SLS Energy

Regional ICT Center of Excellence Bldg, KG 11 Av
Plot No A8, Kigali Special Economic Zone Phase II
Gasabo, Kigali, Rwanda
Tel: +250 788 69 8888
Email: team@slsenergy.io

Introduction

After making a huge profit for the Massive Surveillance Bureau, you realize that your software engineering skills are being undervalued. To maximize your own profit, you launch a cloud-based startup company with one or two colleagues from SLS.

A client has approached your company and several other companies to compete on a project to build a web service for analyzing Twitter data. The client has over one 100MB of raw tweets for your consumption. Your responsibility is to design, develop and deploy a web service that meets the throughput, budget and query requirements of the client. You manage to convince your team to join this competition to demonstrate your superior cloud skills to your potential client by winning the competition.

In this competition, your team needs to build (and optimize) a web service with two tier, a web-tier serving http requests and a storage-tier serving different data queries, as shown in

Figure 1.0.

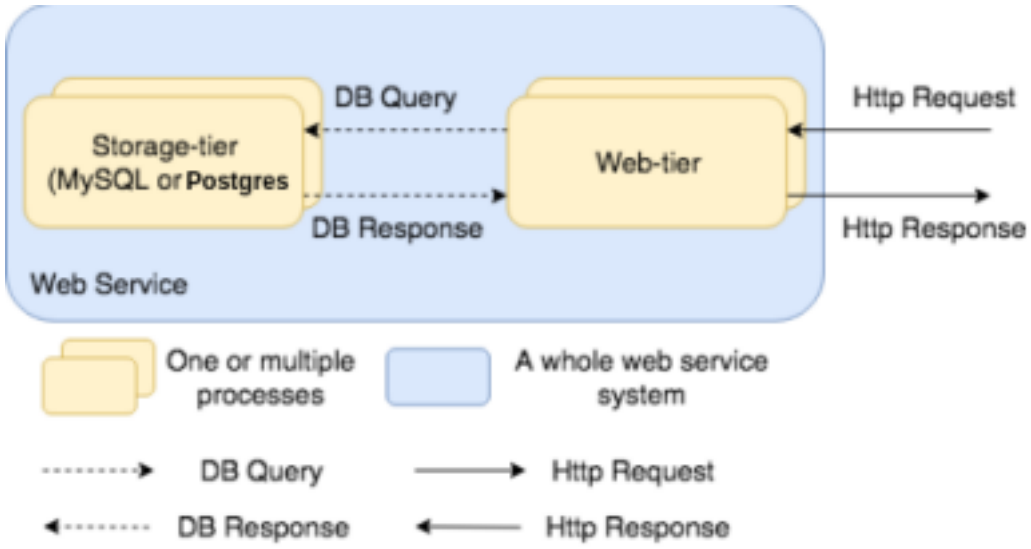

Figure 1.0: The runtime view of a multi-tier web service system

Your web service design should follow these guidelines:

1. A Web Tier: This should be a web service able to receive and respond to queries. Specifically, the service should handle incoming HTTP GET requests and provide suitable responses (as defined in the Query Types section below).
    1. Users access your service using an HTTP GET request through an endpoint

URL. Different URLs are needed for each query type, which are shown in the Query Types section. Query parameters are included within the URL string of the HTTP request.

2. An appropriate response should be formulated for each type of query. The response format should be followed exactly; otherwise, your web service will not provide acceptable responses when tested with our load generator and testing system.
3. The web service should run smoothly for the entire test period, which lasts for several hours.
4. The web service must not refuse queries and should tolerate a heavy load. 2. A Storage Tier: This is used to store the data to be queried.
    1. You will evaluate using any SQL db preferably MySQL or Postgres.
    2. You should compare their performance for different query types for different dataset sizes. Your web service should meet the requirements for throughput and cost/hour for queries at a provided workload.

Dataset

The dataset is available here:

The dataset will be larger than 100MB. There may be **duplicate** or **malformed** records that you need to account for.

The dataset is in JSON format. Each line is a JSON object representing a tweet. Twitter's documentation has a good description of the data format for tweets and related entities in this format. Here is a link to the Twitter API.

Please note that since the tweet texts in the JSON objects are encoded with unicode, different libraries might parse the text slightly differently. To ensure your correctness, we recommend that you use the following libraries to parse your data.

Bug Report

If you encounter any bugs in this system or have any suggestions for improvement, feel free to document it and share it with us via this email:<Chris email goes here>.

Overview of Tasks

Web Tier

This task requires you to build the web-tier of the web service. The web-tier should accept RESTful requests and send back responses.

Recommendations

It will be wise to ensure that your system comes close to satisfying the minimum throughput requirement of heartbeat requests before you move forward.

ETL

This task requires you to load the Twitter dataset into the database using the extract, transform and load (ETL) process in data warehousing. In the extract step, you will extract data from an outside source. For this phase, the outside source is a JSON Twitter dataset of tweets shared previously, containing about 39 thousand tweets. The transform step applies a series of functions on the extracted data to realize the data needed in the target database. The transform step relies on the schema design of the target database. The load phase loads the data into the target database.

You will have to carefully design the ETL process using your local resources. If you feel you don't need this step, feel free to skip it.

Hints

Think about the schema design of your database before attempting the ETL job. How to do ETL correctly and efficiently will be a critical part of your success in this project. Notice that ETL on a large dataset could take over 1h for just a single run, so it will be very painful to do it more than once, although this may be inevitable since you will be refining your schema throughout your development. Think about possible ways to reduce the time and cost of the ETL job since you might have to run it several times.

You may find your ETL job extremely time consuming because of the large dataset and/or the poor design of your ETL process. Due to many reasons that could lead to the failure of your ETL job, please start thinking about your database schema and your ETL job as early as possible.

Try to utilize parallelism as much as possible, loading the data with a single process/thread is a waste of time and computing resources.

Question 1
(User Recommendation System)

Introduction

When you follow someone on Twitter, Facebook or Instagram, the website may recommend some close friends of this user to you. Your web service will implement a similar functionality based on the Twitter dataset we collected.

The request of Q2 contains (1) a user ID, (2) contact tweet type `reply|retweet|both]`, (3) percent-encoded phrase, and (4) a hashtag.

You need to respond with information of every contacted user with a positive ranking score in **descending order**. The ranking score is defined later in this write-up. **For each contacted user with a positive score, you will return (1) user ID, (2) latest screen_name, (3) latest description, and (4) latest contact tweet of the type specified in the query and with the user in the query.**

Request Format

The format is,

```
GET /q2?user_id=<ID>&type=<TYPE>&phrase=<PHRASE>&hashtag=<HASHTAG>
```

An example is,

```
GET /q2?user_id=10000123&type=retweet&phrase=hello%20cc&hashtag=rwa
```

Response format

```
<TEAMID>,<TEAM_AWS_ACCOUNT_ID>\n
user_id_1\tscreen_name_1\tdescription_1\tcontact_tweet_text\
```

```
n
user_id_2\tscreen_name_2\tdescription_2\tcontact_tweet_text
```

where `\n` is the newline character, and `\t` is the tab character. The phrase is percent-encoded. **Each line ends with `\n` except the last one.**

For example,

```
TeamCoolCloud,1234-0000-0001
10000124\tAlanTuring\tComputer Scientist\tI propose to consider the question, 'Can machines think?'\n
10000125\tDijkstra\tAlso a computer scientist\tSimplicity is prerequisite for reliability.
```

Tweet object

Before going into the details, let us understand what a tweet object contains.

- Tweet ID is in `id` or `id_str` field
- Tweet content is in the `text` field
- If a tweet is a **reply** (e.g. A replies to B), then the ID of user B is in `in_reply_to_user_id` or `in_reply_to_user_id_str`
- If a tweet is a **retweet**, the original tweet object is stored in `retweeted_status` ● Hashtags are stored in `entities.hashtags`
- Sender information is stored as a user object in `user`

Please refer to the official Twitter API documentation provided in the links above for more information.

Language of Tweets

For Query 2, we only consider Tweets of these languages:

```
| language | code in lang field |
| ---------- | ------------------ |
| Arabic | ar |
| English | en |
| French | fr |
```

```
| Indonesian | in |
| Portuguese | pt |
| Spanish | es |
| Turkish | tr |
| Japanese | ja |
```

There is a field `lang` in each tweet object. Filter out Tweets whose `lang` field is NOT listed above.

Duplicate Tweets

For duplicate tweets (those with the same tweet_id), retain only one of them.

Malformed Tweets

Tweets satisfying any one of these properties are malformed and should be filtered out.

- Cannot be parsed as a JSON object
- Both `id` and `id_str` of the tweet object are missing or null
- Both `id` and `id_str` of the user object are missing or null
- `created_at` is missing or null
- `text` is missing or null or empty_string
- `hashtag` array missing or null or of length zero/empty

Contact Tweets

Definitions: A **contact tweet** is a tweet that is either a reply tweet or a retweet. A tweet is a **reply tweet** if `in_reply_to_user_id` is not null. A tweet is a **retweet** if `retweeted_staus` is not null.

Note: A tweet cannot be both a reply tweet and a retweet.

Field names below are for the ease of demonstration. Refer to the Twitter API for the exact field name. Say we have the following tweets:

```
| tweet_id | user_id | content | reply_to_id | retweet_to_id |
| -------- | ------- | --------- | ----------- | -------------
```

```
| | 01 | 15618 | cloud | 15213 | null | | 02 | 15640 |
computing | null | 15319 | | 03 | 15513 | is | 15213 | null | |
04 | 15513 | fun | null | null |
```

Then we have the followings:

```
| user_id | contact_tweet_id | contacted_user |
| ------- | ---------------- | -------------- |
| 15213 | 01, 03 | 15618, 15513 |
| 15513 | 03 | 15213 |
| 15319 | 02 | 15640 |
| 15618 | 01 | 15213 |
| 15640 | 02 | 15319 |
```

User Information

User information is stored in a user object as specified in Tweet API and it can appear in **tweet** and **retweeted_status** objects. Therefore,

- For any Tweet, we can find the sender information in `user`
- For a retweet, we can additionally find the original poster's information in `retweeted_status.user`

However, given that our tweet dataset is chronological, some user objects may be outdated and you will need to find the latest information for a user if any. Please use `created_at` in the tweet object to get the most updated user information.

Note that for the user object stored in `retweeted_status`, **use `created_at` in the retweeting object** instead of the one in the `retweeted_status`. For users that appear only in `in_reply_to_user_id` field, return `screen_name` `""` and description `""` if necessary.

Ranking Algorithms

Going back to a use case of our User Recommendation System, a Twitter user, say Alpha, follows Beta. The system should recommend close friends of Alpha that Beta may be interested in. Our ranking system factors in the closeness between Alpha and his friends as well as the interests of Beta. Specifically, the ranking algorithm consists of three parts:

interaction score, hashtag score, and keyword score.

- Interaction score calculates the frequency of interactions between two users (Or Alpha and his friends in the above narratives).
- Hashtag score calculates the common interest shared by two users (Or Alpha and his friends in the above narratives), as reflected by the hashtags in the tweets they posted. ● Keyword score factors in the queried key phrase and hashtag (Or Beta's interests in the above narratives).

The final ranking score between two users is calculated as `final_score = interaction_score * hashtag_score * keywords_score`

For the calculations below, log base is e, and keep 5 decimal points of precision for the final score rounding half up. We will ignore user pairs with a final score of zero. The definitions of the scores are described below.

1. Interaction Score

There are two types of interactions: reply and retweet. The more replies and retweets between two users, the higher their interaction score. **Interaction score** is calculated as `log(1 + 2 * reply_count + retweet_count)` Some examples are below:

1. A replied B 4 times; A retweeted B 3 times `log(1 + 2*4 + 1*3) = 2.485`
2. A replied B twice; B replied A once `log(1 + 2*(2+1) + 1*0) = 1.946` 3. A retweeted B once `log(1 + 2*0 + 1*1) = 0.693`
4. no replies/retweets between A and B `log(1 + 2*0 + 1*0) = 0`

You may spot cases where the reply or retweet is to the same user of the original tweet (self-reply or self-retweet). We will actually count those contact tweets and hence someone may have an interaction score with oneself. It is interesting to see if a user ends up having a higher score than the user's friends.

2. Hashtag Score

The hashtag score is based on the frequency of the same hashtags **(case insensitive)** *posted* by users among all the valid tweets. By *posting*, we mean that it is the user who is stored in user object of a tweet object. Very popular hashtags may not really demonstrate common interests between two users. As such, we do not consider them when calculating the hashtag score. We provide a list of hashtag (case insensitive) to be excluded. **(Please set your browser, parser or text editor to be encoding awareness**

when reading this list!) Note that for the excluded hashtags, we do not filter out those tweets. We just ignore those hashtags in the hashtag score calculation. Here are a few examples. Assume hashtag `zipcode` is a very popular hashtag that we exclude.

```
|  sender_uid  |  reply_to_uid  |  retweet_to_uid  |  hashtags  |  |
---------- | ---------- | ---------- | ---------- |
| 15619
      | 15213 | null | Aws, azure, ZIPCODE |
| 15619
      | null | 15319 | Cloud, Azure |
| 15619
      | null | null | Cloud, GCP |
| 15619
      | null | null | cloud, aws |
| 15319 | 15213 | null | rwa, us | | 15319 | null | 15619
          | AZure |
| 15319 | 15513 | null | Cloud, GCP | | 15319 | null | null | aWs,
zipcode, CLOUD | | 15513 | 15213 | null | rwa, us | | 15513 | null
| 15619
          | haha, ZIPcode |
| 15513 | 10601 | null | zipcode |
```

Given all the tweets above, the hashtag score of the user pairs below are:

```
| uid_1 | uid_2 | same_tag_count | explanation | | ----- | ----- |
-------------- | ---------- |
| 15619
 | 15319 | 13 | aws=3, cloud=5, azure=3, GCP=2 |
| 15619
 | 15513 | 0 | no match |
| 15319 | 15513 | 4 | rwa=2, us=2 |
```

The final `hashtag_score` is calculated as follows.

- If `same_tag_count > 10`, then `hashtag_score = 1 + log(1 + same_tag_count - 10)`.
- Else, `hashtag_score = 1`

We would like you to produce a histogram of hashtag frequency on log scale among all the valid tweet **without hashtag filtering**. Given this histogram, think about how it informs your ETL process in your phase 1 report (Is this histogram common in social data? What if we don't filter out the common tweet? What are good practices for ETL time and resource estimation?)

Note: For the cases of self-reply or self-retweet (the reply or retweet is to the same user of the original tweet), you can assume the **hashtag score** will **always be 1**.

3. Keywords Score

Keywords score is calculated by counting the total number of matches of phrase and also hashtag (provided in the query) across the **contact tweets** of a specific *type*. The *type* is given in the query, and valid values are `[reply|retweet|both]`. Going back to our use case and say Beta is a fan of
cloud computing, hence we may want to examine the content and hashtags of the contact tweets between Alpha and its friends to see with whom Alpha discusses cloud computing the most and recommend it to Beta.

When calculating the keywords score, we count in all the hashtags including those in the filtering list. In the query,

- if *type* equals `reply`, then the score is calculated only based on reply tweets ● if *type* equals `retweet`, then the score is calculated on retweets only
- if *type* equals `both`, then the score is calculated based on all contacted

tweets **Matching rule for the phrase: case sensitive match.**

For example, for phrase `life`

- `lifeisgood` has a match
- `it's my Lifelife` has a match
- `that's my lifelife lesson` has two matches

Another example, for phrase `haha`

- `hahaha` has 2 matches (overlapping matches are possible)
- `haHaha` has no matches
- `Haha bahaha` has 1 match

Matching rule for the hashtag: case insensitive exact match.

For example, if `hashtag` in the request is `cloud`, and a tweet has hashtags `#Cloud #CLOUD #CLOUD #rwanda` (note that duplicate tags are allowed), then this tweet will add 3 to `number_of_matches`.

Between two users, if there are no contact tweets of the type specified in the

query, then `keywords_score = 0`.

Otherwise, `keywords_score = 1 + log(number_of_matches + 1)`.

Keywords score example

Given request & tweet: `GET /q2?user_id=15513&type=retweet&phrase=hello%20cc&hashtag=rwanda`

```
| tweet_id | sender_uid | content | reply_to_uid | retweet_to_uid | hashtags |
| -------- | ---------- | ------------- | ------------ | -------------- | ------------ |
| 01 | 15618 | hello cc, fun | 15513 | null | rwanda, pitt |
| 02 | 15213 | computing | null | 15513 | RWA |
| 03 | 15513 | is | 15213 | null | PA, US, RWA |
| 04 | 15513 | hello cc!! | null | 15640 | rWa, rwa |
```

Result:

```
| user_id | phrase_match | hashtag_match | keywords_score |
| ------- | ------------ | ------------- | -------------- |
| 15618 | null | null | 0 |
| 15213 | 0 | 1 | 1 + log(1+1) |
| 15640 | 1 | 2 | 1 + log(1+3) |
```

Final Score

Again, the calculation of the final score is `final_score = interaction_score * hashtag_score * keywords_score`

We will ignore user pairs with final score equals to 0.

Hint: Please think about the scoring criteria and the implication of threshold before starting your ETL pipeline!

Ordering

Your web service should return the most up-to-date information of users and their latest contact tweet with the user in the query ordered by the score calculated above. Break ties by user ID descending in numerical order.

For the latest contact tweets, break the tie by tweet ID descending in numerical order if they have the same timestamp.

Development flow

- Use Github as the repos you will use to share code.
- Write unit and integration where possible