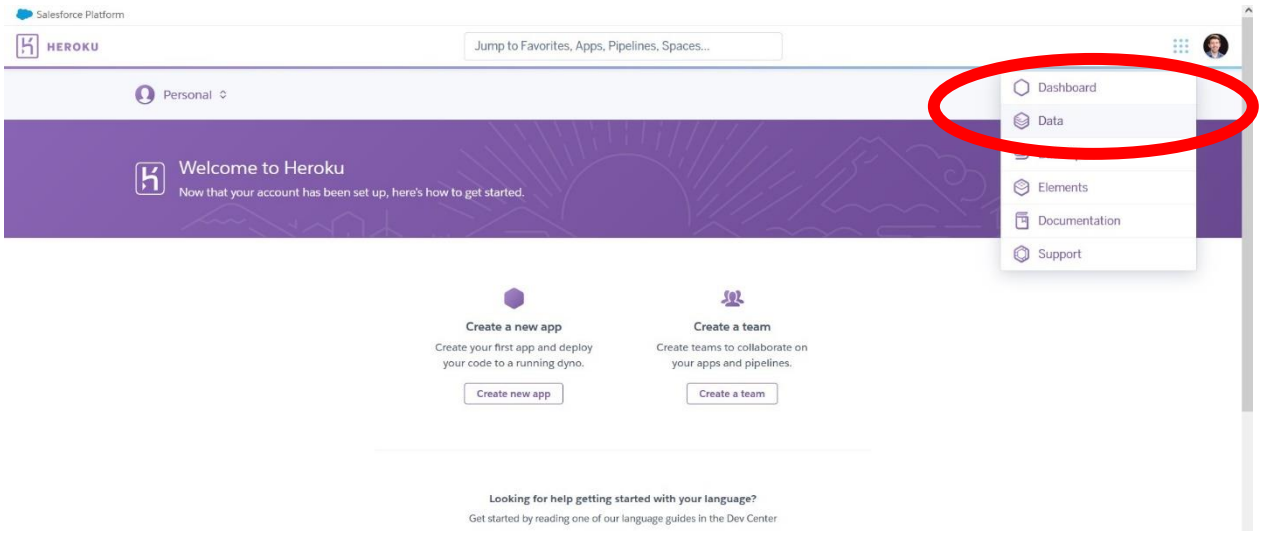
2.7: Heroku Setup Process

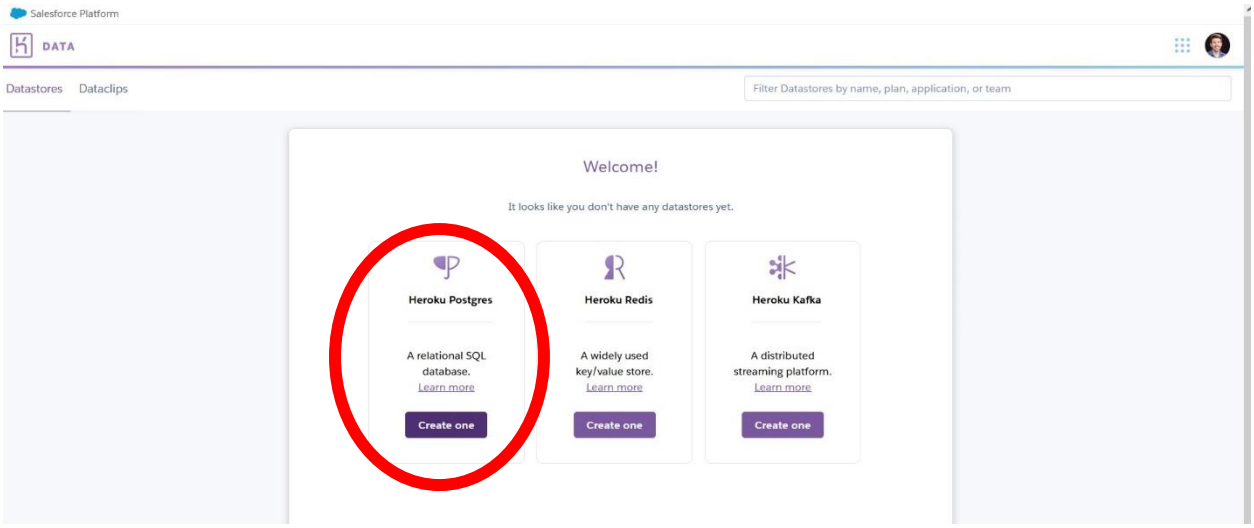
The signup page invites you to "Go from code to running app in minutes," and to "Deploy, scale, and deliver your app to the world." It also asked for my primary development language, for which I entered Python.

The signup process was pretty easy. Once I was within Heroku, I could either "create a new app" or "create a team." I found out that, in order to set up a database, I would first need to connect it to an app, so I went ahead and created one, which I called 'kb-ind-study-app'.



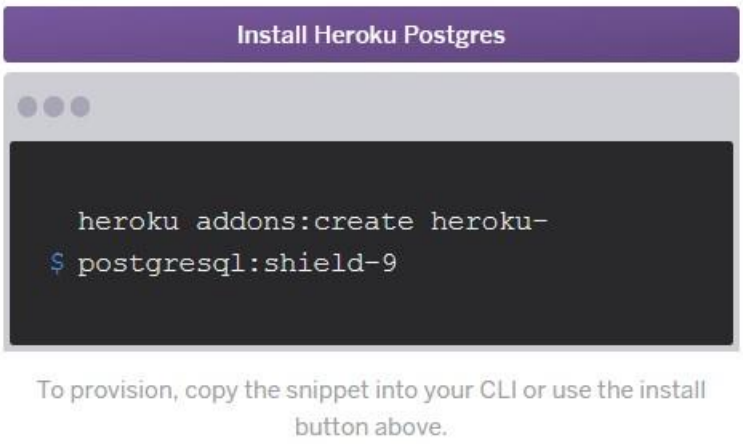
Heroku supports PostgreSQL under its "Heroku Postgres" service and offers extensive PostgreSQL documentation on its website. Once I had my app in place, I could then add Heroku Postgres to it. I was able to get to the Heroku Postgres section by clicking the 9 dots near my icon on the top right, then selecting 'Data' (see image below).





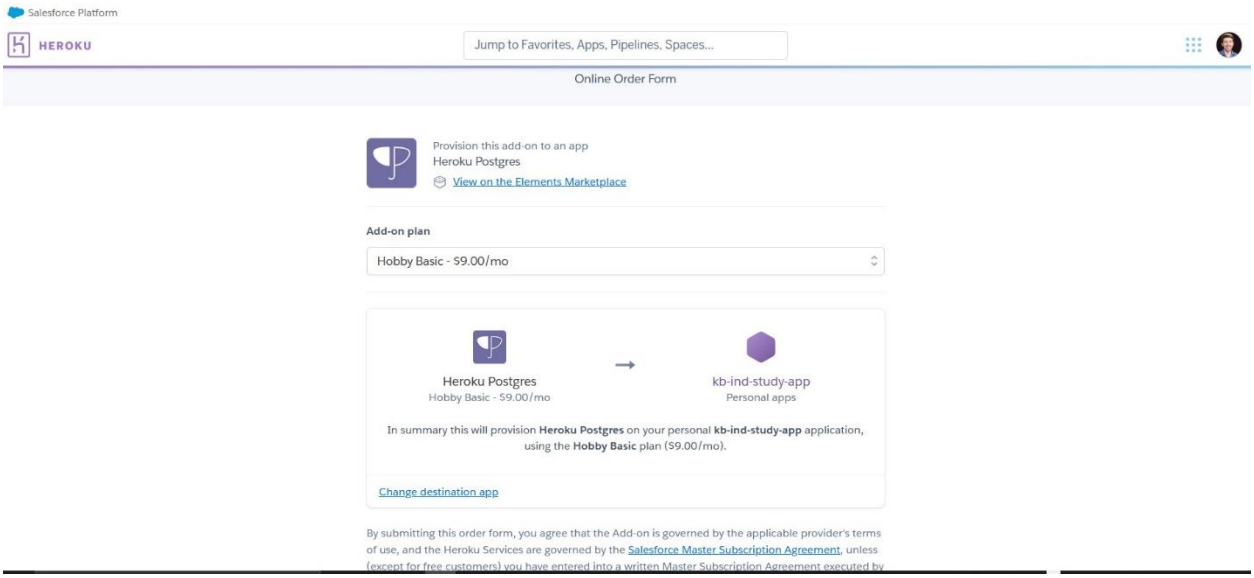
This then took me to the [Heroku Postgres setup page]:

Heroku offered a free 'Hobby Dev' plan, but it came with a row limit of 10,000, so I instead opted for the $9/month "Hobby Basic" plan. This plan allowed for 10 million rows but 0 bytes of RAM. (If I wanted 768 gigabytes of RAM, I could have opted for the "Shield 9" plan for $16,000 a month. If I instead needed RAM, I could have chosen the Standard 0 plan for $50 a month.

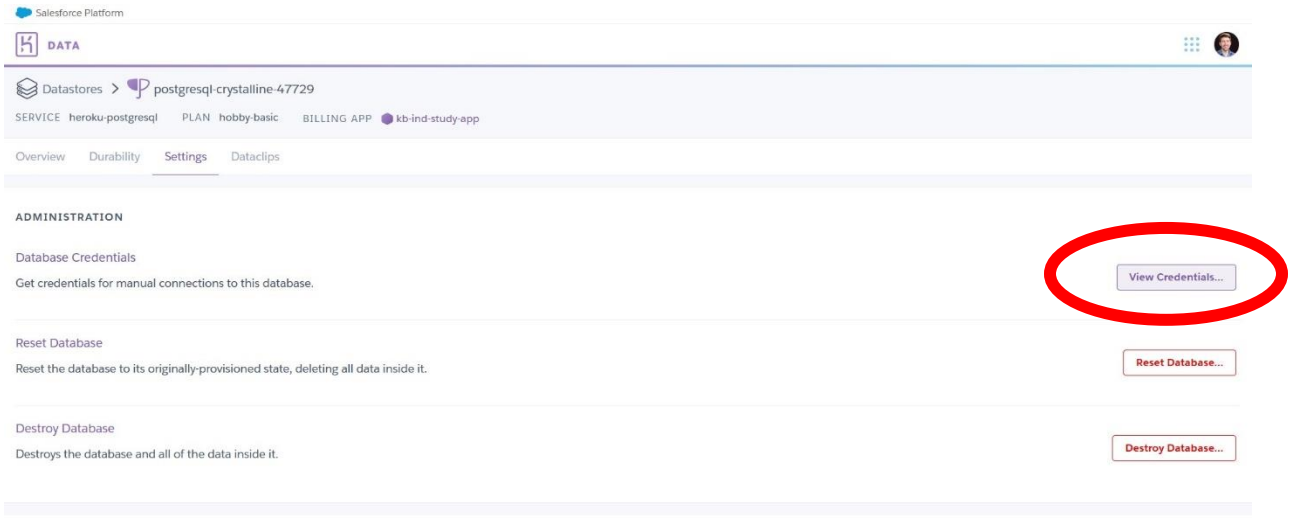


This would have been a very expensive line of code!

After selecting my database plan, I was able to connect it to my app.



My database was given the automatically-generated name 'postgresql-crystalline-47729.' I was able to find my connection parameters by going to 'Settings' and then choosing 'Database Credentials.'

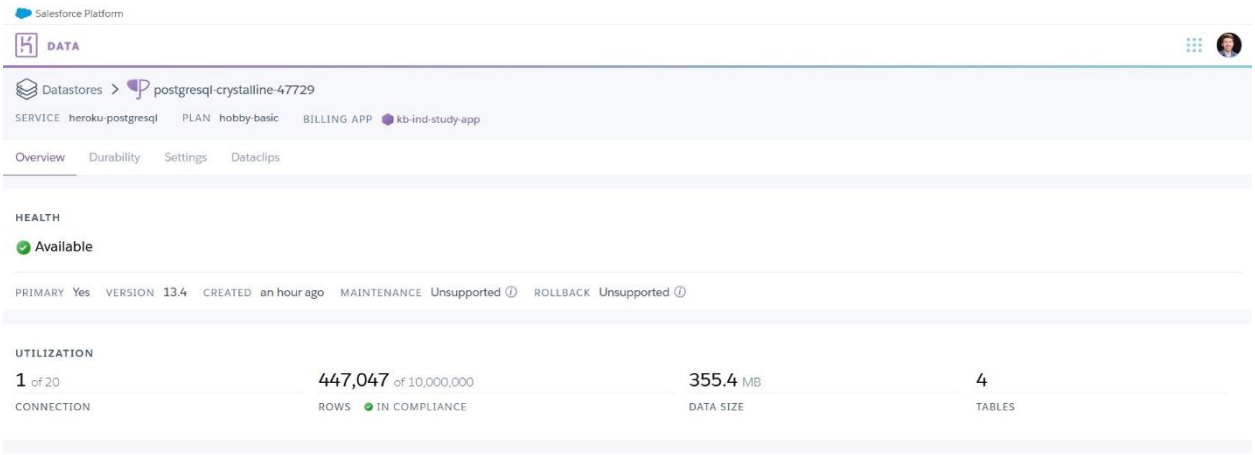


However, the Credentials page also warned me that "these credentials are not permanent," as "Heroku rotates credentials periodically and updates applications where this database is attached."

I noticed that my URI included 'compute-1.amazonaws.com', which suggests that my database was actually hosted on AWS.

When trying to connect to the database using the provided URI, I received an error, but there was an easy fix—I just needed to change 'postgres' in the URI to 'postgresql.'

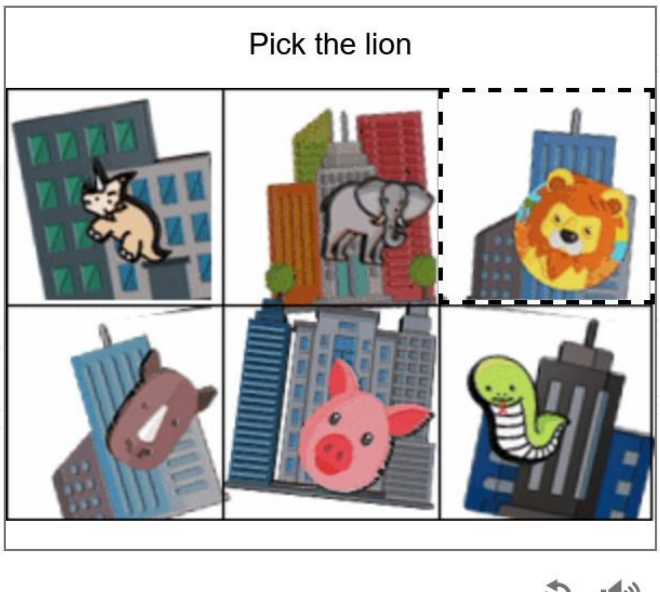
I did like that my database page on data.heroku.com made it easy to view both my row count and my database size (see image below). The 'data size', 'rows', and 'tables' fields updated multiple times as my Python script ran, which made it easier to track the progress of my database import.



Ultimately, despite the additional requirement to create an app, the Heroku setup process was fairly straightforward, as the same table import function that worked for GCP, Azure, and AWS also worked for Heroku. That's the power of SQLAlchemy!

2.8. Databricks Setup Process

I went to databricks.com and set up an account. The signup page announced that Databricks is an "open and unified data analytics platform for data engineering, data science, machine learning, and analytics."



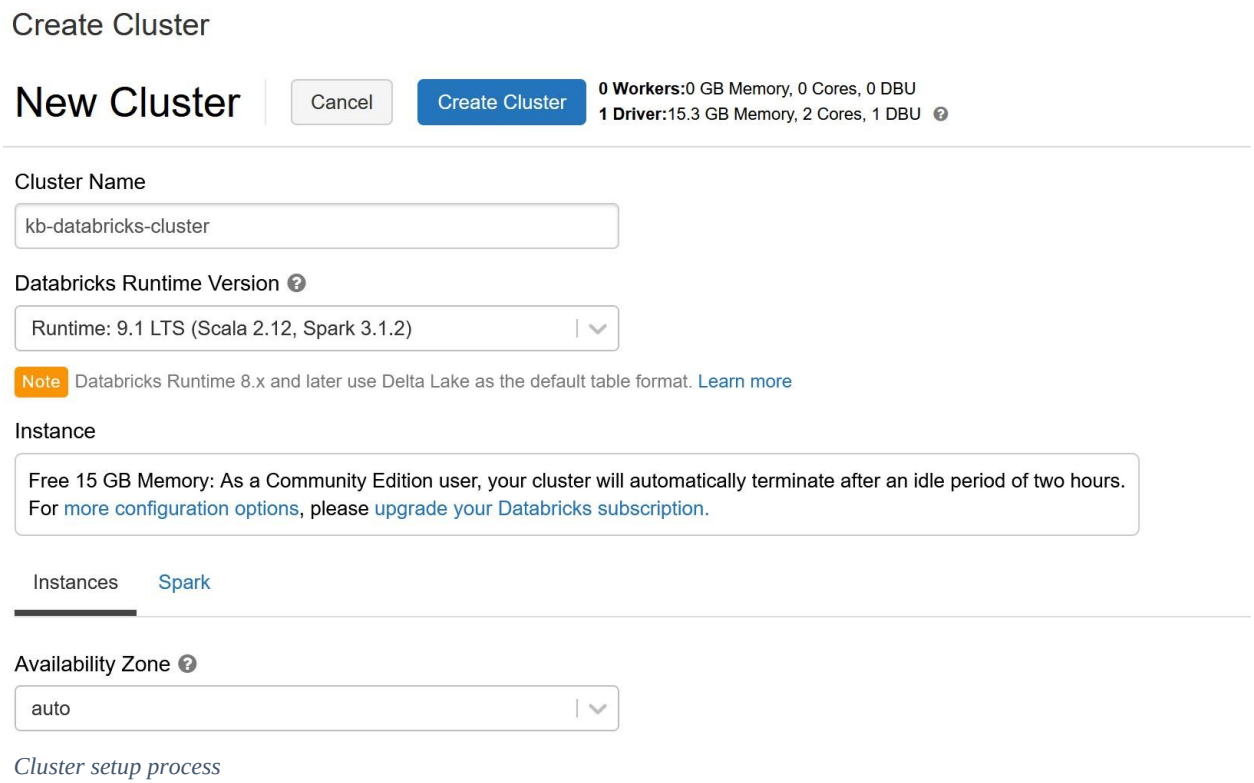
This was the most adorable user verification process I had ever seen.

In order to create a database within Databricks, I first needed to create a 'cluster,' which Databricks defines as "a set of computation resources and configurations on which you run data engineering, data science, and data analytics workloads." Although a free version of Databricks was available, it did not appear to allow account token generation, which I needed for connecting to my database within Python; therefore, I chose to set up a paid account.

When I went to Settings → Manage Account within my portal on https://community.cloud.databricks.com, I was then taken to a subscription signup page that directed me to AWS (as opposed to GCP or Azure). After entering my billing information, I was then asked to connect to my AWS account. I did so by generating an access key on this AWS page. That page also had my AWS Account ID, which was also needed to connect Databricks to AWS.

I then created an AWS bucket as directed by this page. Next, I created a cluster. I chose to use the m5a.large 'worker type' since it used the lowest database units (DBUs) per hour of any of the options I checked.

Launching a cluster appeared to take about 4 minutes each time. I could have kept the



Cluster setup process

cluster running, but I'm not sure how expensive that would have been.

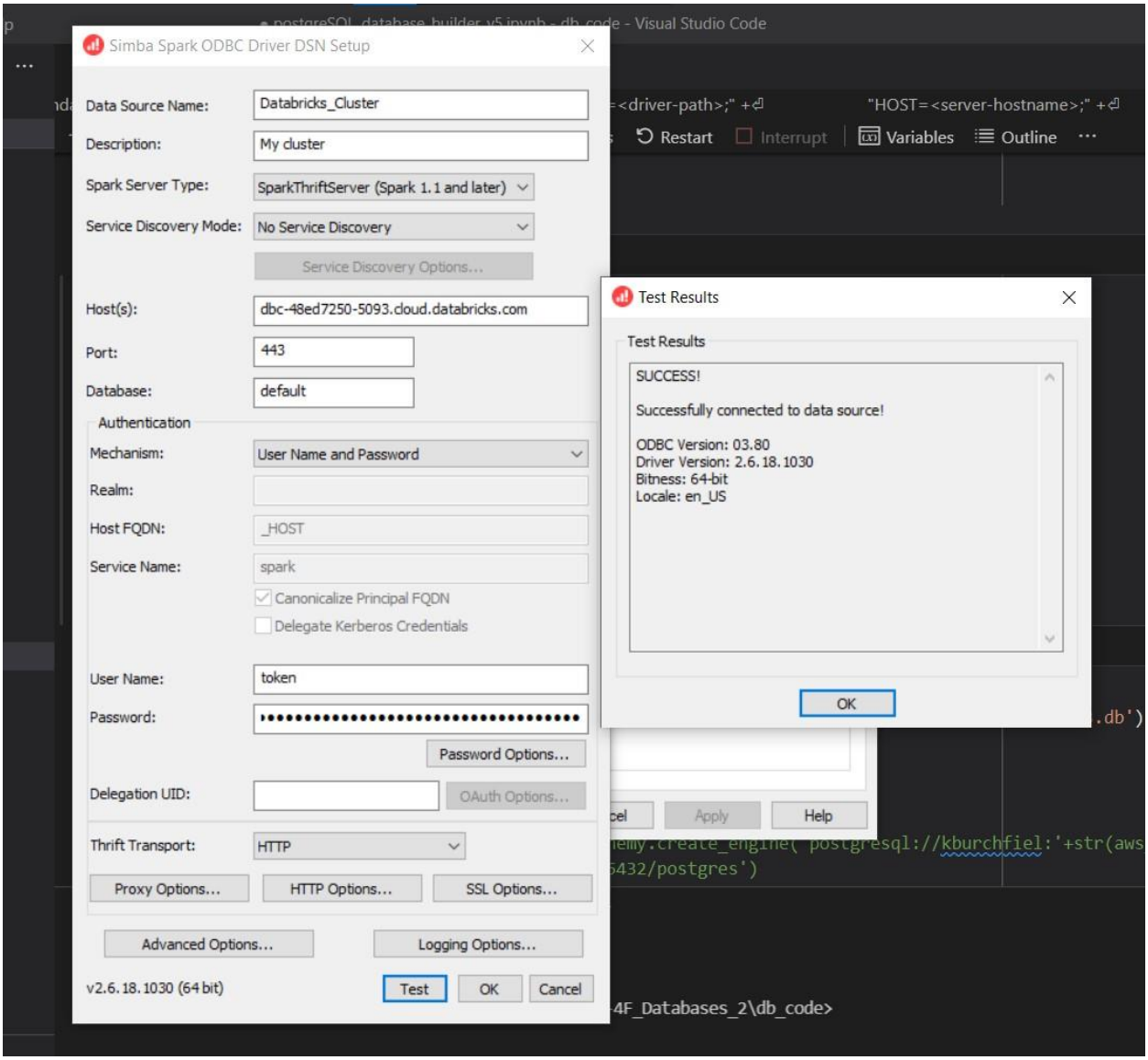
At this point in the process, I had a lot of uncertainty over how much I would spend on both the Databricks side and the AWS side in order to launch and maintain a database. However, it appeared that I wouldn't be charged very much as long as I only ran clusters for short periods of time.

I planned to connect to Databricks using SQLAlchemy as I had for most other databases. A databricks-dbapi Python package makes this possible. However, when I tried to install the library using pip (pip install databricks-dbapi[sqlalchemy], as specified at the above url), I received the following error message:

> "Microsoft Visual C++ 14.0 is required. Get it with "Build Tools for Visual Studio": https://visualstudio.microsoft.com/downloads/"

As noted by the error message, to use this library, I would need to install Build Tools for Visual Studio. Based on the terms of use and on this Stack Exchange thread, it looks like there are restrictions on using these materials for free in commercial settings, so I instead decided to go with Pyodbc.

I set up a Pyodbc connection using the following help page. This process involved setting up a 'Simba Spark ODBC Driver' (see below); my personal access token served as the password.



Unfortunately, Pandas' to_sql command doesn't work with Pyodbc PostgreSQL connections, so I instead had to create tables using explicit 'CREATE TABLE' statements. In order to do so, I also had to update the variable types of my columns using the following reference.

This reference confused me at first because I interpreted 'StringType' and 'FloatType' as the type names that I would need to insert within CREATE TABLE statements. However, I realized based on the examples at the bottom of this page that the actual type names to include were 'string' and 'float.' (This guide shows the actual types.) Databricks appears to use its own SQL format, or possibly Apache Spark SQL.

I used pyodbc to populate my music, photos, and steps tables. The lack of to_sql

functionality made this a more complex process; for each table, I needed to manually create the table, then fill in the values using pyodbc's executemany() function.

In order to successfully populate the steps table, I needed to upload data in chunks. Instead of using executemany() to upload the entire table at once, I ran this function for 100,000 rows at a time.

Although I tried a similar approach for the flights() table, I wasn't able to successfully import it into Databricks due to various error messages. For instance, the following message appeared between 240,000 and 249,999 rows in:

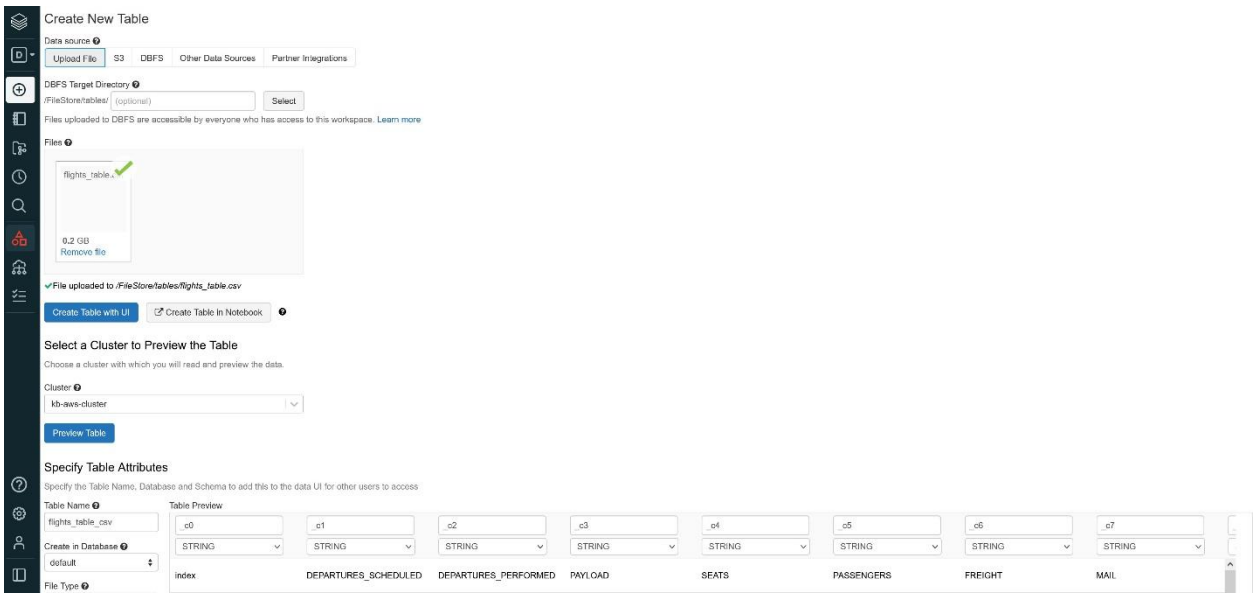
> "OperationalError: ('08S01', '[08S01] [Simba][Hardy] (115) Connection failed with error: Bad Status: Unrecognized response with no error message header. Status code: 502 (115) (SQLExecute)')"
> (See flights_upload_error_message.jpg)

Another message read:

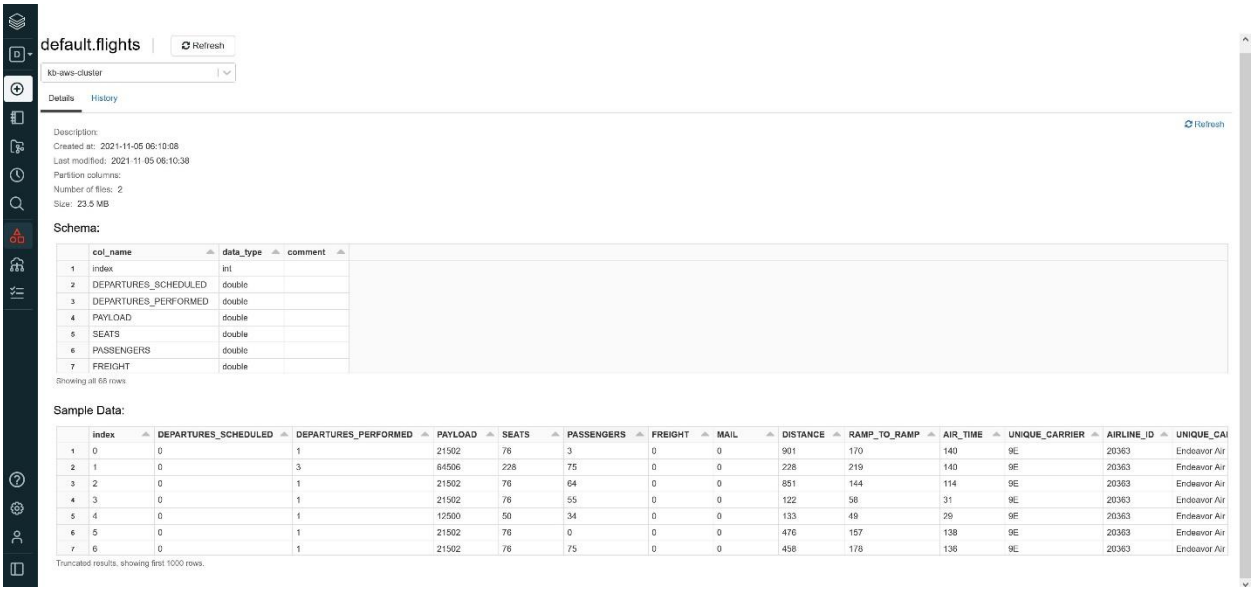
> "OperationalError: ('08S01', '[08S01] [Simba][Hardy] (115) Connection failed with error: SSL_read: error code: 0 (115) (SQLExecute)')." (See this Stack Overflow thread also).

My guess is that the table was so large that it was difficult to complete the upload without running into one error or another. Although increasing the number of rows that executemany() would process at a time may have sped up the process, I ended up running into an Out of Memory error when I increased the row count to 100,000.

Therefore, I ultimately created this table using the Databricks website instead. Within the Databricks portal, I went to Data → Tables and selected the 'Create Table' button. This allowed me to upload a .csv export of my flights table. Next, I instructed Databricks to use the first row of the uploaded file as the header row and to infer the schema from the values. (See images below.) This process was fairly straightforward, but I would still have preferred to accomplish it automatically using Python.

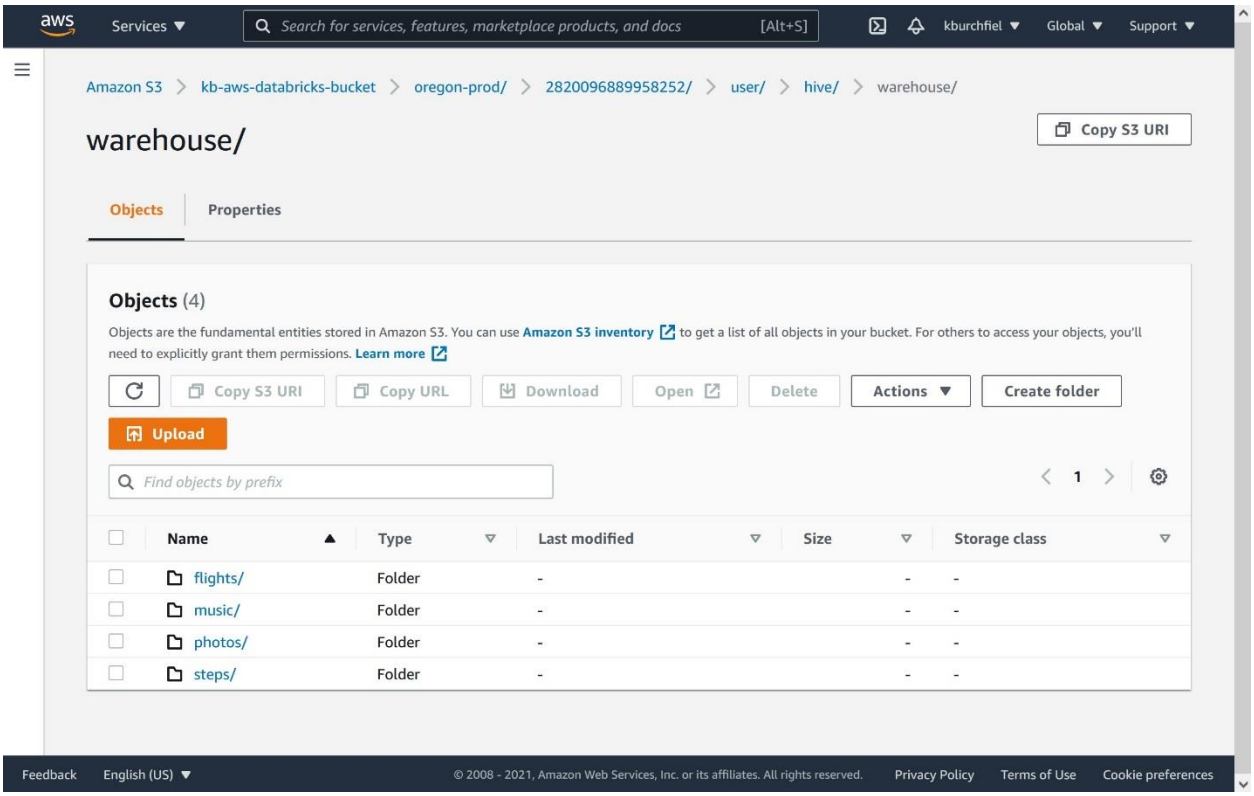


Creating a new table by uploading a .csv file

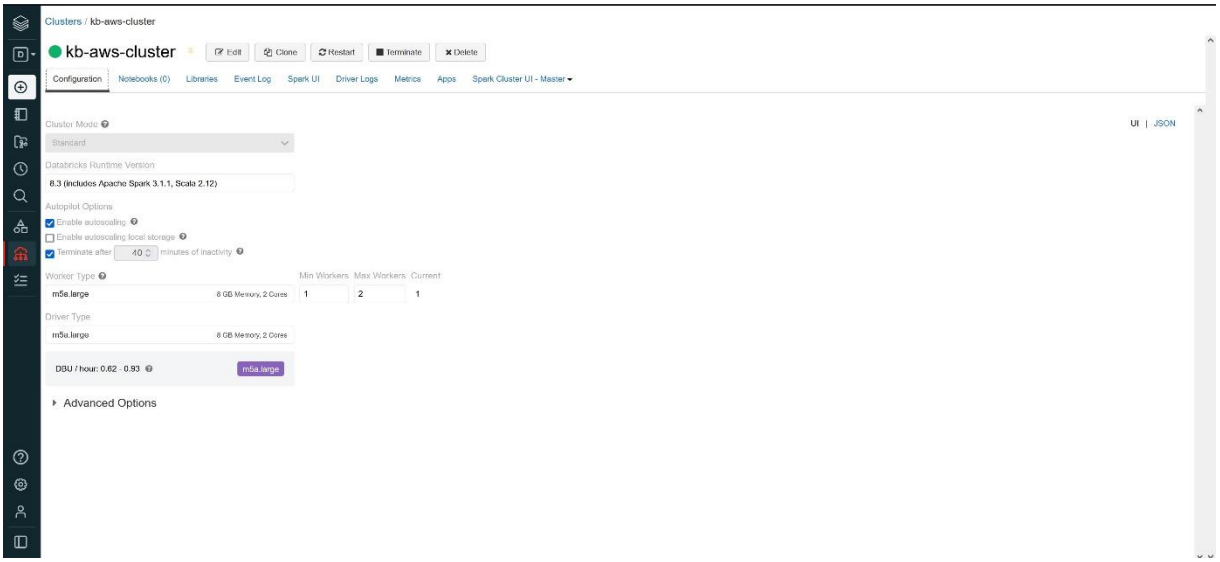


The final version of the table

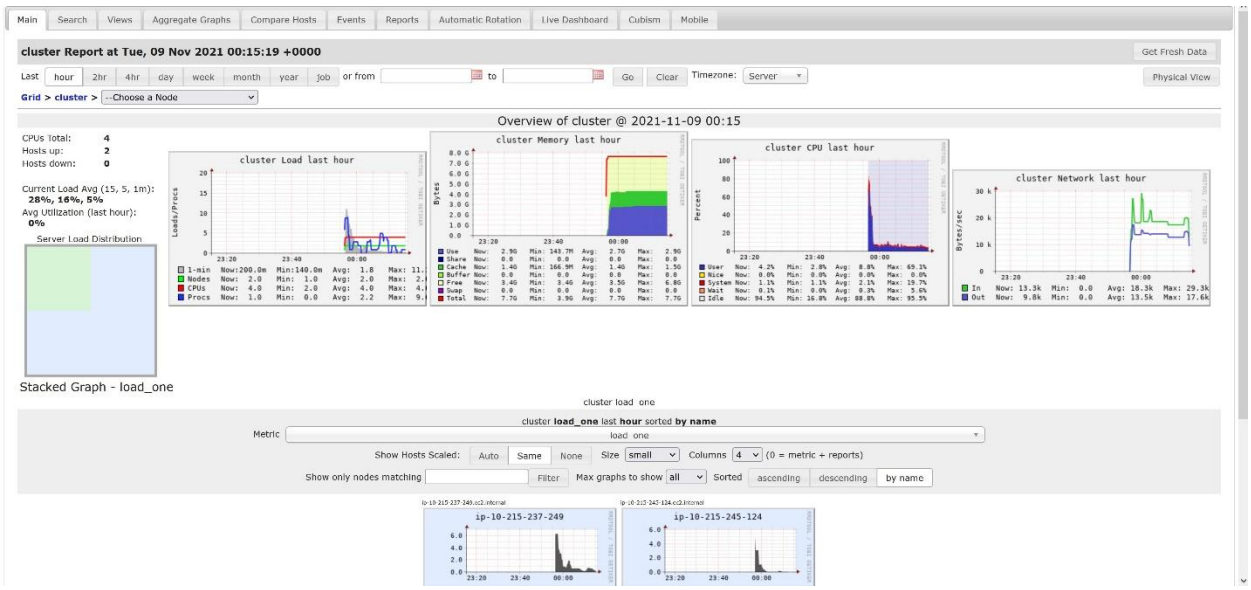
The data in my tables appeared to be stored within my Amazon S3 bucket, as shown by the image below.



Ultimately, I found the Databricks setup process to be much more complex due to my decision to use pyodbc instead of a SQLAlchemy library. Nevertheless, it was a good opportunity to learn more about pyodbc.



A look at the Databricks UI



Databricks has an extensive metrics page.

2.9. Comparing setup processes

My choice to use SQLAlchemy and Python to set up these databases influenced my experiences with the setup process. Ultimately, I found AWS, GCP, and Azure to be fairly straightforward; once I learned how to connect to one of them using PostgreSQL, connecting to the other two proved to be very similar. Setting up databases on Heroku and Snowflake required some extra steps, but both worked well with SQLAlchemy and the to_sql Pandas function.

Meanwhile, the Databricks setup process was considerably more involved, since I chose to use pyodbc rather than SQLAlchemy due to the terms of service attached to MS Build Tools for Visual Studio. Airtable made it easy to view my database data online, but there were significant limits on the amount of data that I could upload, at least using a free account.

Ultimately, if my goal was merely to create a PostgreSQL database using Python and SQLAlchemy, I would likely choose AWS, GCP, or Azure. However, I imagine that Snowflake, Heroku, Airtable, and Databricks would be compelling options for other use cases.

3. Comparing database import and query times

See my database_query_timer.ipynb script for the results of my database timing tests. Please

interpret these results with caution, however. As noted in the disclaimer within that file:

> This notebook is by no means meant as a definitive assessment of the speeds of each database provider. My results were undoubtedly influenced by my configuration settings for each database, and different configuration settings would likely result in different outcomes. In addition, the tests I used may not match real-world usage scenarios. Do not use these results to make decisions about which database provider to use.

Appendix

1. pgloader steps

An alternative to to_sql is pgloader. Documentation for pgloader can be found here, and instructions for converting a SQLite database to PostgreSQL can be found here.

I wasn't able to get pgloader to work on Windows, so I switched to using Ubuntu. Once I had Ubuntu on my computer, I installed: sudo apt-get install postgresql-client.

After opening up Ubuntu, I needed to get access to my computer's Documents folder within my C drive. This required using /mnt (as noted here.). The command I entered within Ubuntu to access my Documents folder was:

```
cd /mnt/c/Users/kburc/D1V1/Documents
```

[Note: In order to use pgloader, I placed both the database and my load file within my Documents folder because the file path pointing to my class folder has a ! in it, which confused the script.]

Next, I ran the following command:

```
pgloader load importkbdatabase2.load
```

load_kb_database.load contains the following code—which I got to work after quite a bit of trial and error (and help from the error messages):

```
load database
```

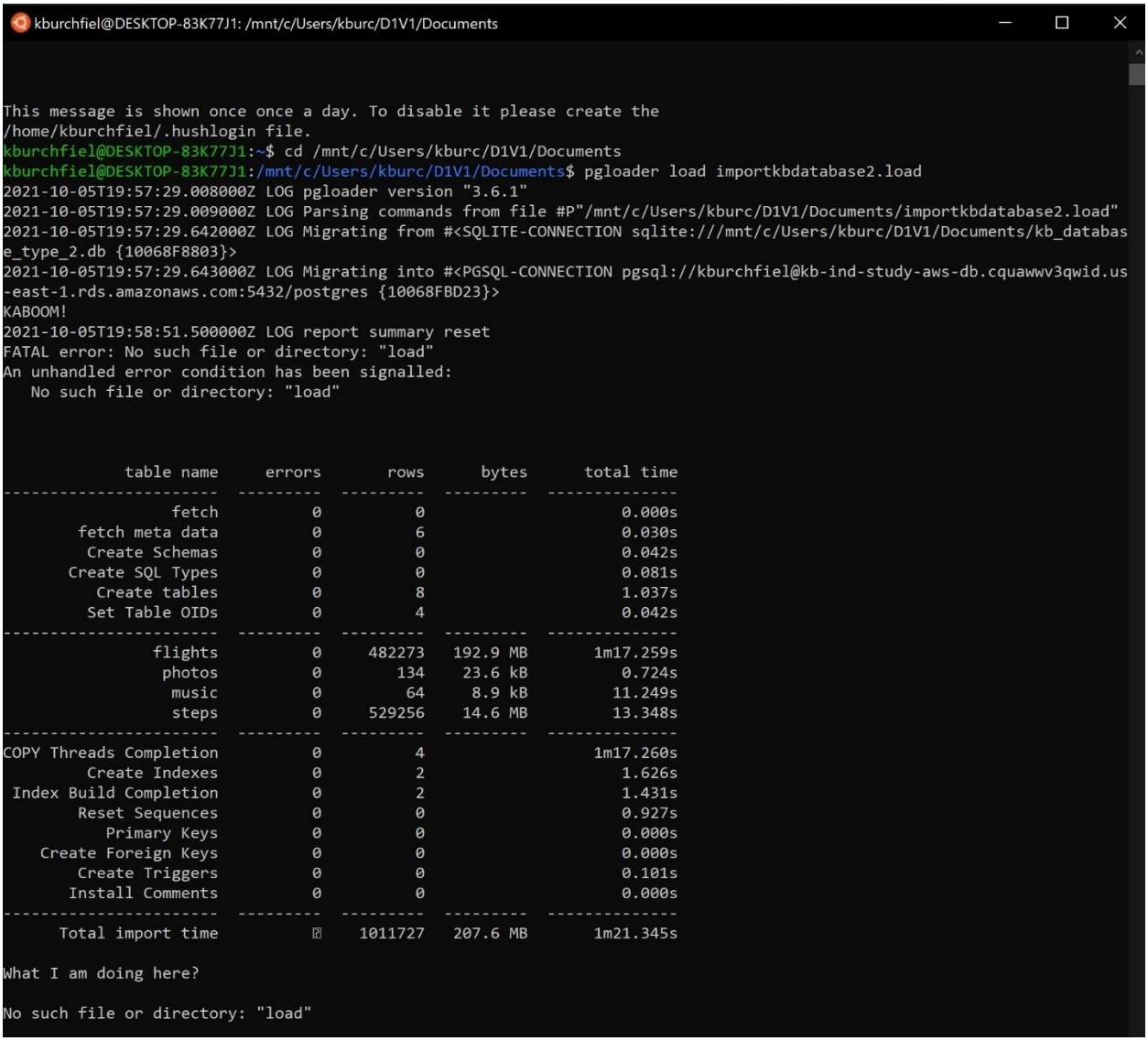
```
    from sqlite://kb_database_type_2.db
    into postgres://kburchfiel:[my password was inserted here, without brackets
surrounding it]@kb-ind-study-aws-db.cquawwv3qwid.us-east-
1.rds.amazonaws.com:5432/postgres

 with include drop, create tables, create indexes, reset sequences

  set work_mem to '16MB', maintenance_work_mem to '512 MB';
```

This answer was helpful when learning how to run a load command.

pgloader succeeded in uploading my tables to AWS, as shown below. However, I ultimately chose to use a Python script rather than the pgloader UI given that Python would require less user input once I had a script in place.


kburchfiel@DESKTOP-83K77J1: /mnt/c/Users/kburc/D1V1/Documents
This message is shown once once a day. To disable it please create the
/home/kburchfiel/.hushlogin file.
kburchfiel@DESKTOP-83K77J1:~$ cd /mnt/c/Users/kburc/D1V1/Documents
kburchfiel@DESKTOP-83K77J1:/mnt/c/Users/kburc/D1V1/Documents$ pgloader load importkbdatabase2.load
2021-10-05T19:57:29.008000Z LOG pgloader version "3.6.1"
2021-10-05T19:57:29.009000Z LOG Parsing commands from file #P"/mnt/c/Users/kburc/D1V1/Documents/importkbdatabase2.load"
2021-10-05T19:57:29.642000Z LOG Migrating from #<SQLITE-CONNECTION sqlite:///mnt/c/Users/kburc/D1V1/Documents/kb_database_type_2.db {10068F8803}>
2021-10-05T19:57:29.643000Z LOG Migrating into #<PGSQL-CONNECTION pgsql://kburchfiel@kb-ind-study-aws-db.cquawwv3qwid.us-east-1.rds.amazonaws.com:5432/postgres {10068FBD23}>
KABOOM!
2021-10-05T19:58:51.500000Z LOG report summary reset
FATAL error: No such file or directory: "load"
An unhandled error condition has been signalled:
No such file or directory: "load"
table name errors rows bytes total time
fetch 0 0 0.000s
fetch meta data 0 6 0.030s
Create Schemas 0 0 0.042s
Create SQL Types 0 0 0.081s
Create tables 0 8 1.037s
Set Table OIDs 0 4 0.042s
flights 0 482273 192.9 MB 1m17.259s
photos 0 134 23.6 kB 0.724s
music 0 64 8.9 kB 11.249s
steps 0 529256 14.6 MB 13.348s
COPY Threads Completion 0 4 1m17.260s
Create Indexes 0 2 1.626s
Index Build Completion 0 2 1.431s
Reset Sequences 0 0 0.927s
Primary Keys 0 0 0.000s
Create Foreign Keys 0 0 0.000s
Create Triggers 0 0 0.101s
Install Comments 0 0 0.000s
Total import time 1011727 207.6 MB 1m21.345s
What I am doing here?
No such file or directory: "load"