DS4300 Homework 1 Methodology

Abstract:

Using a MySQL database, we have constructed an API to insert tweets from a .CSV file (simulating a stream of data) and retrieve collections of these tweets based on followed users (known as a 'timeline'). We have separated this code into multiple files, covering connection and manipulation of the database, defining classes tailored to the project, interfacing with the database, and testing the functionality code. While our performance leaves much to be desired, the implementation is strong enough as we consider improvements.

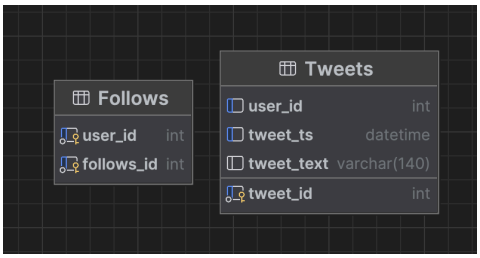
Specifications:

The device that executed the code was a 1.4 GHz Quad-core Intel Core i5, 8 GB 2133 MHz DDR3 RAM, 13-inch 2019 MacBook Pro on MacOS Sonoma 14.2.1. MySQL was used as the database server with InnoDB being the storage engine. The code was written in Python and implemented prepared statements and secondary indexing to improve efficiency.

API Framework

tweet_database.sql

Contains the code for initalizing and implementing the database that the API will be inserting into. A user 'tweetuser' was created with SELECT, INSERT, UPDATE, DELETE, and EXECUTE privileges with the user-defined username and password stored in a .ENV file that will need to be created on implementation. This file creates the Tweets database and two tables Tweets and Follows. It then indexes the database and creates a procedure "get_timeline" to find all tweets associated with all of the users a randomly generated user follows. The diagram of the database is shown below:



tweet_dbutils.py

Contains the definition of the **DBUtils** class, an object that initializes a connection to a given database and provides methods to manipulate that database. Class methods include a function for closing the connection, executing a query that has been built programmatically in Python, insertion of one row into the database, and the insertion of many rows into the database. The creation of indices is also possible, but in our project, we created indices within the .SQL script for building our database schema.

tweet_objects.py

Short file defining classes for **Tweet**, **User**, and **Follows**. **Tweet** is the main class used in this program, containing the tweet text, the ID of the user who tweeted it, and the timestamp it was uploaded into the database. **Follows** is used to collect the users who follow a specific user, but this is a secondary functionality that was added as a bonus feature. Normally imported with tweet_dbutils.py.

tweet_mysql.py

Main API functionality of this library, utilizing the previous files to execute specific actions performed on the database. Methods are contained in the **TweetUserAPI** class, which uses the database connection initialized from **DBUtils.** Within **TweetUserAPI**, there is a method to insert a single tweet into the database, retrieve user IDs from the database, get collections of tweets from users that a given user is following (the 'timeline'), and other easily added but useful operations such as the number of users that a given user is following.

In addition to this class, we have also included a decorator to measure how this library performs at interfacing with the database. This decorator measures the number of method calls and the elapsed time of these functions, and then calculates and returns the number of method calls per second (referenced in the code as "API calls per second").

tweet_test.py

File used to read in data, test our code, and print results. We have used a .ENV file with database connection variables accessed within this code for privacy purposes. Then, we read in a .CSV file to the MySQL database and insert each row individually to simulate the collection of data. After all tweets are entered into the database, we retrieve a set number of random timelines to test accuracy. Finally, we print out the API calls per second to obtain performance results.

Performance:

Leveraging the functionalities of the libraries that we have developed, we were able to insert tweets from a .CSV file (simulating a stream of data) and retrieve collections of these tweets based on followed users. After initializing and running the SQL and Python scripts, we were able to achieve 1490.3 API calls per second for the insertion part of the assignment and 3.9 API calls per second for the second part. Even with indices used in the SQL database, we did not see a notable performance jump. While there's much left to be on the table, the designed and implemented framework works well and returns the desired results in a timely manner.

Next Steps

In the future, we can implement both more functionality to the code and more approaches to optimization. For example, we could have more functions when accessing the database, such as mimicking Twitter's "Lists" functionality where we only return tweets from a specific list of users (not necessarily just those that are followed). We can also extend this API to other programming languages, allowing users of all technical backgrounds to be able to access this data.

In terms of optimization, there may be improved performance results if these scripts were hosted on a powerful virtual machine in the cloud, or with refactored code to cut down on unnecessary and clunky syntax. While our main goal was to get the project up and running, we are open and willing to try and find other new opportunities to increase performance - perhaps using a different database, such as Redis.

Member Contributions

Member	Contributions
Jeffrey Pan	Designed the database structure and implemented a preliminary framework for the API, converting the example into a Twitter-specific framework. Ran preliminary tests for importing records as well as calling tweets, and designed the foundation for tracking API calls per second. Ran the code for performance testing. Collaborated with Ryan in the write-up process.
Ryan Delano	Led design of API performance measurement, collaborated with Jeffrey on code testing and review. Focused on optimization goals, such as the creation of indices and refactoring of extra code. Computer performance impacted ability to test code, but worked with Jeffrey to polish and add ideas and functionalities. Wrote framework for methodology before collaborating on details.