database_uploader

November 22, 2021

1 Database Uploader:

1.1 A script for importing a SQLite database into various database services using SQLAlchemy, Pandas, and other tools

Kenneth Burchfiel

This program provides scripts for importing a SQLite database into the following databases: 1. Amazon Web Services (AWS) 2. Google Cloud Platform (GCP) 3. Microsoft Azure 4. Snowflake 5. Airtable (partial import only due to space restrictions) 6. Heroku 7. Databricks

The AWS, GCP, Azure, and Heroku databases all use PostgreSQL.

[ ]:
```
import time
start_time = time.time() # Allows the program's runtime to be measured
import pandas as pd
import sqlalchemy
from pyairtable import Api, Base, Table # from https://pyairtable.readthedocs.
↪io/en/latest/api.html
import matplotlib.pyplot as plt

import pyodbc
import psycopg2
```

In order to make my database passwords more secure, I will access them by first opening a file that contains the path to my passwords folder, then opening the files within the passwords folder that store the actual passwords. Each file contains only one password. You will of course need to modify this code to point it towards your own passwords.

[ ]:
```
with open(r'C:\Users\kburc\D1V1\Documents\!
↪Dell64docs\Programming\py\kjb3_programs\key_paths\path_to_keys_folder.txt')␣
↪as file:
    path_to_keys_folder = file.readline()
```

[ ]:
```
# with open(path_to_keys_folder+'\\kb_ind_study_aws_db_pw.txt') as file:
#     aws_pw = file.readline()

with open(path_to_keys_folder+'\\kb-cheaper-aws-db_pw.txt') as file:
```

```
    aws_pw = file.readline()

with open(path_to_keys_folder+'\\kb_ind_study_gcp_db_pw.txt') as file:
    gcp_pw = file.readline()

with open(path_to_keys_folder+'\\kb_ind_study_azure_db_pw.txt') as file:
    azure_pw = file.readline()

with open(path_to_keys_folder+'\\snowflake_pw.txt') as file:
    snowflake_pw = file.readline()

with open(path_to_keys_folder+'\\airtable_api_key.txt') as file:
    airtable_key = file.readline()

with open(path_to_keys_folder+'\\heroku_db_pw.txt') as file:
    heroku_pw = file.readline()

with open(path_to_keys_folder+'\\databricks_paid_account_token.txt') as file:
    databricks_paid_account_token = file.readline()
```

The following code block specifies which databases the program should connect to, upload to, and import from. These flags allow you to save time and perhaps money as well by skipping unnecessary uploads and exports.

[ ]:
```
connect_to_sqlite = True
connect_to_aws = False
connect_to_gcp = False
connect_to_azure = False
connect_to_snowflake = False
connect_to_airtable = False
connect_to_heroku = False
connect_to_databricks = False

upload_to_aws = False
upload_to_gcp = False
upload_to_azure = False
upload_to_snowflake = False
upload_to_airtable = False
upload_to_heroku = False
upload_to_databricks = False


import_from_aws = False
import_from_gcp = False
import_from_azure = False
import_from_snowflake = False
import_from_heroku = False
```

```
import_from_databricks = False
```

1.2 Part 1: Establishing Database Connections

The code below establishes connections to each database. Code for generating the SQLite database, whose contents will be exported to the other database types in this program, can be found within the SQLite Database Builder script within this project.

In order to determine which materials to enter into the connection strings, you will need to first create a database online [although that itself may be possible through a Python script], then retrieve the credentials listed for that database.

SQLAlchemy was chosen for most of these databases because it permits the use of the to_sql() Pandas function, which greatly simplifies the database export process.

Note: Due to its size, I could not upload sqlite_database.db, the SQLite database that this script connects to, directly to GitHub. You can instead find it via Google Drive at the following link:

https://drive.google.com/file/d/1THyBZYXXT4le6zQz2SBhQlIjMk2MjhYz/view?usp=sharing

Once you have downloaded it, copy it into the data folder of your copy of the repository.

[ ]:
```
if connect_to_sqlite == True:
    sqlalchemy_sqlite_engine = sqlalchemy.create_engine('sqlite:///
↪data\\sqlite_database.db')  # Based on https://docs.sqlalchemy.org/en/13/
↪dialects/sqlite.html#connect-strings
    # This database was created within sqlite_database_builder.ipynb.


if connect_to_aws == True:
    aws_sqlalchemy_psql_engine = sqlalchemy.create_engine('postgresql://
↪kburchfiel:'+str(aws_pw)+'@kb-cheaper-aws-db.cquawwv3qwid.us-east-1.rds.
↪amazonaws.com:5432/postgres')
# Based on https://stackoverflow.com/a/58208015/13097194
# That answer can also be derived from:
# 1. https://docs.sqlalchemy.org/en/14/core/engines.html
# And: 2. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_sql.
↪html


# SQLAlchemy engine for connecting to Google Cloud Platform SQL database
# The code is based on Google code found at https://cloud.google.com/sql/docs/
↪postgres/samples/cloud-sql-postgres-sqlalchemy-create-tcp. That code is␣
↪licensed under the Apache 2.0 license.
if connect_to_gcp == True:
    gcp_sqlalchemy_psql_engine = sqlalchemy.create_engine(
        # Equivalent URL:
        # postgresql+pg8000://<db_user>:<db_pass>@<db_host>:<db_port>/<db_name>
        sqlalchemy.engine.url.URL.create(
            drivername="postgresql",
```

```
            username='kb_gcp_db',  # e.g. "my-database-user"
            password=gcp_pw,  # e.g. "my-database-password"
            host='34.135.185.218',  # e.g. "127.0.0.1"
            port=5432,  # e.g. 5432
            database='postgres'  # e.g. "my-database-name"
        )
    )

if connect_to_azure == True:
   azure_sqlalchemy_psql_engine = sqlalchemy.create_engine('postgresql://
 ↪kbindstudy:'+str(azure_pw)+'@kb-ind-study-azure-server.postgres.database.
 ↪azure.com/postgres?sslmode=require')
   # This string is based both on the Amazon connection string and the␣
 ↪'PostgreSQL connection URL' shown on the 'Connection strings' page within␣
 ↪the Azure Database site.


# I believe the following code was derived from the AWS/GCP/Azure connection␣
 ↪strings, as the format is identical.
if connect_to_heroku == True:
    heroku_sqlalchemy_psql_engine = sqlalchemy.create_engine('postgresql://
 ↪wtgddsmdmsipoo:'+str(heroku_pw)+'@ec2-23-23-199-57.compute-1.amazonaws.com:
 ↪5432/ddc9jvn58br3tc')

# The following code derived from https://docs.snowflake.com/en/user-guide/
 ↪sqlalchemy.html#connection-string-examples
if connect_to_snowflake == True:
    snowflake_engine = sqlalchemy.create_engine('snowflake://KBURCHFIEL:
 ↪'+str(snowflake_pw)+'@RV85777.east-us-2.azure/KB_SNOWFLAKE_DB/PUBLIC')



if connect_to_databricks == True:
    # Pyodbc databricks setup. Source: https://docs.databricks.com/dev-tools/
 ↪pyodbc.html#windows
    # Pyodbc was used instead of SQLAlchemy because the SQLAlchemy connector␣
 ↪required downloading Microsoft's Visual Studio Build tools, and I don't␣
 ↪think I would be able to use those tools for free in certain commercial use␣
 ↪cases. Therefore, I opted for a pyodbc setup.
    databricks_pyodbc_connection = pyodbc.connect("DSN=Databricks_Cluster",␣
 ↪autocommit=True)
    databricks_pyodbc_cursor = databricks_pyodbc_connection.cursor()
```

Although I strongly prefer using SQLAlchemy when possible due to its compatibility with the to_sql Pandas function, I also wanted to try out an alternate connection method.

The following code derives from https://docs.microsoft.com/en-us/azure/postgresql/connect-python . It demonstrates how psycopg2 can be used as an alternative to SQLAlchemy. The code works but is redundant given that SQLAlchemy already works, so I commented it out.

[ ]:
```
# # Tried connecting via pyodbc but it didn't work, so psycopg2 looks like a␣
↪better option.
# host = "kb-ind-study-azure-server.postgres.database.azure.com"
# dbname = "postgres"
# user = "kbindstudy"
# password = str(azure_pw)
# sslmode = "require"

# # Construct connection string
# conn_string = "host={0} user={1} dbname={2} password={3} sslmode={4}".
↪format(host, user, dbname, password, sslmode)
# psycopg2_azure_connection = psycopg2.connect(conn_string)
# print("Connection established")

# psycopg2_azure_cursor = psycopg2_azure_connection.cursor()
```

1.3 Part 2: Importing data from SQLite database

I will need to import data from my SQLite database in order to upload it into my other databases. First, I will generate a list of all tables within the database. Using this list instead of hard-coding the table names makes the code more portable.

[ ]:
```
sqlite_table_query = sqlalchemy_sqlite_engine.execute("Select name from␣
↪sqlite_schema where type = 'table'")
# This method for extracting all tables derived from https://www.kite.com/
↪python/answers/how-to-list-tables-using-sqlite3-in-python , although I used␣
↪sqlite_schema instead of sqlite_master . See also: https://www.sqlite.org/
↪schematab.html
sqlite_table_tuple_list = sqlite_table_query.fetchall()
sqlite_table_list = [query[0] for query in sqlite_table_tuple_list] # List␣
↪comprehension extracts only the first part of each tuple stored in␣
↪table_list.
sqlite_table_list
```

[ ]:
```
['flights', 'photos', 'steps', 'music']
```

The following code block allows you to determine which tables are already present within a PostgreSQL database.

At this point, you may wish to clear one or more databases of tables that you had previously uploaded. The following function accomplishes this task for PostgreSQL databases. However, the function is not necessary for the import process to work (since to_sql lets you replace pre-existing tables with newer versions, making a prior drop unnecessary).

```
[ ]: def drop_all_public_tables_from_db(connection):
         db_tables_query = connection.execute("Select tablename from pg_tables where␣
      ↪schemaname = 'public'") # See https://www.postgresql.org/docs/8.0/
      ↪view-pg-tables.html

         db_table_tuple_list = db_tables_query.fetchall()
         db_table_list = [query[0] for query in db_table_tuple_list]
         if len(db_table_list) == 0:
             print("No public tables to drop.")
             return
         print(len(db_table_list))
         tables_as_string = (", ").join(db_table_list) # Joining the table names␣
      ↪enables them to be dropped within a single line of SQL code.
         print("dropping tables:",tables_as_string)
         connection.execute("drop table " + tables_as_string)
```

```
[ ]: # drop_all_public_tables_from_db(connection = azure_sqlalchemy_psql_engine)
     # drop_all_public_tables_from_db(connection = gcp_sqlalchemy_psql_engine)
     # drop_all_public_tables_from_db(connection = aws_sqlalchemy_psql_engine)
```

```
4
dropping tables: flights, photos, steps, music
4
dropping tables: flights, photos, steps, music
4
dropping tables: flights, photos, steps, music
```

To test whether a particular connection worked, you can use the following line of code:

```
[ ]: # df_music_from_db = pd.read_sql("Select * from music", con = con_for_export)
     # df_music_from_db
```

The following code iterates through sqlite_table_list to read data from the SQLite database into DataFrames, then append each of these databases to a list of DataFrames (table_to_df_list).

```
[ ]: table_to_df_list = []
     for table in sqlite_table_list:
         table_to_df_list.append(pd.read_sql("Select * from "+str(table), con =␣
      ↪sqlalchemy_sqlite_engine, index_col = 'index'))
```

Here are the four tables now stored in table_to_df_list:

```
[ ]: table_to_df_list[0]
```

```
[ ]:        DEPARTURES_SCHEDULED  DEPARTURES_PERFORMED    PAYLOAD   SEATS  \
     index
     0                       0.0                   1.0    21502.0    76.0
     1                       0.0                   3.0    64506.0   228.0
     2                       0.0                   1.0    21502.0    76.0
```

```
3                          0.0                       1.0    21502.0    76.0
4                          0.0                       1.0    12500.0    50.0
…                       …                     …         …      …
482268                  1166.0                     583.0  1049400.0  5247.0
482269                  1188.0                     594.0  1069200.0  5346.0
482270                  1216.0                     608.0  1094400.0  5472.0
482271                  1258.0                     629.0  1132200.0  5661.0
482272                  2170.0                      44.0    74400.0   264.0

        PASSENGERS  FREIGHT  MAIL  DISTANCE  RAMP_TO_RAMP  AIR_TIME  …  \
index                                                                …
0              3.0      0.0   0.0     901.0         170.0     140.0  …
1             75.0      0.0   0.0     228.0         219.0     140.0  …
2             64.0      0.0   0.0     851.0         144.0     114.0  …
3             55.0      0.0   0.0     122.0          58.0      31.0  …
4             34.0      0.0   0.0     133.0          49.0      29.0  …
…              …      …     …         …           …       …  …
482268      3646.0      0.0   0.0      91.0       27284.0   21338.0  …
482269      3573.0      0.0   0.0      91.0       27799.0   21740.0  …
482270      3827.0      0.0   0.0      91.0       28454.0   22253.0  …
482271      4056.0      0.0   0.0      91.0       29437.0   23021.0  …
482272         0.0      0.0  44.0      59.0        1860.0       0.0  …

       Code_y  Plane_Group_Text Code        Plane_Config_Text  \
index
0           6     Jet, 2-Engine    1  Passenger Configuration
1           6     Jet, 2-Engine    1  Passenger Configuration
2           6     Jet, 2-Engine    1  Passenger Configuration
3           6     Jet, 2-Engine    1  Passenger Configuration
4           6     Jet, 2-Engine    1  Passenger Configuration
…       …                …  …                      …
482268      1  Piston, 2-Engine    1  Passenger Configuration
482269      1  Piston, 2-Engine    1  Passenger Configuration
482270      1  Piston, 2-Engine    1  Passenger Configuration
482271      1  Piston, 2-Engine    1  Passenger Configuration
482272      3   Helicopter/Stol    1  Passenger Configuration

       origin_iata_code origin_lat origin_lon  destination_iata_code  \
index
0                   IAD     38.944    -77.456                    FLL
1                   IAD     38.944    -77.456                    JFK
2                   IAH     29.980    -95.340                    SAV
3                   ILM     34.271    -77.903                    RDU
4                   IND     39.717    -86.294                   None
…                …        …        …                    …
482268              ACK     41.253    -70.060                    BOS
482269              BOS     42.364    -71.005                    ACK
```

```
482270              ACK      41.253    -70.060                     BOS
482271              BOS      42.364    -71.005                     ACK
482272             None         NaN        NaN                    None

        destination_lat  destination_lon
index
0                26.072          -80.153
1                40.640          -73.779
2                32.127          -81.202
3                35.877          -78.787
4                   NaN              NaN
…                   …               …
482268           42.364          -71.005
482269           41.253          -70.060
482270           42.364          -71.005
482271           41.253          -70.060
482272              NaN              NaN

[482273 rows x 62 columns]
```

[ ]:
```
table_to_df_list[1]
```

[ ]:
```
                            file_name     size              created_date  \
index
0       10086848403_b33a695758_o.jpg  1027202  Mon Sep 13 23:45:07 2021
1       10171744985_76f8973d6b_o.jpg   160300  Mon Sep 13 23:45:07 2021
2       10822424126_91be9abb6d_o.jpg   402898  Mon Sep 13 23:45:07 2021
3       10843413524_82caa8b0f8_o.jpg  7405423  Mon Sep 13 23:45:07 2021
4       11309341065_6fcfbee752_o.jpg   466609  Mon Sep 13 23:45:07 2021
…                                …        …                       …
128               S69-34316~orig.jpg  1339165  Mon Sep 13 23:45:08 2021
129               S71-41357~orig.jpg  2103357  Mon Sep 13 23:45:08 2021
130               S71-41759~orig.jpg  1470210  Mon Sep 13 23:45:08 2021
131            sts061-s-104~orig.jpg  2055089  Mon Sep 13 23:45:08 2021
132                    STSCPanel.jpg  4307564  Mon Sep 13 23:45:08 2021

                   modified_date  \
index
0       Mon Sep 13 20:44:36 2021
1       Mon Sep 13 01:42:14 2021
2       Mon Sep 13 01:42:25 2021
3       Mon Sep 13 01:32:56 2021
4       Mon Sep 13 20:38:00 2021
…                            …
128     Mon Sep 13 01:45:54 2021
129     Mon Sep 13 20:48:45 2021
130     Mon Sep 13 20:07:29 2021
```

```
131    Mon Sep 13 20:58:49 2021
132    Mon Sep 13 21:25:57 2021

                                                   gcs_url
index
0      https://storage.googleapis.com/kb_sample_datab…
1      https://storage.googleapis.com/kb_sample_datab…
2      https://storage.googleapis.com/kb_sample_datab…
3      https://storage.googleapis.com/kb_sample_datab…
4      https://storage.googleapis.com/kb_sample_datab…
…                                                    …
128    https://storage.googleapis.com/kb_sample_datab…
129    https://storage.googleapis.com/kb_sample_datab…
130    https://storage.googleapis.com/kb_sample_datab…
131    https://storage.googleapis.com/kb_sample_datab…
132    https://storage.googleapis.com/kb_sample_datab…

[133 rows x 5 columns]
```

[ ]:
```
table_to_df_list[2]
```

[ ]:
```
                           dateTime  value
index
0        2020-01-20 19:05:00.000000      0
1        2020-01-20 19:38:00.000000      0
2        2020-01-20 19:39:00.000000      0
3        2020-01-20 19:40:00.000000      0
4        2020-01-20 19:41:00.000000     61
…                                 …    …
529251   2021-09-16 18:26:00.000000      0
529252   2021-09-16 18:27:00.000000      0
529253   2021-09-16 18:28:00.000000      0
529254   2021-09-16 18:29:00.000000      0
529255   2021-09-16 18:30:00.000000      0

[529256 rows x 2 columns]
```

[ ]:
```
table_to_df_list[3]
```

[ ]:
```
      music_file_name  music_size        music_created_date  \
index
0        sample_0.mp3       94391  Thu Sep 16 15:32:20 2021
1        sample_1.mp3      114244  Thu Sep 16 15:33:02 2021
2       sample_10.mp3       73493  Thu Sep 16 15:34:14 2021
3       sample_11.mp3       94391  Thu Sep 16 15:34:18 2021
4       sample_12.mp3       64088  Thu Sep 16 15:34:22 2021
…                 …          …                        …
```

```
59      sample_62.mp3       73493  Thu Sep 16 15:38:15 2021
60      sample_63.mp3      103795  Thu Sep 16 15:38:19 2021
61       sample_7.mp3       64088  Thu Sep 16 15:34:03 2021
62       sample_8.mp3       73493  Thu Sep 16 15:34:06 2021
63       sample_9.mp3       64088  Thu Sep 16 15:34:10 2021

            music_modified_date  \
index
0      Thu Sep 16 15:32:27 2021
1      Thu Sep 16 15:40:16 2021
2      Thu Sep 16 15:34:14 2021
3      Thu Sep 16 15:34:18 2021
4      Thu Sep 16 15:34:22 2021
…                           …
59     Thu Sep 16 15:38:15 2021
60     Thu Sep 16 15:38:19 2021
61     Thu Sep 16 15:34:03 2021
62     Thu Sep 16 15:34:06 2021
63     Thu Sep 16 15:34:10 2021

                                             music_gcs_url
index
0      https://storage.googleapis.com/kb_sample_datab…
1      https://storage.googleapis.com/kb_sample_datab…
2      https://storage.googleapis.com/kb_sample_datab…
3      https://storage.googleapis.com/kb_sample_datab…
4      https://storage.googleapis.com/kb_sample_datab…
…                                                    …
59     https://storage.googleapis.com/kb_sample_datab…
60     https://storage.googleapis.com/kb_sample_datab…
61     https://storage.googleapis.com/kb_sample_datab…
62     https://storage.googleapis.com/kb_sample_datab…
63     https://storage.googleapis.com/kb_sample_datab…

[64 rows x 5 columns]
```

The following truncated versions of the steps and flights tables will be used for the Airtable import given Airtable's size restrictions.

[ ]:
```
df_steps_truncated = table_to_df_list[sqlite_table_list.index('steps')].iloc[0:
 ↪400] # .index is used to return the entry in table_to_df_list that was based␣
 ↪on the steps table. This works because DataFrames were added to␣
 ↪table_to_df_list in the same order that the tables appear in␣
 ↪sqlite_table_list.
df_steps_truncated.to_csv('data\\df_steps_1st_400_rows.csv')
```

[ ]:
```
df_flights_truncated = table_to_df_list[sqlite_table_list.index('flights')].
↪iloc[0:400]
df_flights_truncated.to_csv('data\\df_flights_1st_400_rows.csv')
```

[ ]:
```
df_steps_truncated
```

[ ]:
```
                         dateTime  value
index
0      2020-01-20 19:05:00.000000      0
1      2020-01-20 19:38:00.000000      0
2      2020-01-20 19:39:00.000000      0
3      2020-01-20 19:40:00.000000      0
4      2020-01-20 19:41:00.000000     61
…                             …      …
395    2020-01-21 06:57:00.000000     50
396    2020-01-21 06:58:00.000000     65
397    2020-01-21 06:59:00.000000     44
398    2020-01-21 07:00:00.000000     10
399    2020-01-21 07:01:00.000000     19

[400 rows x 2 columns]
```

[ ]:
```
df_flights_truncated
```

[ ]:
```
       DEPARTURES_SCHEDULED  DEPARTURES_PERFORMED   PAYLOAD  SEATS  \
index
0                       0.0                   1.0   21502.0   76.0
1                       0.0                   3.0   64506.0  228.0
2                       0.0                   1.0   21502.0   76.0
3                       0.0                   1.0   21502.0   76.0
4                       0.0                   1.0   12500.0   50.0
…                     …                     …       …      …
395                     0.0                   1.0   35000.0  155.0
396                     0.0                   1.0   35000.0  155.0
397                     0.0                   4.0  140000.0  620.0
398                     0.0                   1.0   35000.0  155.0
399                     0.0                   4.0  140000.0  620.0

       PASSENGERS  FREIGHT  MAIL  DISTANCE  RAMP_TO_RAMP  AIR_TIME  …  \
index                                                               …
0             3.0      0.0   0.0     901.0         170.0     140.0  …
1            75.0      0.0   0.0     228.0         219.0     140.0  …
2            64.0      0.0   0.0     851.0         144.0     114.0  …
3            55.0      0.0   0.0     122.0          58.0      31.0  …
4            34.0      0.0   0.0     133.0          49.0      29.0  …
…             …       …     …        …             …         …  …
395          35.0      0.0   0.0    1304.0         250.0     215.0  …
```

```
396           0.0      0.0   0.0    1679.0         212.0     199.0  …
397           0.0      0.0   0.0    1224.0         649.0     585.0  …
398           0.0      0.0   0.0     308.0          73.0      49.0  …
399           0.0      0.0   0.0    1067.0         646.0     567.0  …

      Code_y  Plane_Group_Text Code        Plane_Config_Text origin_iata_code  \
index
0          6     Jet, 2-Engine    1  Passenger Configuration              IAD
1          6     Jet, 2-Engine    1  Passenger Configuration              IAD
2          6     Jet, 2-Engine    1  Passenger Configuration              IAH
3          6     Jet, 2-Engine    1  Passenger Configuration              ILM
4          6     Jet, 2-Engine    1  Passenger Configuration              IND
…      …               …  …                      …              …
395        6     Jet, 2-Engine    1  Passenger Configuration              BDL
396        6     Jet, 2-Engine    1  Passenger Configuration              BRO
397        6     Jet, 2-Engine    1  Passenger Configuration              BRO
398        6     Jet, 2-Engine    1  Passenger Configuration              BRO
399        6     Jet, 2-Engine    1  Passenger Configuration              BRO

      origin_lat origin_lon  destination_iata_code  destination_lat  \
index
0         38.944    -77.456                    FLL           26.072
1         38.944    -77.456                    JFK           40.640
2         29.980    -95.340                    SAV           32.127
3         34.271    -77.903                    RDU           35.877
4         39.717    -86.294                   None              NaN
…          …         …                    …              …
395       41.939    -72.683                    TUL           36.198
396       25.907    -97.426                    EWR           40.692
397       25.907    -97.426                   None              NaN
398       25.907    -97.426                    IAH           29.980
399       25.907    -97.426                    MIA           25.793

       destination_lon
index
0              -80.153
1              -73.779
2              -81.202
3              -78.787
4                  NaN
…                …
395            -95.888
396            -74.169
397                NaN
398            -95.340
399            -80.291
```

```
[400 rows x 62 columns]
```

2 Step 2: Use SQLAlchemy to write data in table_to_df_list to other database types

Using a series of functions (defined below), I can now upload the SQLite tables (in DataFrame form) to each database. This process is simple for databases that support SQLAlchemy, but more complex for databases that do not.

The function below, upload_to_database, uses the to_sql Pandas function to upload each table stored in table_to_df_list to a given database; the name for each table is sourced from sqlite_table_list. Tables already present in the databases are replaced. The function also outputs the length of time needed to upload all tables to the database.

upload_to_database is compatible with databases that support SQLAlchemy. Within this program, it can be used with the AWS, GCP, and Azure database connections created above, and likely with the Heroku connection as well.

```
[ ]: def upload_to_database(table_to_df_list, con_for_export):
         print("Uploading tables to database using the following connection:
      ↪",con_for_export)
         upload_start_time = time.time()
         for i in range(len(table_to_df_list)):
             table_to_df_list[i].to_sql(sqlite_table_list[i], con = con_for_export,␣
      ↪if_exists = 'replace')
             # if_exists = 'replace' is meant to overwrite an older version of a␣
      ↪table with the newer version. The storage logs on at least one database␣
      ↪provider indicated that running this function caused an increase in my␣
      ↪database size despite this replacement clause. However, this may have been␣
      ↪due to other factors.
         upload_end_time = time.time()
         upload_run_time = upload_end_time - upload_start_time
         upload_run_minutes = upload_run_time // 60
         upload_run_seconds = upload_run_time % 60
         print("Completed upload at",time.ctime(upload_end_time),"(local time)")
         print("Total run time:",'{:.2f}'.format(upload_run_time),"second(s)␣
      ↪("+str(upload_run_minutes),"minute(s) and",'{:.2f}'.
      ↪format(upload_run_seconds),"second(s))")

     # Amount of time needed to run this set of code (on my apartment's Ethernet␣
      ↪connection):
     # AWS: 225.9s
     # GCP: 325.1s
     # Azure: 269.8s
```

```
# It took 198.3 seconds when uploading an older but similar version of my␣
↪database to Azure, but I was uploading the tables at the Butler Library at␣
↪Columbia, where the WiFi
# upload speed is faster than at home.)
```

The Snowflake import process was somewhat more involved due to the need to convert all columns to uppercase form.

```
[ ]: def upload_to_snowflake(table_to_df_list):
         print("Uploading tables to Snowflake")
         upload_start_time = time.time()
         con_for_export = snowflake_engine
         from snowflake.connector.pandas_tools import pd_writer
         for i in range(len(table_to_df_list)):
         # From https://docs.snowflake.com/en/user-guide/python-connector-api.html
         # pd_writer is necessary to include as the method
             table_to_df_list[i].columns = list(map(str.upper, table_to_df_list[i].
     ↪columns))
             # Columns need to be converted to uppercase so that they will work with␣
     ↪Snowflake.
             # This method comes from CodinginCircles at https://stackoverflow.com/a/
     ↪63404628/13097194
             table_to_df_list[i].to_sql(sqlite_table_list[i], con = con_for_export,␣
     ↪if_exists = 'replace', index = False, method = pd_writer)
         # Had to set index to False in order to avoid an error message explaining␣
     ↪that Snowflake doesn't support indices
         upload_end_time = time.time()
         upload_run_time = upload_end_time - upload_start_time
         upload_run_minutes = upload_run_time // 60
         upload_run_seconds = upload_run_time % 60
         print("Completed upload at",time.ctime(upload_end_time),"(local time)")
         print("Total run time:",'{:.2f}'.format(upload_run_time),"second(s)␣
     ↪("+str(upload_run_minutes),"minute(s) and",'{:.2f}'.
     ↪format(upload_run_seconds),"second(s))")
```

The upload_to_airtable() function below is more complex than upload_to_databas() due to the use of batch_create in place of to_sql. The code only imports the music and flights tables from the SQLite database; the other two tables (in truncated form due to Airtable's capacity restrictions) were uploaded using the web interface, which may also have been the best way to upload the music and flights tables.

```
[ ]: # The Airtable API documentation (https://pyairtable.readthedocs.io/en/latest/
     ↪api.html#pyairtable.api.Api.batch_create) proved helpful in writing the␣
     ↪following function.

     def upload_to_airtable():
         airtable_base_id = 'appoqaUn0vZ1TeP31' # Found via link on API page
```

```
    # that I accessed from my Account page

    music_airtable = Table(airtable_key, airtable_base_id, 'music')
    df_music_as_strings = table_to_df_list[sqlite_table_list.index('music')].
 ↪copy().astype('str')
# This change prevents an error from occurring during the Airtable upload␣
 ↪process.
# music_size is originally in int64 format, so I converted it to string format␣
 ↪beforehand. The above code converts
# all columns to strings, but these can be converted back to other types as
# needed.
    df_music_as_dict_list = []

    for i in range(len(df_music_as_strings)):
        df_music_as_dict_list.append(df_music_as_strings.iloc[i].to_dict())
    music_airtable.batch_create(df_music_as_dict_list, typecast=True)


    photos_airtable = Table(airtable_key, airtable_base_id, 'photos')
    df_photos_as_dict_list = []

    df_photos_as_strings = table_to_df_list[sqlite_table_list.index('photos')].
 ↪copy().astype('str')
    for i in range(len(df_photos_as_strings)):
        df_photos_as_dict_list.append(df_photos_as_strings.iloc[i].to_dict())
    photos_airtable.batch_create(df_photos_as_dict_list, typecast=True)
```

The upload_to_databricks function uses pyodbc instead of SQLAlchemy. Although I was able to upload the photos, music, and steps tables to Databricks using the function, I ran into connectivity issues when uploading the flights table; as a result, I exported the flights table to a .csv file and uploaded it via the web interface instead. This method could also have been used for the other tables, of course.

```
[ ]: def upload_to_databricks():
    databricks_tables_query = databricks_pyodbc_cursor.execute("show tables in␣
 ↪default").fetchall() # Shows all tables currently within the Databricks␣
 ↪database. https://spark.apache.org/docs/3.0.0-preview/
 ↪sql-ref-syntax-aux-show-tables.html
    databricks_table_list = [row[1] for row in databricks_tables_query] #␣
 ↪databricks_tables_query returns a tuple for each table in the database. This␣
 ↪list comprehension adds the 2nd entry within that row (containing the table␣
 ↪name) into a new list.


    if 'photos' in databricks_table_list: # Checks whether the table has␣
 ↪already been created, and drops it if so
        databricks_pyodbc_cursor.execute("Drop table photos")
```

```
    databricks_pyodbc_cursor.execute("CREATE TABLE photos(file_name string, size float, created_date string, modified_date string, gcs_url string);") # Took 47 seconds

    databricks_pyodbc_cursor.fast_executemany = True
    databricks_pyodbc_cursor.executemany("insert into photos(file_name, size, created_date, modified_date, gcs_url) values (?, ?, ?, ?, ?)", table_to_df_list[sqlite_table_list.index('photos')].values.tolist())
    # The above lines are based on https://github.com/mkleehammer/pyodbc/wiki/Cursor#executemanysql-params-with-fast_executemanytrue
    # The idea of using .values.tolist() to convert a Pandas DataFrame into a sequence that would work with executemany came from ansen at https://stackoverflow.com/a/30185727/13097194
    # An alternative to tolist() would be to use .iloc or .loc to locate and add each value individually, but .values.tolist() is much simpler.
    # index('photos') retrieves the index number corresponding to the location of the 'photos' table within table_to_df_list.

    if 'music' in databricks_table_list(): # Checks whether the table has already been created, and drops it if so
        databricks_pyodbc_cursor.execute("Drop table music")

    databricks_pyodbc_cursor.execute("CREATE TABLE music(music_file_name string, music_size float, music_created_date string, music_modified_date string, music_gcs_url string);") # Took 47 seconds
    databricks_pyodbc_cursor.fast_executemany = True
    databricks_pyodbc_cursor.executemany("insert into music(music_file_name, music_size, music_created_date, music_modified_date, music_gcs_url) values (?, ?, ?, ?, ?)", table_to_df_list[sqlite_table_list.index('music')].values.tolist())

    if 'steps' in databricks_table_list(): # Checks whether the table has already been created, and drops it if so
        databricks_pyodbc_cursor.execute("Drop table steps")

    databricks_pyodbc_cursor.execute("CREATE TABLE steps(dateTime string, value int);")
    for i in range(0, 500000, 100000): # This for loop allows rows to be uploaded in chunks. This method was chosen because trying to upload all 529,256 rows at once from the steps table resulted in errors.
        databricks_pyodbc_cursor.fast_executemany = True
        start_time = time.time()
        print("Now uploading rows",i,"to",i+99999) # i+100000 won't be added during this iteration of the loop due to how slice notation works in Python.
```

```
        databricks_pyodbc_cursor.executemany("insert into steps(dateTime,␣
↪value) values (?, ?)", table_to_df_list[sqlite_table_list.index('steps')].
↪iloc[i:i+100000].values.tolist())
        end_time = time.time()
        print("Time (in seconds) to upload this set of rows:
↪",end_time-start_time)
    print("Now uploading rows 500,000 to 529,255:")
    databricks_pyodbc_cursor.executemany("insert into steps(dateTime, value)␣
↪values (?, ?)", table_to_df_list[sqlite_table_list.index('steps')].
↪iloc[500000:529256].values.tolist())

    # The steps upload code took 6m55.9s to run when uploading 10000 rows at a␣
↪time, but only 2m2.9 s when uploading 100,000 rows at a time. (This was␣
↪partly because the code started uploading rows earlier within the 100,000␣
↪row condition, which I imagine was just due to random chance.)
```

Creating code to upload the flights table required more preparatory work. Unfortunately, due to connection errors that arose during the upload process, I was not able to run this code successfully. However, it could theoretically work in the absence of connectivity problems, so I have included it in commented form.

```
[ ]: # # Because there are 62 columns within the flights table, I did not want to␣
↪manually type out the column names and types when defining the tables, nor␣
↪did I want to type exactly 62 question marks as part of the table population␣
↪process. Instead, I used code to generate variables storing these strings,␣
↪then inserted those variables into the SQL queries.

# databricks_tables_query = databricks_pyodbc_cursor.execute("show tables in␣
↪default").fetchall() # Shows all tables currently within the Databricks␣
↪database. https://spark.apache.org/docs/3.0.0-preview/
↪sql-ref-syntax-aux-show-tables.html
# databricks_table_list = [row[1] for row in databricks_tables_query] #␣
↪databricks_tables_query returns a tuple for each table in the database. This␣
↪list comprehension adds the 2nd entry within that row (containing the table␣
↪name) into a new list.


# string_for_flights_table_definition = " string, ".
↪join(table_to_df_list[sqlite_table_list.index('flights')].columns)
# string_for_flights_table_definition+= ' string'
# print(string_for_flights_table_definition)

# question_marks_for_flights_table_population = '?' + (', ?' * 61)
# print(question_marks_for_flights_table_population)

# string_for_flights_table_population = ", ".
↪join(table_to_df_list[sqlite_table_list.index('flights')].columns)
```

```
# print(string_for_flights_table_population)

# # The following code was meant to create and populate the flights table, but␣
 ↪I encountered multiple connection errors when trying to run it; therefore, I␣
 ↪instead imported this table using the Databricks web UI. I originally tried␣
 ↪to use executemany to add 100,000 rows at a time, but received an Out of␣
 ↪Memory error (likely because there were 62 columns per row). Therefore, I␣
 ↪modified the code to add only 10,000 rows at a time. I later increased it to␣
 ↪50,000 rows. Regardless of the setting I chose, however, I continued to␣
 ↪encounter error messages that appeared related to the connection to␣
 ↪Databricks. More debugging may have resolved these issues.

# if 'flights' in databricks_table_list: # Checks whether the database has␣
 ↪already been created
#     databricks_pyodbc_cursor.execute("Drop table flights")
# databricks_pyodbc_cursor.execute("CREATE TABLE␣
 ↪flights("+string_for_flights_table_definition+");")

# df_flights_as_strings_to_list = table_to_df_list[sqlite_table_list.
 ↪index('flights')].astype('str').values.tolist() # Changing the DataFrame␣
 ↪columns to string types and converting the output to a list beforehand may␣
 ↪save time.
# # The idea of using tolist() to convert a Pandas DataFrame into a sequence␣
 ↪that would work with executemany came from ansen at https://stackoverflow.
 ↪com/a/30185727/13097194
# # Changed the type of each column to a string in order to avoid type␣
 ↪compatibility errors


# for i in range(0, 450000, 50000):
#     databricks_pyodbc_cursor.fast_executemany = True
#     start_time = time.time()
#     print("Now uploading rows",i,"to",i+49999)
#     databricks_pyodbc_cursor.executemany("insert into␣
 ↪flights("+string_for_flights_table_population+") values␣
 ↪("+question_marks_for_flights_table_population+")",␣
 ↪df_flights_as_strings_to_list[0:50000])
#     end_time = time.time()
#     print("Time (in seconds) to upload this set of rows:",end_time-start_time)
# print("Now uploading rows 450,000 to 482,273:")
# databricks_pyodbc_cursor.executemany("insert into␣
 ↪flights("+string_for_flights_table_population+") values␣
 ↪("+question_marks_for_flights_table_population+")",␣
 ↪df_flights_as_strings_to_list[450000:482274])
```

```
# # I received an error message that appeared to relate to a connection issue,␣
↪so I restarted both my Jupyter Notebook kernel and the Databricks cluster. I␣
↪also changed the Databricks timeout time from 40 minutes to 400 minutes so␣
↪that a timeout wouldn't occur during this operation.
```

```
[ ]: # table_to_df_list[sqlite_table_list.index('flights')].to_csv('flights_table.
↪csv') # Exports the flights table to .csv format so that it can be uploaded␣
↪to Databricks using the web UI. Alternately, you can upload the␣
↪routes_planes_coordinates.csv file to Databricks, as this file was the␣
↪original source of the flights table.
```

The following block of code shows how, when adding tables into Databricks, a for loop could be used in place of executemany(). However, this code took quite a while to run; hence, executemany() appears to be the better option (and also a simpler one). For example, this code took 257.2 seconds to execute compared to only 14.1 when fast_executemany was set to True and executemany was used.

```
[ ]: # databricks_pyodbc_cursor.execute("Drop table music")

# if 'music' not in databricks_table_list(): # Checks whether the database has␣
↪already been created
#     databricks_pyodbc_cursor.execute("CREATE TABLE music(music_file_name␣
↪string, music_size float, music_created_date string, music_modified_date␣
↪string, music_gcs_url string);") # Took 47 seconds

#     for i in range(len(table_to_df_list[sqlite_table_list.index('music')])):
#         databricks_pyodbc_cursor.execute("insert into music(music_file_name,␣
↪music_size, music_created_date, music_modified_date, music_gcs_url) values (?
↪, ?, ?, ?, ?)", table_to_df_list[sqlite_table_list.index('music')].iloc[i].
↪astype(str).tolist())
#         # The above line is based on https://github.com/mkleehammer/pyodbc/
↪wiki/Cursor
#         # 'astype(str)' was added in to avoid an error related to int64␣
↪values. See https://stackoverflow.com/a/68504686/13097194

# else:
#     print("Table already present within database. Drop the table in order to␣
↪update it.")
```

The following block of code calls the above upload functions for all databases whose upload flags (defined earlier in the code) are set to True.

```
[ ]: if upload_to_aws == True:
    print("\n Uploading tables to AWS")
    upload_to_database(table_to_df_list = table_to_df_list, con_for_export =␣
↪aws_sqlalchemy_psql_engine)

if upload_to_gcp == True:
```

```
    print("\n Uploading tables to GCP")
    upload_to_database(table_to_df_list = table_to_df_list, con_for_export =␣
 ↪gcp_sqlalchemy_psql_engine)

if upload_to_azure == True:
    print("\n Uploading tables to Azure")
    upload_to_database(table_to_df_list = table_to_df_list, con_for_export =␣
 ↪azure_sqlalchemy_psql_engine)

if upload_to_heroku == True:
    print("\n Uploading tables to Heroku")
    upload_to_database(table_to_df_list = table_to_df_list, con_for_export =␣
 ↪heroku_sqlalchemy_psql_engine)

if upload_to_snowflake == True:
    print("\n Uploading tables to Snowflake")
    upload_to_snowflake(table_to_df_list = table_to_df_list)

if upload_to_airtable == True:
    print("\n Uploading tables to Airtable")
    upload_to_airtable()

if upload_to_databricks == True:
    print("\n Uploading tables to Databricks")
    upload_to_databricks()
```

```
 Uploading tables to AWS
Uploading tables to database using the following connection:
Engine(postgresql://kburchfiel:***@kb-cheaper-aws-db.cquawwv3qwid.us-
east-1.rds.amazonaws.com:5432/postgres)
Completed upload at Mon Nov  8 21:50:57 2021 (local time)
Total run time: 251.78 second(s) (4.0 minute(s) and 11.78 second(s))

 Uploading tables to GCP
Uploading tables to database using the following connection:
Engine(postgresql://kb_gcp_db:***@34.135.185.218:5432/postgres)
Completed upload at Mon Nov  8 21:55:39 2021 (local time)
Total run time: 281.70 second(s) (4.0 minute(s) and 41.70 second(s))

 Uploading tables to Azure
Uploading tables to database using the following connection:
Engine(postgresql://kbindstudy:***@kb-ind-study-azure-
server.postgres.database.azure.com/postgres?sslmode=require)
Completed upload at Mon Nov  8 22:00:26 2021 (local time)
Total run time: 286.63 second(s) (4.0 minute(s) and 46.63 second(s))
```

When I ran upload_to_database for AWS, GCP, and Azure back-to-back, I obtained the following upload times:

AWS: 247.55s (4m7.55s) GCP: 282.12s (4m42.12s) Azure: 293.15s (4m53.15s)

The flights, steps, music, and photos tables did not exist in these databases prior to this upload, but I obtained similar times when pre-existing copies of the tables were already present.

2.1 Examples of alternate import strategies using psycopg2

The two functions below show two ways of using psycopg2 to upload the flights table to the Azure database. My preference is to use to_sql instead given its simplicity, but a demonstration of psycopg2's functionality may still be helpful.

[ ]:
```
def alt_azure_flights_import():

# Because there are 62 columns within the flights table, I did not want to␣
↪manually type out the column names and types when defining the tables, nor␣
↪did I want to type exactly 62 question marks as part of the table population␣
↪process. Instead, I used code to generate variables storing these strings,␣
↪then inserted those variables into the ensuing SQL queries.

    string_for_flights_table_definition = " text, ".
↪join(table_to_df_list[sqlite_table_list.index('flights')].columns)
    string_for_flights_table_definition+= ' text'
    print(string_for_flights_table_definition)

    question_marks_for_flights_table_population = '%s' + (', %s' * 61)
    print(question_marks_for_flights_table_population)

    string_for_flights_table_population = ", ".
↪join(table_to_df_list[sqlite_table_list.index('flights')].columns)
    print(string_for_flights_table_population)

    azure_tables_query = pd.read_sql("Select tablename from pg_tables where␣
↪schemaname = 'public'", con = psycopg2_azure_connection) # From https://
↪stackoverflow.com/a/24462829/13097194 and https://www.postgresql.org/docs/
↪current/infoschema-tables.html
    azure_table_list = azure_tables_query['tablename'].tolist()
    print("Current list of tables:")
    print(azure_table_list)
    if 'flights_psycopg2' in azure_table_list: # Checks whether the database␣
↪has already been created
        psycopg2_azure_cursor.execute("Drop table flights_psycopg2;")
        psycopg2_azure_connection.commit()
    psycopg2_azure_cursor.execute("CREATE TABLE␣
↪flights_psycopg2("+string_for_flights_table_definition+");")
```

```
    df_flights_as_strings_to_list = table_to_df_list[sqlite_table_list.
↪index('flights')].astype('str').values.tolist()
    # The idea of using tolist() to convert a Pandas DataFrame into a sequence␣
↪that would work with executemany came from ansen at https://stackoverflow.
↪com/a/30185727/13097194
    # Changing the DataFrame columns to string types (in order to avoid type␣
↪compatibility errors) and converting the output to a list beforehand should␣
↪save time compared to making this change each time the following for loop␣
↪runs.


    new_start_time = time.time()
    for i in range(len(df_flights_as_strings_to_list)):
        psycopg2_azure_cursor.execute("insert into␣
↪flights_psycopg2("+string_for_flights_table_population+") values␣
↪("+question_marks_for_flights_table_population+")",␣
↪df_flights_as_strings_to_list[i])
        # The above line is based on https://www.psycopg.org/docs/usage.html .␣
↪Note that variables were used in place of hard-coded table columns, which␣
↪saved me the trouble of having to write out those tables manually.

        # The timing script below allows the function's progress to be tracked.
        if i % 1000 == 0:
            new_end_time = time.time()
            print("Uploaded",i,"rows so far.")
            print("Time (in seconds) elapsed so far:
↪",new_end_time-new_start_time)

    psycopg2_azure_connection.commit() # Necessary in order to keep the newly␣
↪created table within the database after the program ends

    # This code ended up taking 230 minutes (3h50m) and 46.2 seconds to run!␣
↪During a subsequent run, it took 205m (3h25m) and 6.5 seconds. Clearly, this␣
↪isn't the most efficient way to upload tables using psycopg2.
```

The following function runs considerably faster than the previous one, as it uses execute_batch rather than execute. See the above function for additional comments and citations.

[ ]:
```
def alt_azure_flights_import_execute_batch():
    string_for_flights_table_definition = " text, ".
↪join(table_to_df_list[sqlite_table_list.index('flights')].columns)
    string_for_flights_table_definition+= ' text'
    print(string_for_flights_table_definition)

    question_marks_for_flights_table_population = '%s' + (', %s' * 61)
    print(question_marks_for_flights_table_population)
```

```
    string_for_flights_table_population = ", ".
↪join(table_to_df_list[sqlite_table_list.index('flights')].columns)
    print(string_for_flights_table_population)
    print("Creating list of tables already present in the database:")
    azure_tables_query = pd.read_sql("Select tablename from pg_tables where␣
↪schemaname = 'public'", con = psycopg2_azure_connection) # From https://
↪stackoverflow.com/a/24462829/13097194 and https://www.postgresql.org/docs/
↪current/infoschema-tables.html
    azure_table_list = azure_tables_query['tablename'].tolist()
    print("Current list of tables:")
    print(azure_table_list)
    print("Checking whether flights_psycopg2 is already in the list:")

    if 'flights_psycopg2' in azure_table_list: # Checks whether the database␣
↪has already been created
        print("Dropping pre-existing copy of flights_psycopg2 from database:")
        psycopg2_azure_cursor.execute("Drop table flights_psycopg2;")
        # Adding a semicolon here may have been necessary
        # in order to prevent the code from hanging.
        psycopg2_azure_connection.commit()
    print("Creating new copy of flights_psycopg2:")
    psycopg2_azure_cursor.execute("CREATE TABLE␣
↪flights_psycopg2("+string_for_flights_table_definition+");")

    print("converting table to list")
    df_flights_as_strings_to_list = table_to_df_list[sqlite_table_list.
↪index('flights')].astype('str').values.tolist()

    new_start_time = time.time()
    print("Now uploading rows")
    psycopg2.extras.execute_batch(psycopg2_azure_cursor, "insert into␣
↪flights_psycopg2("+string_for_flights_table_population+") values␣
↪("+question_marks_for_flights_table_population+")",␣
↪df_flights_as_strings_to_list, page_size = 1000) # See https://www.psycopg.
↪org/docs/extras.html#fast-exec
    new_end_time = time.time()
    print("Time (in seconds) elapsed:",new_end_time-new_start_time)

    psycopg2_azure_connection.commit()

    # This block of code took 6 minutes and 59.1 seconds to run with page_size␣
↪set to 10000, and 7 minutes and 15.4 seconds to run with page_size set to␣
↪1000. (Some other changes were made to this cell in between the two runs.)␣
↪Meanwhile, using to_sql to upload just the flights table to the Azure␣
↪database took 4 minutes and 5.8 seconds.
```

The following commented-out lines can be used to test out these functions.

```
[ ]: # psycopg2_azure_cursor.execute("Drop table flights_psycopg2;")
     # psycopg2_azure_connection.commit()
```

```
[ ]: # azure_tables_query = pd.read_sql("Select tablename from pg_tables where␣
      ↪schemaname = 'public'", con = psycopg2_azure_connection) # From https://
      ↪stackoverflow.com/a/24462829/13097194 and https://www.postgresql.org/docs/
      ↪current/infoschema-tables.html
     # azure_table_list = azure_tables_query['tablename'].tolist()
     # print("Current list of tables:")
     # print(azure_table_list)
```

```
[ ]: # alt_azure_flights_import()
```

```
[ ]: # psycopg2_azure_cursor.execute("drop table flights_psycopg2")
     # psycopg2_azure_connection.commit()
```

```
[ ]: # test_of_psycopg2 = pd.read_sql("Select * from flights_psycopg2",␣
      ↪con=psycopg2_azure_connection)
     # test_of_psycopg2
```

3 Step 3: Test out database imports

Finally, the following code blocks make it possible to test whether data was successfully uploaded to 6 of the above databases. Because I found Airtable to be limited as a database host, I did not create an import function for it. If you choose to create one, consider applying pyAirtable's .all() function: https://pyairtable.readthedocs.io/en/latest/api.html

For comparisons of the time needed to import these databases and execute other queries on them, see the Database Query Timer notebook.

```
[ ]: def read_from_db(table_list, con_for_import):
         print("Importing tables from database using the following connection:
      ↪",con_for_import)
         import_start_time = time.time()
         imported_table_df_list = []
         for table in table_list:
             imported_df = pd.read_sql("select * from "+table+";",␣
      ↪con=con_for_import)
             if imported_df.columns[0] == 'index': # If the table already has an␣
      ↪index
                 # column, set that column as the index.
                 imported_df.set_index('index',inplace=True)
             else: # Otherwise, if the index column is unnamed, give it the name␣
      ↪'index.'
                 imported_df.index.name = 'index'
             imported_table_df_list.append(imported_df)
         import_end_time = time.time()
```

```
    import_run_time = import_end_time - import_start_time
    import_run_minutes = import_run_time // 60
    import_run_seconds = import_run_time % 60
    print("Completed import at",time.ctime(import_end_time),"(local time)")
    print("Total run time:",'{:.2f}'.format(import_run_time),"second(s)␣
↪("+str(import_run_minutes),"minute(s) and",'{:.2f}'.
↪format(import_run_seconds),"second(s))")
    return (imported_table_df_list, import_run_time)
    # These components could also be returned as a list or class.
```

For each database, read_from_db will only be called if the database's import flag is set to True.

[ ]:
```
if import_from_aws == True:
    aws_imported_table_list, aws_run_time = read_from_db(sqlite_table_list,␣
↪con_for_import = aws_sqlalchemy_psql_engine)

if import_from_gcp == True:
    gcp_imported_table_list, gcp_run_time = read_from_db(sqlite_table_list,␣
↪con_for_import = gcp_sqlalchemy_psql_engine)

if import_from_azure == True:
    azure_imported_table_list, azure_run_time = read_from_db(sqlite_table_list,␣
↪con_for_import = azure_sqlalchemy_psql_engine)

if import_from_heroku == True:
    heroku_imported_table_list, heroku_run_time =␣
↪read_from_db(sqlite_table_list, con_for_import =␣
↪heroku_sqlalchemy_psql_engine)

if import_from_snowflake == True:
    snowflake_imported_table_list, snowflake_run_time =␣
↪read_from_db(sqlite_table_list, con_for_import = snowflake_engine)

if import_from_databricks == True:
    databricks_imported_table_list, databricks_run_time =␣
↪read_from_db(sqlite_table_list, con_for_import =␣
↪databricks_pyodbc_connection)
```

```
Importing tables from database using the following connection:
Engine(snowflake://KBURCHFIEL:***@RV85777.east-
us-2.azure/KB_SNOWFLAKE_DB/PUBLIC)
Completed import at Mon Nov  8 21:46:17 2021 (local time)
Total run time: 57.57 second(s) (0.0 minute(s) and 57.57 second(s))
```

read_from_db returns a list of DataFrames, each of which stores a particular table. The commented-out code blocks below show how to access and examine these tables.

```
[ ]: # imported_table_list = aws_imported_table_list.copy()
     # imported_table_list[3]
```

```
[ ]: # snowflake_imported_table_list[0]
```

```
[ ]:         departures_scheduled  departures_performed   payload  seats  \
     index
     0                        2.0                   2.0  138035.0    0.0
     1                        2.0                   2.0  187270.0    0.0
     2                        2.0                   2.0  181759.0    0.0
     3                        2.0                   2.0  241160.0    0.0
     4                        2.0                   2.0  184573.0    0.0
     …                          …                     …         …      …
     482268                   2.0                   2.0  188880.0    0.0
     482269                   2.0                   2.0  246417.0    0.0
     482270                   2.0                   2.0  140401.0    0.0
     482271                   2.0                   2.0  147871.0    0.0
     482272                   2.0                   2.0  254323.0    0.0

             passengers   freight     mail  distance  ramp_to_ramp  air_time  …  \
     index                                                                    …
     0              0.0   60427.0      0.0     690.0         203.0     181.0  …
     1              0.0  103389.0      0.0     278.0         115.0      81.0  …
     2              0.0   47551.0      0.0     175.0          96.0      66.0  …
     3              0.0   82038.0  48462.0     582.0         209.0     154.0  …
     4              0.0  101921.0      0.0     372.0         144.0     116.0  …
     …                …         …        …         …             …         …  …
     482268         0.0   47615.0      0.0    2603.0         648.0     593.0  …
     482269         0.0  100673.0      0.0     323.0         122.0     103.0  …
     482270         0.0   73540.0      0.0    1624.0         380.0     349.0  …
     482271         0.0   34010.0      0.0     396.0         153.0     111.0  …
     482272         0.0  120154.0      0.0     501.0         227.0     165.0  …

            code_y  plane_group_text  code      plane_config_text origin_iata_code  \
     index
     0           6     Jet, 2-Engine     2  Freight Configuration              MFE
     1           6     Jet, 2-Engine     2  Freight Configuration              MSP
     2           6     Jet, 2-Engine     2  Freight Configuration              OKC
     3           6     Jet, 2-Engine     2  Freight Configuration              OMA
     4           6     Jet, 2-Engine     2  Freight Configuration              ONT
     …       …                 …    …                      …                …
     482268      6     Jet, 2-Engine     2  Freight Configuration              HNL
     482269      6     Jet, 2-Engine     2  Freight Configuration              LAN
     482270      6     Jet, 2-Engine     2  Freight Configuration              LAS
     482271      6     Jet, 2-Engine     2  Freight Configuration              LRD
     482272      6     Jet, 2-Engine     2  Freight Configuration              MDT
```

```
        origin_lat  origin_lon  destination_iata_code  destination_lat  \
index
0           26.176     -98.239                    LIT           34.729
1           44.880     -93.217                   None              NaN
2           35.393     -97.601                    DFW           32.896
3           41.302     -95.894                   None              NaN
4           34.056    -117.601                    MHR           38.554
…              …           …                     …                …
482268      21.316    -157.927                    ONT           34.056
482269      42.779     -84.587                   None              NaN
482270      36.080    -115.152                   None              NaN
482271      27.544     -99.461                    DFW           32.896
482272      40.193     -76.763                   None              NaN

        destination_lon
index
0               -92.224
1                   NaN
2               -97.037
3                   NaN
4              -121.297
…                   …
482268         -117.601
482269              NaN
482270              NaN
482271          -97.037
482272              NaN

[482273 rows x 62 columns]
```

[ ]:

```
end_time = time.time()
run_time = end_time - start_time
run_minutes = run_time // 60
run_seconds = run_time % 60
print("Completed run at",time.ctime(end_time),"(local time)")
print("Total run time:",'{:.2f}'.format(run_time),"second(s)␣
↪("+str(run_minutes),"minute(s) and",'{:.2f}'.
↪format(run_seconds),"second(s))") # Only meaningful when the program is run␣
↪nonstop from start to finish
```