Fernfachhochschule Schweiz
Zürich | Basel | Bern | Brig

Mitglied der SUPSI

1.Semesterarbeit

Erste Schritte - Twitter API Stream

Aufgabestellungen aus dem Fach EDS – Einführung in Data Science

Inhaltsverzeichnis

1. ABSTRACT 2
2. EINLEITUNG 3
3. MATERIAL 3
4. METHODEN 3
5. RESULTATE 5
6. DISKUSSION 7
7. QUELLEN 8
8. ANHANG 8

1. Abstract

Ein erster Einblick in die Twitter API wird erarbeitet mit Fokus auf die Einrichtung eines Streams. Erreicht wird dies mit Python auf der PyCharm IDE. Verwendet wird die Authentifizierung über OAuth 2.0 und das Twitter API 2.0 zur Erstellung eines Streams. Betrachtet wird die Angabe von Regeln, um die gewünschten Tweets live abzurufen. Weiter kann bestimmt werden welche zusätzlichen Felder zurückgegeben werden ausserhalb des Standards. Der Systemaufbau von Twitter wird aufgezeigt und darin die Verarbeitung von Tweets aufgezeigt. Eine Auflistung der Datenbanken und ihre Verwendung wird beschrieben.

2. Einleitung

Um mich vertraut mit der Twitter API zu machen, werden via Twitter Stream ausgewählte Topics abgerufen. Wo sind die Einschränkungen der API. Wie wird die Anfrage von Twitter abgearbeitet. Wäre dies ohne NoSQL Datenbanken machbar.

3. Material

Um auf den Stream zuzugreifen wurde mit PyCharm [1] und Python gearbeitet. Für mein Beispiel wird die Twitter API v2 [3] verwendet. Das Arbeiten mit der Twitter API setzt einen Twitter Developer Account [4] voraus zur Authentifizierung mit seinem persönlichen Bearer Token [2] via OAuth 2.0 [5]. Als Vorbereitung habe ich mich eingehend mit gefilterten Streams auseinandergesetzt. Zur Zeit dieser Arbeit ist das Twitter API v2 gerade in der Entstehung.

4. Methoden

Als Vorbereitung einen Stream aufzubauen braucht es die Informationen zur *authentification* und den persönlichen *rules*. Es werden Anfragen an die Twitter API v2 geschickt die mit dem *BEARER_TOKEN* der eigenen Twitter App authentifiziert werden.

authentification:

```
def create_headers(bearer_token):
    headers = {"Authorization": "Bearer {}".format(bearer_token)}
    return headers
```

In *headers* wird der persönliche BEARER_TOKEN mitgegeben. Dieser muss der Applikation bereitgestellt werden. Der eleganteste Weg dazu ist diesen der Applikation als Systemvariable zugänglich zu machen. In meinem Beispiel habe ich die Variable fest vergeben im Code. Requests werden mit OAuth 2.0 authentifiziert.

rules:

```
def set_rules(headers, delete, bearer_token):
    # You can adjust the rules if needed
    sample_rules = [
        {"value": "#New40k", "tag": "#New40k"},
    ]
    payload = {"add": sample_rules}
    response = requests.post(
        "https://api.twitter.com/2/tweets/search/stream/rules",
        headers=headers,
        json=payload,
    )
```

Um an die gewünschten Tweets zu gelangen, müssen Regeln festgelegt werden.

```
{"value": "#New40k", "tag": "New40k"},
```

In diesem Beispiel wurde nach Tweets mit dem Hashtag '#New40k' gesucht. Die Regel trägt die Bezeichnung 'New40k'. In den gefundenen Tweets wird immer auch die Regel angegeben aufgrund derer er gefunden wurde.

Bei der Initialisierung der Streams können zusätzlich die zurückgegebenen Felder bestimmt werden. Standardmässig werden die Felder *id* und *text* zurückgegeben.

```
def get_stream(headers, set, bearer_token):
    response = requests.get(
        "https://api.twitter.com/2/tweets/search/stream",
headers=headers, stream=True,
    )
    print(response.status_code)
    if response.status_code != 200:
        raise Exception(
            "Cannot delete rules (HTTP {}): {}".format(
                response.status_code, response.text
            )
        )
    for response_line in response.iter_lines():
        if response_line:
            json_response = json.loads(response_line)
            print(json.dumps(json_response, indent=4, sort_keys=True))
```

Für die Standardabfrage reicht das zeigen auf */tweets/search/stream*. Um weitere Felder hinzuzufügen braucht es die Angabe der zugehörigen Felder. Diese setzen sich aus einem *key* und *value* zusammen. Der *key* bestimmt von welchem Twitter-Objekt und das *value* welche Eigenschaft davon gewählt wird.

```
"https://api.twitter.com/2/tweets/search/stream?tweet.fields=
created_at&expansions=author_id&user.fields=created_at"
```

Hier wurden zur Abfrage drei weitere Felder hinzugefügt. Die genaue Bedeutung dieser und weiter Felder können in der Dokumentation unter *Data Dictionary* [6] nachgeschaut werden.

5. Resultate

So ist es nun möglich gezielt Tweets abzurufen via Stream. Die abgefangenen Tweets werden im Beispiel in der Konsole ausgegeben im JSON-Format.


```
{
   "data": {
      "author_id": "1605414661",
      "created_at": "2020-08-26T11:44:49.000Z",
      "id": "1298587197021593604",
      "text": "RT @Gonders: Annnnnd, Bladeguard number 3. The gang ready to.... I
dunno... guard something? \n\n#warmongers #WarhammerCommunity
#wepaintmini\u2026"
   },
   "includes": {
      "users": [
         {
            "created_at": "2013-07-19T08:05:36.000Z",
            "id": "1605414661",
            "name": "Sim Lauren",
            "username": "PaintySim"
         }
      ]
   },
   "matching_rules": [
      {
         "id": 1298571939083157505,
         "tag": "New40k"
      }
   ]
}
```


Einige Beschränkungen sind dem Twitter API v2 gegeben. Für jede Applikation ist nur eine Verbindung gleichzeitig erlaubt. Pro 15 Minuten sind maximal 50 Verbindungsanfragen sowie 450 Regeländerungen erlaubt. Pro Abfrage können maximal 25 Regeln mitgegeben werden. Hierbei können bei grösseren Abfragen Verzögerungen von bis zu 10 Sekunden auftreten.

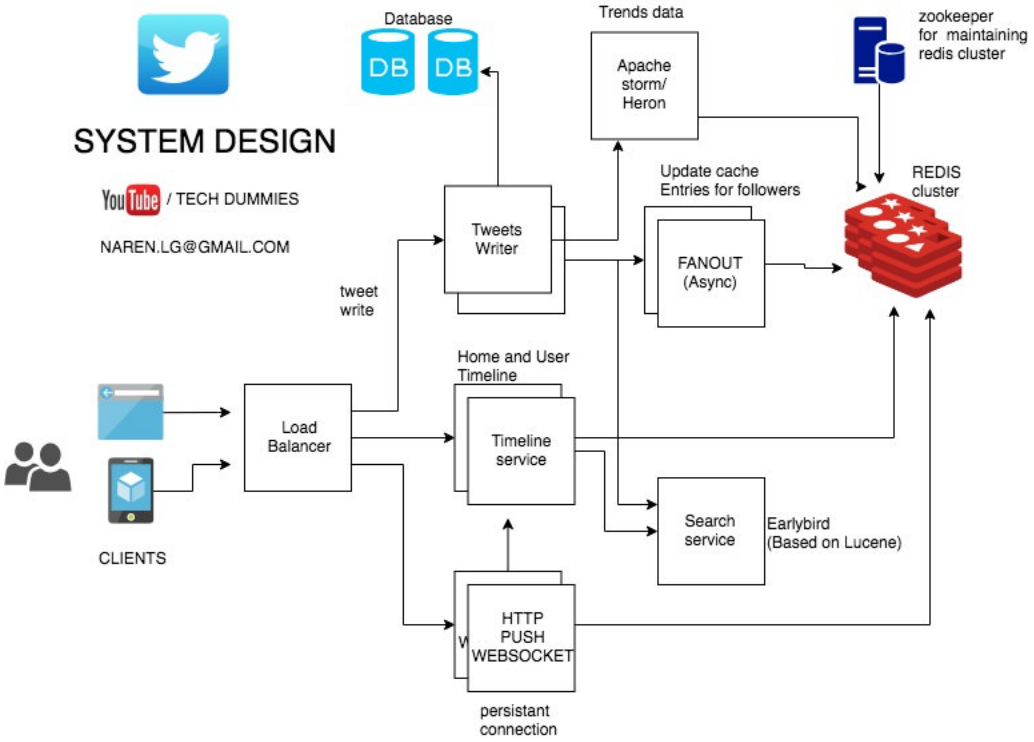

Figure 1 - Twitter System Design

Wie genau ein Tweet verarbeitet wird, bestimmt die Anzahl der Follower. Erreicht ein User Millionen von Followers erhält er den Celebritie-Status. Es würde zuviel Zeit und Rechenleistung in Anspruch nehmen jeden Tweet eines solchen Benutzer in jede Timeline seiner Followers zu schreiben.

Benutzer ohne Celebritie-Status:
Schickt ein Benutzer einen Tweet ab gelangt er zum Load Balancer. Dieser verteilt ihn an an DB und Cache. Durch einen Fanout schreibt Redis den Tweet in die Timelines der Followers des Tweeters. So haben alle Followers den neuen Tweet beim nächsten Aufruf bereits in ihrer Timeline.

Benutzer mit Celebritie-Status:
Um die Verteilung von Tweets an Millionen Followers gleichzeitig zu gewährleisten, werden lediglich Vermerke verteilt. Befindet sich ein Celebritie in der Liste der Benutzer denen gefolgt wird, werden diese beim Aufruf der eigenen Timeline in Realtime geladen. So wird die Belastung auf das System verteilt, auf die versetzte Abrufzeit der Follower.

Gemäss [9] werden folgende Datenbanken bei Twitter verwendet.

- Gizzard ist das verteilte Datenspeicher-Framework von Twitter, das auf MySQL (InnoDB) aufbaut. InnoDB wurde gewählt, weil es keine Daten korrumpiert. Gizzard ist lediglich ein Datenspeicher. Daten werden gespeichert und wieder ausgelesen werden.
- Kassandra wird für Schreibvorgänge mit hoher Geschwindigkeit und Lesevorgänge mit niedriger Geschwindigkeit verwendet. Einer der grössten Vorteile ist, dass Cassandra auf preiswerterer Hardware als MySQL läuft.
- Hadoop wird benutzt, um unstrukturierte und große Datensätze mit Hunderten von Milliarden von Zeilen zu verarbeiten.
- Vertica wird für Analysen, große Aggregationen und Joins verwendet.

6. Diskussion

Es gibt mehrere Möglichkeiten einen Twitter Stream abzurufen. Ich habe mich für den Twitter eigene API 2.0 entschieden und ich bin zufrieden mit dem Ergebnis. Es würde die Möglichkeit bestehen mit Tweepy [10] auf dasselbe Ergebnis zu kommen. Hier kommen die eigenen Präferenzen ins Spiel. Mit dem erlangten Wissen aus dieser Arbeit kann nun weitergearbeitet werden.
Die Grenzen, die durch die Twitter-API gesetzt werden, müssen für zukünftige Arbeiten berücksichtigt werden. Es konnte ein Verständnis aufgebaut werden wie ein Tweet abgearbeitet wird von Twitter. NoSQL Datenbanken sind ein fester Bestandteil der Systemarchitektur. Die Bewältigung einer solchen Flut an Daten macht es unumgänglich diese Technologie einzusetzen.

7. Quellen

[1]	PyCharm: https://www.jetbrains.com/pycharm/
[2]	TwitterApp: API Key, API Key Secret https://developer.twitter.com/en/portal/projects/1292869185048018945/apps/18545563/keys
[3]	Twitter Streaming API: https://developer.twitter.com/en/docs/twitter-api/tweets/filtered-stream/introduction
[4]	Twitter Dev: https://developer.twitter.com/en
[5]	OAuth 2.0: https://developer.twitter.com/en/docs/authentication/oauth-2-0
[6]	Data Dictionary: https://developer.twitter.com/en/docs/twitter-api/data-dictionary/object-model/tweet
[7]	Tweet Object: https://developer.twitter.com/en/docs/twitter-api/data-dictionary/object-model/tweet
[8]	Twitter API Grenzen: https://developer.twitter.com/en/docs/twitter-api/tweets/filtered-stream/migrate
[9]	Twitter Systemaufbau: https://medium.com/@narengowda/system-design-for-twitter-e737284afc95
[10]	Tweepy: http://docs.tweepy.org/en/latest/

8. Anhang

- Beispielcode [3]

```
import requests
import os
import json

# To set your enviornment variables in your terminal run the following line:
# export 'BEARER_TOKEN'='<your_bearer_token>'
BEARER_TOKEN='YOUR_BEARER_TOKEN'

def create_headers(bearer_token):
    headers = {"Authorization": "Bearer {}".format(bearer_token)}
    return headers
def get_rules(headers, bearer_token):
    response = requests.get(
        "https://api.twitter.com/2/tweets/search/stream/rules", headers=headers
    )
    if response.status_code != 200:
        raise Exception(
            "Cannot get rules (HTTP {}): {}".format(response.status_code, response.text)
        )
    print(json.dumps(response.json()))
    return response.json()
def delete_all_rules(headers, bearer_token, rules):
    if rules is None or "data" not in rules:
        return None
    ids = list(map(lambda rule: rule["id"], rules["data"]))
    payload = {"delete": {"ids": ids}}
    response = requests.post(
        "https://api.twitter.com/2/tweets/search/stream/rules",
        headers=headers,
        json=payload
    )
    if response.status_code != 200:
        raise Exception(
            "Cannot delete rules (HTTP {}): {}".format(
                response.status_code, response.text
            )
        )
    print(json.dumps(response.json()))
def set_rules(headers, delete, bearer_token):
    # You can adjust the rules if needed
    sample_rules = [
        {"value": "#New40k", "tag": "New40k"},
    ]
    payload = {"add": sample_rules}
    response = requests.post(
        "https://api.twitter.com/2/tweets/search/stream/rules",
        headers=headers,
        json=payload,
    )
    if response.status_code != 201:
        raise Exception(
            "Cannot add rules (HTTP {}): {}".format(response.status_code, response.text)
        )
    print(json.dumps(response.json()))
def get_stream(headers, set, bearer_token):
    response = requests.get(

"https://api.twitter.com/2/tweets/search/stream?tweet.fields=created_at&expansions=author_id&user.fields=creat
ed_at", headers=headers, stream=True,
    )
    print(response.status_code)
    if response.status_code != 200:
        raise Exception(
            "Cannot delete rules (HTTP {}): {}".format(
                response.status_code, response.text
            )
        )
    for response_line in response.iter_lines():
        if response_line:
            json_response = json.loads(response_line)
            print(json.dumps(json_response, indent=4, sort_keys=True))
def main():
    #bearer_token = os.environ.get("BEARER_TOKEN") #Delete Comment if Environment Variable is set
    bearer_token = BEARER_TOKEN #Add Comment if Environment Variable is set
    headers = create_headers(bearer_token)
    rules = get_rules(headers, bearer_token)
    delete = delete_all_rules(headers, bearer_token, rules)
    set = set_rules(headers, delete, bearer_token)
    get_stream(headers, set, bearer_token)

if __name__ == "__main__":
    main()
```