Twitter Analytics Documentation

Release 0.1

Deepak Saini, Abhishek Gupta

Jun 28, 2018

CONTENTS:

Twitter generates millions of tweets each day. A vast amount of information is hence available for different kinds of analyses. However, this also brings up three challenges. First, the system needs to be efficient enough to be able to absorb data at such high rates. Second, to be able to perform complex analysis on this data, it needs an appropriate representation in the database. Third, the system needs to provide an intuitive abstraction of the data so that the users can specify various complex analyses without having to write complex programs using different software tools. This will allow even slightly non-technical users to be able to use the system.

In this demonstration, we introduce a system that tries to tackle the above challenges. The system uses a couple of data stores - Neo4j and MongoDB - to persist the data in a way that allows complex historical queries to be answered efficiently. It also provides an intuitive abstract representation of this data, allowing users to formulate complex queries. Also, running on streaming data, the system provides an abstraction which allows the user to specify real time events in the stream, for which he wishes to be notified. We demonstrate both of these abstractions using an example of each, on real world data.

NOTE : For all experiments mentioned in this document, we use a Intel i7 processor with 16 GB RAM.

CHAPTER

ONE

INTRODUCTION TO TWITTER ANALYTICS SYSTEM

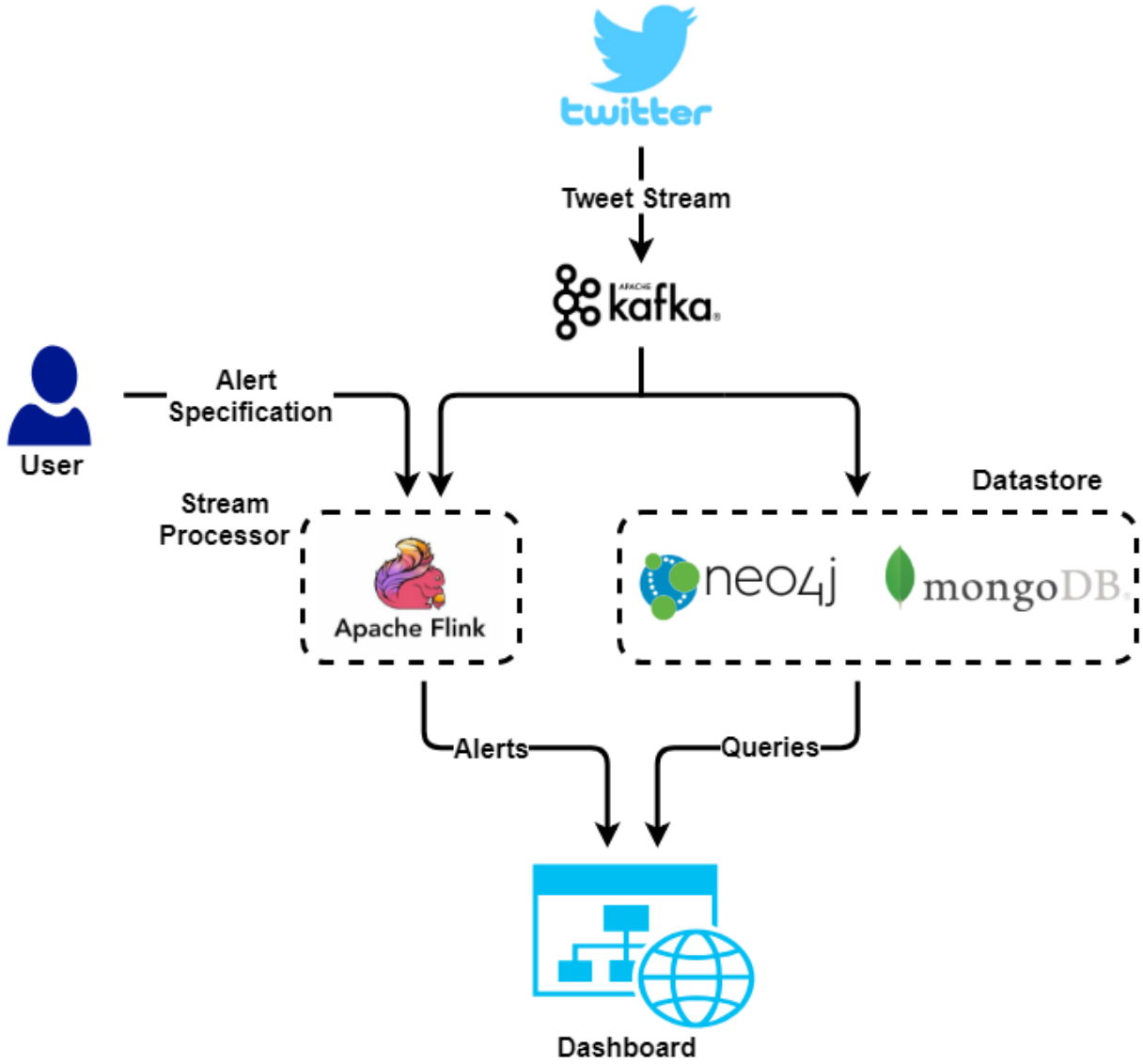
Nowadays social media generates huge amount of data which can be used to infer a number of trending social events and to understand the connections between different entities. One such useful platform is Twitter where users can follow other users and tweet on any topic, tagging it with hashtags, mentioning other users or using URLs. This data can be spatially represented as a network of entities (like users, tweets, hashtags and URLs) which are interconnected in complex ways. A number of programming tools are often used by developers to perform various tasks on a subset of this data.

However, in reality, this data is not static with time, giving it a temporal dimension as well. The system thus needs to capture both the spatially and temporally evolving aspects of this network in its database schema. The same abstraction also needs to be extended to the query specification process, making it easier to specify queries without writing queries in complex database specific languages. Hence, instead of limiting the users to a limited set of APIs, this system would allow any complex query to be specified.

Our system continuously streams tweets from the Twitter Streaming API, persists them into a database schema compliant with the above needs and gives users a couple of abstractions to work with. The first abstraction is over the live stream of tweets allowing them to detect custom live events (Eg. finding hashtags going viral) in the stream and get notified about them. The second abstraction is over the historical view over the data stored from the stream. The second abstraction looks at the data as a network of users, tweets and their attributes, along with the network's temporal dimension, allowing users to specify complex queries in an abstract way.

Together, these two abstractions would provide an intuitive analytics platform for the users.

1.1 Major parts of the system



Lets begin by describing the major components of the system.

- The streaming tweet collector: We have a connection to the twitter streaming API which keeps on collecting tweets from the twitter pipe. For more details refer to the section on twitter stream.
- The alert generation system: The collected tweets are pushed to **Apache Kafka** for downstream processes. This tweet stream is processed by **Apache Flink**, which is an open-source, distributed and high-performance stream processing framework, to look for user specified alerts in the tweet stream.
- The datastores: The stream is also persisted in a couple of data stores to make queries later. **Neo4j**, which is a graph database management system supported by the query language Cypher, is used to store network based information from these tweets. **MongoDB**, which is a document oriented database, is used to store document based information to answer simpler aggregate queries.
- The dashboard: These alerts and queries are then accessible to the user from a web application based dashboard. Please refer to the section *Dashboard Website* in which we explain the functionalities of the system through use cases.

CHAPTER

TWO

READ DATA FROM TWITTER API

There are a couple of ways to read data from the Twitter APIs:

- User Timeline: Fetching the timeline (all tweets) of a given user till that time. The Twitter API end-point for this is GET statuses/user_timeline
- Stream Sample: Streaming a random 1% sample of all live tweets. The Twitter API end-point for the same is GET statuses/sample.

If you wish to gather data of a specific set of users, then you have to use the User Timeline API, otherwise for live stream use the Stream Sample API. We experimented with both the techniques.

To have access to Twitter APIs, you need to create an account on apps.twitter.com to use their OAuth based authorization system. Details for the same can be found on this link. This will generate a unique set of Access Token, Access Secret, Consumer Key and Consumer Secret for you, which need to be used while making the API calls. We made use of Python Twitter Tools library which exposes Python functions which make the API call for us based on the parameters.

2.1 User Timeline API

For a given user, you can find the following:

- User information: Eg. no. of tweets, no. of followers, no. of friends, no. of likes, location etc.
- All tweets by the user till that point of time.
- All friends and followers of that user at that point of time.

Rate Limit: Each API endpoint has its own rate limit which limits the number of calls you can make to Twitter for that API in a period of 15 minutes. The rate limit can be checked here. You need to adhere to these rate limits, else your account may be blacklisted. We do this in our application as follows: For each API endpoint, we maintain an array which keeps the history of number of calls made in each minute in the past 15 minutes. For each request of an API call, we check the array, delete entries older than 15 minutes and keep checking the total number of requests in past 15 minutes until it is less than the rate limit threshold. Only then we make the API call.

Incremental data: You may need to periodically make calls to the Twitter API to get the new tweets that were tweeted by the users since the last time you called the API. Twitter provides a mechanism to do this by using the parameter "since_id" in the statuses/user_timeline API call. The API returns tweets that have an id > since_id. Thus you can keep store of the maximum tweet id that you have seen for each user and make the next API call using that as the value of since_id to get the new tweets.

Refer to the file: 'Read Twitter Stream/main.py' for the code. The main function expects a file containing a list of user screen names separated by new lines. It generates a folder the data as mentioned in the main function. Each time you run the file, it accumulates the new data in this directory.

2.2 Stream Sample API

The end-point GET statuses/sample is free, however it returns only about 1% of the actual stream. You can buy the enterprise version called Decahose to have access to 10% of the stream.

Batch writes: In order to speed up the process of persisting the tweets from memory to disk, we make use of batch writes. Our streaming application will keep buffering tweets till the batch size is reached. Once the the batch size is reached, it flushes all the buffered tweets to the current tweet file. Additionally, it keeps track of number of tweets written to the current tweet file and once it exceeds a threshold, it starts with a new file and starts flushing to it. This will prevent a single file from becoming too big in size.

Refer to the file 'Read Twitter Stream/streaming.py' for the code. The code will write the data in the same directory, flushing the data periodically to a file. After a threshold number of tweets have been written to the current output file, it generates a new file and starts flushing the tweets to it. This will prevent a single file from becoming too big in size.

2.3 User Timeline API Code Documentation

Here we provide a documentation of the code. Module to fetch data of a specified list of users. Data includes user's profile information, all tweets on user's timeline till now and list of ids of user's followers and friends.

In the main function, configure the path of the directory of the data folder to be created and the path of a file containing a list of user screen names separated by new lines. It generates a folder for the data as specified in the main function. Each time you run the file, it accumulates the new data in this directory.

Create a file named SECRETS which contains your Twitter OAuth related keys in the following order, separated by new lines: <ACCESS_TOKEN> <ACCESS_SECRET> <CONSUMER_KEY> <CONSUMER_SECRET>

Running the code:

- First ensure Python Twitter Tools is installed. (https://github.com/sixohsix/twitter)
- Before running, you may want to change the name of the file (containing the user screen names) in the main function. That file should contain one screen name in each line.

Command to run: `python main.py`

class userstimeline.**DateTimeEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: `json.encoder.JSONEncoder`

default(*o*)

encode(*o*)

Return a JSON string representation of a Python data structure.

```
>>> from json.encoder import JSONEncoder
>>> JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

item_separator = ', '

iterencode(*o*, *_one_shot=False*)

Encode the given object and yield each string representation as available.

For example:

```
for chunk in JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

key_separator = ': '

class userstimeline.**UserTimelineAPI**(*data_directory*)
Bases: object

Class to fetch data of a specified list of users.

clear_everyting()
Clears the data folder

fetch_persist_friends_and_followers(*user_screen_names*, *time*)
Fetches and persists users' friends and followers

Parameters

- **user_screen_names** – list of screen names of users
- **time** – wall clock time when this file started running

fetch_persist_tweets(*user_screen_names*, *time*, *type_*)
Fetches and persists tweets (excluding tweets already persisted)

Parameters

- **user_screen_names** – list of screen names of users
- **time** – wall clock time when this file started running
- **type** – one of 'tweets' or 'favourites' to fetch user's own tweets or favorited tweets respectively

fetch_persist_users(*user_screen_names*, *time*)
Fetches and persists user information (calling this multiple times will keep adding new entries so that you can compare over time)

Parameters

- **user_screen_names** – list of screen names of users
- **time** – wall clock time when this file started running

my_favourites_fetcher(*retry_no=0*, ***kwargs*)
Waits until rate limit is clear and then calls the Twitter API to fetch user's favorited tweets. Retries 3 times in case of exceptions.

Parameters

- **retry_no** – Number of retries done till now in case of exceptions
- **kwargs** – Contains screen_name, count (maximum 200 allowed), since_id (minimum id of tweet to look for), max_id (maximum id of tweet to look for)

Returns A list of tweet dictionaries

my_followers_fetcher(*retry_no=0*, ***kwargs*)
Waits until rate limit is clear and then calls the Twitter API to fetch user's followers' ids. Retries 3 times in case of exceptions.

Parameters

- **retry_no** – Number of retries done till now in case of exceptions

- **kwargs** – Contains screen_name, cursor (used for paginated results, -1 to fetch the latest batch)

Returns A dictionary containing 'ids' (ids of followers) and 'next_cursor' (cursor of next batch)

my_friends_fetcher(*retry_no=0*, ***kwargs*)
Waits until rate limit is clear and then calls the Twitter API to fetch user's friends' ids. Retries 3 times in case of exceptions.

Parameters

- **retry_no** – Number of retries done till now in case of exceptions
- **kwargs** – Contains screen_name, cursor (used for paginated results, -1 to fetch the latest batch)

Returns A dictionary containing 'ids' (ids of friends) and 'next_cursor' (cursor of next batch)

my_tweet_fetcher(*retry_no=0*, ***kwargs*)
Waits until rate limit is clear and then calls the Twitter API to fetch user's tweets. Retries 3 times in case of exceptions.

Parameters

- **retry_no** – Number of retries done till now in case of exceptions
- **kwargs** – Contains screen_name, count (maximum 200 allowed), since_id (minimum id of tweet to look for), max_id (maximum id of tweet to look for)

Returns A list of tweet dictionaries

my_user_fetcher(*retry_no=0*, ***kwargs*)
Waits until rate limit is clear and then calls the Twitter API to fetch the given users' info. Retries 3 times in case of exceptions.

Parameters

- **retry_no** – Number of retries done till now in case of exceptions
- **kwargs** – Contains screen_name which is comma separated list of screen names

Returns A list of user info dictionaries

wait_for_rate_limit(*type_*)
Common function for all API calls that waits (sleeps) until the restriction of rate limiting for that type is clear. For each request of each API call, it checks the corresponding array in COUNTS_DICT, deletes entries older than WINDOW_LEN minutes and keeps checking the total no of requests in past WINDOW_LEN minutes until it is less than the rate limit threshold set in COUNTS_LIMIT. Sleeps in between.

Parameters **type** – The type of API call. One of 'USERS_LOOKUP', 'TWEETS', 'FOLLOWERS', 'FRIENDS', 'FAVOURITES'.

userstimeline.**extract_hash_tags**(*screen_name*, *date_start*, *date_end*)

userstimeline.**get_user_screen_names**(*filename*)

userstimeline.**plot_user_field**(*screen_name*, *date_start*, *date_end*, *field_names*)

2.4 Stream Sample API Code Documentation

Here we provide a documentation of the code. Module to stream 1% sample of tweets using Twitter's Streaming API. The code will write the data in the same directory.

Create a file named SECRETS which contains your Twitter OAuth related keys in the following order, separated by new lines: <ACCESS_TOKEN> <ACCESS_SECRET> <CONSUMER_KEY> <CONSUMER_SECRET>

Running the code:

- First ensure Python Twitter Tools is installed. (https://github.com/sixohsix/twitter)
- Before running, you may set the following parameters inside the file:
  - tweet_count = 1000000000 # total number of tweets to read
  - batch_size = 10000 # how many tweets to write together in 1 go
  - file_tweet_count = 500000 # max no of tweets to write in 1 file

Command to run: `python streaming.py`

class streaming.**Logger**(*batch_size*, *file_tweet_count*)

Bases: object

Class to periodically write tweets to files and to log anything (automatically timestamped). The function write_tweet will keep buffering tweets till batch_size is reached. Once the batch size is reached, it flushes all the buffered tweets to the current tweet_file_name. Additionally, it keeps track of number of tweets written to the current file and once it exceeds the threshold file_tweet_count, it starts with a new tweet_file and starts flushing to it. This will prevent a single file from becoming too big in size.

configure_new_file()

Changes the configuration to start logging and writing tweets to new files (filenames will contain timestamp). Also puts a '[' at the beginning of the tweet file to start a new list.

finish_current_file()

Finshes the current tweet file by ending the list with a ']'.

log(*s*)

Logs the string supplied as argument to the current log file.

Parameters **s** – The string to log

write_tweet(*tweet*)

Adds the tweet to the buffer. If buffer is filled, then flushes the buffer to the current tweet file. If the current tweet file exceeds the tweet count threshold, finishes the current file and a configures a new one.

Parameters **tweet** – The tweet to write to file

streaming.**signal_handler**(*signal*, *frame*)

CHAPTER

THREE

PUTTING TWEETS TO KAFKA

The tweets read from the Twitter API are written to files on the disk for any further processing. These tweets are then read from the files and put on a Kafka topic ("tweets_topic") for the downstream processes. Ideally, in a running application, the tweets from the API should directly be put on the Kafka topic without writing to the files. However, collecting data once in the files helps in easy testing of different components.

We now provide the documentation for the application that reads tweets from the files and puts them to a Kafka topic.

3.1 Kafka Tweet Producer Documentation

Here we provide a documentation of the Kafka Tweet Producer. Module to read tweets from files and post them to a Kafka topic (currently "tweets_topic") for any downstream processes.

Note: Currently, only the Flink applications are reading tweets from this Kafka topic. In our proposed architecture, MongoDB and Neo4j ingestion applications should also read tweets from this Kafka topic. However this is not integrated yet, hence MongoDB and Neo4j applications currenlty read tweets from files themselves. This should be integrated as such.

---

class kafka_tweets_producer.**Producer**(*tweet_folder_name*, *max_q_size*)

Bases: object

Class that on initialization, spawns 2 threads. One thread keeps reading tweets from the files and puts them to a queue. Second thread keeps reading tweets from the queue and puts them to the Kafka topic.

exception kafka_tweets_producer.**ServiceExit**

Bases: Exception

Custom exception which is used to trigger the clean exit of all running threads and the main program.

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

kafka_tweets_producer.**service_shutdown**(*signum*, *frame*)

CHAPTER

FOUR

INGESTING DATA INTO NEO4J

4.1 Data stored in neo4j

Our aim in the project is to capture the dynamics of an evolving social network. These dynamics can be a combination of :

- Spatial dynamics : captured by network based information.
- Temporal dynamics : The spatial information present at some interval of time in the past. This makes sense as the network keeps on changing and the user might want to see the state of some part of it at some point in past.

We store the complete twitter network in graph database neo4j.

For sake of understanding, lets divide the complete network into three parts:

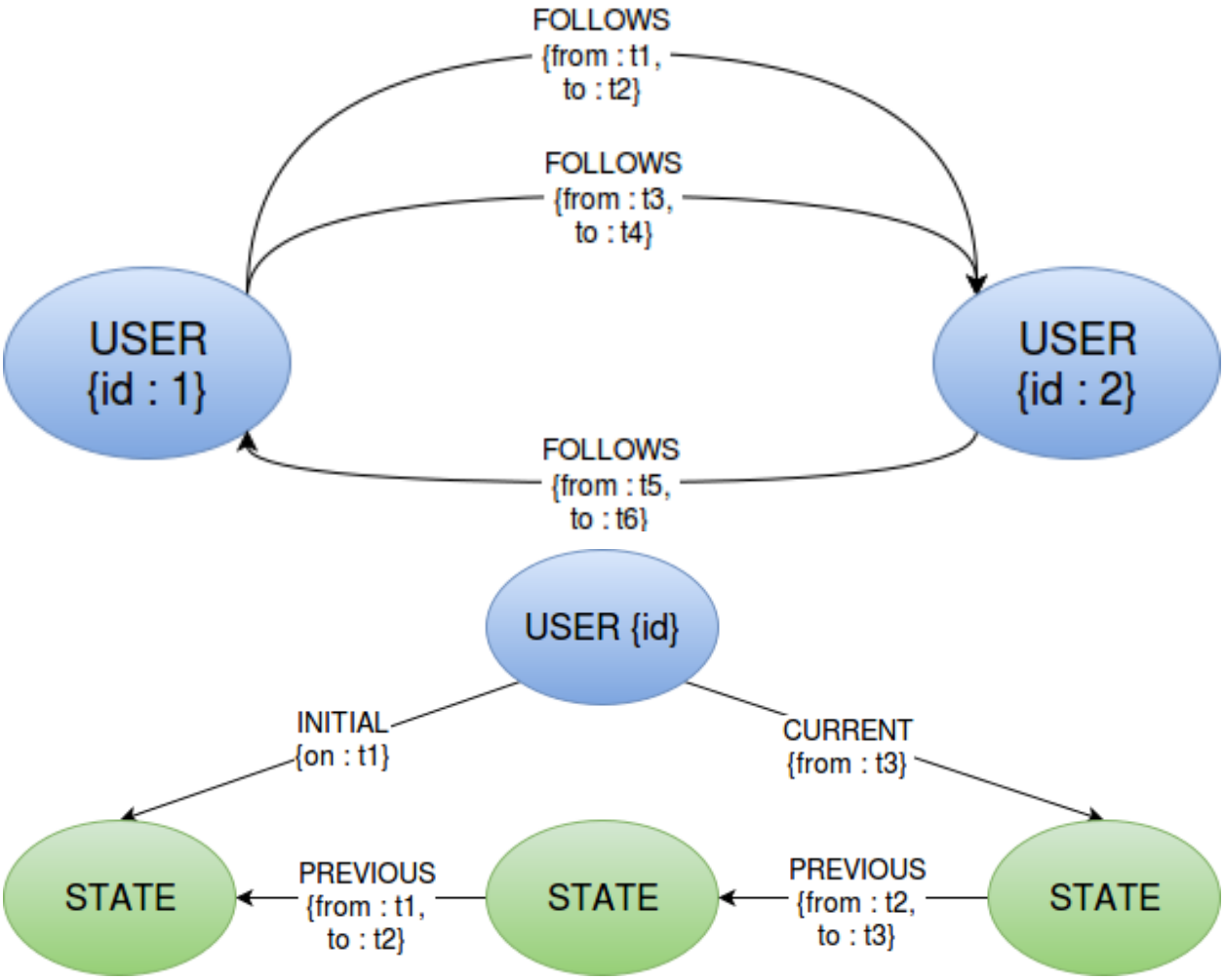
- User network : Stores the user information and connections between the user nodes
- Tweet network : Stores the tweets, their attributes and interconnections between tweets and their connections with user nodes as well.
- Indexing network : Stores the time indexing structure utilized to answer queries having a temporal dimension. Its instructive to imagine the user and tweet network on a plane and the indexing network on top of this plane.

Let's look at these networks in some detail

4.1.1 User network

The user network contains following nodes:

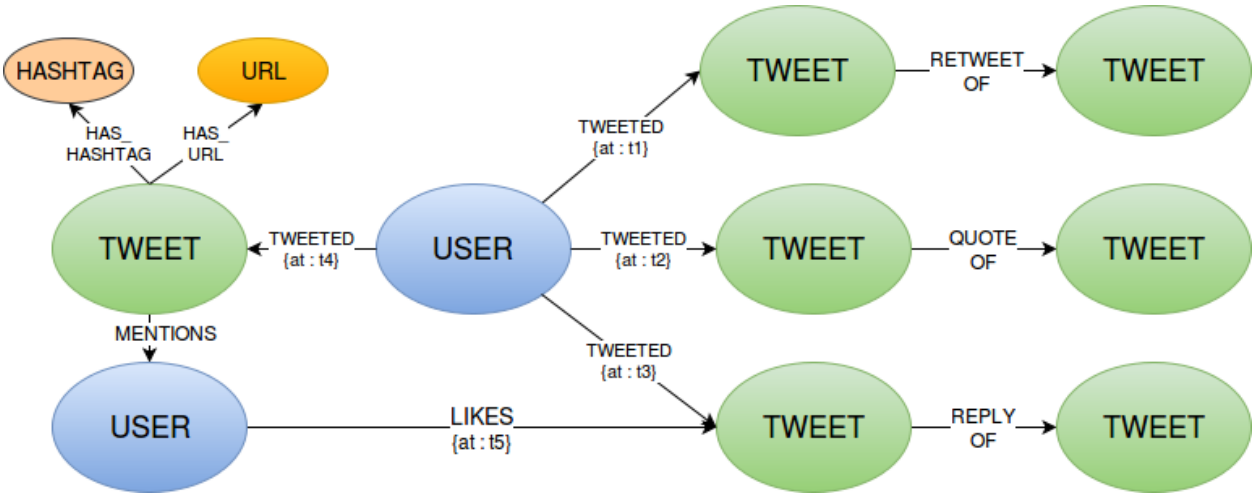
- USER node : It contains user_id and screen_name.
- USER_INFO node: It contains the user info like number of followers/friends, number of tweets, location and other meta data. These nodes form a link list in which new nodes are added, when information of the user is collected.



4.1.2 Tweet network

The tweet network contains following nodes:

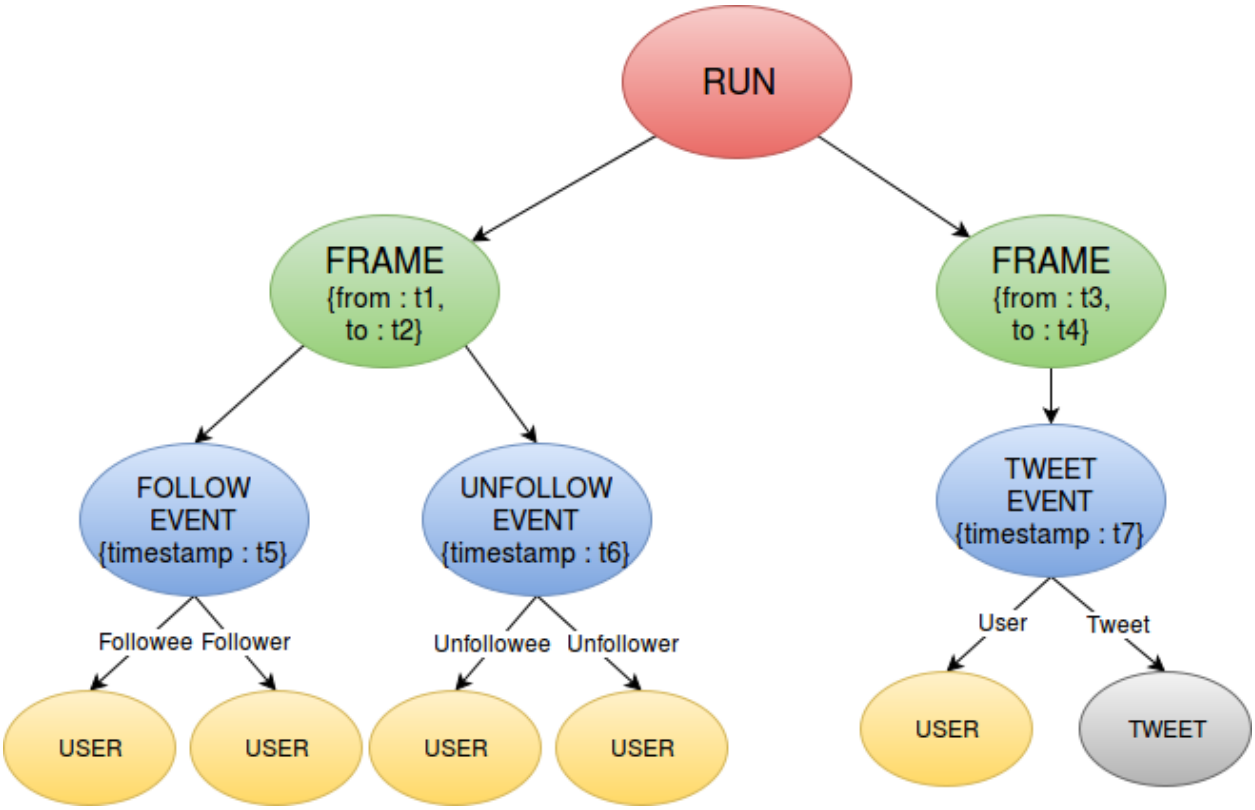
- TWEET node : It contains the tweet info like its id, the raw text of the tweet, the location from which the tweet is made(currently we are storing it in raw fashion, which in future can be extended by creating LOCATION nodes just like HASHTAG et al.)
- HASHTAG node : It contains the text of the hashtag.
- URL node : Contains the full and the shortened url.



4.1.3 Indexing network

The Indexing network contains:

- RUN node : It is a unique node in the entire network. Just a starting placeholder for the root of the indexing structure.
- FRAME node : A frame time interval is to be specified by the user. Each frame indexes time slices equal to the specified interval.
- FOLLOW_EVENT node : represents that Follower started following Followee.
- UNFOLLOW_EVENT node : represents that Unfollower stopped following Unfollowee.
- TWEET_EVENT node : represents that user tweeted tweet.



4.2 Ingesting the data into neo4j : Logic

We get a tweet from the twitter firehose. One simple thing could be to make a connection to the on-disk database and insert the tweet. This can be achieved using *session.run(<cypher tweet insertion query>)*, because run in neo4j emulates a auto-commit transaction.

4.2.1 Improving ingestion rate using transaction

A simple way to make this more efficient would be the use of transactions. The idea behind using transactions is to keep on accumulating the incoming tweets in a graph in memory. After some fixed number of tweets, the transaction is committed to the disk. Clearly this leads to faster ingestion rate:

- Accumulating to memory and then writing the batch to disk is faster as compared to writing tweet by tweet to disk due to the efficiency in disk head seaks.
- Further, when creating the in-memory local graph from the transaction batch, neo4j does some changes in the order in which to write the changes to ensure efficiency.

But, the clear downside of this is that the queries being answered will lag behind at most the transaction size. This happens because the tweets are being inserted in real time manner and the queries are also being answered simultaneously. But this is not a major issue as it induces a lag of only <10 secs(assuming transaction size of ~30k).

4.2.2 Indexing

We create uniqueness constraints on the following attributes of these nodes:

Node	Attribute
FRAME	start_t
TWEET	id
USER	id
HASHTAG	text
URL	url

4.3 Running the ingestion script

There are 2 applications to ingest data into Neo4j.

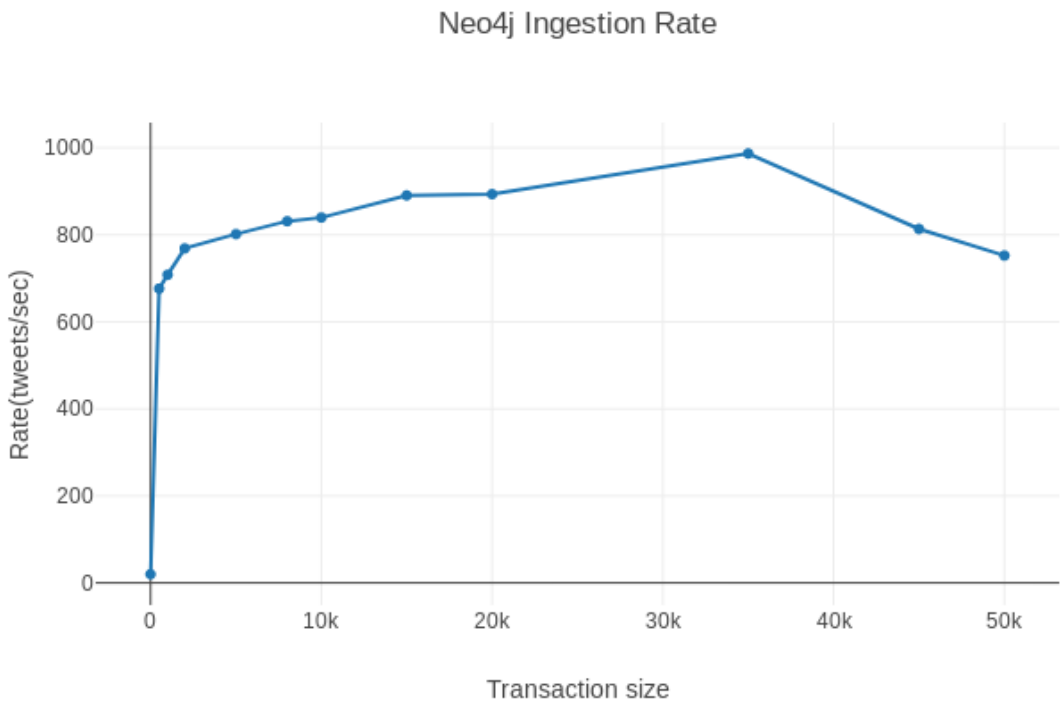
4.3.1 Streaming data

If the data has been collected using the Stream Sample API (as explained in *Stream Sample API*), then this application needs to be used to ingest the tweets into Neo4j. Navigate to the Ingestion/Neo4j and make changes to the file ingest_neo4j_streaming.py. Specifically, provide the folder containing the tweets containing files. We are simulating the twitter stream by reading the tweets from a file on the disk and storing those in memory. This makes sense as we can't possibly get tweets from the twitter hose at a rate greater than reading from memory, thus this in no way can be a bottleneck to the ingestion rate. Then just run the file *python ingest_neo4j_streaming.py* to start ingesting. A logs file will be created which will keep on updating to help the user gauze the ingestion rate.

4.3.2 User Timeline data

If the data has been collected using the User Timeline API (as explained in *User Timeline API*), then this application needs to be used to ingest the tweets into Neo4j. Navigate to the Ingestion/Neo4j and make changes to the file ingest_neo4j_user_timeline.py. This file ingests all the data present in the 'data' folder in the same directory. However, if the data is collected in an incremental fashion (eg. running the data collection application day by day), then this file needs to be changed accordingly to only ingest the data of the new timestamps.

4.4 Neo4j Ingestion Rates



Observe that the ingestion rate peaks at 1000 tweets/sec at a transaction size of around 35k. This is mainly due to the limitation of memory size. The authors observe that keeping a larger transaction size leads to lag on the system indicative of use of swap space. Thus, the maximum ingestion rate can be enhanced just by putting in more memory, albeit with decreasing returns.

4.5 Code Documentation for Neo4j data ingestion

Here we provide a documentation of the code for ingesting tweets collected using Streaming API. Module to insert data, collected using streaming API, into Neo4j database.

The *ingest_neo4j_streaming* module contains the classes:

- *ingest_neo4j_streaming.Twitter*

One can use the `ingest_neo4j_streaming.Twitter.ingest_tweet()` to insert a new tweet into the database.

An example usage where we want to insert all tweets from all files in a folder tweet_folder:

```
>>> t = Twitter(50000)
>>> t.get_constraints()
>>> t.get_profile()
>>> read_tweets(<tweet_folder>, t)
>>> t.get_profile()
>>> t.close_session()
```

class ingest_neo4j_streaming.**Twitter**(*batch_size=200*)
Bases: object

Class containing functions to insert tweets into neo4j. We open a connection to the neo4j database through py2neo and the official neo4j driver. Dealing with transactions is easy in py2neo, so it is used to make and commit transactions. While the connection to the neo4j driver is used just in clearing out the graph.

Parameters **batch_size** – number of tweets to take into transaction before commiting it

clear_graph()
Delete the complete graph. Albiet, keep the indices.

close()
See if there are some tweets not comitted in the trnx and if yes, commit those.

close_session()
Close the neo4j driver session

create_constraint(*node*, *attrib*)
Create constraint on attrib in node node

Parameters **node** – node type on which to create the constraint

Parem attrib node attibute whose constraint is to be created

create_constraints()
Create uniqueness constraints on the attributes of nodes. Note that ceating a constraint automatically creates an index on it

drop_constraint(*node*, *attrib*)
Drop constraint on attrib in node node

Parameters **node** – node type on which to delete the constraint

Parem attrib node attibute whose constraint is to be deleted

get_constraints()
Get the constrainsts on different types of nodes.

get_profile()
The number of total, user, tweet, hashtag nodes in the graph

insert_tweet(*tweet*, *favourited_by=None*, *fav_timestamp=None*)
The main function to insert the tweet. Begin a transaction, when atleast batch_size number of tweets are collected, commit the transaction. Start collecting the new tweets after that.

We have two cases depending if the tweet to be inserted is a retweet or not. We mention the steps taken to insert the tweet in the two case:

- **The tweet is a retweet**
  - Create a tweet_event under a appropriate frame
  - Merge node for this tweet. Maybe the tweet node already partially exists because some other tweet is its reply.
  - Create favorite relation if needed
  - Proceed only if the tweet was not already created
  - Create user and then the relationships
  - Find node of original tweet and link
- **The tweet is not a retweet**

- Create a tweet_event under a appropriate frame
- Merge node for this tweet
- Create favorite relation if needed
- Proceed only if the tweet was not already created
- Create user and then the relationships
- Create links to hashtags, mentions, urls
- Create link to quoted tweet in case this tweet quotes another tweet
- Create link to original tweet in case this is a reply tweet

Parameters

- **tweet** – the json of the tweet to be inserted
- **favourited_by** – userid of the user who favourited the tweet
- **fav_timestamp** – time at which the tweet was favourited

Returns None

Todo: Currently we are making use of transactions only. We get decent peak ingestion rate of around 1000 tweets/sec. But this can be increased by overlapping the collection and ingestion part as we do in case of mongoDB. But the same scheme can't be used here as the transaction created is not a native python object and hance can't be passed between python multiprocessing module processes. So, one idea is to create a csv with the tweets, and then call the use_csv function in neo4j to ingest. But this is a contrived way of doing this by doing same task twice.

---

ingest_neo4j_streaming.**flatten_json**(*json_obj*)
Function to flatten the tweet. Used in case we want to store the complete tweet JSON in the TWEET node. This is because neo4j doesn't allow nested jsons to be stored

ingest_neo4j_streaming.**getDateFromTimestamp**(*timestamp*)

ingest_neo4j_streaming.**getFrameStartEndTime**(*timestamp*)

ingest_neo4j_streaming.**log**(*text*)
Why use this when we can use logging? There is a peculiar bug when open neo4j bolt server with logging

ingest_neo4j_streaming.**read_tweets**(*path*, *twitter*, *filename=''*)
Read tweets from the directory in path and inert all tweets in all files in the first level of path into neo4j.

Parameters

- **path** – the path of the directory
- **twitter** – a Twitter object
- **filename** – optional, if want to insert tweets from a single file

Here we provide a documentation of the code for ingesting data collected using User Timeline API. Module to insert data, collected using user timeline API, into Neo4j database.

This is the schema of the Neo4j graph:

USER NETWORK Node labels - USER(id), USER_INFO(dict) Relationships - CURR_STATE(from), INITIAL_STATE(on), PREV(from,to)

FOLLOWER NETWORK Node labels - Relationships - FOLLOWS(from,to), FOLLOWED(from,to) // FOLLOWS.to will always be the last time data was collected

TWEET NETWORK Node labels - TWEET(id,created_at,is_active), TWEET_INFO(dict), HASHTAG(text), URL(url,expanded_url), //MEDIA(url, media_url)//, PLACE(id,name,country) -> This is not the location of tweet but the location with which the tweet is tagged (could be about it) Relationships - TWEETED(on), LIKES(on), INFO, REPLY_TO(on), RETWEET_OF(on), QUOTED(on), HAS_MENTION(on), HAS_HASHTAG(on), //HAS_MEDIA(on)//, HAS_URL(on), HAS_PLACE(on)

FRAME NETWORK Node labels - RUN, FRAME(start_t,end_t), TWEET_EVENT(timestamp), FOLLOW_EVENT(timestamp), UNFOLLOW_EVENT(timestamp), FAV_EVENT(timestamp) Relationships - HAS_FRAME, HAS_TWEET, TE_USER, TE_TWEET, HAS_FOLLOW, FE_FOLLOWED, FE_FOLLOWS, HAS_UNFOLLOW, UFE_UNFOLLOWED, UFE_UNFOLLOWS, HAS_FAV, FAV_USER, FAV_TWEET

class ingest_neo4j_user_timeline.**Neo4jIngestion_UserTimeline**(*data_directory*, *user_screen_names_file_path*)

Bases: object

clear_db()

close_session()

create_indexes()

create_tweet(*tweet*, *favourited_by=None*, *fav_timestamp=None*)

flatten_json(*json_obj*)

getFrameStartEndTime(*timestamp*)

readDataAndCreateGraph(*STAGE_BY_STAGE*)

simulateExample()

sync_session(*type_=None*)

update_followers(*user_id*, *follower_ids*, *timestamp*)

update_friends(*user_id*, *friend_ids*, *timestamp*)

update_user(*id*, *user_info_dict*, *timestamp*)

ingest_neo4j_user_timeline.**log**(*text*)

CHAPTER

FIVE

INGESTING DATA INTO MONGODB

5.1 Why store in MongoDB

In mongoDB we store only the data which can be extracted quickly from incoming tweets without much processing. We answer "non-network" based i.e. simple aggregate queries using MongoDB instead of Neo4j.

This means that any query which can be answered using mongoDB can also be answered using the network data in neo4j. This has been done to ensure that some very common queries can be answered quickly. Also, neo4j has a limit on the parallel sessions that can be made to the database, so in case we decide to do away with mongoDB, those queries would have to be answered from neo4j and would unnecessarily take up the sessions.

5.2 Data Format in mongoDB

We have three collections in mongoDB:

- **To store the hashtags. Each document in this collection stores the following information:**
  - the hashtag
  - the timestamp of the tweet which contained the hashtag
  - the sentiment associated with the tweet containing the hashtag
- **To store urls**
  - the url
  - the timestamp of the tweet which contained the hashtag
  - the sentiment associated with the tweet containing the hashtag
- **To store user mentions**
  - the user mention
  - the timestamp of the tweet which contained the hashtag
  - the sentiment associated with the tweet containing the hashtag

Given this information in mongoDB, we can currently use it to answer queries like:

- Most popular hashtags(and their sentiment) in total
- Most popular hashtags(and their sentiment) in an interval of time
- Most popular urls in total
- Most popular urls in an interval of time

- Most popular users in total(in terms of their mentions)
- Most popular users in an interval of time(in terms of their mentions)

5.3 mongoDB v/s neo4j

Note that just the bare minimum information that is currently being stored in the mongoDB. It can easily be extended to store more information. MongoDB provides strong mechanisms to aggregate and extract information from the database.

So, even if we decide to store some pseudo-structural information, like the user of the tweet in hashtags collection and then answer queries like the sentiment associated will all the tweets of an user, we expect the query execution time to be atleast as fast as answering the query in neo4j, though in case of neo4j also, answering such query would also take only a single hop, which means that the execution time would be small anyways. This is precisely the reason why we don't currently store such information in mongoDB.

But, as the size of the system grows, it would surely be beneficial to store much more condensed data in mongoDB and use it to answer more complex queries.

5.4 Ingesting the data into mongoDB : Logic

[scheme 1] A simple approach would be to ingest a tweet into the database as and when it comes in real time. But clearly(and as mentioned in mongoDB documentation) this is suboptimal, as we are connecting to the on-disk database frequently.

5.4.1 Improving ingestion rate using transactions

[scheme 2] An easy solution to this would be to keep collecting the data in memory and then write it to the database periodically in batches.

But observe that, the time it takes the process to open a connection to database and then write the data to it, no new tweets are being collected in memory.

5.4.2 Improving ingestion rate using parallel multiple process

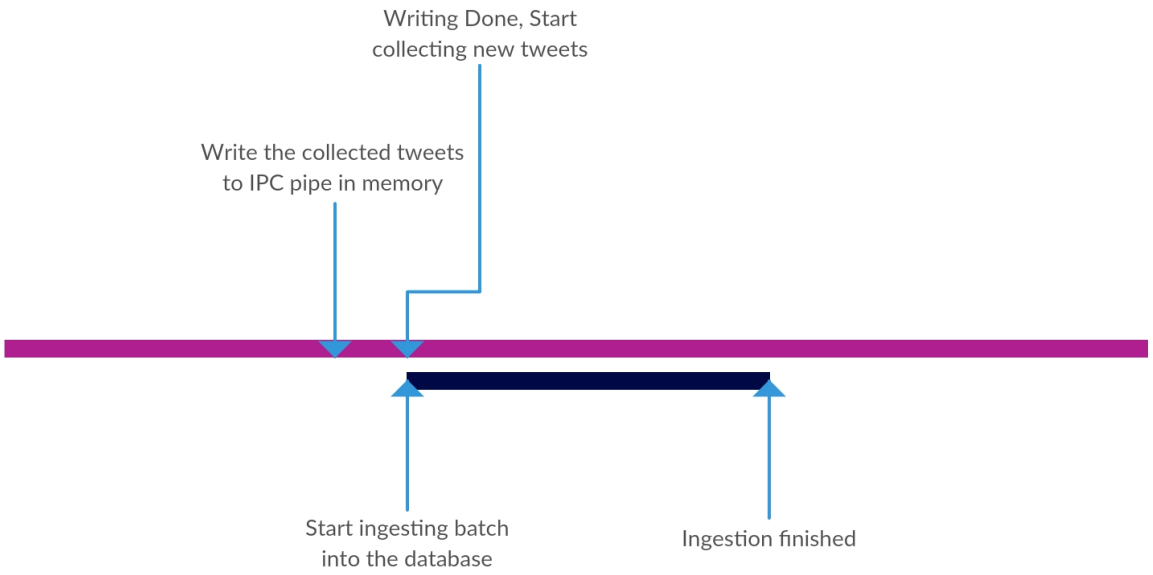
[scheme 3] So finally we take the approach of utilizing multiple processes to write data to mongoDB.

Observe here the distinction between a thread and a process. While using multiple threads, the threads are run(usually, if we discount the kernel threads spawned by python) on a single core in python, due to Global Interpreter Lock and thus, though we get virtual parallelism, we don't get real parallelism. Thus, due to the limitation of the language, we are using process to get the parallelism between writing to database and collecting new tweets. A clear disadvantage of using process over threads will become clear below.

To explain the final multi-process approach, we have two processes running:

- Accumulator process - It collects the tweets in an in-memory data structure. Also, in the beginning at t=0, it spawns a timer thread, which generates an interrupt after every pre-specified T time.
- Connector process - It takes a list of tweets through a pipe, opens connection to the database and writes the tweets to the database.

How the system works can be understood through this image:



So, the timer process in the accumulator process generates an interrupt after every T seconds, at this instant, the accumulator stops collecting tweets and writes those to Inter process communication(IPC) pipe. This is generally fast as IPC pipe are implemented in memory. Now, the other end of the pipe is in the connector process. After the writing process has been complete, it receives the tweets and starts writing those to the on-disk database as a batch, which again ensures that the process is faster as compared to writing single tweet at a time in a loop. Concurrently, while the connector process is writing the tweets, the accumulator process starts accumulating new tweets.

So in this way the the process of writing to database in connector process is overlapped with the the accumulation of tweets in accumulator process. Note that we have a small gap equivalent to time taken to write to IPC, in which the accumulator process is not collecting the tweets. The whole process can further be made efficient by removing this gap, but since we are getting tweet ingestion rate much more than the rate of tweets coming on twitter and the gain from removing the gap would not be much, we don't implement it.

To answer queries like the most popular hashtags in total, or most popular hashtags in a large interval. It would be beneficial to have aggregates over a larger interval. For example, say we want to get the most popular hashtags in an year, it would be helpful in that setting to have an aggregated document containing 100 most popular hashtags in each month, then we can consider a union of these 12 documents plus some counting from the interval edges to get the most popular hashtags. Clearly, this will fasten the query answering rate. Though, this would not always give the exactly accurate results and can also not be used to get the counts of hashtags, but can be used to get most popular k hashtags as the size of data grows. To implement it, simply spawn another thread in the connector process to read data from the hashtags collection at a specific time interval(like 1 week), aggregate the data and store the aggregated information into a new collection. We provide the code for this, but don't currently use this mechanism.

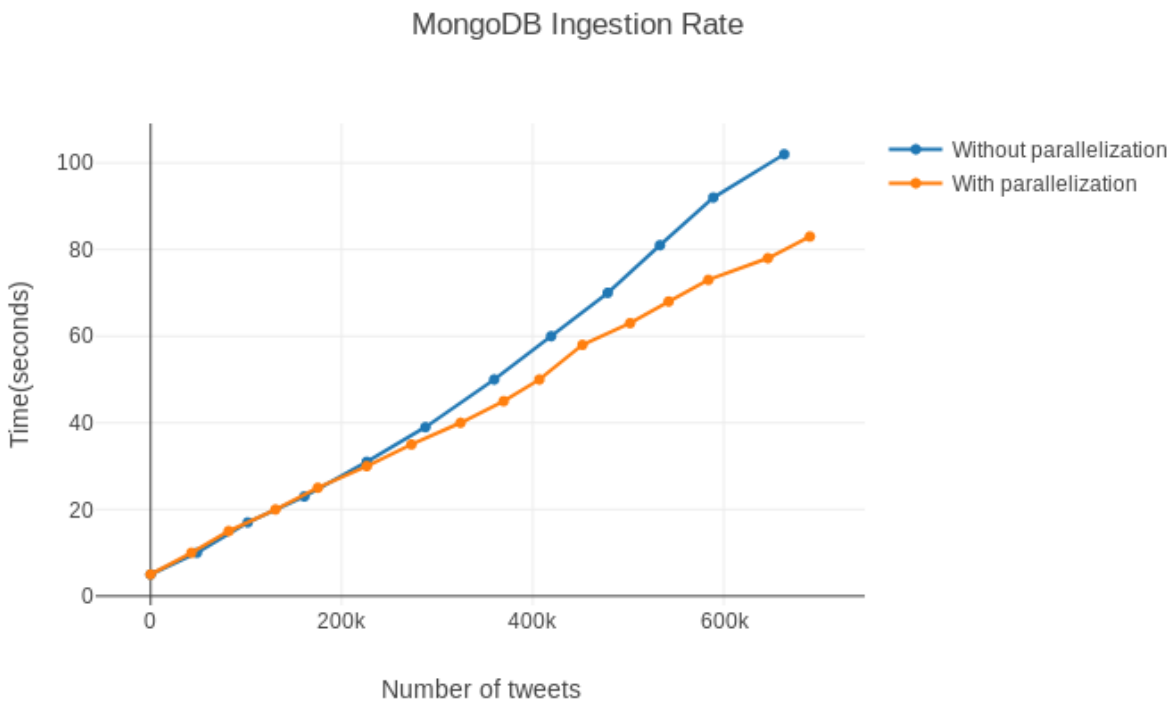
5.5 Ingesting the data into mongoDB : Practical side

On practical side, to ingest data into mongoDB, navigate to the Ingestion/MonogDB and make changes to the file ingest_raw.py. Specifically, provide the folder containing the tweets containing files. We are simulating the twitter stream by reading the tweets from a file on the disk and storing those in memory. This makes sense as we can't possibly get tweets from the twitter hose at a rate greater than reading from memory, thus this in no way can be a bottleneck to the ingestion rate. Then just run the we need to run the file *python ingest_raw.py* to start ingesting. A logs file will be created which will keep on updating to help the user gauze the ingestion rate.

Please observe that the process of ingesting into neo4j and mongoDB are similar, with just variations in which code to run.

5.6 MongoDB Ingestion Rates

As expected, the ingestion rate into mongoDB while overlapping writing into database and accumulating data is faster than without parallelization. The plot below shows a comparison between scheme 2 and scheme 3 as described above. Observe that as more and more tweets are inserted, the difference between the two scheme grows as the time saved in overlapping inserting the accumulating keeps on adding up in advantage of scheme 3.



Clearly the ingestion rate depends on the time after which the interrupt to start write the collected tweets to database is generate(called T in *Improving ingestion rate using parallel multiple process*).

Finally we get an ingestion rate of around 7k-12k(around x10 of that of neo4j) tweets/second on average, depending on T.

5.7 Code Documentation for mongoDB ingestion

Module to ingest data into MongoDB. We try to overlay the collection and ingestion of tweets while ingeting data. For more details, see the documenation.

The `ingest_raw` module contains the classes:

- `ingest_raw.Ingest`

One can use the `ingest_raw.Ingest.insert_tweet()` to insert a new tweet into the database.

An example usage where we want to insert all tweets from all files in a folder tweet_folder:

```
>>> i = Ingest(10)
>>> i.clear_db()
>>> read_tweets(i, <tweet_folder>)
```

class ingest_raw.**Ingest**(*interval*)

Bases: object

Class to insert tweets into mongoDB.

Parameters **interval** – time interval after which to generate interupt for starting ingestion of tweets

aggregate()

The function to be called in case, we choose to aggregate the counts at a larger inteeval.

Note: interval1>>interval. interval is like order of seconds and interal1 is like order of hours.

---

clear_db()

Delete all the collections

exit()

Join the worker process

insert_tweet(*tweet*)

Function to collect incoming real time tweets. Update the in memory dictionaries.

Parameters **tweet** – the json of the tweet.

..note:: If we choose to keep new information about the tweets, we need to modify this, along with *ingest_raw.Ingest.worker()*.

populate()

The code executed by the collector process. After each interupt it spawns a new timer thread to generate the new interupt. Also, it puts the collected tweets into the IPC pipe and starts collecting new tweets.

worker(*q*)

The function called inside he ingestor(worker) process. This function is alled after every <interval> seconds. It loops to look for inputs from the pipe end. Once inputs are there, it opens connection to database and commits the batch recieved from the pipe to the on-disk database.

Parameters **q** – the inter-process communication pipe

class ingest_raw.**Timer**(*interval*, *function*, *args=None*, *kwargs=None*, *iterations=1*, *infinite=False*)

Bases: multiprocessing.context.Process

Calls a function after a specified number of seconds:

```
>>> t = Timer(30.0, f, args=None, kwargs=None)
>>> t.start()
>>> t.cancel() #stops the timer if it is still waiting
```

authkey

cancel()

Stop the timer if it hasn't already finished.

daemon

Return whether process is a daemon

exitcode

Return exit code of process or *None* if it has yet to stop

ident
Return identifier (PID) of process or *None* if it has yet to start

is_alive()
Return whether process is alive

join(*timeout=None*)
Wait until child process terminates

name

pid
Return identifier (PID) of process or *None* if it has yet to start

run()

sentinel
Return a file descriptor (Unix) or handle (Windows) suitable for waiting for process termination.

start()
Start child process

terminate()
Terminate process; sends SIGTERM signal or uses TerminateProcess()

ingest_raw.**calculate_sentiment**
Function to calculate sentiment of a tweet. Just calculate the number of positive and negative words, matching against a pre-curated list.

Parameters

- **positive_words** – the list of positive sentiment words
- **negative_words** – the list of negative sentiment words
- **tweet_text** – the raw test of the tweet splitted tokenized, hashtags and user-mentions removed

ingest_raw.**getDateFromTimestamp**(*timestamp*)

ingest_raw.**read_tweets**(*ingest*, *path*, *filename=''*)
Read tweets from the directory in path and inert all tweets in all files in the first level of path into neo4j.

Parameters

- **path** – the path of the directory
- **twitter** – a Twitter object
- **filename** – optional, if want to insert tweets from a single file

ingest_raw.**threaded**(*fn*)

CHAPTER

SIX

NEO4J: API TO GENERATE CYPHER QUERIES

Here we explain the API to generate cypher queries for Neo4j.

6.1 Template of a general query

6.1.1 Basic Abstraction

Any query can be thought of as a 2 step process -

- Extract the relevant sub-graph satisfying the query constraints (Eg. Users and their tweets that use a certain hashtag)
- Post-processing of this sub-graph to return desired result (Eg. Return "names" of such users, Return "number" of such users)

In a generic way, the 1st step can be constructed using AND,OR,NOT of multiple constraints(Though in our code right now, only AND is supported as Neo4j doesn't support OR and NOT directly).We now specify how each such constraint can be built.

We look at the network in an abstract in two dimensions.

- There are "Entities" (users and tweets) which have "Attributes" (like user has screen_name,follower_count etc. and tweet has hashtag,mentions etc.).
- The entities have "Relations" between them which have the only attribute as time/time-interval (Eg. Follows "relation" between 2 user "entities" has a time-interval associated).

So each constraint can be specified by specifying a pattern consisting of

- Two Entities and their Attributes
- Relation between the entities and its Attribute (which is the time constraint of this relation)

To make things clear we provide an example here. Suppose our query is - Find users who follow a user with id=1 and have also tweeted with a hashtag "h" between time t1 and t2. We first break this into AND of two constraints:

- User follows a user with id=1
- User has tweeted with a hashtag "h" between time t1 and t2.

We now specify the 1st constraint using our entity-attribute abstraction.

- Source entity - User, Attributes - None
- Destination entity - User, Attributes - id=1
- Relationship - Follows, Attributes - None

We now specify the 2nd constraint using our entity-attribute abstraction.

- Source entity - User, Attributes - None
- Destination entity - Tweet, Attributes - hashtag:"h"
- Relationship - Follows, Attributes - b/w t1,t2

6.1.2 Naming entities

The missing thing in this abstraction is that we need to be able to distinguish that the Users in the two constraints above refer to different users. To do so, we "name" each entity (like a variable). So we have:

- **Constraint 1:**
  - Source entity - u1:User, Attributes - None
  - Destination entity - u2:User, Attributes - id=1
  - Relationship - Follows, Attributes - None
- **Constraint 2:**
  - Source entity - u1:User, Attributes - None
  - Destination entity - u3:Tweet, Attributes - hashtag:"h"
  - Relationship - Follows, Attributes - b/w t1,t2

6.1.3 Variable attributes

In the example above, we considered a fixed hashtag "h". But the user can provide a variable attribute to an entity by giving it a name enclosed in curly braces `{}`. In such a case, the attribute is treated as a variable to the query and the name inside the braces is treated as the name of the said variable. For example, had we input the hashtag as `{hash1}`, it means that our query has a variable named "hash1".

6.1.4 Returns

The user can write a comma separated list of entities/attributes to return. An entity can be specified directly by its name. Eg. `my_tweet_entity`. An entity's attribute can be specified as `<entitiy_name>.<attribute>`. Eg. `u.screen_name`.

The user can also group on certain entity/attribute by using an aggregate function on some other entity/attribute. Eg. `return u.id, count(distinct h)` will group on u.id and return count of distinct h among all records returned for each u.id.

6.2 Creating a custom query through dashboard API : Behind the scenes

A user can follow the general template of a query as provided above to build a query. when a user provides the inputs to specify the query, the following steps are executed on the server:

- Cleanup and processing of the inputs provided by the user.
- The variables(User/Tweet), their attributes and the relations are stored in a database. These stored objects can be later used by the user.

Finally, to create the query the user need to specify the query name and the return variables. The query specified by the user in terms of constraints is converted into a Cypher neo4j graph mining query:

- All variable attributes are unwinded. This connects to the fact that we are expecting inputs as a list of native objects, hence the unwind for all inputs to the query. This ensures a Cartesian cross in the query.
- The time indexed part of the query is generated through the time indexing structure with frames and events.
- The network part is generated through the user network and tweet network based on the relationships.
- The return variables specified by the user are just concatenated with a return statement in the cypher.

It is to be noted here, that we don't do any kind of checking if the constraints specified by the user to build the query are valid. Checking this in a generic manner without executing the query is difficult. So, this is delegated to the neo4j query engine itself and the user will get empty result in case the constrains are invalid.

6.3 Code Documentation for Neo4j query generation

Here we provide a documentation of the code. Module to generate cypher code for inputs taken from user though dashboard API.

The *generate_queries* module contains the classes:

- *generate_queries.CreateQuery*

One can use the *generate_queries.CreateQuery.create_query()* to build a cypher query.

Example illustrating how to create a query which gives the userids and their tweet counts who have used a certian hashtag.

```
>>> actors=[("u","USER"),("t","TWEET"),("t1","TWEET")]
>>> attributes=[[],[("hashtag","{hash}")],[]]
>>> relations=[("u","TWEETED","t","",""),("u","TWEETED","t1","","")]
>>> cq = CreateQuery();
>>> return_values="u.id,count(t1)"
>>> ret_dict = cq.create_query(actors,attributes,relations,return_values)
>>> pprint(ret_dict,width=150)
```

Example of a query which uses time indexing in a relationship:

```
>>> actors = [('x', 'USER'), ('u1', 'USER')] + [('t1', 'TWEET'), ('t2', 'TWEET')]
>>> attributes = [[('id', '12')], [('id', '24')]]+[[('hashtag', 'BLUERISING')], [(
↪'retweet_of', 't1'), ('has_mention', 'u1')]]
>>> relations = [('x', 'FOLLOWS', 'u1', '', ''), ('x', 'TWEETED', 't2', '24', '48')]
```

class generate_queries.**CreateQuery**

Bases: object

Class containing functions to generate query.

conditional_create(*entity*)

Condionally provide the attributes of the node if not already created, else directly use the name of the variable create earlier. If already create, pass empty list of properties.

Parameters **entity** – the entity which to check and create

Returns the code for the node as neo4j node enclosed in ()

create_query(*actors*, *attributes*, *relations*, *return_values*)

Takes a list of attributes and relationships between them and return a cypher code as string. For the format of the lists see the examples.

Parameters

- **actors** – the variable names, types of the attributes
- **attributes** – the properties of the actors
- **relations** – the relations between the entities along with time index for the relations
- **return_values** – a direct string containing the return directive.

Returns index of the bond in the molecule

Note: Here we are expecting that if user has not specified the times on the dashboard, then we pass epmty string. If you store some other default in dashboard database then change this accordingly.

Note: The return _values is directly used as a string in the cypher query, so the user can use AS and other similar cypher directives while specifying the query.

Todo: Support to compose queries using OR. For example, currently compostion of relationships or attribute properties like all tweets(t) which are retweets of t1 or quoted t2, is not supported. Use cypher union for this.

---

generate_node(*var*, *type*, *props*)

Helper function for *generate_queries.CreateQuery.conditional_create()*

Parameters

- **var** – the variable name of the entity
- **type** – the type of the entity. Observe we pass type as :USER and NOT as USER
- **props** – the properties of the entity.

Returns the code for the node as neo4j node enclosed in ()

CHAPTER

SEVEN

GENERATING QUERIES IN MONGODB

As mentioned in mongoDB ingestion section, in mongoDB we store only the data without any structural information which can be extracted quickly from incoming tweets without much processing. Further, we store in mongoDB to ensure that some very common queries can be answered quickly.

This leads to these important properties of the mongoDB part of datastore:

- Only very specific queries can be answered using only the mongoDB. The specific queries further depend on the data which is being stored, which is further decided by which queries are seen frequently and need to be sped up. Given that currently we have the hashtag, url and user mention collection in mongoDB with the entity name, timestamp, the sentiment associated with the tweet in which the mentioned entity occurred; we can answer only specific queries like the most popular hashtag(and its sentiment) occurring in an interval(which can be the entire time as well).
- This further means that the mongoDB schema and datastore can easily be modified and extended. For example, if I decide to store the named entities in the tweets as well, all we need is to make a new collection.

Contrast this with neo4j where the schema is more or less for fixed for all practical purposes and the user can't easily change it.

Given the above two properties, it makes little sense to develop a complete generic API to input queries from the user as it would certainly be an overkill. So currently we provide APIs only to take only specific queries from the user. The queries which can currently be answered using mongoDB are follows(the things mentioned inside <> are the inputs to the query):

- Give the <number> most popular hashtags in total
- Give the <number> most popular hashtags in the time interval <Begin Time> and <End Time>
- Give the timetamps at which <hashtag> is used between <Begin Time> and <End Time>
- Give the timetamps, associated positive and negative sentiment of a <hashtag> between <Begin Time> and <End Time>

Similarly, queries analogous to the above can also be answered for urls and user mentions.

7.1 Generic API for mongoDB : Idea

For sake of completeness we also provide the way to generate a generic API to get mongoDB queries. For example take at this code to answer the query to get the most popular hashtags in an interval:

```
pipeline = [{"$match":{"timestamp":{"$gte":t1,"$lte":t2}}},{"$group": {"_id": "
↪$hashtag", "count": {"$sum": 1}}},
{"$sort": {"count":-1}},{"$limit":limit}]
l =  self.db.ht_collection.aggregate(pipeline)["result"]
return {"hashtag":[x["_id"] for x in l],"count":[x["count"] for x in l]}
```

As can be seen the query answering has three parts :

- The aggregation pipeline: This is the part with needs to be input from the user. As we have limited number of constructs that, can be used in aggregation, Taking those as inputs along with their parameters is not that difficult. Some of the construct, though can also be filled in automatically.
- Aggregating the collection based on the pipeline: This has fixed code and can be generated easily.
- Unzipping the result based on the output variables name input by the user: Again, fixed code and thus easily generated.

Along with the reasons mentioned above regarding the non requirement of generic monogDB query creator, another reason is that to generate the queries, would invariably require generation of python code, something like the above snippet and then modifying a file with a new function to connect to the database and execute the python code. This would further require modifying the source code in the dashboard website and the DAG execution python functions as well to register the new function, opening several fronts from which bugs can creep in.

7.1.1 MongoDB query execution code documentation

Here we provide a documentation of the code used for this functionality. Module to execute mongoDB queries. The idea is to keep the mongo interface minimal and easily extensible, and thus only pre specified queries can be answered through mongoDB, rather than generic ones.

The *execute_queries* module contains the classes:

- *execute_queries.MongoQuery*

One can use the different function in the class to execute different queries

Example illustrating how to answer different queries.

```
>>> q = MongoQuery()
>>> print(q.mp_ht_in_total(limit=10)) # get 10 most popular hashtags
>>> print(q.mp_um_in_total(10)) # get 10 most popular users
>>> print(q.mp_ht_in_interval(10, 1500486521,1501496521)) # get 10 most popular␣
↪hashtags in interval
>>> print(q.ht_in_interval("baystars",1500486521,1501496521)) # get the timestamps at␣
↪which baystars is used in interval
>>> print(q.ht_with_sentiment("baystars",1500486521,1501496521)) # get the timestamps␣
↪and sentiment at which baystars is used in interval
```

class execute_queries.**MongoQuery**

Bases: object

Class to answer mongoDB queries. Make connection to the database and keep on answering queris untill the object is deleted

clear_db()

Delete all the collections

ht_in_interval(*hashtag*, *begin*, *end*)

Give the timetamps at which <hashtag> is used between <begin> and <end>

Parameters

- **hashtag** – hashtag for the query
- **begin** – the begining unix time timestamp of the interval
- **end** – the ending unix time timestamp of the interval

ht_with_sentiment (*hashtag*, *begin*, *end*)
Give the timetamps at which <hashtag> is used and and sentiment of tweet in which <hashtag> occured between <begin> and <end>

Parameters

- **hashtag** – hashtag for the query
- **begin** – the begining unix time timestamp of the interval
- **end** – the ending unix time timestamp of the interval

mp_ht_in_interval (*limit*, *begin*, *end*)
Function to give the most popular hashtags in the time interval <begin> and <end>

Parameters

- **limit** – number of records to return
- **begin** – the begining unix time timestamp of the interval
- **end** – the ending unix time timestamp of the interval

mp_ht_in_total (*limit*)
Give <limit> most popular hashtags in total

Parameters **limit** – number of records to return

mp_um_in_total (*limit*)
Give <limit> most popular users(in iterms of mentions) in total

Parameters **limit** – number of records to return

CHAPTER

EIGHT

ABOUT POST PROCESSING FUNCTIONS

8.1 Need of post processing function

Some processing can be done in a cypher query in case of neo4j, and further in case of mongoDB, there is functionality to write custom functions to be included in the aggregation pipeline. But we provide the user the ability to create post processing function. The major reasons behind this are:

- It may be easy to do some projection on data output by a query post the execution, rather than coding it in the cypher in case of neo4j, or the aggregation pipeline in case of mongoDB.
- On similar lines as above, the user may need to aggregate multiple outputs from different queries in a post processing function in a custom manner not supported by the query mechanism of the databases.

8.2 Format of post processing functions

We treat the post processing functions as queries. The idea behind treating post processing functions as a query is to provide simplistic abstraction while creating a DAG. Thus while creating a DAG, a user has to just compose queries which can either be query to any of the two databases or a post processing function as well. Thus, given the DAG abstraction, the user can feed the output of the query(ies) into a post processing node.

Further to support this abstraction, we require the post processing function to accept a dictionary of lists of native python objects(named "inputs") and return a dictionary(named "ret") in same format. The function should further be named as "func". This requires that the user specifies the input and output variable names while creating the post processing function. This will be explained in detail in the DAG section.

Another way(instead of asking the user to explicitly provide the input and output variable names) in which post processing function could have been created is to just take as input the code of the function, parse it to get the number of inputs and their names. This is relatively easy. But, the issue is to get the output variables. This is a difficult problem and exactly this is used to generate automatic documentation of python code. But has been observed, even it misses the names of return variables. Its easy in case named variables are returned but the issue is when expressions are returned(for example the code contains `return l[:10]`, its not clear what should be the name of the return variable). Thus, out adopted method of dealing with dictionaries with named variables provides a clean abstraction over the the alternative.

Here is an example of a post processing function to output the union of lists input to it:

```
def func(input):
    """
    Function to take union of two lists.
    :param: input - a dictionary with the attribute names as keys and their values as␣
↪dict-values.
    :return: a dictionary with output variables as keys.
```

```
    """
    l1 = input["list1"]
    l2 = input["list2"]
    for x in l2:
        if x not in l1:
            l1.append(x)

    ret = {}
    ret["l_out"] = l1
    return ret
```

Post processing functions are also used to display custom metric. To view a custom metric, the user is required to specify a post processing function which accepts as inputs the outputs of any of the queries in the DAG and outputs a x and y coordinates to be used for plotting.

Here is another example of a post processing function to create a custom metric to plot the users with their number of tweets:

```
def func(inputs):
    inputs = list(zip(inputs["userid"], inputs["count"]))
    inputs.sort(key=lambda item:item[1], reverse=True)
    x_vals = []
    y_vals = []
    for i in range(10):
        x_vals.append(str(inputs[i][0]))
        y_vals.append(inputs[i][1])

    ret = {}
    ret["x_vals"] = x_vals
    ret["y_vals"] = y_vals
    return ret
```

8.3 Executing post processing function

To execute the post processing function, we just provide include the inputs in the context being passed to the function. The execution requires these three steps:

- Compilation : Any code errors are output to the user at this point.
- Passing the inputs to the function and executing its code.
- Obtaining the outputs : The output ret dictionary is pushed on the context by the function.

This can be seen in this code snippet used to execute the post processing functions:

```
context = {"inputs":copy.deepcopy(inputs)}
try:
    compile(function_code,'','exec')
    exec(functiona_code + "\n" + "ret = func(inputs)", context)
    for out in outputs:
        ret[out] = context[out]
except Exception as e:
    print("Exception while executing Post proc function: %s, %s"%(type(e),e))
```

CHAPTER

NINE

COMPOSING MULTIPLE QUERIES : DAG

9.1 Basic terminology

When we say **Query**, it means an one of the following three things:

- MongoDB query : A query not capable of giving any network information
- Neo4j query : A network based and/or time indexed query on the twitter network
- Post processing function : A python function which takes outputs of query(ies) as inputs and transforms them to give the output

DAG stands for directed acyclic graph. Thus it a directed graph with no cycles. The idea behind a DAG is to compose multiple queries to build a complex queries. A DAG has nodes and has directed connections connections between the nodes. Each node as a query associated with it.

9.2 Idea behind a DAG

As mentioned above, our main idea is to provide the user an easy abstraction to build complex queries. But apart from this there are several functions that the abstraction of a DAG seems to serve, which we list below:

- Provide an abstraction to build complex queries from simple queries.
- A particular database may be suited to answer particular type of queries. In fact this is the main reason behind storing data in mongoDB to answer commonly encountered queries. We expect the user to have a basic understanding of the database schemas and thus be able to have an idea of efficiency of the two databases in answering specific queries. Having such knowledge, the user can compose queries from different databases in sake of efficiency.
- It may be easy to do some projection of data output by a query post the execution, rather than coding it in the cypher in case of neo4j, or the aggregation pipeline in case of mongoDB. Thus, given the DAG abstraction, the user can feed the output of the query into a postprocessing node.
- On similar lines as above, the user may need to aggregate multiple outputs from different queries in a postprocessing function in a custom manner not supported by the query mechanism of the databases.
- Breaking a big query into smaller ones may be beneficial from the end user point of view because by doing so we can show the incremental results of the smaller parts(as they are executed) to the user instead of waiting for the entire big query to execute.

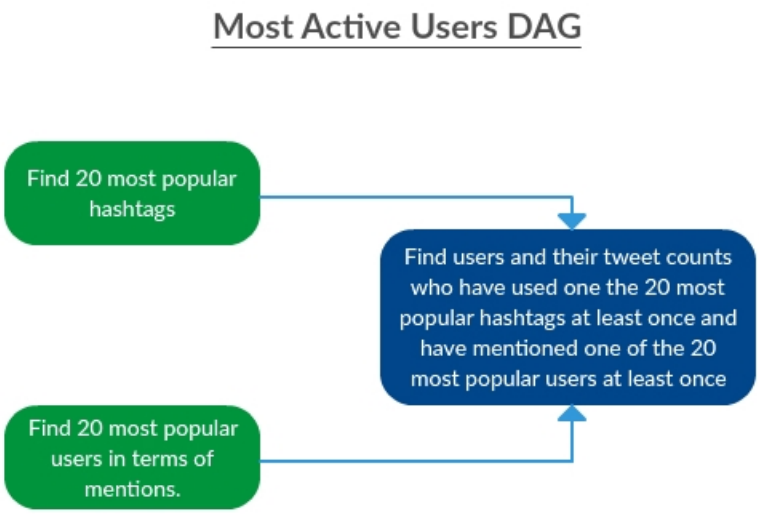
In this abstraction, a single query can also be treated as a DAG, one having a single node and no connections.

We store the queries that the user creates through the dashboard. The user can then specify the structure of the DAG network by uploading a file in which he specifies how outputs and inputs of queries are connected. We provide the details in the next section.

9.3 Building a DAG from queries

A DAG is composition of queries in which we need to specify how the outputs of queries upstream feed into the inputs of the downstream ones.

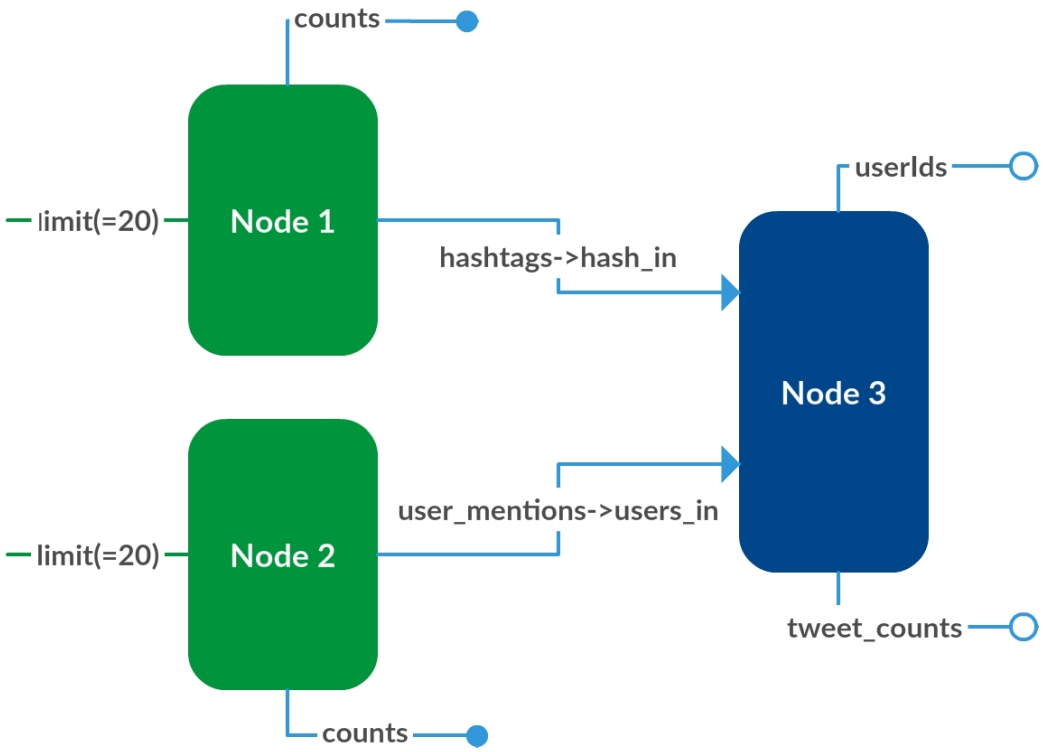
We explain how to build the queries with the help on an example. Let us build a DAG to get the most active users. Refer to this image(the green queries represent mongoDB queries and blue ones represent neo4j queries):



First we need to build the three queries separately, let us say we have the built queries as:

- **mongoDB query(most_popular_hashtags_20 - Node 1) - 20 most popular hashtags in total**
  - INPUTS : limit(number of records to return)
  - OUTPUTS : hashtags(list of popular hashtags, arranged by count in decreasing order), counts(list of their corresponding counts)
- **mongoDB query(most_popular_mentions_20 - Node 2) - 20 most popular users(in terms of number of mentions) in total**
  - INPUTS : limit(number of records to return)
  - OUTPUTS : user_metions(list of popular users, arranged by count in decreasing order), counts(list of their corresponding counts)
- **neo4j query(active_users - Node 3) - userIds and their tweet counts who have used one of the popular hashtags atleast onc**
  - INPUTS : hash_in(list of 20 most popular hashtags), users_in(list of 20 most popular users)
  - OUTPUTS : userIds(list of required users), tweet_counts(total number of their tweets)

This query is demonstrated by the block diagram below also:



As mentioned in neo4j query generation section, we expect all the inputs to the neo4j query to be list of native objects. We put a similar constraint on the inputs to post processing function. Keeping this in mind, to ensure consistency and a seamless flow of information, the outputs of each query(mongoDB, neo4j or postprocessing function) is expected to be a list. Thus each node in the DAG accepts a dictionary as input in which the values are lists and similarly returns a dictionary with list values. The keys in both dictionary is the name of the inputs/outputs, as specified in the query generation.

The only place where the list input breaks is in case of mongoDB query as they require some basic inputs which can directly be provided as native objects(for example the limit input to the above two mongoDB queries).

Further we need to specify which outputs of the queries are to be returned.

The example input file to create the above DAG looks something like this:

```
3
n1 most_popular_hashtags_20
n2 most_popular_mentions_20
n3 active_users
INPUTS:
CONNECTIONS:
n1.hashtag n3.hashtag
n2.userId n3.um_id
RETURNS:
n3.userId
n3.count
```

Observe the structure of the file.

- Line 1 contains the number of nodes in the DAG.
- The following number of nodes lines contain the name of the nodes and each node corresponds to which query.
- Then a line contains the keyword "INPUTS:". The inputs to the queries in the DAG are specified here. For example, had the `n1.hashtag` variable been taken as an input rather than feeding from the output of an upstream query, it would have been specified as `n1.hashtag ["hash1","hash2"]`.
- Then a line containing the keyword "CONNECTION:". Below the connections in the DAG are specified.
- And finally a line contains the keyword "RETURNS:". Below, we specify the outputs of the queries which are to be returned.

Please note that, all the outputs of all the queries can be seen in XComs in airflow and also in the logs of the DAG run. But we provide the user to specify the things of interest to the user, through the RETURN variables. This will be useful in case we provide a functionality to observe the variation of a quantity over periodic DAG runs in future. Presently we don't have such a functionality in our Dashboard, though providing such a functionality shouldn't be difficult.

9.4 DAG in airflow

To create a DAG in airflow, we need to create a file in the dags folder in the AIRFLOW_HOME directory. So, when the user specifies the DAG through out dashboard, we generate a python file in the mentioned folder. The newly created DAG is registered with airflow after sometime(airflow has a heartbeat thread running, which looks for new DAGs in the folder periodically) We generate the code to specify the dag in airflow something like this.

```
task_0 = PythonOperator(
    task_id='node_{}'.format("n1"),
    python_callable=execute_query,
    op_kwargs={'node_name':"n1"},
    provide_context = True,
    dag=dag)

task_1 = PythonOperator(
        task_id='node_{}'.format("n2"),
        python_callable=execute_query,
        op_kwargs={'node_name':"n2"},
        provide_context = True,
        dag=dag)

task_2 = PythonOperator(
        task_id='node_{}'.format("n3"),
        python_callable=execute_query,
        op_kwargs={'node_name':"n3"},
        provide_context = True,
        dag=dag)
task_0 >> task_2
task_1 >> task_2
```

In the above code, the execute query is the function in which we execute queries and pass on their outputs to XComs to be used by the downstream nodes.

```
# Pushing onto XComs
context['task_instance'].xcom_push(k,v)
# Pulling from XComs
context['task_instance'].xcom_pull(task_ids=get_task_from_node(mapp[0]),dag_id =
↪"active_users_dag",key=k)
```

Apart from this, some of the DAG properties which needs to be specified in airflow are generated as the default ones. For example the `start_date` property is specified as the current time.

Airflow provides certain other useful properties which may be of interest to the user(when the system becomes huge). For example, the user can set an email-address on which a notification will be sent in case of the success and/or failure of tasks. But, taking all these inputs though our dashboard to generate the DAG, is like create a complete front-end wrapper over the airflow system, which is not out aim. If the user does wish to use the more involved airflow properties, he/she can always edit the source of the generated dag file. Also, if the need arises to provide the user such functionality through our dashboard, then modifying the code to generate an additional line in the dag python file is easy.

Further on airflow, different views of the DAG can be observed, some of the views which are of particular interest to us are the following :

- Tree view - A view which tells the parallel streams in the DAG. We can specify how many parallel worker threads to have in airflow.
- Graph view - A graph view specifying the connections between the nodes of the DAG.
- Gant view - activates after the DAG has been executed, tells how much time taken by each query to execute.

Also,airflow provides the functionality to schedule the DAG runs periodically and properly stores the logs of each run. This can be leveraged in scenarios in which the user wants to run the same compositional query periodically.

9.5 Creating custom metric

Custom metric can be created on top of the DAG. A custom metric is nothing but a graphical view of the data output from the DAG execution.

To view a custom metric, the user is required to specify the following things:

- A DAG : The outputs of any queries in the DAG can be used to create the custom metric.
- A post processing function : accepts as inputs the outputs of any of the queries in the DAG and outputs a x and y coordinates to be used for plotting.
- Either mapping between the inputs of the post processing function and the outputs of the queries in the DAG or fixed native values to the inputs.

To display the custom metric, the DAG is executed to feed data into the post processing function. The user can choose to view the metric in either of these formats:

- Plot : The x and y coordinates are plotted using plotly through an Ajax call and displayed on the dashboard.
- Table : The values are displayed in table format again using an Ajax call.

An example of creating a custom metric will be provided in the *Dashboard Website* section.

9.6 Code Documentation for DAG abstraction

Here we provide a documentation of the code used to generate, execute the DAG. Module to generate and store the DAG created by the user. Constains functions to generate DAG for the airflow dashboard also.

The *create_dag* module contains the classes:

- *create_dag.DAG*

When we instantiate an object of the class, the network source of the DAG is parsed to get parameters. One can use the function *create_dag.DAG.feed_forward()* to execute the networkx DAG and the function *create_dag.DAG.generate_dag()* to generate an airflow DAG.

Example illustrating how to create a DAG in which no input to no query is constant:

```
>>> queries = {"q1":["query 1",["inp1","inp2"],["out1","out2"]],
                        "q2":["query 2",["inp1"],["out1","out2"]],
                        "q3":["query 3",["inp1","inp2","inp3"],["out1"]]}
>>> types = {"q1":"mongoDB","q2":"PostProcesing","q3":"neo4j"}
>>> constants = {}
>>> # input_graph.txt contains the network of the DAG. For format see the␣
↪documentation
>>> dag = DAG(open("input_graph.txt","r").read(),queries,types,constants)
```

Example of a query in which we use constants:

```
>>> # queries and types as before
>>> constants = {"q1":{"inp1":3}}
>>> # input_graph.txt contains the network of the DAG. For format see the␣
↪documentation
>>> dag = DAG(open("input_graph.txt","r").read(),queries,types,constants)
```

Now to execute the graph provide a function which can execute the queries.

```
>>> dag.feed_forward(<execute>)
```

Create a DAG in airflow, get the plotly div for the DAG to be displayed on the dashboard

```
>>> dag.generate_dag(<dag_name>)
>>> dag.plot_dag()
```

class create_dag.**DAG**(*network_file_source*, *queries*, *types*, *constants*)

Bases: object

This class contains the functions to deal with the abstraction of DAG in our system. Some of the functions to this end, some of the functions are in the views.py file also.

The __init__ is called to initilaize a DAG object. It reads the source of the DAG network file passed to it as a string. It parses the string and extracts te nodes, connections, inputs and returns from the file.

Parameters

- **network_file_source** – a string contains the network specificaion of the DAG
- **queries** – a dictionary of queries with keys as the query names/postprocessing function names
- **types** – a dictionary containing the types of different queries
- **constants** – a dictionary of constant inputs to a query

For the queries parameter, the expectation depends on the type of query:

- For neo4j queries it will be the query code, input, output list
- For mongoDB queries it will be the partially formatted query specification, input, output list
- For post processing functions it will be the function_definition, input, output list

Note: The constants dictionary will contain only the mongoDB queries in current format

Note: We don't do very comprehensive checking if the network is valid. In case it is not a DAG, the user is notified of it. Other than that, there will be some python errors if some other issue is there.

feed_forward(*execute*)
Do a topological sort and then do a BFS of the DAG to execute all the queries. Expect that there is a function to evaluate a query given its inputs(as a dictionary) and returns a dictionary of outputs

Parameters **execute** – a function to execute the queries

generate_dag(*dag_name*)
Create a python file for the DAG in the dags directory of AIRFLOW_HOME. Generate the airflow code for the dag in the file. The templates used in the function are taken in the *create_dag* module.

Parameters **dag_name** – the name of the dag to be generated

Note: Currently it has the relative address of the folder as being contained in the myapp folder. Change it appropriately if you decide to change the airflow home

get_drawable_dag(*G*, *queries*, *types*, *edges*)
Helper function for *create_dag.DAG.plot_dag()*. It gets the locations of the various figures in the plotly plot of the DAG.

Parameters

- **queries** – the queries in the DAG
- **types** – the types of queris. Not used, but can choose to get different colored rectangles for different query types
- **edges** – the edges in the DAG

Returns a directed graph with the connections between inputs and outputs, the locations of bounding rectangles for the queries

plot_dag()
Get the plotly div for the DAG. Get the locations using the helper function and then just plot those and return the html div for the plot

Returns the html div of the DAG plotly plot

CHAPTER

TEN

GENERATING ALERTS USING APACHE FLINK

Tweets are continuously streamed from Twitter. Our system provides a functionality wherein it can detect certain user specified events in the live tweet stream. For example, user can make a specification to find viral hashtags in the sream. We leverage a couple of open-source technologies to detect user specified alerts in the Twitter stream. Apache Flink is an open-source, distributed and high performing stream processing tool. Apache Kafka, also open-source, is another streaming tool which can be used as a message queue for communication between programs in a highly available distributed fashion.

Tweets are continuously streamed from Twitter using the Twitter Streaming API and pushed to a Kafka topic ("tweets_topic") for downstream processes. This tweet stream is processed by Flink programs to detect the specified alerts (one Flink program per alert specification). Alert specifications are given by the user using an abstraction that we describe next.

10.1 Alert Specification Abstraction

Our alert specification abstraction is inspired from Flink's own specification. Each tweet is considered to have 4 attributes - UserId, Hashtags, URLs and User mentions. To specify an alert, user needs to specify the following:

- Filter - values of 0 or more tweet attributes to filter the tweets relevant for the alert.
- Group keys - 0 or more of tweet attributes on which to group and split the tweet stream (1 sub-stream per group).
- Window length (in seconds) - to divide each sub-stream into multiple windows, each of fixed length.
- Window slide (in seconds) - to specify how often to start a new window (windows may overlap if slide < length).
- Count - threshold of count of tweets in any window.

As soon as the count is reached in any window, an alert is generated.

10.2 Back-end process

The specification made by the user in our abstraction is processed as follows:

- The specification is translated to a Flink Java application, compiled using Maven, uploaded and run by Flink server running locally.
- This Flink Java application continuously streams the tweets from the Kafka topic ("tweets_topic"), processes them according to the specification and posts any alerts on a different Kafka topic ("alerts_topic").
- There is a simple Python application which continuously polls for alerts on this Kafka topic and persists any found alerts to MongoDB to be displayed by the dashboard.

10.3 Example - Finding viral hashtags

Let us consider an example where we wish to be notified an alert if any hashtag is getting viral in the twitter stream. Suppose we define a hashtag as viral, if it is used more than 100 times in a span of 60 seconds. Now to describe this in our abstraction, we need to specify the following:

- Filter = None; as we need to consider all tweets.
- Group keys = Hashtag; as we need to create a sub-stream for each hashtag.
- Window length = 60
- Window slide = 60; say, for non-overlapping windows
- Count = 100

10.4 Flink Code Generator Documentation

Here we provide a documentation of the Flink code generator. Module to generate Java code for Flink from the alert specification taken from user on the dashboard. You need to have jinja2 python module and Maven installed.

class flink_code_gen.**FlinkCodeGenerator**

Bases: object

Class to translate alert specification given by user to Java code for Flink. It uses a template defined in flink_template.txt. The user provided specification is translated to Java code and placed at appropriate positions inside the template. This renders the complete Java file. The Java project is then created containing this file and compiled using Maven.

compile_code(*alert_name*)

Compiles the java project for the given alert using maven and creates the jar file.

Parameters **alert_name** – The name of the alert whose code is to be compiled.

Returns The path of the jar file resulting from the compilation.

delete_code(*alert_name*)

Deletes the java project for the given alert name.

Parameters **alert_name** – The name of the alert whose code is to be deleted.

write_code(*alert_name*, *filter_string*, *group_keys*, *window_length*, *window_slide*, *threshold*)

Generates the java code for the given alert specification and writes tha java project for it in the alert's base path. Note: It can raise exception like alert already exists with the given name.

Parameters

- **alert_name** – Name of the alert to be created.
- **filter_string** – Filter specification for the alert. Refer to flink_code_gen.FlinkCodeGenerator._get_filter_code().
- **group_keys** – List of keys to group on. Refer to flink_code_gen.FlinkCodeGenerator._get_duplication_code().
- **window_length** – Length of window in seconds. The threshold will be looked at each window in each sub-stream.
- **window_slide** – Number of seconds after which to start each new window.
- **threshold** – Count threshold for tweets in each window to generate the alert.

10.5 Flink API Documentation

Here we provide a documentation of the Flink API. Module to communicate with the Flink server running locally to do tasks like uploading an alert's jar, running the jar as a Flink job, cancelling a running job and checking the status of jobs.

class flink_api.**FlinkAPI**(*hostname='localhost'*, *port=8081*)

Bases: object

Class to communicate with Flink server.

cancel_job(*job_id*)

Cancels the Flink job with the given job_id.

Parameters **job_id** – job_id of the Flink job to cancel.

check_job_status_all()

Check the status of all jobs run in the past.

Returns Dictionary having key as alert_name (which was supplied in run_jar) and value as status of the last job of that alert.

run_jar(*alert_name*, *flink_jar_id*)

Runs the given jar_id on Flink.

Parameters

- **alert_name** – Name of the alert to which the jar_id belongs.
- **flink_jar_id** – jar_id to be run on Flink.

Returns The job_id of the Flink job started as returned by the Flink server.

upload_jar(*jar_path*)

Uploads the jar of the alert to the Flink server running locally.

Parameters **jar_path** – Path of the jar file to upload.

Returns The jar_id as returned by Flink server.

10.6 Flink Alerts Consumer

Here we provide a documentation of the Kafka consumer (for the topic 'alerts_topic") for Flink alerts. Module to read alerts from the Kafka topic ("alerts_topic") where Flink applications post alerts and then put them in MongoDB.

kafka_flink_alerts_consumer.**insert_records**(*records*)

CHAPTER

ELEVEN

BENCHMARKING THE QUERY ANSWERING

11.1 Neo4j queries

We divide the number queries to be answered into two types. Simple neo4j queries and complex neo4j queries. We have these queries and a set list of hashtags and userIds present in the system. We generate a list of queries to be answered by randomly picking attribute to create the query. Thus creating a single query consists of these two steps:

- Pick a templated cypher query
- Randomly pick the values of inputs to the query from a static list to create the query.

Similarly, we create a list of queries. The templated cypher queries are put into the simple or complex basket by seeing the time taken to answer a single query.

Having obtained the queries, we spawn multiple threads each of which opens a connection to the neo4j database in form of a session. Pops a query from the queue and delegates it to be answered by the database. To observe the optimal number of connections to be opened to the database, we plot the query answering rate verses the number of parallel connections.

11.1.1 Simple Queries

The simple queries considered are these:

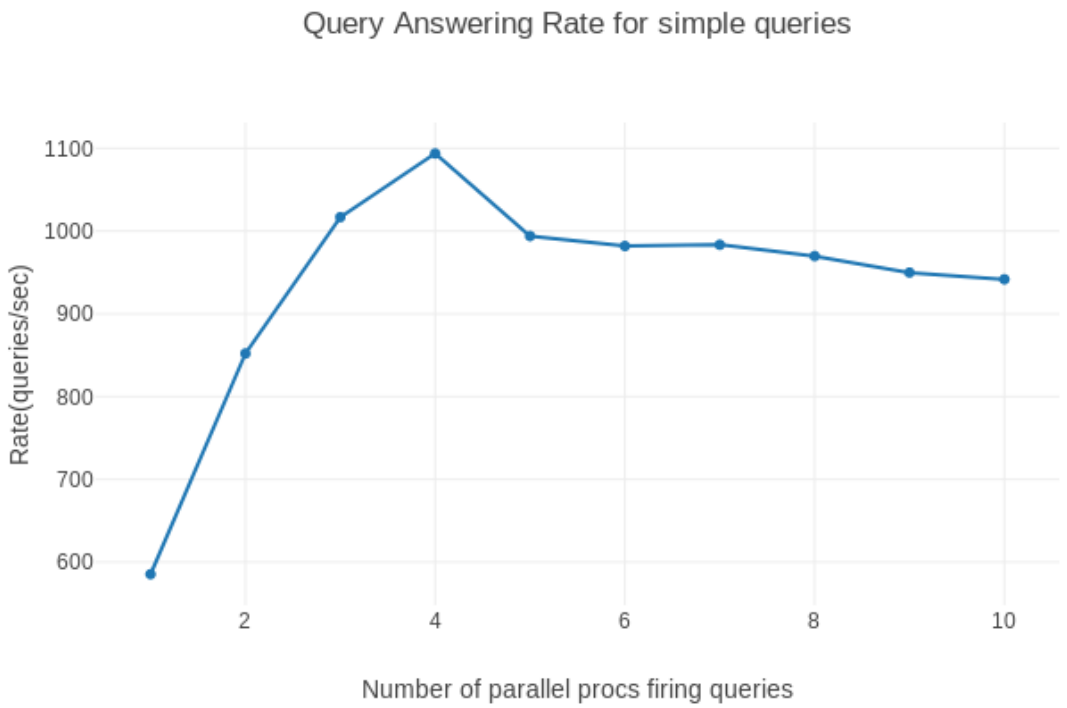
- Return count of distinct users who have used a hashtag
- Return count of ditinct users who follow a certain user
- Return the number of times a user was followed in a given interval
- Find the number of current followers of users who have used certain
- Find the count of users who tweeted in a given interval

The cypher code of these queries can also be seen here.

```
# Return count of distinct users who have used a hashtag
q1 = """match (u:USER)-[:TWEETED]->(t:TWEET)-[:HAS_HASHTAG]->(:HASHTAG{text:"{{h}}"})
with distinct u as u1 return count(u1)"""

# Return count of ditinct users who follow a certain user
q2 = """match (x:USER)-[:FOLLOWS]->(:USER {id:{{u1}}}), (x)-[:FOLLOWS]->(:USER {id:{
↪{u2}}})
with distinct x as x1 return count(x1)"""

# Return the number of times a user was followed in a given interval
q3 = """
```

```
match (fe:FOLLOW_EVENT)-[:FE_FOLLOWED]->(u:USER {id:{{u}}})
where fe.timestamp > {{t1}} and fe.timestamp < {{t2}}
return count(fe)
"""

# Find the number of current followers of users who have used certain hahstag
q4 = """
match (x:USER {id:{{u}}})-[:TWEETED]->(:TWEET)-[:HAS_HASHTAG]->(h:HASHTAG), (f:USER)-
↪[:FOLLOWS]->(x), (f)-[:TWEETED]->(:TWEET)-[:HAS_HASHTAG]->(h)
with distinct f as f1 return count(f1)
"""

# Find the count of users who tweeted in a given interval
q5 = """
match (te:TWEET_EVENT)-[:TE_TWEET]->(:TWEET)-[:RETWEET_OF]->(t:TWEET), (te)-[:TE_
↪USER]->(:USER {id:{{u}}}), (x:USER)-[:TWEETED]->(t)
where te.timestamp < {{t1}} and te.timestamp > {{t2}}
with distinct x as x1 return count(x1)
```

Using these templated queries we generate the list of simple queries to be fired as described above and observe the query ansering rate with number of parallel sessions. This graph is obtained:



As we obtain the best query rate is obtained on opening 4 parallel sessions of about 1100 queries/second in answering of simple queries.

11.1.2 Complex Queries

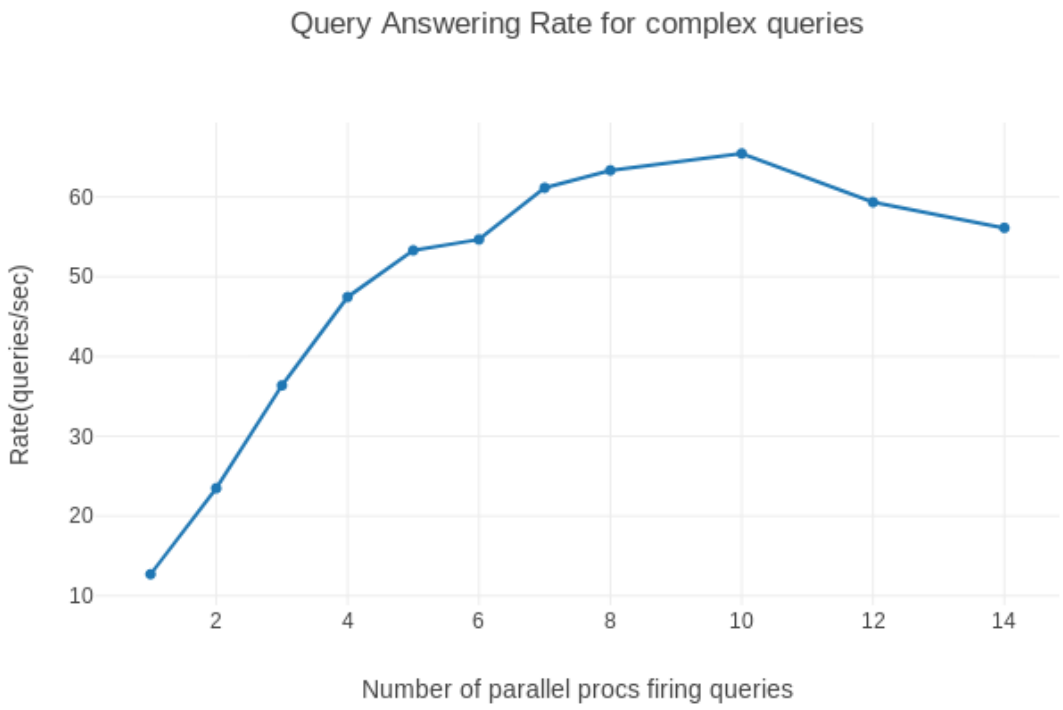
For complex queries we try to consider those queries which require more than one hop in the network. The complex queries considered are these:

- Find common followers of two users
- Return the users which follow a user u and tweeted t(which mentions same u) b/w t1 and t2
- Find users which have tweeted tweet t1(retweet of another tweet t containing hashtag hash AND such that t1 itself contains the same hashtag hash) b/w time1 and time2 and follows u
- Find users which follow user with id u1 and follow user u which tweeted b/w t1 and t2 containing hashtag hash

The cypher code of these queries is as follows.

```
# Find common followers of two users
q6 = """
MATCH (u1 :USER {id:{{id1}}}),(u2 :USER {id:{{id2}}}), (user :USER)
WHERE (u1) <-[:FOLLOWS]- (user) AND (user) -[:FOLLOWS]-> (u2)
RETURN count(user)
"""

# Find the users which follow a user u and tweeted t(which mentions same u) b/w t1␣
↪and t2
q7="""
MATCH (run:RUN) -[:HAS_FRAME]-> (frame1:FRAME)
WHERE  frame1.end_t >= {{t1}} AND frame1.start_t <= {{t2}}
MATCH (frame1) -[:HAS_TWEET]-> (event1 :TWEET_EVENT), (event1) -[:TE_USER]-> (x :USER
↪{}), (event1) -[:TE_TWEET]-> (t :TWEET {}), (x :USER {}) -[:FOLLOWS]-> (u :USER {id:
↪{{u}}}), (t) -[:HAS_MENTION]-> (u :USER {id:{{u}}})
RETURN  COUNT(x)
"""

# Find users which have tweeted tweet t1(retweet of another tweet t containing␣
↪hashtag hash AND such that t1 itself contains the same hashtag hash) b/w time1 and␣
↪time2 and follows u
q8 = """
MATCH (run:RUN) -[:HAS_FRAME]-> (frame1:FRAME)
WHERE  frame1.end_t >= {{time1}} AND frame1.start_t <= {{time2}}
MATCH (frame1) -[:HAS_TWEET]-> (event1 :TWEET_EVENT), (event1) -[:TE_USER]-> (x :USER
↪{}), (event1) -[:TE_TWEET]-> (t1 :TWEET {}), (x :USER {}) -[:FOLLOWS]-> (u :USER
↪{id:{{u}}}), (t :TWEET {}) -[:HAS_HASHTAG]-> (:HASHTAG {text:'{{hash}}'}), (t1␣
↪:TWEET {}) -[:HAS_HASHTAG]-> (:HASHTAG {text:'{{hash}}'}), (t1) -[:RETWEET_OF]-> (t)
RETURN  COUNT(x)
"""

# Find users which follow user with id u1 and follow user u which tweeted b/w t1 and␣
↪t2 containing hashtag hash
q9 = """
MATCH (run:RUN) -[:HAS_FRAME]-> (frame1:FRAME)
WHERE  frame1.end_t >= {{t1}} AND frame1.start_t <= {{t2}}
MATCH (frame1) -[:HAS_TWEET]-> (event1 :TWEET_EVENT), (event1) -[:TE_USER]-> (u :USER
↪{}), (event1) -[:TE_TWEET]-> (t :TWEET {}), (x :USER {}) -[:FOLLOWS]-> (u1 :USER
↪{id:{{u1}}}), (x) -[:FOLLOWS]-> (u), (t :TWEET {}) -[:HAS_HASHTAG]-> (:HASHTAG
↪{text:'{{hash}}'})
RETURN  COUNT(x)
"""
```

Using these templated queries we generate the list of complex queries to be fired as described above and observe the query answering rate with number of parallel sessions. The following graph is obtained:



As we obtain the best query rate is obtained on opening 10 parallel sessions of about 65 queries/second in answering of complex queries.

Further observe that the peak performance in query answering is obtained at lesser number of parallel connections as compared to the compex case. This is because there is overhead in maintaining parallel connections in neo4j, which involves maintaining the sesions and delegating the queries to the database. This overhead is much more prominent in case when the queries itself take much less time in answering them as compared to the complex case where the overhead gets ammortised better.

11.2 MongoDB queries

MongoDB queries are generally answered very fast in comparison to the neo4j queries, which is owing to the intended schemas of the two databases. Thus, no further observations were made in mongoDB query answering apart from observing that all the queries are answered in mili second scale.

11.3 Code Documentation for benchmarking

Here we provide a documentation of the code used in this sections functionality. Module to benchmark the query answering rate for neo4j.

- Generate a list of queries.
- Open multple neo4j sessions.
- See what is the peak rate.

query_answering.**answer_queries**(*query_l*)
Function to answer all queries from a list of queries in sequential manner

Parameters **query_l** – the list of cypher queries

query_answering.**answer_queries_par**(*query_l*, *num_procs*)
Function to answer all queries from a list of queries in concurrent manner. Spawn <num_procs> number of processes. Each process opens a session and executes a cypher query.

Parameters

- **query_l** – the list of cypher queries
- **num_procs** – number of processes to create

Note: There will only be atmost k number of real parallel process in the system, where k is number of cores. This number is further limited by the session management of neo4j, which is what we observe in the profile difference between simple and complex queries.

---

query_answering.**answer_query**(*query*)
Answer a single cypher query <query>

Parameters **query** – cypher query to be answered

query_answering.**generate_random_queries**(*total*)
Generate a list containing <total> cypher queries. Consider templated cypher queries and lists of attributes. Randomly put in the attributs into the cypher query tempate to get an executable query.

Parameters **total** – number of queris to generate

Note: Obiviously, one need to change the list of attibutes accordingly, if they choose to benchmark on a different dataset.

CHAPTER

TWELVE

DASHBOARD WEBSITE

12.1 Major parts of the dashboard website

We have organised the dashboard website into tabs with each tab containing associated APIs and functionality. Each tab is futher divided into sub tabs. We enlist the major tabs and subtabs and the functionality contained there in, to give an overview of the hierarchy of the website.

12.1.1 Hashtags

This tab contains the functionality to view major statistics associated with hashtags. Though as we will see, the functionality in this tab can entirely be emulated by creating a suitable DAG, but we choose to keep a separate option to get some common stats about the common entities like hastags, user mentions and urls.

The Hashtags tab contains three subtabs:

- Top 10 : Takes as inputs the start time and the end time, to output the 10 most popular hashtags in the said interval with the number of tweets containing the hashtag
- Usage Plot : Takes as input a hashtag, start time and end time, to output the plot of how the usage(as number of tweets in which the hashtag occurs) of the hashtag has changed over the interval.
- Sentiment Plot : Takes as input a hashtag, start time and end time, to output the plot of how the sentiment associated with the hashtag has changed over the interval.

12.1.2 Mentions

The Mentions tab contains the major statistics concerning user mentions. It has the follwoing three subtabs:

- Top 10 : Takes as inputs the start time and the end time, to output the 10 most popular users in the said interval with the number of tweets in which the user is mentioned.
- Usage Plot : Takes as input a user, start time and end time, to output the plot of how the mention frequency(as number of tweets in which the user is mentioned) of the user has changed over the interval.
- Sentiment Plot : Takes as input a user, start time and end time, to output the plot of how the sentiment associated with the user has changed over the interval.

12.1.3 URLs

The URLs tab contains the major statistics concerning urls. It has the follwoing three subtabs:

- Top 10 : Takes as inputs the start time and the end time, to output the 10 most popular urls in the said interval with the number of tweets containing the url.

- Usage Plot : Takes as input a url, start time and end time, to output the plot of how the usage(as number of tweets in which the url occurs) of the url has changed over the interval.
- Sentiment Plot : Takes as input a url, start time and end time, to output the plot of how the sentiment associated with the url has changed over the interval.

12.1.4 Alerts

12.1.5 DAG

We expalin about DAGs in detail in section *Composing multiple queries : DAG*. We assume the reader has read though the section and is aware with the terminology.

This tab contains the functionalities to create and view DAGs. It has the following subtabs:

- Create Neo4j Query : Contains the APIs to create a neo4j queries. The user provides the inputs for query cration thorugh a simple form.
- Create MongoDB Query : Contains the APIs to create mongoDB queries.
- Create Post-Processing Function: Contains the APIs to create a post processing function. The form contains a file upload field thorugh which the file containing the python code for the function needs to be uploaded.
- All Queries : A color coded list of all queries created by the user, along with their input and output variables names. The user can delete queries from here.
- Create DAG : Compose the queries seen in the list of queries to create a DAG. The structure need to be specified in a file which needs to be uploaded.
- View DAG : Contains a list of DAGs created by the user. Also contains a button through which the user can go the airflow dashboard. Apart from that, with each DAG there is a "View" button which redirects to a page containg the strucutre code and the plotly graph of the DAG.
- Create Custom Metric : Contains a form in which the user needs to specify a DAG and a post processing function to create metric and view it either in plot/graph format.

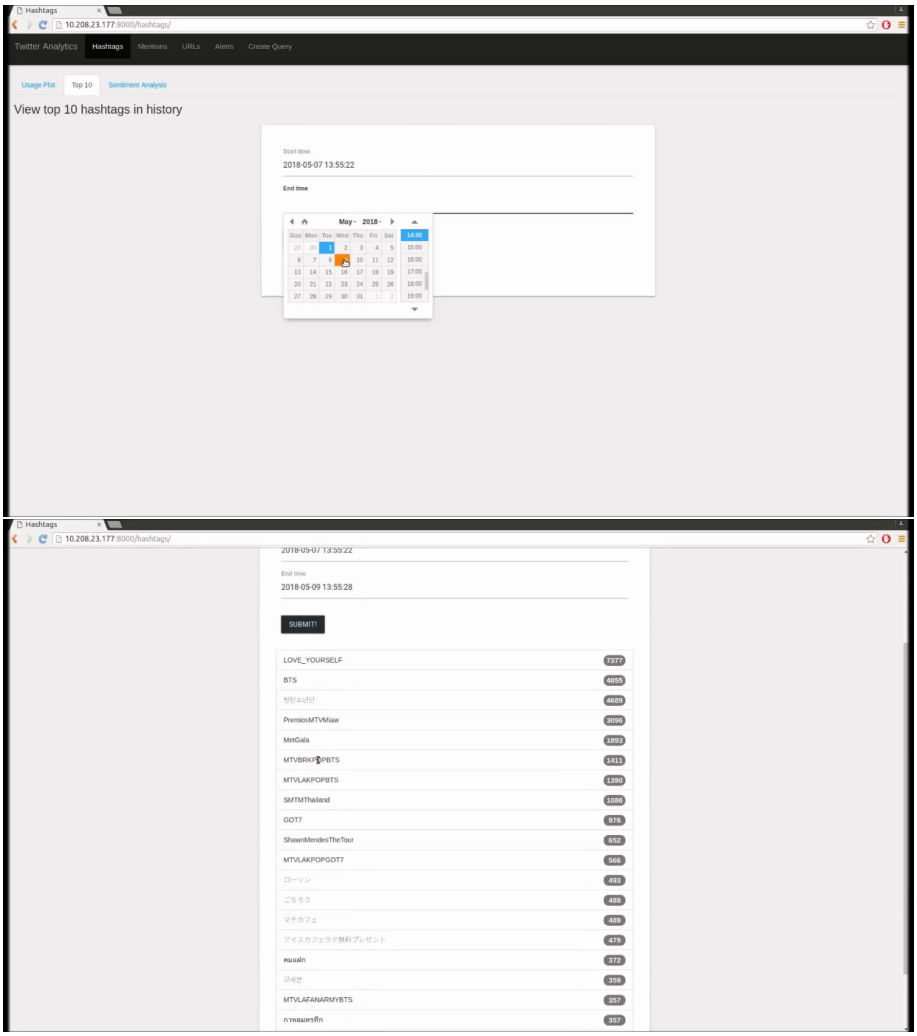
12.2 Use Cases

Here we walk through some major use cases of the system with snapshots to give the reader some headway on how to use the system. The system has been designed, keeping in mind that the end user may not be much proficient in computer technology and has been strucutured to be intuitive and simple. Nonetheless, the authors feel that these use case should be enough to get the user started.

Also, please notice that the easlier use cases may be used in the later ones. So it's better the reader goes through these in order.

12.2.1 Viewing top 10 popular hashtags

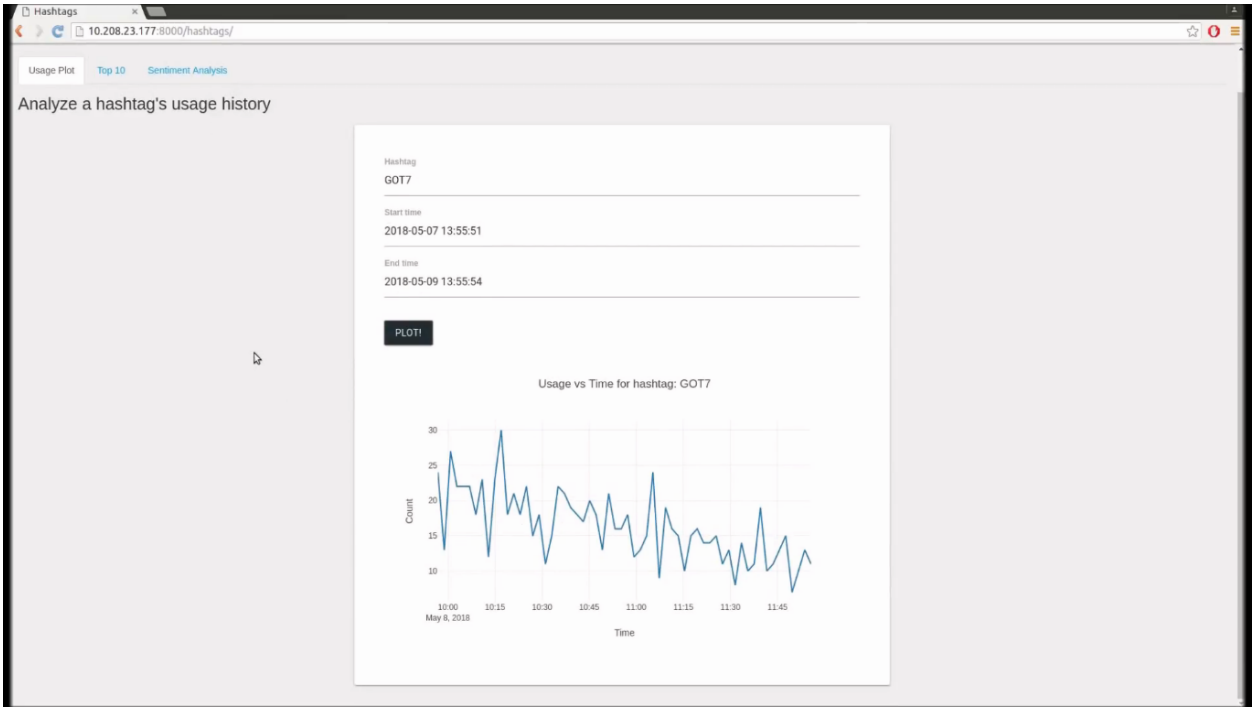
Go to the Hashtags/Top 10, enter the required fields. The list of top 10 most popular hahstags will be displayed below.



Lets take a hashtag and view its statsitics in below couple of use cases.

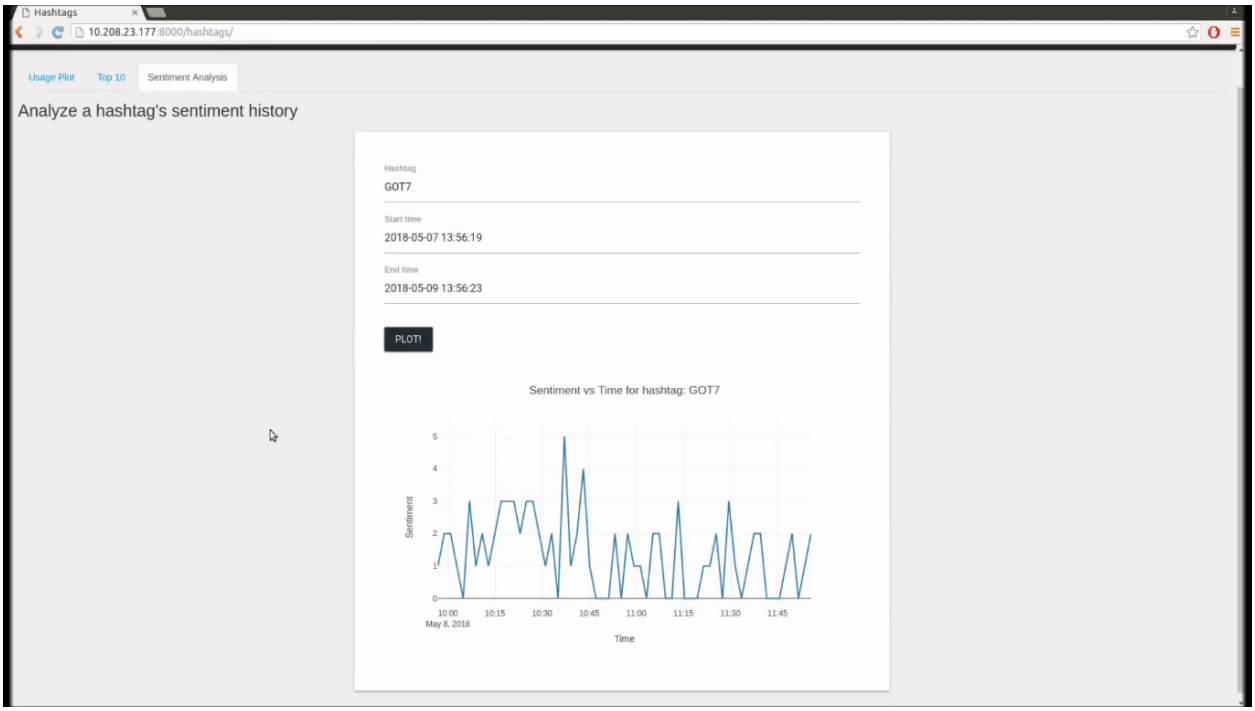
12.2.2 Viewing usage history of hashtag

Let's see how the usage of hashtag "GOT7" has changed over a period of 2 days.



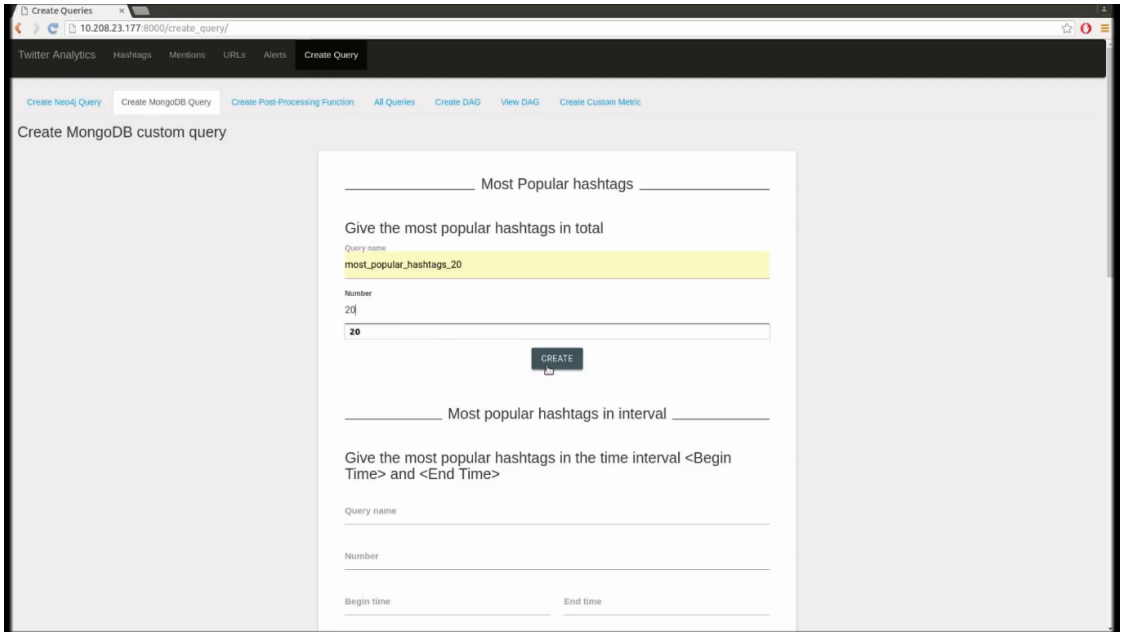
12.2.3 Viewing sentiment history of hashtag

Let's see how the sentiment about hashtag "GOT7" has changed over the same period of 2 days.



12.2.4 Creating a mongoDB query

Let's create a mongo DB query named "most_popular_hashtags_20" to give us the 20 most popular hashtags. Specify the variables and click "create" to create the query.

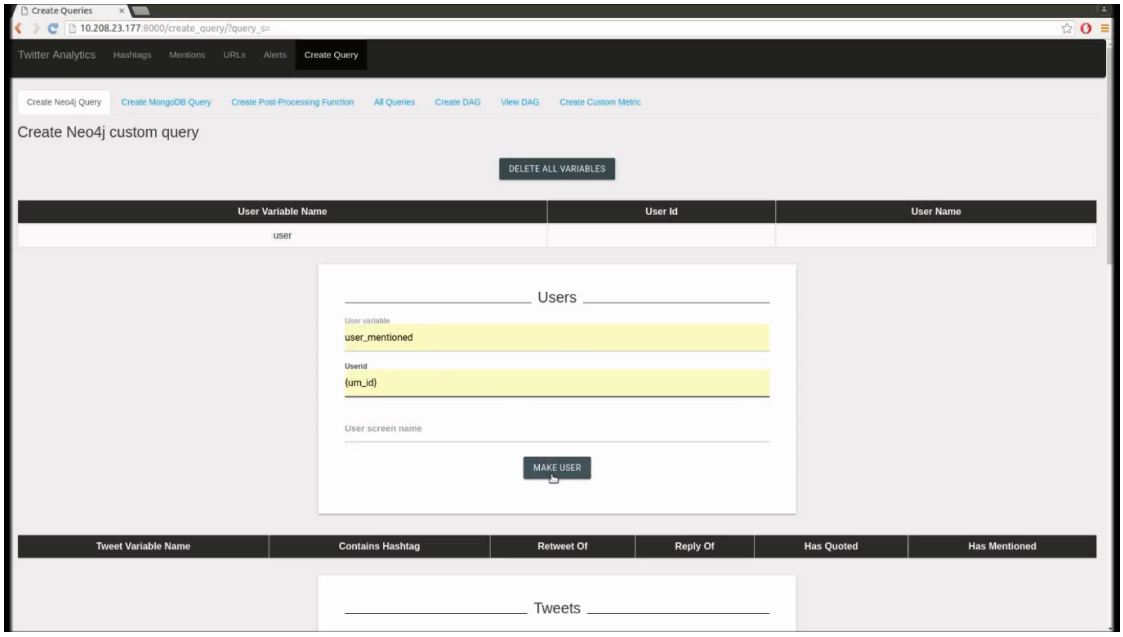


Similarly other mongoDB queries can be created.

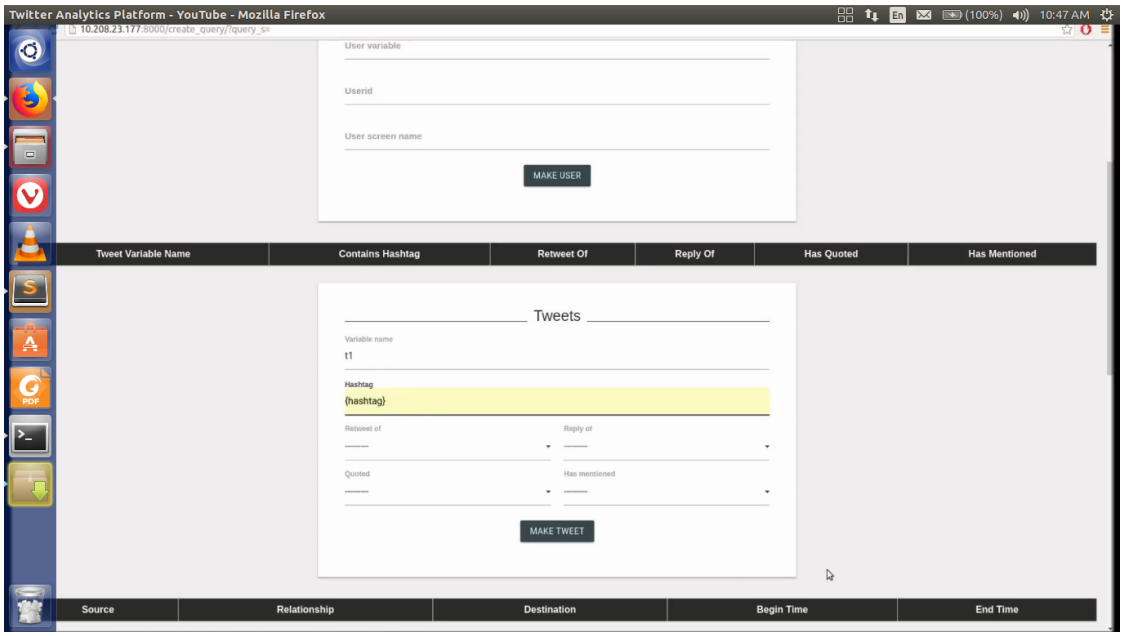
12.2.5 Creating neo4j queries

To crate a neo4j query we need to create user and tweet entities and relationships between them. Here we show how to create the neo4j to get userIds and their tweet counts who have used one of the hashtags from a list of hashtags atleast once and have tweeted with one of the popular user mentions from a list of userIds atleast once, mentioned in *Building a DAG from queries*.

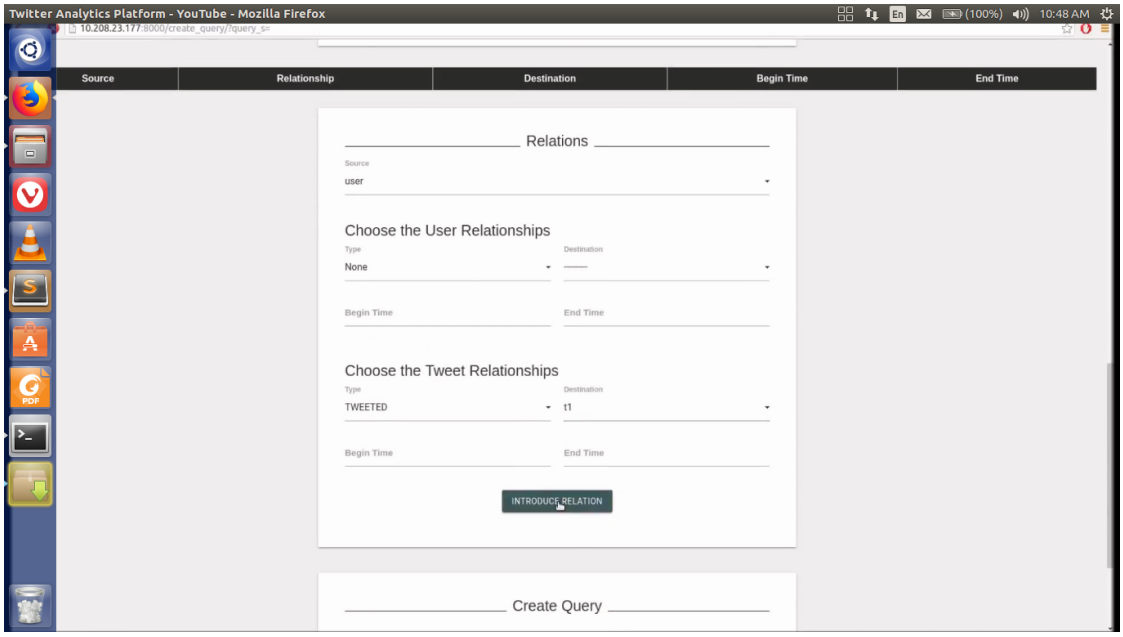
Create a user variable named user with no attributes. Also create a user variable named user_mentioned having variable attribute {um_id}. The curly braces specify that the attribute is variable.



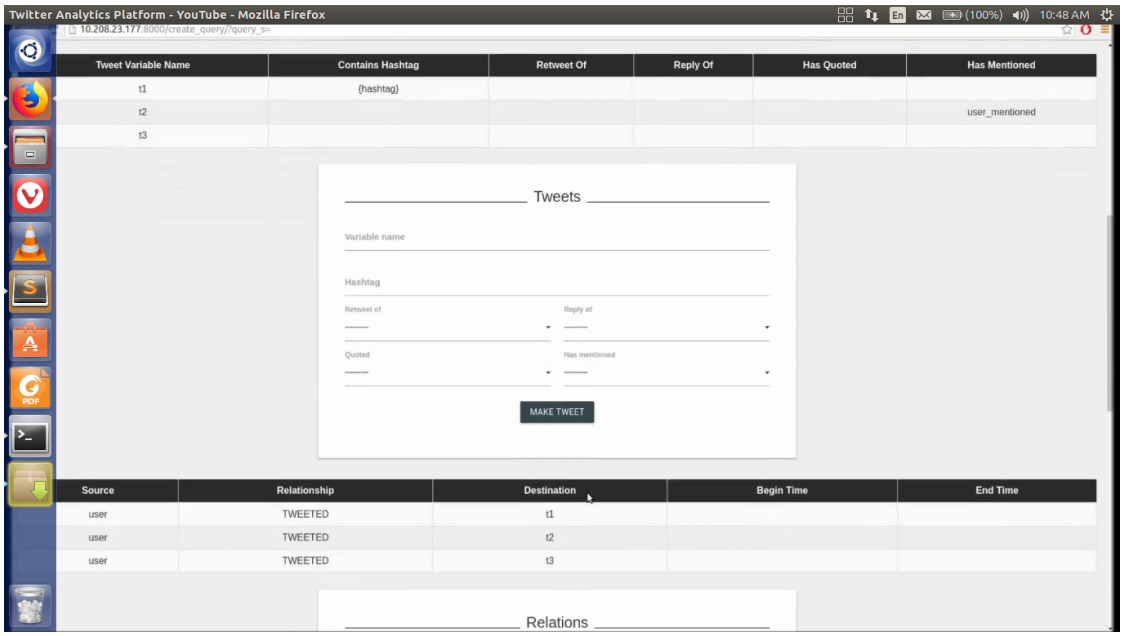
Let us now create some tweets. Create a tweet named t1 having variable hashtag {hashtag}. Create a tweet t2 which has a mention of user User_mentioned, which was created above. Also, create a tweet t3 having no attributes.



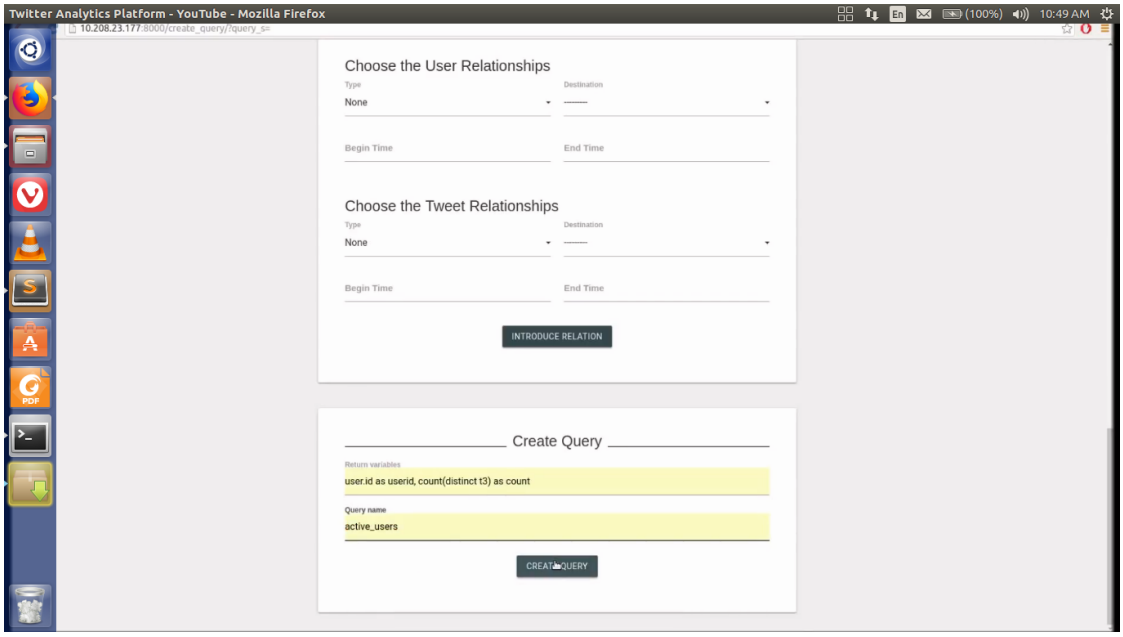
Lets now create some relation ships. Create the relationships, user tweeted tweet t1, user tweeted tweet t2 and user tweeted t3.



So finally we have 2 user variables, 3 tweet variables and 3 relationships between the entities. This can be seen in this image where a screenshot of the tweets and relationships listing is shown.

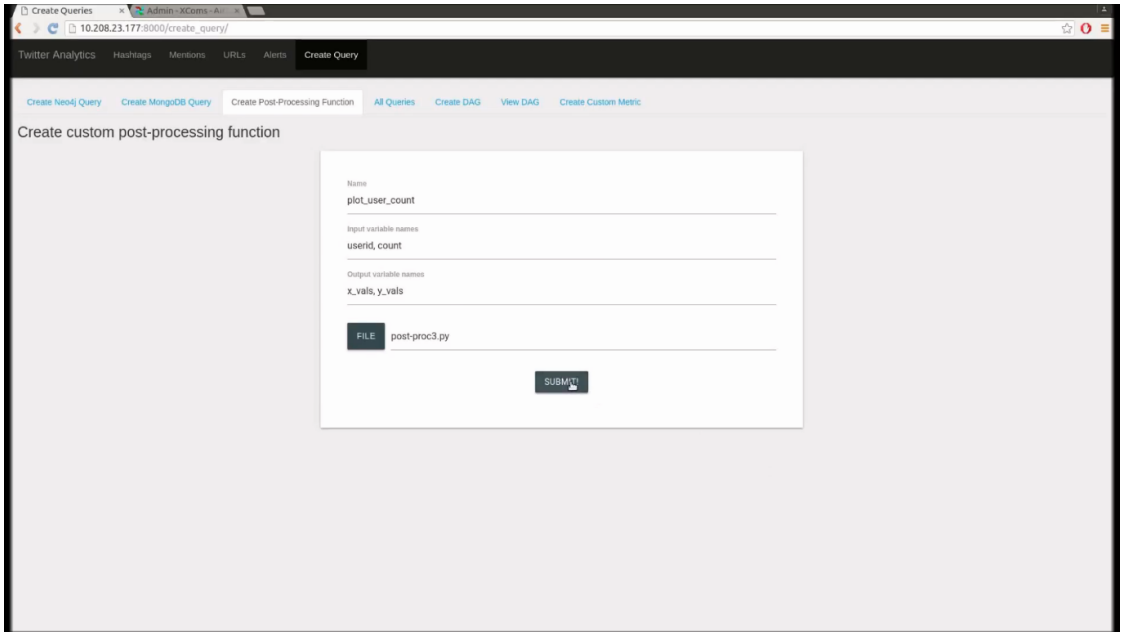


To create the query specify the return variables and the query name.



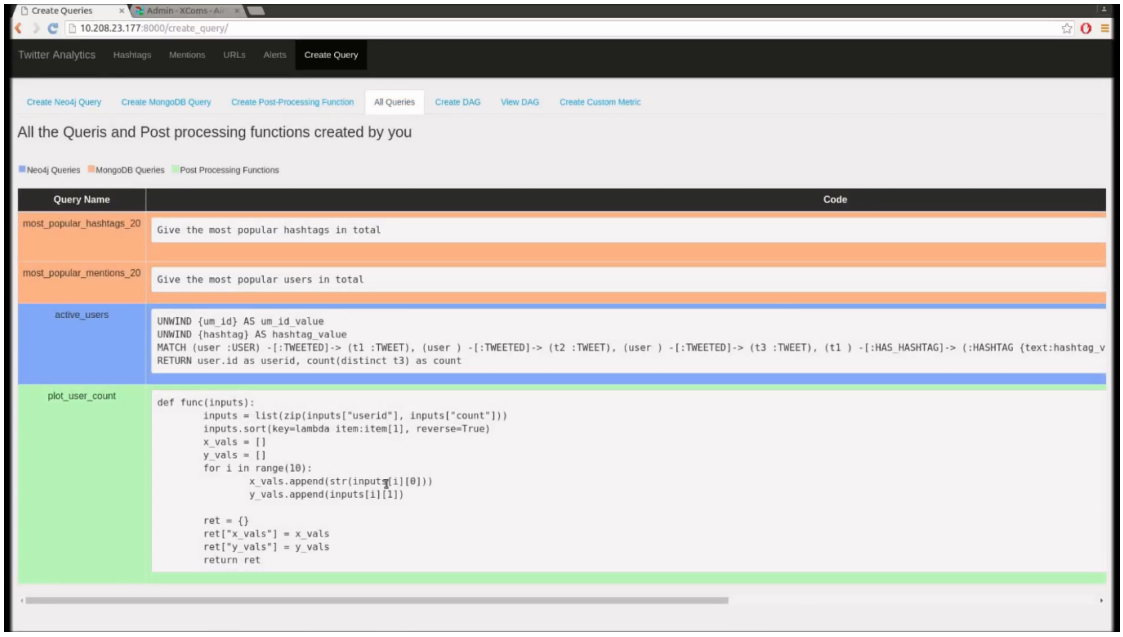
12.2.6 Create Post processing function

Select a file containing the python code to create a post processing function



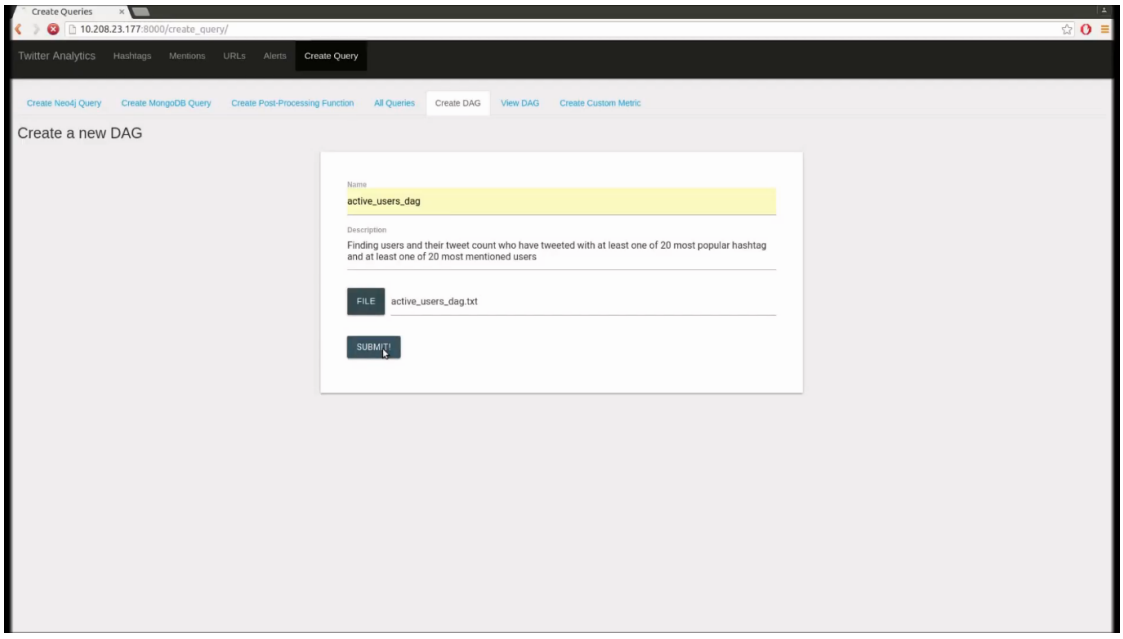
12.2.7 View Queries

To view the queries navigate to DAG/All Queries. As you can see here, currently we have 4 queries, 2 mongo DB and 1 neo4j and 1 post processing function. Additionally, you can see the cypher code generated for the neo4j query and the code of the post processing function in this screenshot:



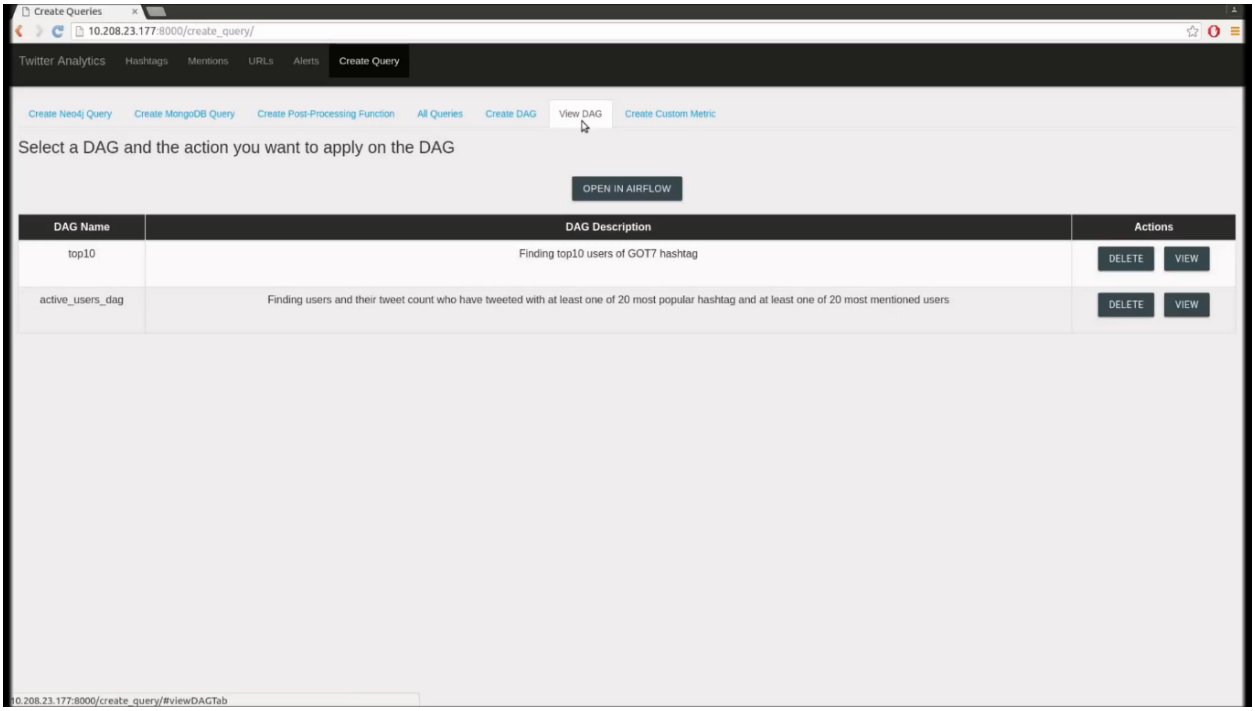
12.2.8 Create DAG

Now let us create the DAG to get the most active users as mentioned in *Building a DAG from queries*. Input the name of the DAG as "activer_users_dag", optionally the description and the file containing the structure specification of the DAG.

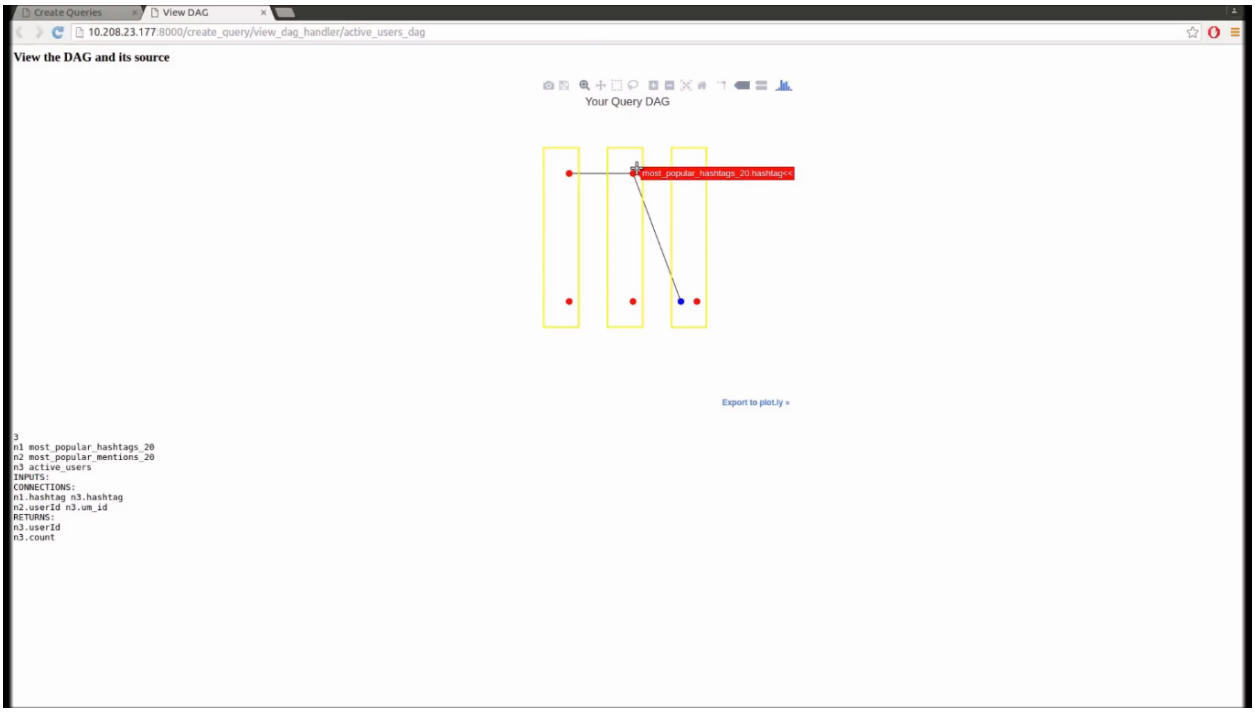


12.2.9 View DAGs

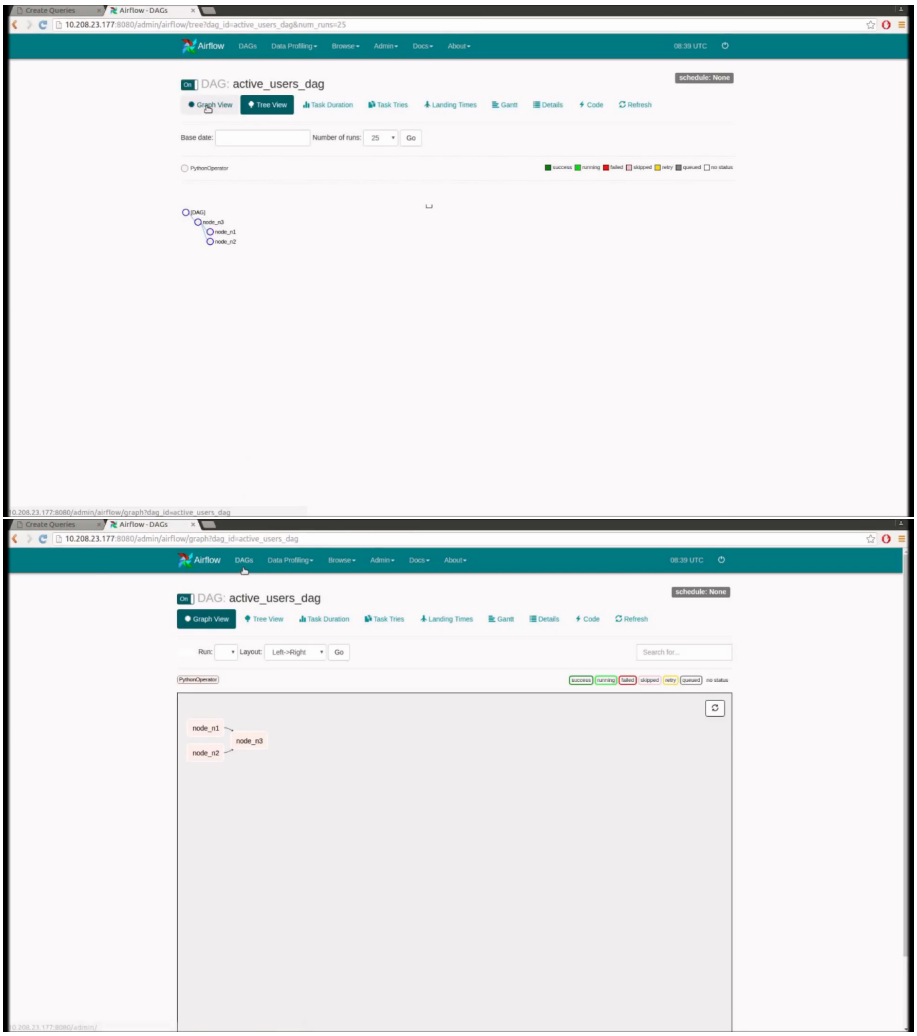
Navigate to the View DAG subtab to view all the created DAGs.



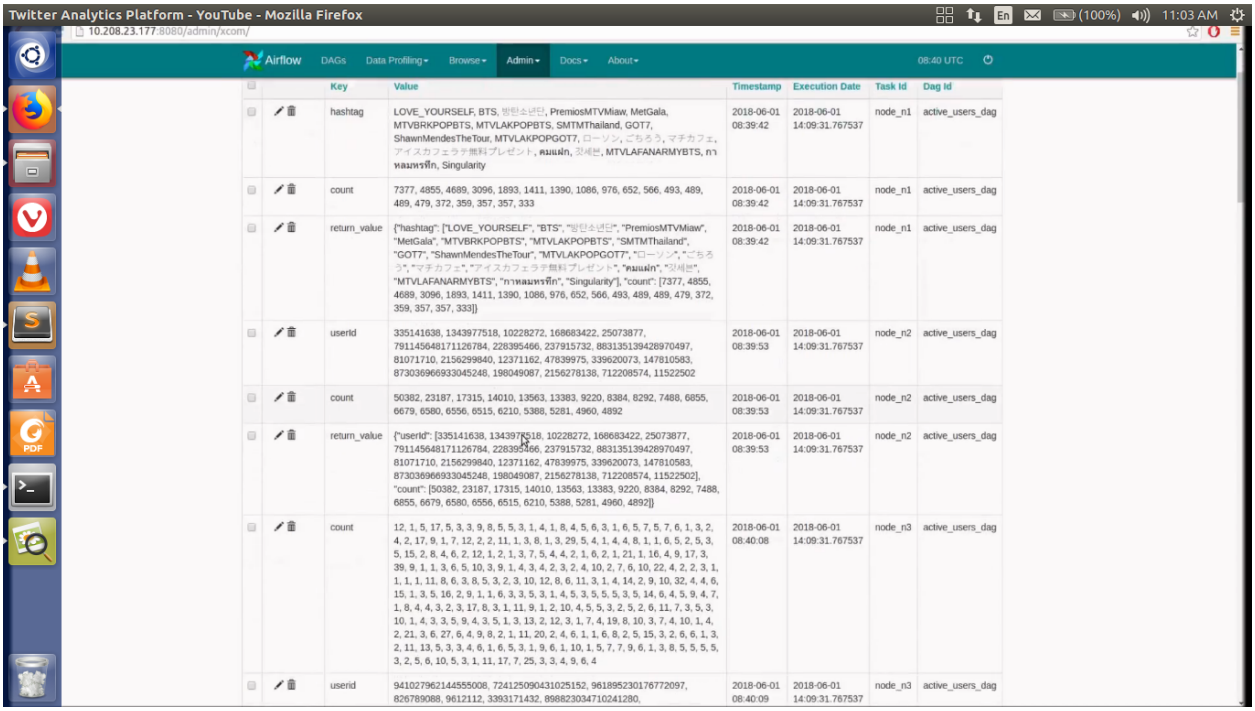
We can view a DAG by clicking "View" button against it. You can see how outputs from one query are feeding into the inputs of another query. Beneath in the screenshot you can see the structure of the DAG in code as well.



Now let us view our DAG in airflow. Here you can see the tree view and the graph view of the DAG.

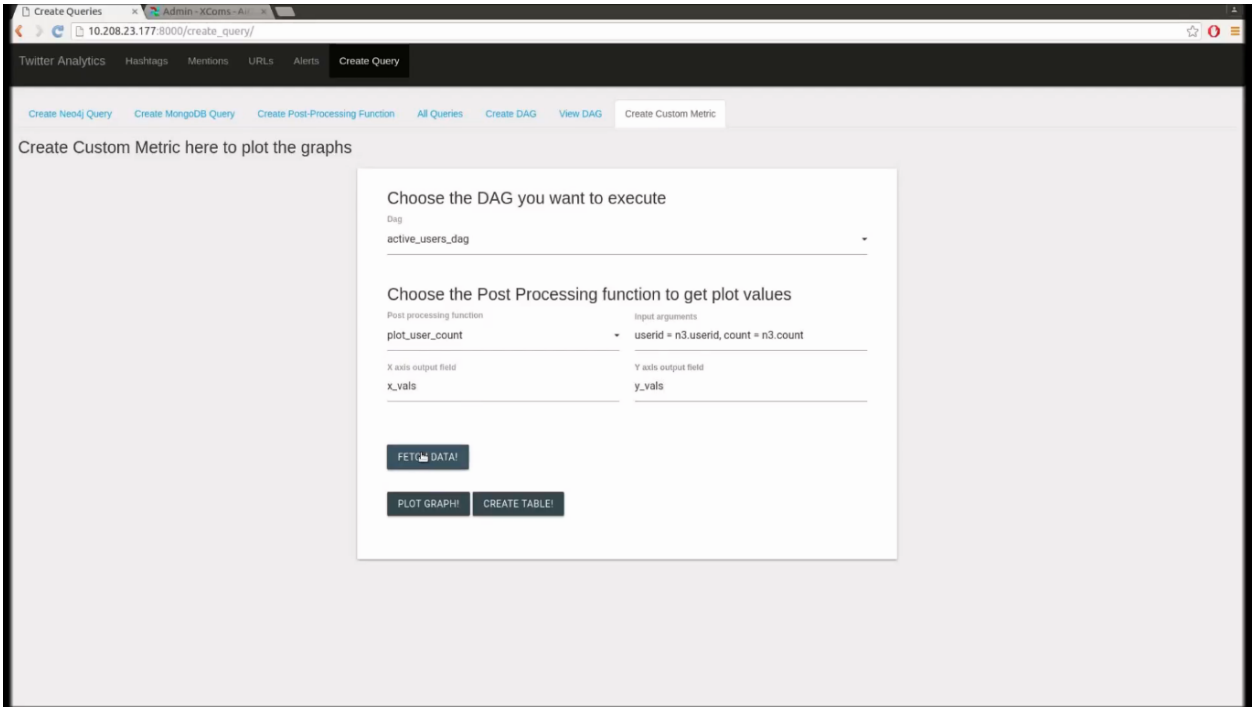


Execute the DAG in airflow and navigate to XComs list to see the outputs of all the queries. A screensot of the XComs list is provided here.



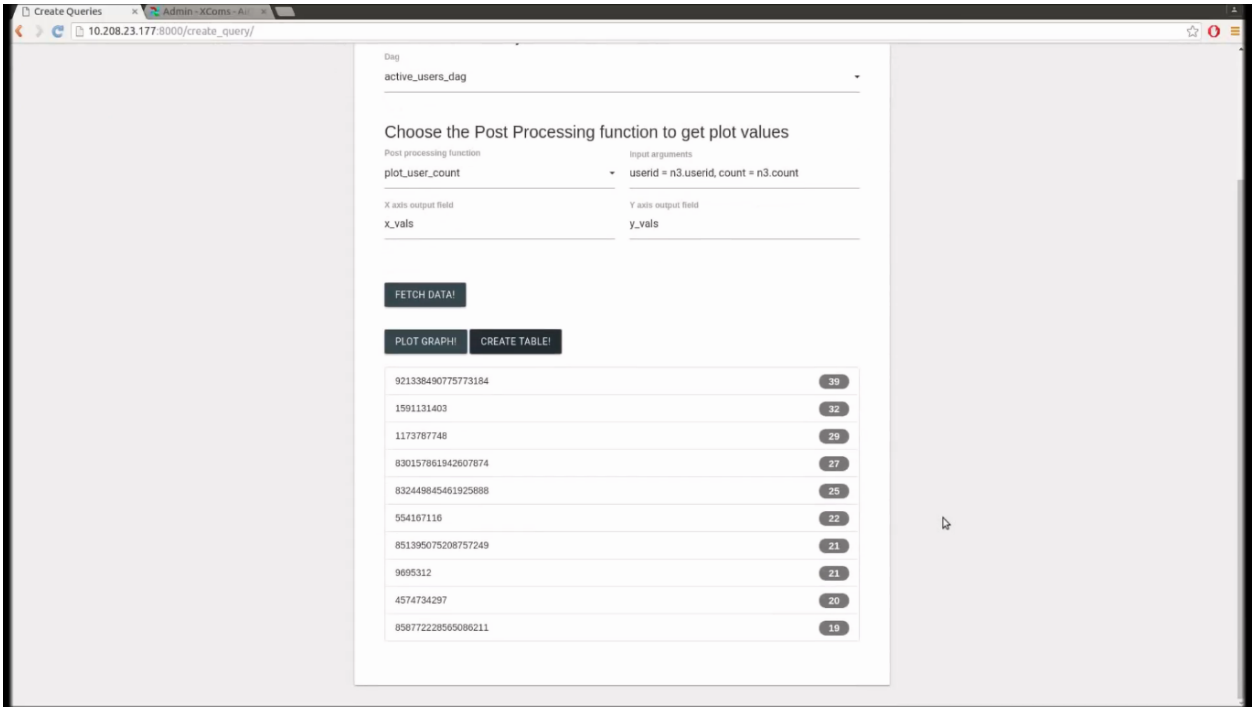
12.2.10 Create Custom metric

To create the custom metric, we need to specify the DAG which we want to execute, choose a post processing function which outputs the x and y coordinates and create a mapping between the outputs of the DAG and inputs of the post processing function. Shown here is how to create a custom metric on the most active users DAG to plot the 10 top active user Ids with their number of tweets:



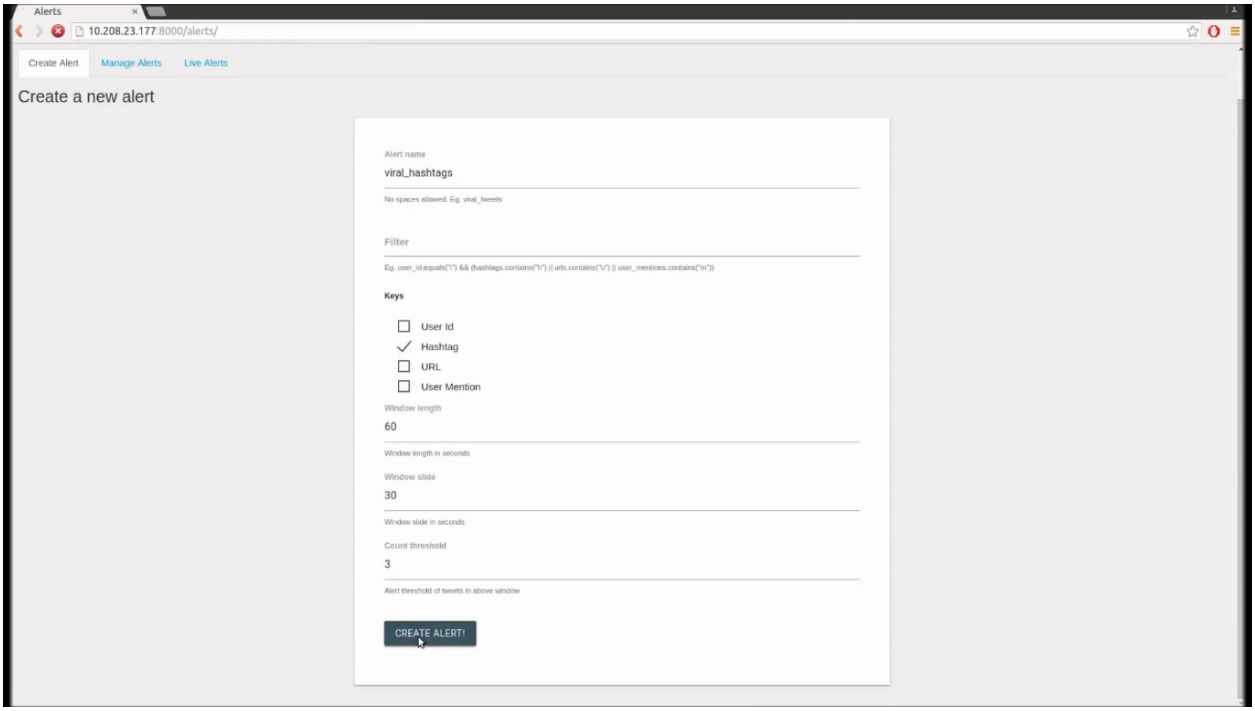
When you click Fetch data the DAG will be executed to feed data into the post processing function. You can now view

in a plot or table format by clicking on "PLOT GRAPH!" and "CREATE TABLE!" respectively. The table will look something like this:



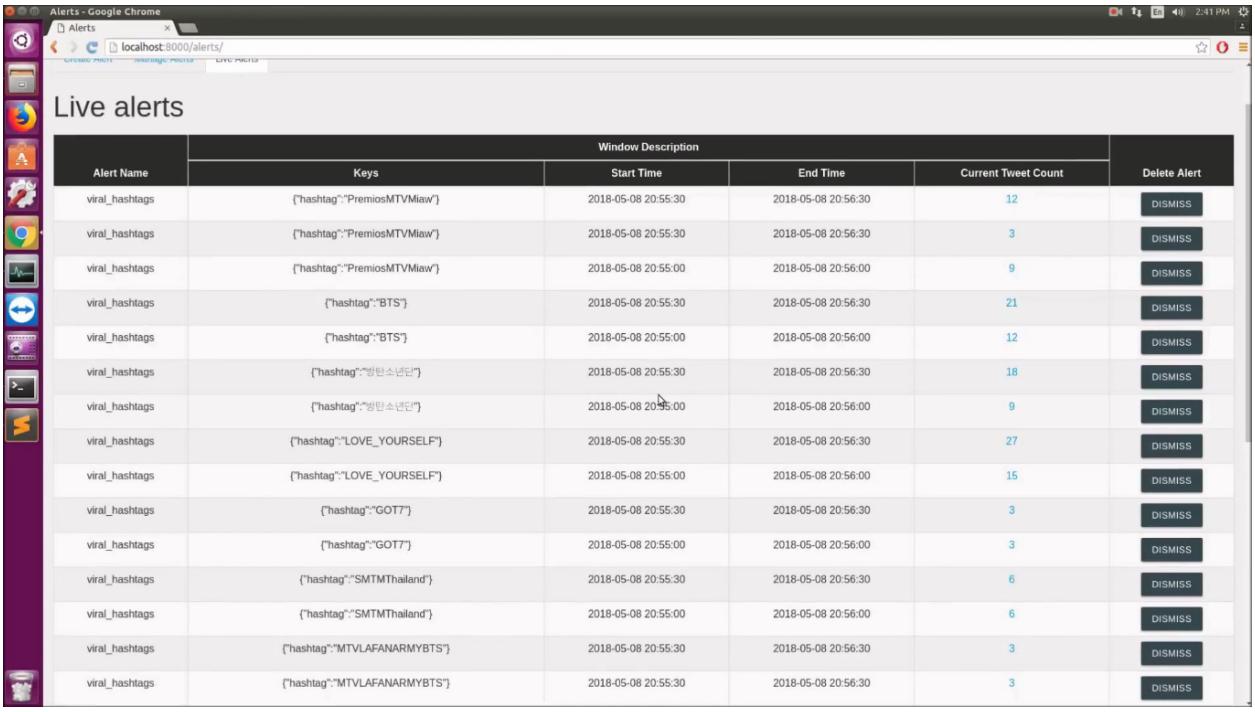
12.2.11 Create Alert

To create an alert on the tweet stream, we need to specify the alert name, the filter, choose keys on which to group and partition the tweet stream, the window length, the window slide and the count threshold. Let's create a hashtag "viral_hashtags" to notify when a hashtag frequency exceeds 3 in the past window of 60 seconds, the window sliding ahead by 30 seconds.



12.2.12 View Alerts

The alerts are generated as real time tweets are put into the kafka queue.



In the end, the best way to figure out the system is to get your hands dirty with the system! To get the system on your local system, the reader should see *Getting the system running*

CHAPTER

THIRTEEN

GETTING THE SYSTEM RUNNING

13.1 Setting up the environment

We recommend getting conda(or miniconda if you are low on disk space). Installing conda is easy, and the installation instructions can be found on the download page itself. After installation of conda, run the following commands:

```
// create a new virtual environment
conda create --name twitter_analytics python==3.6 --file requirements_conda.txt
// clone the repo
git clone https://github.com/abhi19gupta/TwitterAnalytics.git
// cd into the repo
cd TwitterAnalytics
// activate the virtual envt
source activate twitter_analytics
// install the required modules.
pip install -r requirements_pip.txt
.
.
// deactivate the virtual environment
source deactivate
```

Please note that the requirements mentioned above are a superset of system's actual requirements. It may contain some ML related modules which were used while trying NER on tweets, but are not used in the final system.

The entire system has been designed and tested on Ubuntu 16.04 machine. It may work on Windows with appropriate tweaks.

Apart from these, the user also needs to install the following:

- MongoDB: Follow the installation instructions on this MongoDB installation link.
- Neo4j: Follow the installation instructions on this Neo4j installation link.
- Apache Flink: Download the Flink 1.4.2 binary from this Flink binary link. If this does not work, download the latest version from the Flink download page. Once downloaded, simply extract the zip and the installation is complete. You need to use the path of this installation in the shell script as mentioned in *Running Flink and Kafka*.
- Apache Kafka: Download the Kafka 2.11 binary from thie Kafka binary link. If this does not work, download the latest version from the Kafka download page. Once downloaded, simply extract the zip and the installation is complete. You need to use the path of this installation in the shell script as mentioned in *Running Flink and Kafka*.
- Apache Airflow: Follow the installation instructions on this Airflow installation link.

13.2 Running the system

To set the system up, the following steps need to be taken.

13.2.1 Collecting data

Navigate to 'Read Twitter Stream' and run `python streaming.py` to collect streaming data or run `python userstimeline.py` to collect data for a set of users.

More details can be found here: *Read data from Twitter API*.

13.2.2 Ingesting data

Before proceeding, start the Neo4j and MongoDB servers locally (using service commands in Ubuntu. Eg. `service neo4j start`, `service mongod start`).

Once the data is collected, you can ingest it into the databases as follows:

- Neo4j: Navigate to Ingestion/Neo4j/ and run `python ingest_neo4j_streaming.py` to ingest data collected using streaming API or run `python ingest_neo4j_user_timeline.py` to ingest data collected using User Timeline API.
- MongDB: In MongoDB we provide functionality to store only streaming data. Navigate to Ingestion/MongoDB and run `python ingest_raw.py`.

More details can be found here: *Ingesting data into MongoDB* and *Ingesting data into Neo4j*

13.2.3 Running the dashboard

To run the website on a local server on your machine, navigate to Dashboard_Website/ and run `python manage.py runserver`.

13.2.4 Running Flink and Kafka

To test the Flink applications generated on the dashboard, you need to do the following:

- Start the Kafka and Flink servers locally using the shell script in 'Kafka/' folder. (Configure the path of the Kafka and Flink directories in the shell script).
- You can now create alert specifications on the dashboard.
- Now, to put some tweets on the Kafka topic 'tweets_topic'. Navigate to 'Kafka/' and run `python kafka_tweets_producer.py` after configuring the data directory in its main function.

CHAPTER

FOURTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

H

I

J

K

L

M

N

P

Q

R

S

T

U

W