database_query_timer

November 29, 2021

1 Database Query Timer

Kenneth Burchfiel

In this Jupyter notebook, I will test out the speed of 6 database types (AWS, GCP, Azure, Heroku, Snowflake, and Databricks) using a variety of conditions, then plot the results.

1.1 Disclaimer:

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.**

[ ]:
```
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
from snowflake.connector.pandas_tools import pd_writer
import pyodbc
import speedtest # https://github.com/sivel/speedtest-cli/wiki
import mpld3
from mpld3 import plugins
from mpld3 import fig_to_html
```

The following cell contains configuration variables that determine which trials are run and under what conditions.

[ ]:
```
full_import_test = False
import_test_trials_count = 5

simple_query_test = False
# Simple query test parameters are set prior to each instance of the test
```

```
complex_query_test = False
complex_query_test_query_count = 50
complex_query_test_trials_count = 4
```

Next, I'll extract my database passwords and then connect to each database using those 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()
```

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.

[ ]:
```
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.
```

```
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

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"
    )
)

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.

heroku_sqlalchemy_psql_engine = sqlalchemy.create_engine('postgresql://
↪wtgddsmdmsipoo:'+str(heroku_pw)+'@ec2-23-23-199-57.compute-1.amazonaws.com:
↪5432/ddc9jvn58br3tc')

snowflake_engine = sqlalchemy.create_engine('snowflake://KBURCHFIEL:
↪'+str(snowflake_pw)+'@RV85777.east-us-2.azure/KB_SNOWFLAKE_DB/PUBLIC')

databricks_pyodbc_connection = pyodbc.connect("DSN=Databricks_Cluster",␣
↪autocommit=True)
databricks_pyodbc_cursor = databricks_pyodbc_connection.cursor()
```

Now I'll connect to my local SQLite database to retrieve a list of tables in that database. This list will correspond with the tables in the 6 online databases.

[ ]:
```
sqlite_table_query = sqlalchemy_sqlite_engine.execute("Select name from␣
↪sqlite_schema where type = 'table'")
```

```
# This method for extracting all tables comes from https://www.kite.com/python/
↪answers/how-to-list-tables-using-sqlite3-in-python
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']

2 Part 1: Full database import test

I will first measure how long it takes to download the entire contents of the flights, music, steps, and photos tables stored in the 6 online databases into DataFrames.

The following function imports all tables from a given database into DataFrames and times how long it takes to do so.

[ ]:
```
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.
```

The next function, time_database_imports, applies read_from_db to each database and outputs a dictionary containing the times it took to download each set of tables.

```
[ ]: def time_database_imports(trials_count):
         import_times_dict_list = []
         trials_count # Can be increased to create average upload times
         for i in range(1, trials_count+1):
             print("\nTrial",i)
             print("\n\nAWS Import:")
             aws_imported_table_list, aws_run_time = read_from_db(sqlite_table_list,␣
     ↪con_for_import = aws_sqlalchemy_psql_engine)
             import_times_dict_list.append({"Service":"AWS", "Time":␣
     ↪aws_run_time,"Trial":i})
             print("\n\nGCP Import:")
             gcp_imported_table_list, gcp_run_time = read_from_db(sqlite_table_list,␣
     ↪con_for_import = gcp_sqlalchemy_psql_engine)
             import_times_dict_list.append({"Service":"GCP","Time":
     ↪gcp_run_time,"Trial":i})
             print("\n\nAzure Import:")
             azure_imported_table_list, azure_run_time =␣
     ↪read_from_db(sqlite_table_list, con_for_import =␣
     ↪azure_sqlalchemy_psql_engine)
             import_times_dict_list.append({"Service":'Azure', "Time":
     ↪azure_run_time,"Trial":i})
             print("\n\nHeroku Import:")
             heroku_imported_table_list, heroku_run_time =␣
     ↪read_from_db(sqlite_table_list, con_for_import =␣
     ↪heroku_sqlalchemy_psql_engine)
             import_times_dict_list.append({"Service":'Heroku', "Time":
     ↪heroku_run_time,"Trial":i})
             print("\n\nSnowflake Import:")
             snowflake_imported_table_list, snowflake_run_time =␣
     ↪read_from_db(sqlite_table_list, con_for_import = snowflake_engine)
             import_times_dict_list.append({"Service":'Snowflake', "Time":
     ↪snowflake_run_time,"Trial":i})
             print("\n\nDatabricks Import:")
             databricks_imported_table_list, databricks_run_time =␣
     ↪read_from_db(sqlite_table_list, con_for_import =␣
     ↪databricks_pyodbc_connection)
             import_times_dict_list.append({"Service":'Databricks', "Time":
     ↪databricks_run_time,"Trial":i})

         df_import_times = pd.DataFrame(import_times_dict_list)
         df_import_times
         return df_import_times
```

Since internet connection times might impact my download/query times, I created a function that runs an internet speed test (from speedtest.com). Running this function before and after each set of database tests helps me identify changes in internet speed that might influence the results of the tests.

```
[ ]: def run_speedtest():
        # Code is based on https://github.com/sivel/speedtest-cli/wiki
        servers = []
        threads = None
        s = speedtest.Speedtest()
        s.get_servers(servers)
        s.get_best_server()
        s.download(threads=threads)
        s.upload(threads=threads)
        s.results.share()
        results_dict = s.results.dict()
        print("Download speed (in Mbps):", round(results_dict['download']/
     ↪1000000,2))
        print("Upload speed (in Mbps):", round(results_dict['upload']/1000000,2))
```

The following code block calculates the time required to download the flights, steps, photos, and music tables from each database into DataFrames. Setting trials_count to a number greater than 1 allows an average download speed to be calculated from multiple tests.

```
[ ]: if full_import_test == True:
        print("Pre-test speedtest:")
        run_speedtest()
        df_import_times = time_database_imports(trials_count =␣
     ↪import_test_trials_count)
        print("Post-test speedtest:")
        run_speedtest()
        df_import_times.to_csv('metrics\\full_import_results.csv') # Saving the␣
     ↪results to a
        # .csv file means the data can be analyzed in the future without having
        # to re-run the tests
```

Mean import times for each database are shown below.

```
[ ]: df_import_times = pd.read_csv('metrics\\full_import_results.csv') # Check␣
     ↪headers
     mean_import_times = df_import_times[['Service', 'Time']].copy().
     ↪groupby('Service').mean()
     mean_import_times.rename(columns={'Time':'Import_Time'},inplace=True)
     mean_import_times.sort_values('Import_Time',inplace=True)
     mean_import_times.reset_index(inplace=True)
     mean_import_times['Import_Time_Rank'] = mean_import_times.index+1 # Converts␣
     ↪the index (which starts with 0) into a ranking that starts with 1
     mean_import_times
```

```
[ ]:      Service  Import_Time  Import_Time_Rank
     0  Snowflake    17.480364                 1
     1      Azure    40.748104                 2
     2        AWS    41.006784                 3
```

```
3         GCP    41.315156                   4
4      Heroku    54.513024                   5
5  Databricks    65.536882                   6
```

The plot_results() function below plots the contents of DataFrames formatted like df_import_times as a series of lines, each of which represents the download or query times for a particular database. plot_results() saves each chart as (1) a static .png image and (2) an interactive .html file that can be viewed in a web browser. The .html file displays each query/download time when the user hovers over a particular marker on the graph.

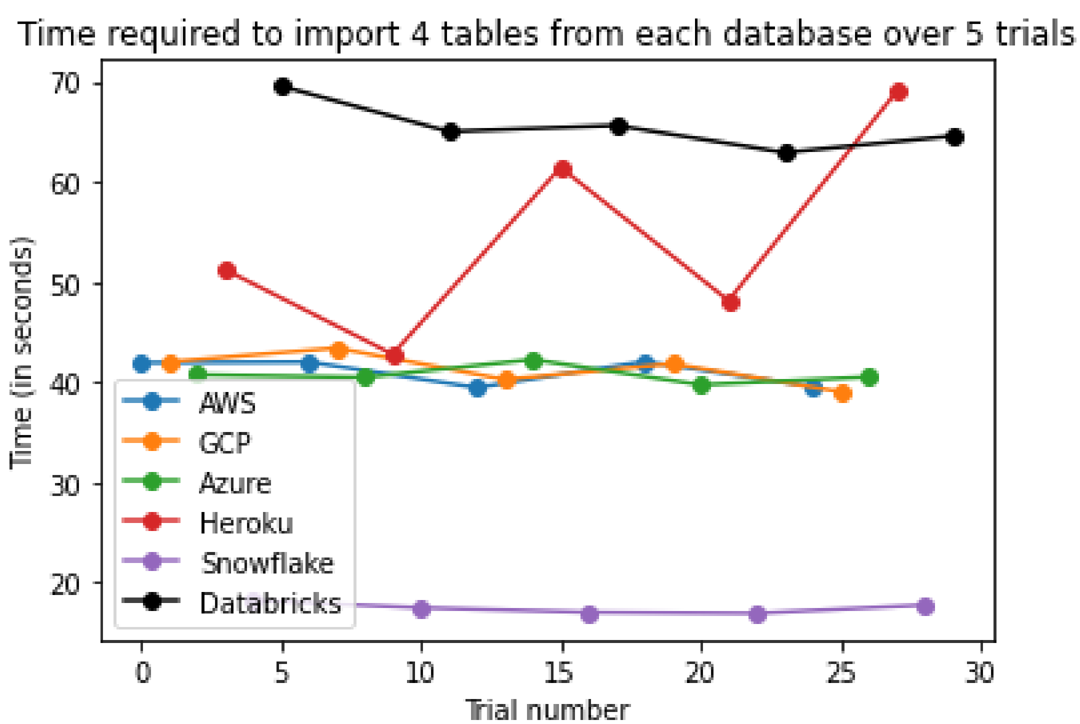
```
[ ]: def plot_results(df, title, save_string):
         fig, axes = plt.subplots()
         fig.set_facecolor('white')
         line_list = []
         line_list.append(axes.plot(df.query("Service == 'AWS'")['Time'],␣
      ↪label='AWS', marker='o'))
         plugins.connect(fig, plugins.PointLabelTooltip(line_list[0][0], labels =␣
      ↪list(df.query("Service == 'AWS'")['Time'].round(3)))) # A second [0] is␣
      ↪needed for the tooltips to show! See https://nbviewer.org/gist/aflaxman/
      ↪f7d6fda45a69223c4200 and https://mpld3.github.io/modules/API.html. Note that␣
      ↪they include a [0] even though there's just one line in their examples.
         line_list.append(axes.plot(df.query("Service == 'GCP'")['Time'], label =␣
      ↪'GCP', marker='o'))
         plugins.connect(fig, plugins.PointLabelTooltip(line_list[1][0], labels =␣
      ↪list(df.query("Service == 'GCP'")['Time'].round(3))))
         line_list.append(axes.plot(df.query("Service == 'Azure'")['Time'], label =␣
      ↪'Azure', marker='o'))
         plugins.connect(fig, plugins.PointLabelTooltip(line_list[2][0], labels =␣
      ↪list(df.query("Service == 'Azure'")['Time'].round(3))))
         line_list.append(axes.plot(df.query("Service == 'Heroku'")['Time'], label =␣
      ↪'Heroku', marker='o'))
         plugins.connect(fig, plugins.PointLabelTooltip(line_list[3][0], labels =␣
      ↪list(df.query("Service == 'Heroku'")['Time'].round(3))))
         line_list.append(axes.plot(df.query("Service == 'Snowflake'")['Time'],␣
      ↪label = 'Snowflake', marker='o'))
         plugins.connect(fig, plugins.PointLabelTooltip(line_list[4][0], labels =␣
      ↪list(df.query("Service == 'Snowflake'")['Time'].round(3))))
         line_list.append(axes.plot(df.query("Service == 'Databricks'")['Time'],␣
      ↪label = 'Databricks', marker='o', color = 'black')) # The default 6th color␣
      ↪is hard to differentiate from another color, at least when I block blue␣
      ↪light from my monitor
         plugins.connect(fig, plugins.PointLabelTooltip(line_list[5][0], labels =␣
      ↪list(df.query("Service == 'Databricks'")['Time'].round(3))))
         plt.title(title) # The wrap=True parameter doesn't appear to affect .html␣
      ↪charts; therefore, I wrapped text for those titles by inserting an explicit␣
      ↪newline character (\n) where necessary.
         plt.ylabel("Time (in seconds)")
```

```
    plt.xlabel("Trial number")
    plt.legend()
    print("Line_list[0]:",line_list[0])
    mpld3.save_html(fig, 'metrics\\'+save_string+'.html')
    plt.savefig('metrics\\'+save_string+'.png',dpi=400)
    # plt.show()
    return fig


# for i in range(len(fig_line_list)):
#         plugins.connect(test_fig, plugins.LineLabelTooltip(fig_line_list[0],␣
 ↪label = 'test'))
#mpld3.save_html(test_fig, 'new_output_test.html')
# mpld3.display()
# Once you get this to work, integrate it with the rest of the plot code, then␣
 ↪save the html file within that code (based on the same title that you're␣
 ↪giving the other charts--just with 'html' instead)
```

[ ]:
```
import_plot = plot_results(df_import_times, title = "Time required to import 4␣
 ↪tables from each database over "+str(import_test_trials_count)+" trials",␣
 ↪save_string = 'full_import_time')
```

```
Line_list[0]: [<matplotlib.lines.Line2D object at 0x000001759F47B8E0>]
```



The above chart shows that Snowflake's import times were consistently the lowest among all 5 providers, whereas Databricks imports tended to take the longest. AWS, GCP, and Azure all had very similar import times.

3 Part 2: Simple query tests

The next two tests evaluate how long it takes to execute a simple query within each database.

The query_test function applies a particular query to a database and times how long it takes for that query to be stored (in the query_result variable). Setting query_count above 1 allows the query to be applied multiple times.

```
[ ]: def query_test(query, query_count, connection, offset = False):
         query_start_time = time.time()
         if offset == True: # This condition causes a different row to be returned␣
     ↪each time. For this to work, query_count has to be lower than the length of␣
     ↪the results of the query.
             for i in range(query_count):
                 query_result = connection.execute(query+" limit 1 offset "+str(i))
         else:
             for i in range(query_count):
                 query_result = connection.execute(query)
         query_end_time = time.time()
         query_run_time = query_end_time - query_start_time
         return query_run_time
```

The database_query_test function applies query_test to each database and stores the times needed to execute the function into a DataFrame. Its trials_count parameter determines how many times the function will apply query_count to each database.

```
[ ]: def database_query_test(query, query_count, trials_count, offset = False,␣
     ↪verbose = False):
         query_times_dict_list = []
         for i in range(1, trials_count+1):
             print("\nTrial",i)
             print("\nRunning AWS Query")
             query_start_time = time.time()
             aws_run_time = query_test(query, query_count, connection =␣
     ↪aws_sqlalchemy_psql_engine, offset = offset)
             query_times_dict_list.append({"Service":"AWS", "Time":␣
     ↪aws_run_time,"Trial":i})
             query_end_time = time.time()
             query_length = query_end_time - query_start_time
             if verbose == True:
                 print("Query time (in seconds):",query_length)
                 print("Average time per query:",query_length/query_count)

             print("\nRunning GCP Query")
```

```
        query_start_time = time.time()
        gcp_run_time = query_test(query, query_count, connection =␣
↪gcp_sqlalchemy_psql_engine, offset = offset)
        query_times_dict_list.append({"Service":"GCP", "Time":␣
↪gcp_run_time,"Trial":i})
        query_end_time = time.time()
        query_length = query_end_time - query_start_time
        if verbose == True:
            print("Query time (in seconds):",query_length)
            print("Average time per query:",query_length/query_count)

        print("\nRunning Azure Query")
        query_start_time = time.time()
        azure_run_time = query_test(query, query_count, connection =␣
↪azure_sqlalchemy_psql_engine, offset = offset)
        query_times_dict_list.append({"Service":'Azure', "Time":
↪azure_run_time,"Trial":i})
        query_end_time = time.time()
        query_length = query_end_time - query_start_time
        if verbose == True:
            print("Query time (in seconds):",query_length)
            print("Average time per query:",query_length/query_count)

        print("\nRunning Heroku Query")
        query_start_time = time.time()
        heroku_run_time = query_test(query, query_count, connection =␣
↪heroku_sqlalchemy_psql_engine, offset = offset)
        query_times_dict_list.append({"Service":'Heroku', "Time":
↪heroku_run_time,"Trial":i})
        query_end_time = time.time()
        query_length = query_end_time - query_start_time
        if verbose == True:
            print("Query time (in seconds):",query_length)
            print("Average time per query:",query_length/query_count)

        print("\nRunning Snowflake Query")
        query_start_time = time.time()
        snowflake_run_time = query_test(query, query_count, connection =␣
↪snowflake_engine, offset = offset)
        query_times_dict_list.append({"Service":'Snowflake', "Time":
↪snowflake_run_time,"Trial":i})
        query_end_time = time.time()
        query_length = query_end_time - query_start_time
        if verbose == True:
            print("Query time (in seconds):",query_length)
            print("Average time per query:",query_length/query_count)
```

```
        print("\nRunning Databricks Query")
        query_start_time = time.time()
        databricks_run_time = query_test(query, query_count, connection =␣
↪databricks_pyodbc_connection, offset = offset)
        query_times_dict_list.append({"Service":'Databricks', "Time":
↪databricks_run_time,"Trial":i})
        query_end_time = time.time()
        query_length = query_end_time - query_start_time
        if verbose == True:
            print("Query time (in seconds):",query_length)
            print("Average time per query:",query_length/query_count)

    df_simple_query_results = pd.DataFrame(query_times_dict_list)
    return df_simple_query_results
```

The first simple query test will apply a basic query to each database only once, then repeat this process 20 times in order to obtain an average query length for each database.

[ ]:
```
simple_query = "Select * from photos limit 1"
simple_query_count = 1
simple_query_test_trials_count = 20

if simple_query_test == True:
    print("Pre-test speedtest:")
    run_speedtest()
    simple_query_results = database_query_test(query = simple_query,␣
↪query_count = simple_query_count, trials_count =␣
↪simple_query_test_trials_count, offset = False)
    print("Post-test speedtest:")
    run_speedtest()
    simple_query_results.to_csv('metrics\\simple_query_test_1x.csv')
```

[ ]:
```
simple_query_results_1x = pd.read_csv('metrics\\simple_query_test_1x.csv')
mean_simple_query_results_1x = simple_query_results_1x[['Service', 'Time']].
↪copy().groupby('Service').mean().sort_values('Time')
mean_simple_query_results_1x.rename(columns={'Time':
↪'Simple_Query_1x_Time'},inplace=True)
mean_simple_query_results_1x.sort_values('Simple_Query_1x_Time',inplace=True)
mean_simple_query_results_1x.reset_index(inplace=True)
mean_simple_query_results_1x['Simple_Query_1x_Rank'] =␣
↪mean_simple_query_results_1x.index+1 # Converts the index (which starts with␣
↪0) into a ranking that starts with 1
mean_simple_query_results_1x
```

[ ]:

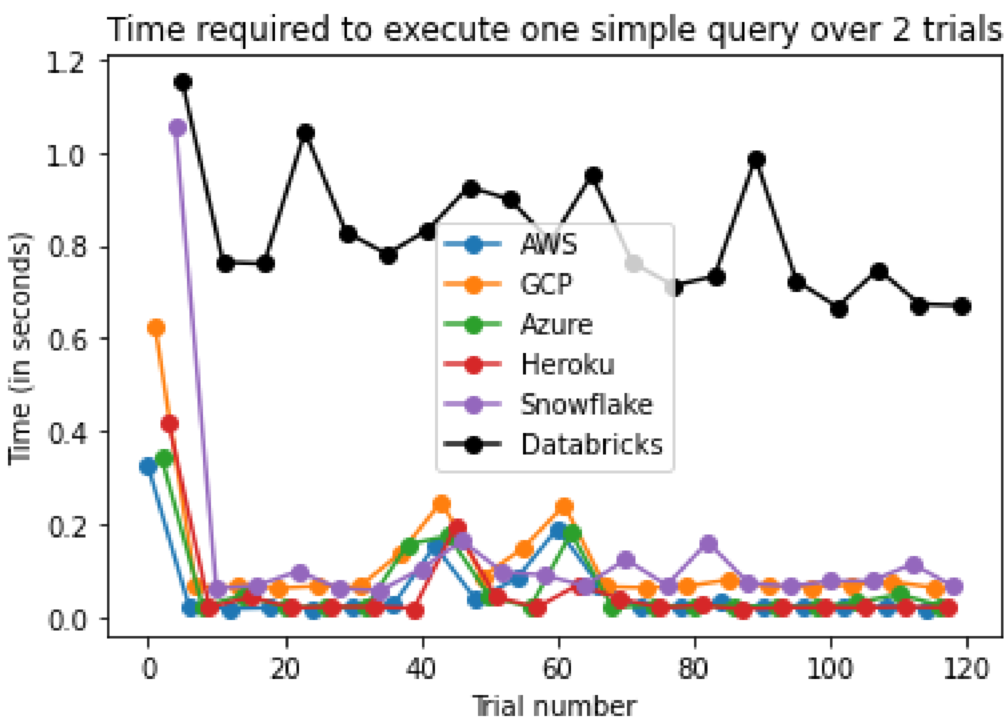
	Service	Simple_Query_1x_Time	Simple_Query_1x_Rank
0	Heroku	0.056835	1

```
1         AWS                0.060395                          2
2       Azure                0.066731                          3
3         GCP                0.123177                          4
4   Snowflake                0.139276                          5
5  Databricks                0.821429                          6
```

This table, sorted by execution time, shows the mean amount of time required to (1) execute the simple query on each database and (2) assign the result to a variable. Databricks queries took significantly longer than did queries of the other 5 databases, whereas Heroku had the fastest query speeds. GCP and Snowflake were slower than AWS and Azure.

```
[ ]: simple_query_1x_plot = plot_results(df = simple_query_results_1x, title = 'Time␣
      ↪required to execute one simple query over␣
      ↪'+str(simple_query_test_trials_count)+' trials', save_string =␣
      ↪'simple_query_results_1x')
```

```
Line_list[0]: [<matplotlib.lines.Line2D object at 0x000001759F5DE4F0>]
```



The above two plots (which are identical) show that, for every database, the first of the 20 queries took the longest.

4 Longer simple query test

Now that I have calculated the length of time required for one query, I will determine the length of time needed to calculate that same query 1000 times in a row. The trials count was reduced from

20 to 2 to save time; even so, the following code block still took about 38 minutes to run.

[ ]:
```
simple_query_count = 1000
simple_query_test_trials_count = 2

if simple_query_test == True:
    print("Pre-test speedtest:")
    run_speedtest()
    simple_query_results = database_query_test(query = simple_query,␣
 ↪query_count = simple_query_count, trials_count =␣
 ↪simple_query_test_trials_count, offset = False, verbose = True)
    print("Post-test speedtest:")
    run_speedtest()
    simple_query_results.to_csv('metrics\\simple_query_test_1000x.csv')
```

[ ]:
```
simple_query_results_1000x = pd.read_csv('metrics\\simple_query_test_1000x.csv')
mean_simple_query_results_1000x = simple_query_results_1000x[['Service',␣
 ↪'Time']].copy().groupby('Service').mean().sort_values('Time')
mean_simple_query_results_1000x.rename(columns={'Time':
 ↪'Simple_Query_1000x_Time'},inplace=True)
mean_simple_query_results_1000x['Simple_Query_1000x_Time_Per_Query'] =␣
 ↪mean_simple_query_results_1000x['Simple_Query_1000x_Time'] /␣
 ↪simple_query_count
mean_simple_query_results_1000x.
 ↪sort_values('Simple_Query_1000x_Time',inplace=True)
mean_simple_query_results_1000x.reset_index(inplace=True)
mean_simple_query_results_1000x['Simple_Query_1000x_Rank'] =␣
 ↪mean_simple_query_results_1000x.index+1 # Converts the index (which starts␣
 ↪with 0) into a ranking that starts with 1
mean_simple_query_results_1000x
```

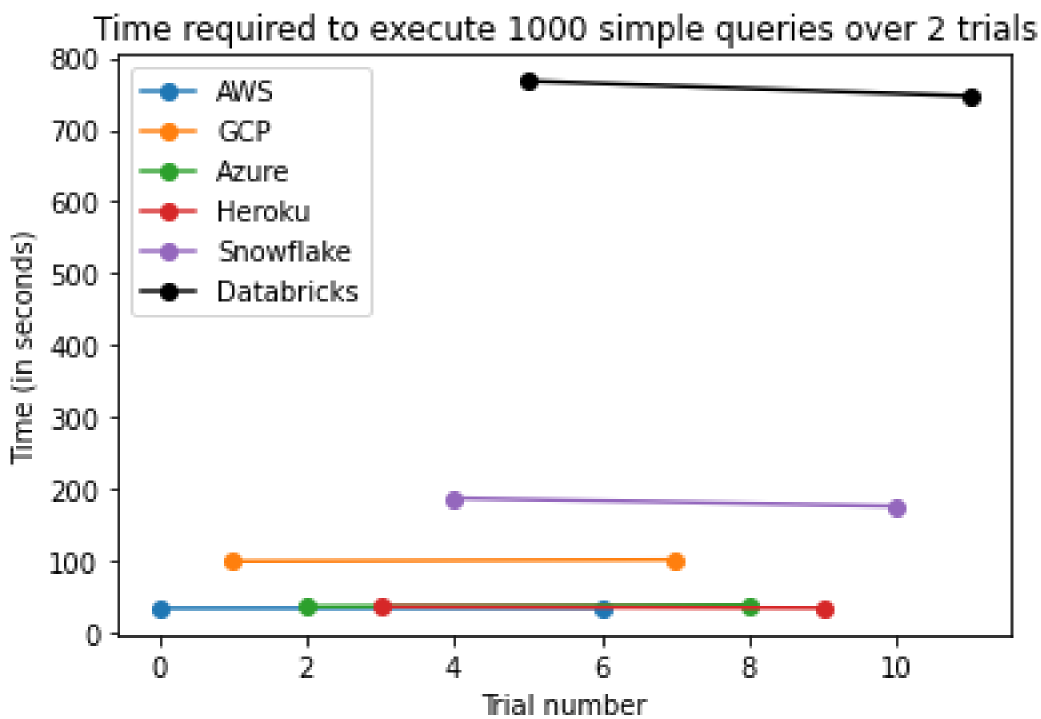
[ ]:
```
      Service  Simple_Query_1000x_Time  Simple_Query_1000x_Time_Per_Query  \
0         AWS                33.991590                           0.033992
1      Heroku                35.360053                           0.035360
2       Azure                37.267689                           0.037268
3         GCP               100.523524                           0.100524
4   Snowflake               181.722105                           0.181722
5  Databricks               757.514256                           0.757514

   Simple_Query_1000x_Rank
0                        1
1                        2
2                        3
3                        4
4                        5
5                        6
```

The above DataFrame shows the average amount of time needed to execute 1000 simple queries on each database. The ranking of the providers was similar to the single-query ranking, except that AWS's average time was now faster than Heroku. Databricks remained consistently slower than the other 5 databases, and GCP and Snowflake were considerably slower than AWS, Heroku, and Azure.

```
[ ]: simple_query_1000x_plot = plot_results(df = simple_query_results_1000x, title =
     ↪'Time required to execute '+str(simple_query_count)+' simple queries over
     ↪'+str(simple_query_test_trials_count)+' trials', save_string =
     ↪'simple_query_results_1000x')
```

```
Line_list[0]: [<matplotlib.lines.Line2D object at 0x00000175A06635B0>]
```



The query times remained relatively similar across the two trials, which is not surprising given that each time shown is the result of 1,000 queries (rather than just one as in the earlier simple query test).

4.1 Complex query test

Having examined the times needed to process a very simple query, I will now execute a more complex query. This query determines which airports have the highest average number of passengers per Delta flight. It involves aggregate functions along with WHERE, GROUP BY, and ORDER BY statements, and is applied to a table with over 29 million cells. Here is an example of the query applied to the flights table within my local sqlite database:

```
[ ]: complex_query = "Select sum(\"PASSENGERS\"), sum(\"DEPARTURES_PERFORMED\"),␣
 ↪sum(\"PASSENGERS\")/sum(\"DEPARTURES_PERFORMED\") as␣
 ↪passengers_per_departure, destination_iata_code from flights where␣
 ↪\"UNIQUE_CARRIER_NAME\" = 'Delta Air Lines Inc.' group by␣
 ↪destination_iata_code order by passengers_per_departure desc" # When using␣
 ↪read_sql within PostgreSQL databases, if the original column name is␣
 ↪capitalized or in all caps, you'll need to put the column name in quotes␣
 ↪(preceded by backticks so they're not mistaken as beginning/ending a new␣
 ↪string). This is because read_sql converts all column entries to lowercase.␣
 ↪See https://stackoverflow.com/questions/68635773/
 ↪column-does-not-exist-sqlalchemy-postgresql-trouble-with-quotation-marks
# The quotes aren't necessary for SQLite; hence, the following code would also␣
 ↪work:

complex_query_for_sqlite = "Select sum(PASSENGERS), sum(DEPARTURES_PERFORMED),␣
 ↪sum(PASSENGERS)/sum(DEPARTURES_PERFORMED) as passengers_per_departure,␣
 ↪destination_iata_code from flights where UNIQUE_CARRIER_NAME == 'Delta Air␣
 ↪Lines Inc.' group by destination_iata_code order by passengers_per_departure␣
 ↪desc"



test_result = pd.read_sql(complex_query, con = sqlalchemy_sqlite_engine)
test_result.head(30)
```

```
[ ]:     sum("PASSENGERS")  sum("DEPARTURES_PERFORMED")  passengers_per_departure  \
    0                282.0                          1.0                282.000000
    1                550.0                          2.0                275.000000
    2                540.0                          2.0                270.000000
    3              96456.0                        360.0                267.933333
    4              92048.0                        357.0                257.837535
    5                257.0                          1.0                257.000000
    6             241441.0                        961.0                251.239334
    7             112596.0                        469.0                240.076759
    8              85956.0                        359.0                239.431755
    9            1426354.0                       6009.0                237.369612
    10            125616.0                        533.0                235.677298
    11            171485.0                        732.0                234.269126
    12               234.0                          1.0                234.000000
    13             83992.0                        363.0                231.382920
    14               454.0                          2.0                227.000000
    15               224.0                          1.0                224.000000
    16            150358.0                        683.0                220.143485
    17             74878.0                        345.0                217.037681
    18            430895.0                       2004.0                215.017465
    19             78187.0                        365.0                214.210959
    20            259142.0                       1216.0                213.110197
```

```
21        130728.0           617.0    211.876823
22        152780.0           724.0    211.022099
23         50251.0           240.0    209.379167
24        869353.0          4173.0    208.328061
25        903740.0          4393.0    205.722741
26         33532.0           163.0    205.717791
27           204.0             1.0    204.000000
28         66306.0           327.0    202.770642
29         34958.0           173.0    202.069364

   destination_iata_code
0                    GOA
1                    MVD
2                    YHZ
3                    TLV
4                    JNB
5                    LGW
6                    FCO
7                    MXP
8                    SYD
9                    AMS
10                   BCN
11                   HND
12                   YQX
13                   EZE
14                   VCP
15                   MDZ
16                   PEK
17                   VCE
18                   NRT
19                   LIM
20                   GRU
21                   MAD
22                   DUB
23                   HKG
24                   CDG
25                   HNL
26                   PRG
27                   GYE
28                   STR
29                   TXL
```

[ ]:
```
if complex_query_test == True:
    print("Pre-test speedtest:")
    run_speedtest()
```

```
    complex_query_results = database_query_test(query = complex_query,␣
↪query_count = complex_query_test_query_count, trials_count =␣
↪complex_query_test_trials_count, offset = False, verbose = True)
    print("Post-test speedtest:")
    run_speedtest()
    complex_query_results.to_csv('metrics\\complex_query_test.csv')
```

[ ]:

```
complex_query_results = pd.read_csv('metrics\\complex_query_test.csv')
mean_complex_query_times = complex_query_results[['Service', 'Time']].copy().
↪groupby('Service').mean().sort_values('Time')
mean_complex_query_times.rename(columns={'Time':
↪'Complex_Query_Time'},inplace=True)
mean_complex_query_times['Time_Per_Complex_Query'] =␣
↪mean_complex_query_times['Complex_Query_Time'] /␣
↪complex_query_test_query_count
mean_complex_query_times.sort_values('Complex_Query_Time',inplace=True)
mean_complex_query_times.reset_index(inplace=True)
mean_complex_query_times['Complex_Query_Rank'] = mean_complex_query_times.
↪index+1 # Converts the index (which starts with 0) into a ranking that␣
↪starts with 1
mean_complex_query_times
```

[ ]:

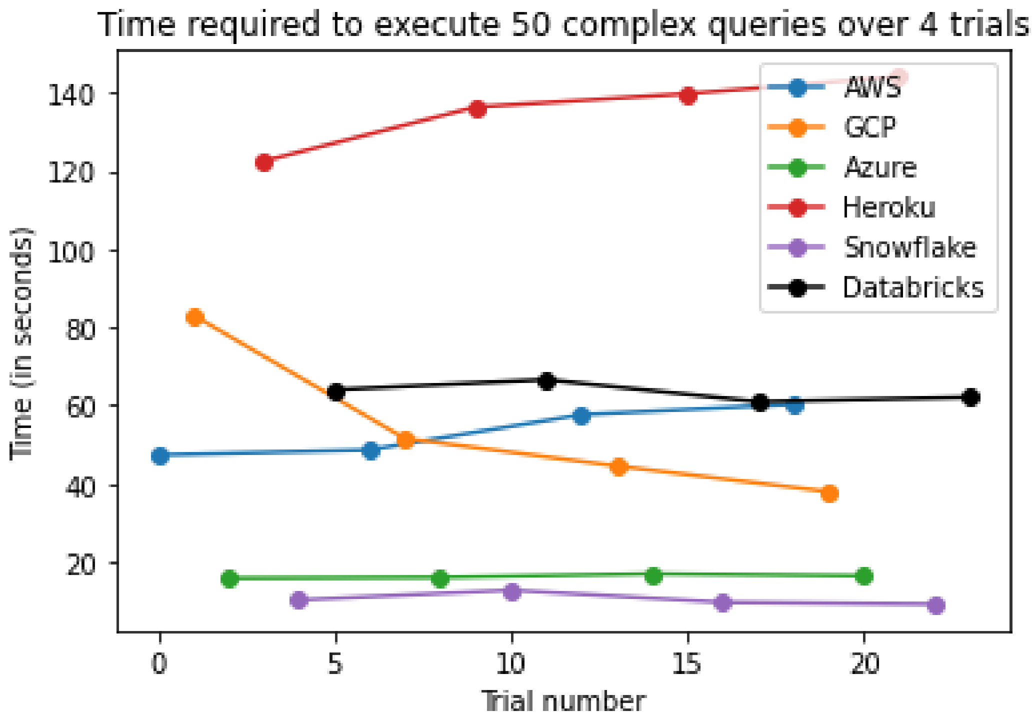
	Service	Complex_Query_Time	Time_Per_Complex_Query	Complex_Query_Rank
0	Snowflake	10.372686	0.207454	1
1	Azure	16.240557	0.324811	2
2	AWS	53.415292	1.068306	3
3	GCP	54.259636	1.085193	4
4	Databricks	63.377897	1.267558	5
5	Heroku	135.697018	2.713940	6

The complex queries took longer than the simple queries for all 6 providers. However, the complex query time/simple query time ratio varied widely, as df_query_time_comparison shows below. It was lowest for Snowflake (1.142), which exhibited the lowest complex query time overall, and highest for Heroku (76.75), which had the highest overall complex query time. Azure also had a significantly lower time than did AWS and GCP.

[ ]:

```
complex_query_plot = plot_results(complex_query_results, title = 'Time required␣
↪to execute '+str(complex_query_test_query_count)+' complex queries over␣
↪'+str(complex_query_test_trials_count)+' trials', save_string =␣
↪'complex_query_time')
```

```
Line_list[0]: [<matplotlib.lines.Line2D object at 0x00000175A0732280>]
```



The above chart shows that, while Azure, Snowflake, and Databricks had relatively consistent complex query times, the query times for AWS and Heroku increased over the 4 trials whereas GCP's decreased.

The following code block merges together all of the mean import/query time DataFrames, then creates a composite ranking of the 6 providers based on the sum of their 4 individual rankings.

[ ]:
```
df_query_time_comparison = mean_import_times.
 ↪merge(mean_simple_query_results_1x, on = 'Service').
 ↪merge(mean_simple_query_results_1000x, on = 'Service').
 ↪merge(mean_complex_query_times, on = 'Service')
df_query_time_comparison.insert(1, 'Sum of ranks',␣
 ↪df_query_time_comparison['Import_Time_Rank'] +␣
 ↪df_query_time_comparison['Simple_Query_1x_Rank'] +␣
 ↪df_query_time_comparison['Simple_Query_1000x_Rank'] +␣
 ↪df_query_time_comparison['Complex_Query_Rank'])
df_query_time_comparison['Complex/Simple Query Ratio'] =␣
 ↪df_query_time_comparison['Time_Per_Complex_Query']/
 ↪df_query_time_comparison['Simple_Query_1000x_Time_Per_Query']
df_query_time_comparison.sort_values('Sum of ranks', inplace=True)
df_query_time_comparison.reset_index(drop=True,inplace=True)
df_query_time_comparison.insert(1, 'Composite ranking',␣
 ↪df_query_time_comparison.index+1)
df_query_time_comparison.to_csv('metrics\\overall_database_query_rankings.csv')
```

```
df_query_time_comparison
```

[ ]:

```
       Service  Composite ranking  Sum of ranks  Import_Time  Import_Time_Rank  \
0          AWS                  1             9    41.006784                 3
1        Azure                  2            10    40.748104                 2
2    Snowflake                  3            12    17.480364                 1
3       Heroku                  4            14    54.513024                 5
4          GCP                  5            16    41.315156                 4
5   Databricks                  6            23    65.536882                 6

   Simple_Query_1x_Time  Simple_Query_1x_Rank  Simple_Query_1000x_Time  \
0              0.060395                     2                33.991590
1              0.066731                     3                37.267689
2              0.139276                     5               181.722105
3              0.056835                     1                35.360053
4              0.123177                     4               100.523524
5              0.821429                     6               757.514256

   Simple_Query_1000x_Time_Per_Query  Simple_Query_1000x_Rank  \
0                           0.033992                        1
1                           0.037268                        3
2                           0.181722                        5
3                           0.035360                        2
4                           0.100524                        4
5                           0.757514                        6

   Complex_Query_Time  Time_Per_Complex_Query  Complex_Query_Rank  \
0           53.415292                1.068306                   3
1           16.240557                0.324811                   2
2           10.372686                0.207454                   1
3          135.697018                2.713940                   6
4           54.259636                1.085193                   4
5           63.377897                1.267558                   5

   Complex/Simple Query Ratio
0                   31.428534
1                    8.715623
2                    1.141599
3                   76.751592
4                   10.795411
5                    1.673312
```

[ ]:

```
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
```

```
Completed run at Mon Nov  8 19:05:46 2021 (local time)
Total run time: 4.69 second(s) (0.0 minute(s) and 4.69 second(s))
```