Zach Ingbretsen Exercise 2

For instructions on how to run the code, please see README.md! This document speaks about the technology and the code, but not specifically about how to run it.

Purpose

Being able to process streaming data is becoming increasingly important as the amount and velocity of data being collected by companies and individuals surge. Processing these data (e.g., Twitter data) in more traditional ways is likened to trying to drink from a fire hose. Specialized stream processing is required to process these data in such a way as to derive insights quickly and respond to changes in realtime.

The idea behind this project is to demonstrate a streaming processing pipeline of Twitter data. This project implements a simple word counter to parse tweets into machine-readable words, count the occurrences of each of those words, and then store those counts in a database. This framework could easily be expanded to process and filter data for business purposes.

Architecture

Software/Dependencies

The backbone of this project is Apache Storm. Apache Storm is stream processing software that is modular and robust to failures. For this project, we are also using streamparse, which allows us to use python code to run Storm jobs. Storm uses Java primarily (and other JVM-supported languages), but streamparse allows us to iterate more quickly with a more user-friendly language. This whole project could be written in Java with purely Apache Storm, but Python is much easier to prototype in.

With Apache Storm, the user defines spouts (which are the sources of streaming data) and bolts (which process the data). In this case, each spout is a Twitter stream that uses the Python package Tweepy to pull data from Twitter. There are two bolts: one to parse the tweets into individual words, and another to count those emitted words and log them to the database.

There are three spouts to pull data from Twitter. These spouts emit full, unedited tweets. There are then three bolts to parse the tweets. These bolts use the shuffle grouping, so that the tweets from the spouts are distributed evenly across the parsing bolts. There are two bolts to count the words. These bolts use a "fields" grouping, so that all instances of a given word will go the the same bolt.

This makes the counting process easier. The words are collected, buffered, and are written to the database.

In this project, the data write to a PostgreSQL database. PostgreSQL is an open source and feature-filled database that is easy to write to and query from. We are using the Python package Psycopg2 to interact with Postgres. We can create our database and table in Python using Psycopg2, write to it from inside our streamparse bolt, and then query the database from Python CLI scripts. Using Python end-to-end makes for a cohesive and coherent project.

To query the database, there are two scripts (finalresults.py and histogram.py) that use the module FetchResults that was written for this exercise. FetchResults works inside a context manager so safely open and close the connection to the database, and provides some sorting options. The finalresults.py script can be run with the flags `--numerical` to sort by count instead of alphabetically and `--desc` to sort in descending order instead of ascending order.

Relevant directories and files

The code is organized into one streamparse project, 2 python packages, and other individual scripts for setup/querying. Most importantly, the streamparse code is in the extweetwordcount directory. To run the streaming code, the user should enter that directory and run the command `sparse run`.

Streamparse and other scripts use the credentials, and FetchResults modules (created for this exercise). The credentials package mainly exists for running the hello-stream-twitter.py file for testing your Twitter credentials. The FetchResults package provides a nice interface to the database. The finalresults.py and histogram.py both import this package to query the database and report their results.

Also of potential interest, in the wordcount.py bolt is the pgUpdater class. I wrote this to more easily and safely interact with the Postgres database. The PostgresUpdater class acts as a buffer to store words and update the database in batches, rather than writing every word to the database upon receipt.

Rather than update the database upon receipt of every word, we maintain a buffer of words and their counts. When a single word has been encountered a specified number of times, we update the database entry for that word and delete the word from our buffer.

If, instead, we have a total number of different words in our buffer that exceeds some threshold, we dump our full buffer to the database. This will make sure that our buffer sends the less common words to the database if it is taking them a long time to reach the threshold to be sent individually.

Directory structure:

```
.
+---- Architecture.md
+---- Architecture.pdf
+---- credentials
|   `---- Twittercredentials.py
+---- extweetwordcount
|   +---- config.json
|   +---- example.config
|   +---- fabfile.py
|   +---- project.clj
|   +---- README.md
|   +---- src
|   |   +---- bolts
|   |   |   +---- parse.py
|   |   |   +---- postgres
|   |   |   |   `---- postgresUpdater.py~
|   |   |   `---- wordcount.py
|   |   `---- spouts
|   |       `---- tweets.py
|   +---- tasks.py
|   +---- topologies
|   |   `---- tweetwordcount.clj
|   `---- virtualenvs
|       `---- tweetwordcount.txt
+---- FetchResults
|   +---- FetchResults.py
+---- finalresults.py
+---- hello-stream-twitter.py
+---- histogram.py
+---- postgres_credentials.config
+---- README.md
+---- setup_config.sh
+---- setup_database.py
+---- setup.sh
+---- startup.sh
+---- twitter_credentials.config
```