
Twitter Analytics Documentation

Release 0.1

Deepak Saini, Abhishek Gupta

Jun 15, 2018

CONTENTS:

1	Introduction to twitter analytics system	3
1.1	Major parts of the system	4
2	Read data from twitter streaming API	5
3	Ingesting data into Neo4j	7
3.1	Data stored in neo4j	7
3.2	User network	7
3.3	Tweet network	8
3.4	Indexing network	9
3.5	Ingesting the data into database	10
3.6	Ingestion Rates	10
4	Ingesting data into MongoDB	11
4.1	Why store in MongoDB	11
4.2	Data Format in mongoDB	11
4.3	mongoDB v/s neo4j	12
4.4	Ingesting the data into mongoDB	12
4.5	MongoDB Ingestion Rates	13
5	Neo4j: API to generate cypher queries	15
5.1	Template of a general query	15
5.2	Creating a custom query through dashboard API : Behind the scenes	16
6	Generating queries in mongoDB	17
7	About postprocessing functions	19
7.1	Need of post processing function	19
7.2	Format of post processing functions	19
7.3	Executing post processing function	20
8	Composing multiple queries : DAG	21
8.1	Basic terminology	21
8.2	Idea behind a DAG	21
8.3	Building a DAG from queries	22
8.4	DAG in airflow	24
8.5	Creating custom metric	25
9	Generating alerts using flink and kafka	27
10	Benchmarking the query answering	29

11 Dashboard Website	31
11.1 Major parts of the dashboard website	31
11.1.1 Hashtags	31
11.1.2 Mentions	31
11.1.3 URLs	31
11.1.4 Alerts	32
11.1.5 DAG	32
11.2 Use Cases	32
11.2.1 Viewing top 10 popular hashtags	32
11.2.2 Viewing usage history of hashtag	33
11.2.3 Viewing sentiment history of hashtag	34
11.2.4 Creating a mongoDB query	35
11.2.5 Creating neo4j queries	35
11.2.6 Create Post processing function	38
11.2.7 View Queries	38
11.2.8 Create DAG	39
11.2.9 View DAGs	39
11.2.10 Create Custom metric	42
11.2.11 Create Alert	43
11.2.12 View Alerts	44
12 Getting the system running	45
12.1 Setting up the environment	45
12.2 Running the dashboards	45
13 Indices and tables	47

Twitter generates millions of tweets each day. A vast amount of information is hence available for different kinds of analyses. The end users of these analyses however, may or may not be technically proficient. This necessitates the need of a system that can absorb such large amounts of data and at the same time, provide an intuitive abstraction over this data. This abstraction should allow the end users to specify different kinds of analyses without going into the technicalities of the implementation.

In this demonstration, we introduce a system that tries to meet precisely the above needs. 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. Also, acting as a data-store for the tweet network, the system provides another abstraction which allows the user to formulate complex queries on this historical data. We demonstrate both of these abstractions using an example of each, on real world data.

Here we provide a comprehensive documentation of each component of the system along with a documentation of the code.

**CHAPTER
ONE**

INTRODUCTION TO TWITTER ANALYTICS SYSTEM

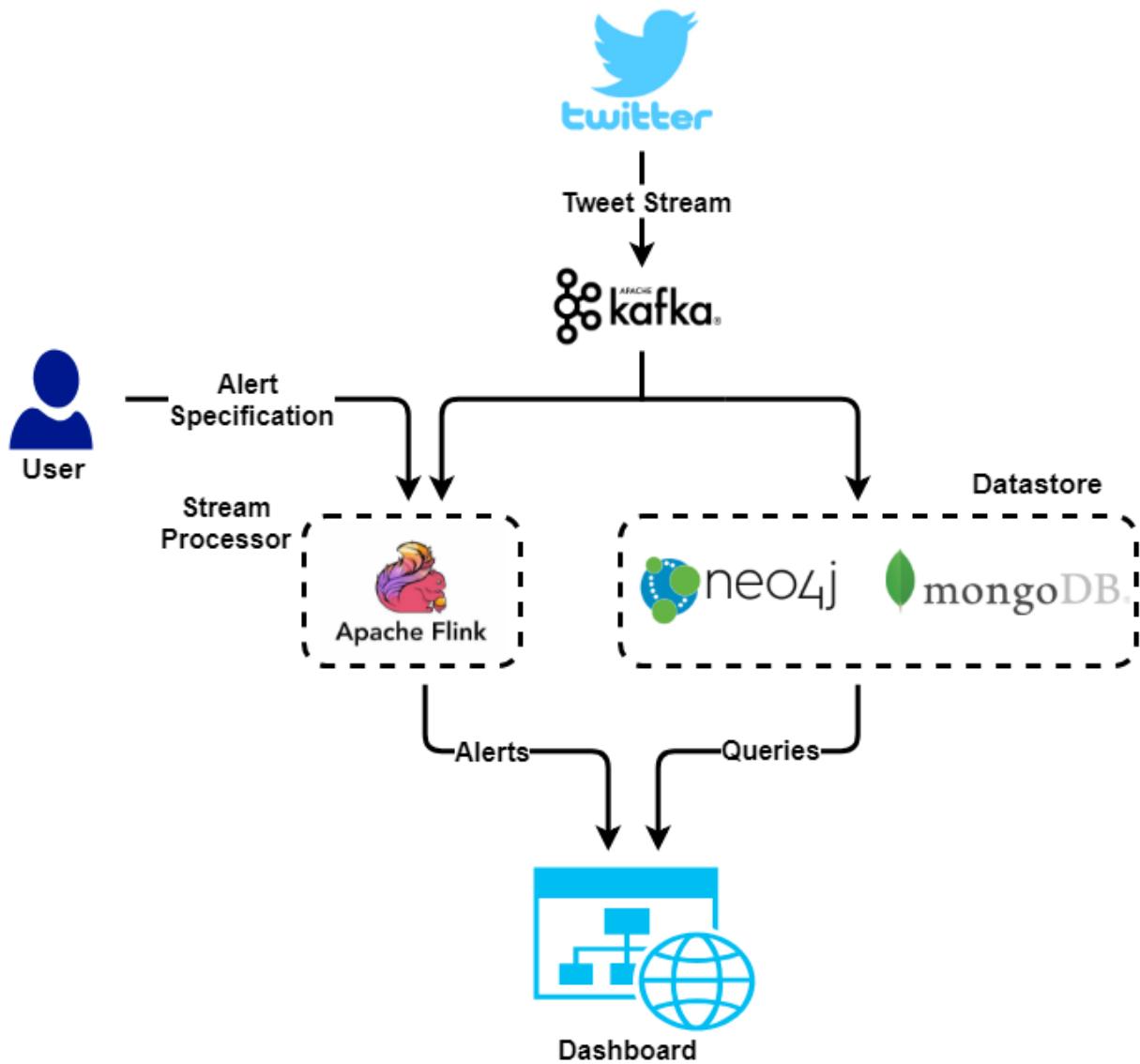
Nowadays social media generates huge amount of data which can be used to infer a number of trending social events and to understand the connections between different entities. One such useful platform is Twitter where users can follow other users and tweet on any topic, tagging it with hashtags, mentioning other users or using URLs. This creates a complex network of entities (like users, tweets, hashtags and URLs) which are interconnected in complex ways in a temporally evolving fashion. A number of programming tools are used by developers to answer queries or do analysis on top of this data.

However, there are a lot of people who are not so technically proficient to be able to write and maintain such a system. They generally have to rely on existing systems which either expect them to write queries in SQL (or other query languages) or offer only a partial view into the data through a limited number of available queries. Due to this reason, these people have to rely on developers to make meaning out of this data.

Our system has been designed for people with limited programming expertise. It continuously streams tweets from the Twitter Streaming API and gives users a couple of abstractions to work with. The first is an abstraction 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 is an abstraction over the historical view over the data stored from the stream. The second abstraction looks at the data as a network of users and tweets along with their attributes.

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 STREAMING API

INGESTING DATA INTO NEO4J

3.1 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.

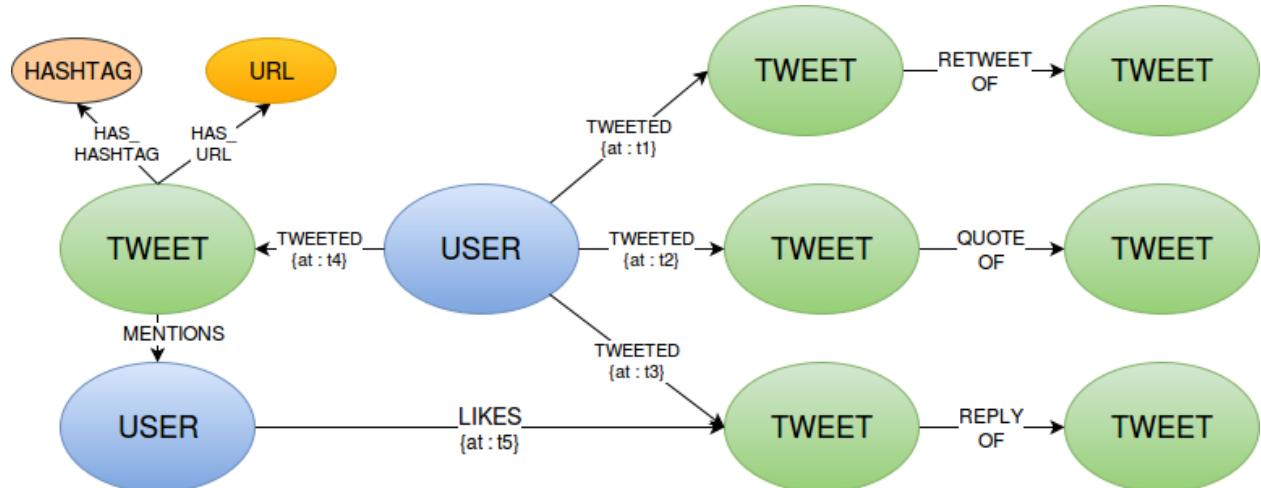
We store the complete twitter network in graph database neo4j.

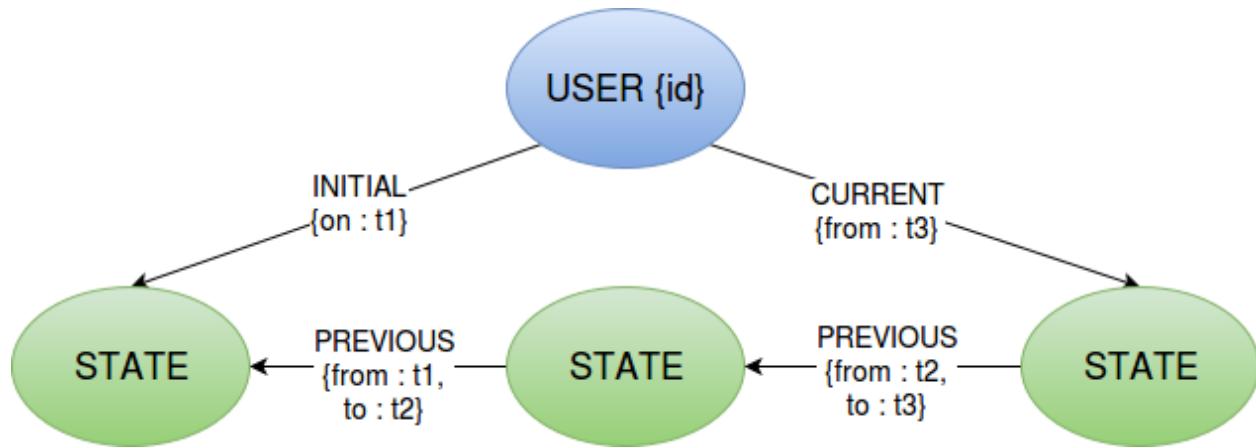
For sake of understanding, lets divide the complete network into three parts:

- 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 utilised 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.

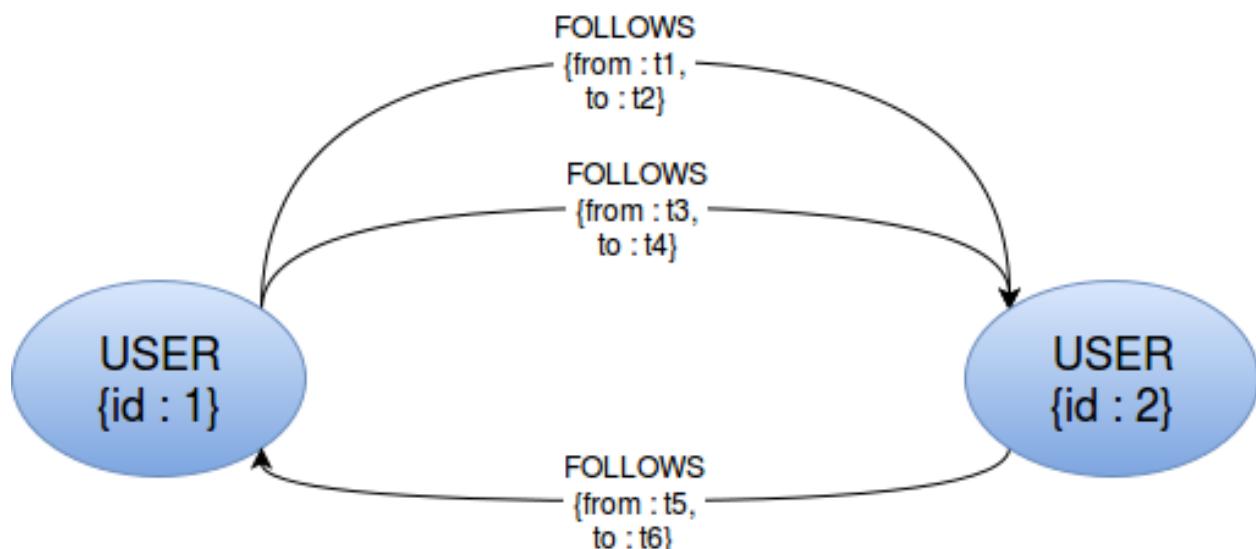
Let's look at these networks in some detail

3.2 User network

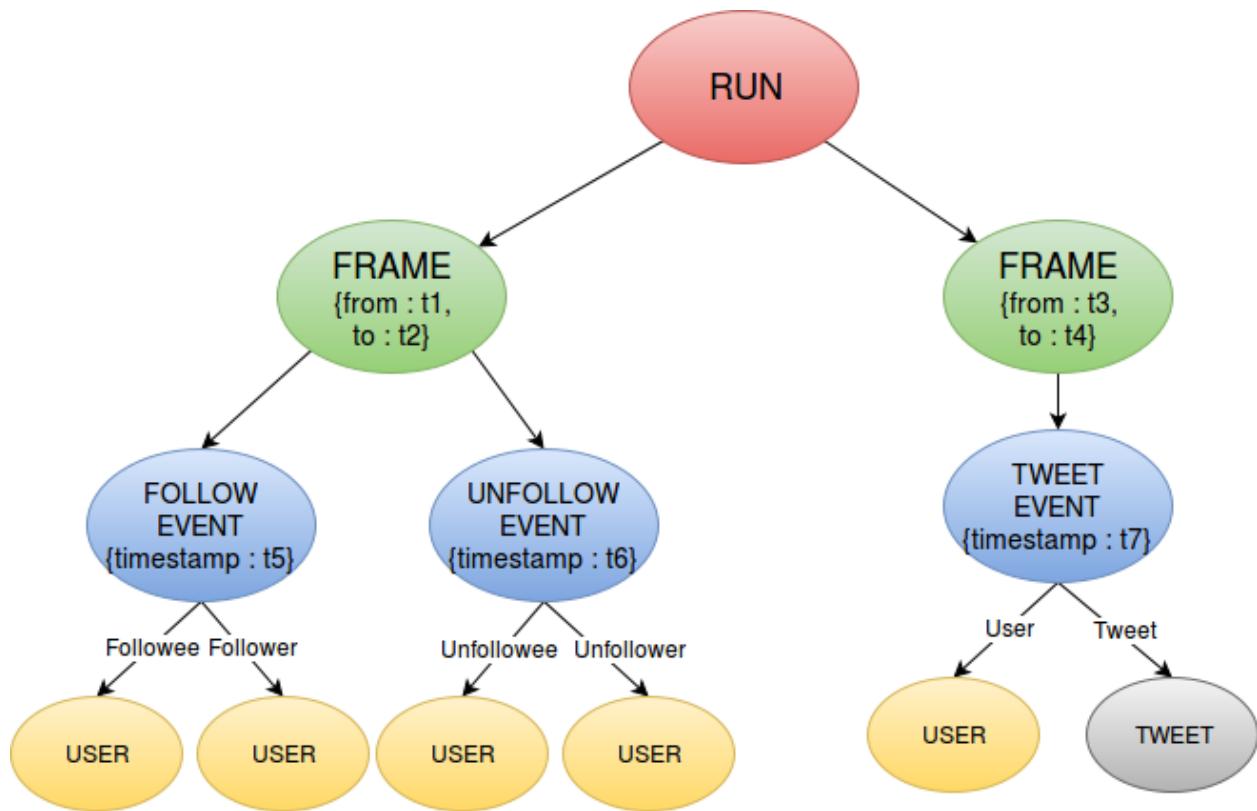




3.3 Tweet network

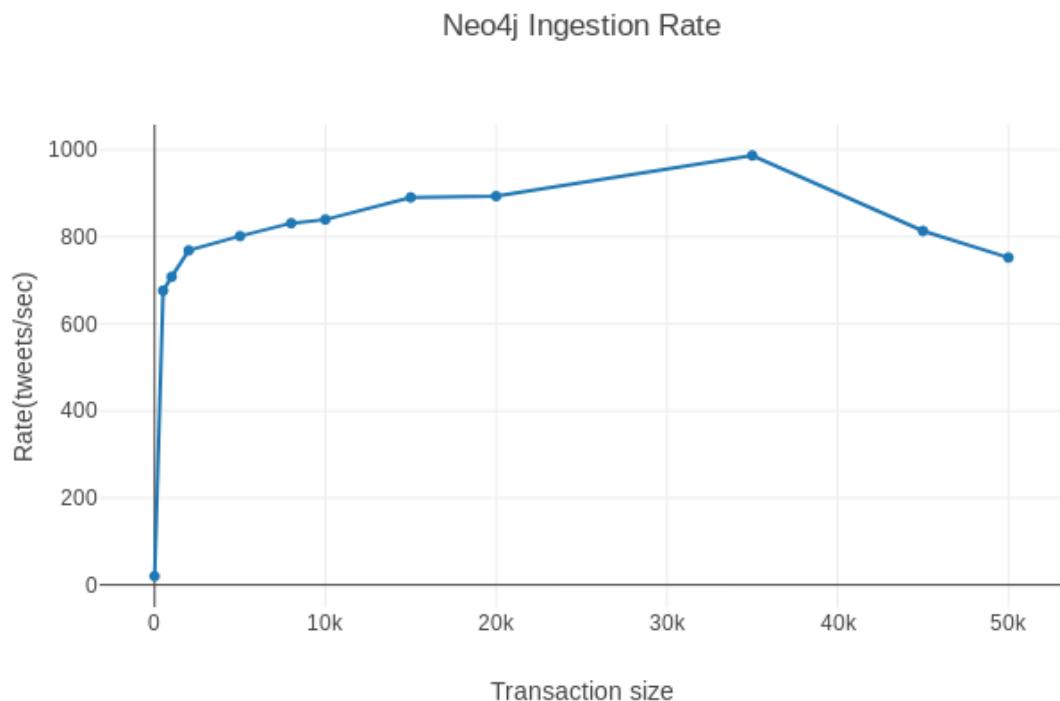


3.4 Indexing network



3.5 Ingesting the data into database

3.6 Ingestion Rates



INGESTING DATA INTO MONGODB

4.1 Why store in MongoDB

In mongoDB we store only the data which can be extracted quickly from incoming tweets without much processing.

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.

4.2 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)

4.3 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 with 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.

4.4 Ingesting the data into mongoDB

A simple approach would be to ingest a tweet into the database as 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.[scheme 1]

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

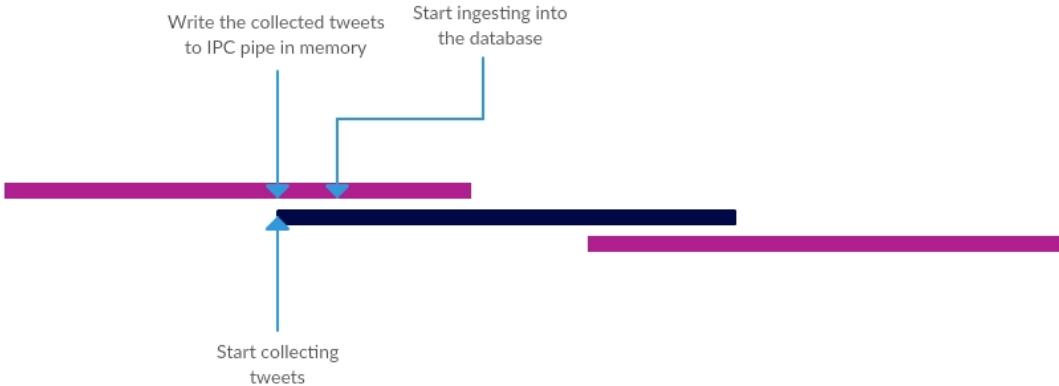
So finally we the approach of utilising multiple processes to write data to mongoDB.[scheme 3]

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 three 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:



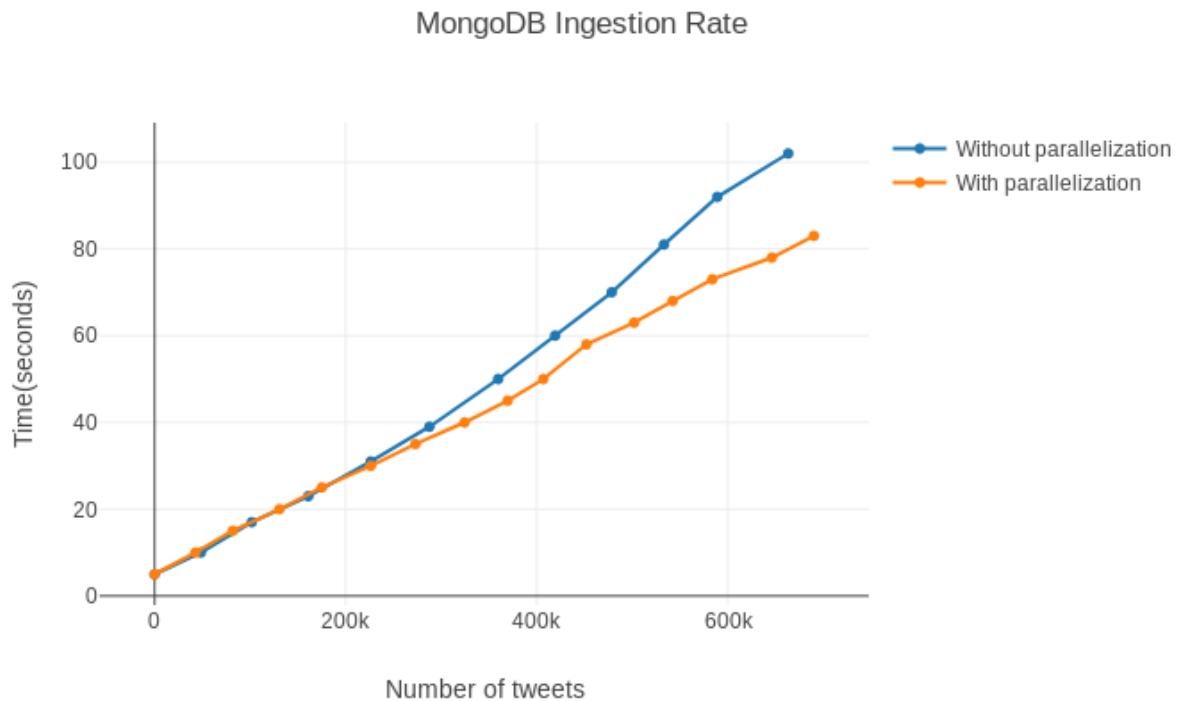
So, the timer process in the accumulator process generates an interrupt after every T seconds, at this instant, the accumulator stops collecting tweets and writes those to Inter process communication(IPC) pipe. This is generally fast as IPC pipe are implemented in memory. Now, the other end of the pipe 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 a 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.

4.5 MongoDB Ingestion Rates

As expected, the ingestion rate into mongoDB whilw 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 advntage 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 above).

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

NEO4J: API TO GENERATE CYpher QUERIES

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

5.1 Template of a general query

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. 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

The missing thing in this abstraction is that we have not taken into account that the source entity in both the constraints refers to the same User. 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

5.2 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) and the relations are stored in a database. These stored objects can be later used by the user.
- The query specified by the user is converted into a Cypher neo4j graph mining query.
- Connection is established with the neo4j server and the query is executed on the database.
- The results obtained are concatenated and are displayed.

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.

Contast this with neo4j where the schema is more or less 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 queies 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 timestamps at which <hashtag> is used between <Begin Time> and <End Time>
- Give the timestamps, associated positive and negative sentiment of a <hashtag> between <Begin Time> and <End Time>

Similarly, queris analogous to the above can also be answered for urls and user mantions.

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 constructs, 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 MongoDB 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.

ABOUT POSTPROCESSING FUNCTIONS

7.1 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 postprocessing function in a custom manner not supported by the query mechanism of the databases.

7.2 Format of post processing functions

We treat the postprocessing 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 postprocessing 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 dictonary(named “ret”) in same format. The function should further be named as “func”. This requires that the user specifies the input and output varibile names while creating the post processing function. This will be expalined 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 varibales. 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 varibales. Its easy in case named varibales are returned but the issue is when epressions are returned(for example the code contains `return 1[: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 alternatve.

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 dictioeny with the attribute names as keys and their values as
    ↪dict-values.
    :return: a dictionary with output variables as keys.
```

(continues on next page)

(continued from previous page)

```

"""
11 = input["list1"]
12 = input["list2"]
for x in 12:
    if x not in 11:
        11.append(x)

ret = {}
ret["l_out"] = 11
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

```

7.3 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(function_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))

```

COMPOSING MULTIPLE QUERIES : DAG

8.1 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 between the nodes. Each node as a query associated with it.

8.2 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.

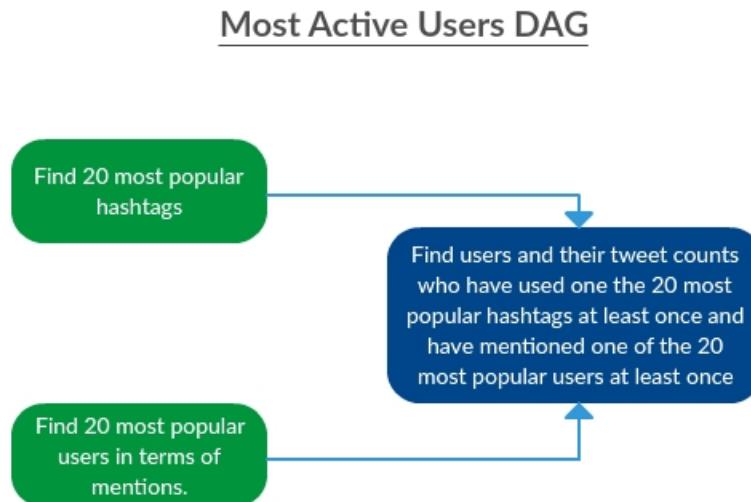
In this abstraction, a single query can also be treated as a DAG, one having a single node and no connections.

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.

8.3 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.

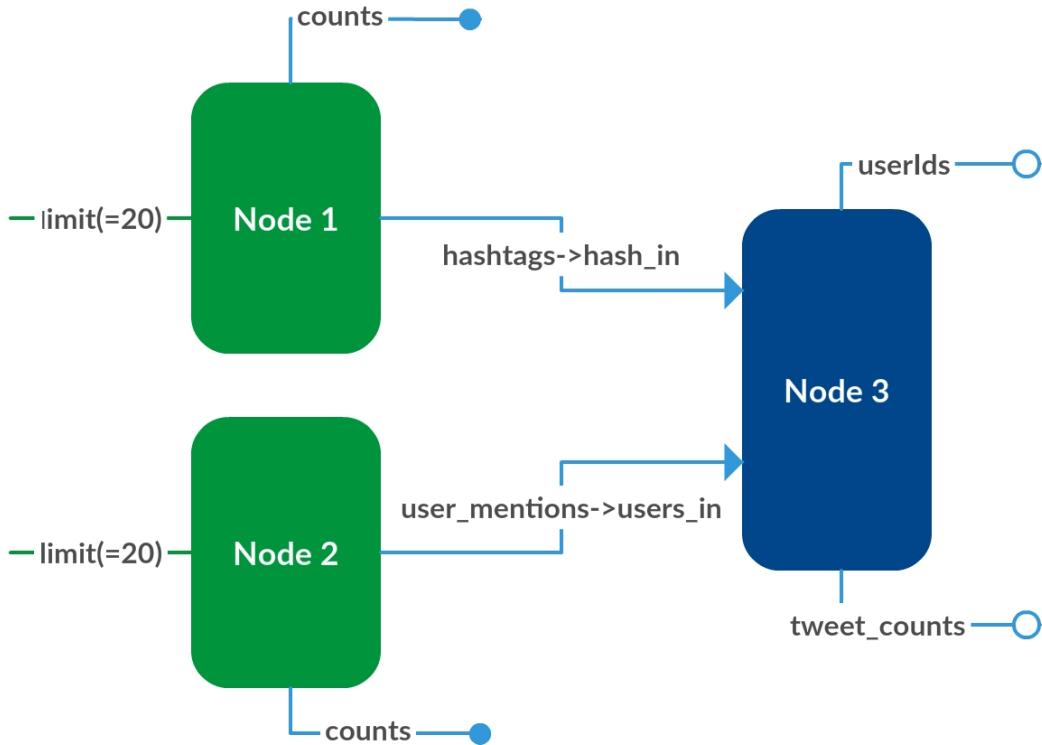
We explain how to build the queries with the help on an example. Let us build a DAG to get the most active users. Refer to this image(the green queries represent mongoDB queries and blue ones represent neo4j queries):



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_mentions(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 once**
 - 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 dictioanry 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 struture 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 feefind 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.

8.4 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 our 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_{0}'.format("n1"),  
    python_callable=execute_query,  
    op_kwargs={'node_name':'n1'},  
    provide_context = True,  
    dag=dag)  
  
task_1 = PythonOperator(  
    task_id='node_{0}'.format("n2"),  
    python_callable=execute_query,  
    op_kwargs={'node_name':'n2'},  
    provide_context = True,  
    dag=dag)  
  
task_2 = PythonOperator(  
    task_id='node_{0}'.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 through our dashboard to generate the DAG, is like creating a complete front-end wrapper over the airflow system, which is not our 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.

8.5 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 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.

CHAPTER
NINE

GENERATING ALERTS USING FLINK AND KAFKA

**CHAPTER
TEN**

BENCHMARKING THE QUERY ANSWERING

DASHBOARD WEBSITE

11.1 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 further 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.

11.1.1 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 hashtags, 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.

11.1.2 Mentions

The Mentions tab contains the major statistics concerning user mentions. It has the following 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.

11.1.3 URLs

The URLs tab contains the major statistics concerning urls. It has the following 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.

11.1.4 Alerts

11.1.5 DAG

We explain about DAGs in detail in section [*Composing multiple queries : DAG*](#). We assume the reader has read through 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 creation through 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 through 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 containing the structure 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.

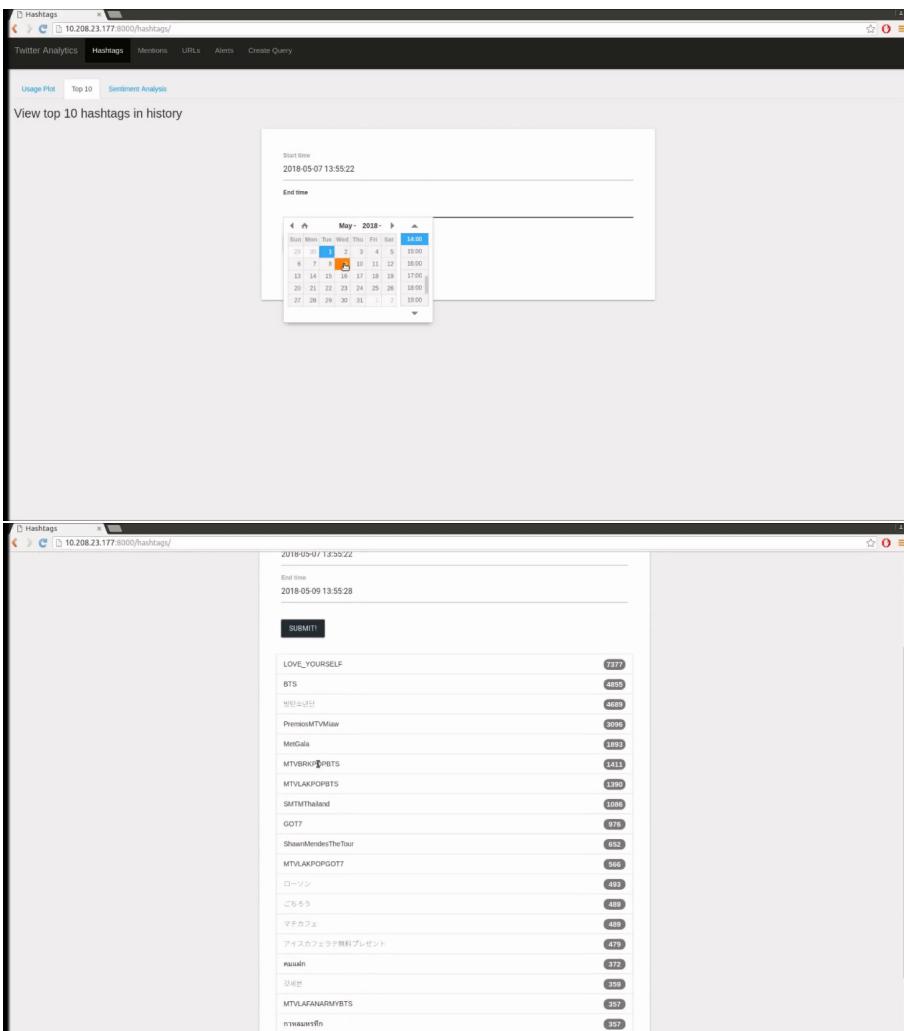
11.2 Use Cases

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 structured to be intuitive and simple. Nonetheless, the authors feel that these use cases should be enough to get the user started.

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

11.2.1 Viewing top 10 popular hashtags

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



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

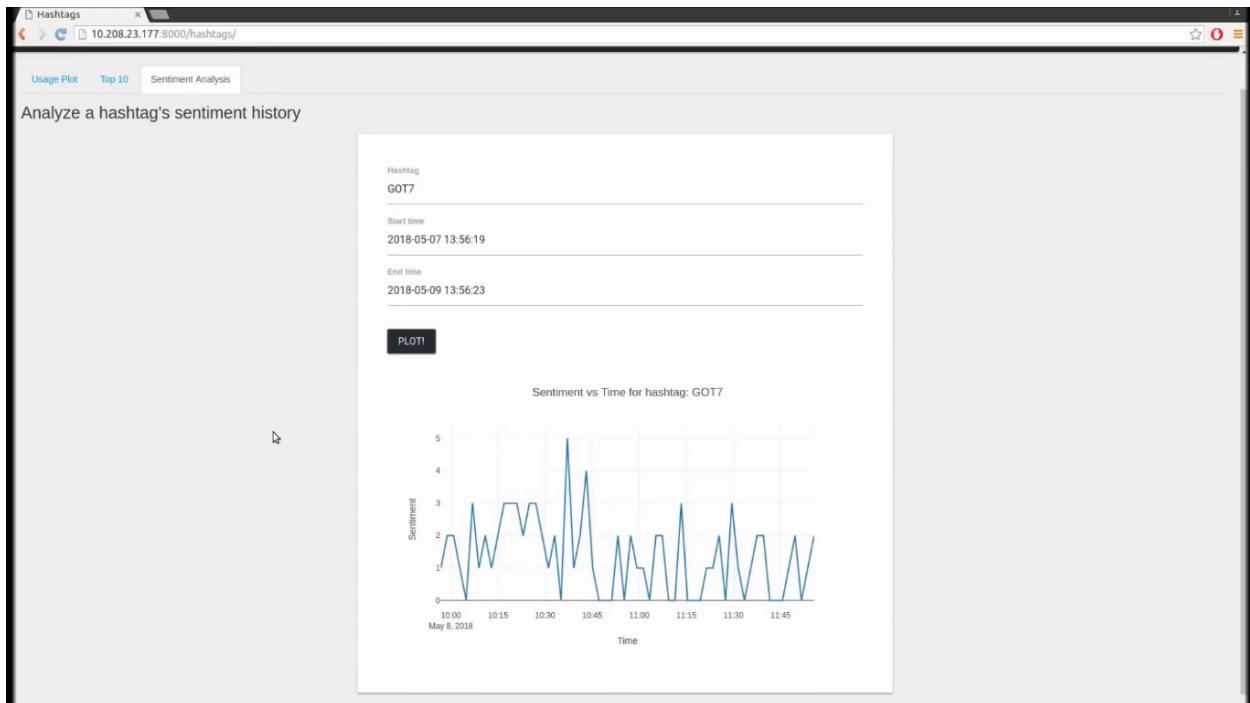
11.2.2 Viewing usage history of hashtag

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



11.2.3 Viewing sentiment history of hashtag

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



11.2.4 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.

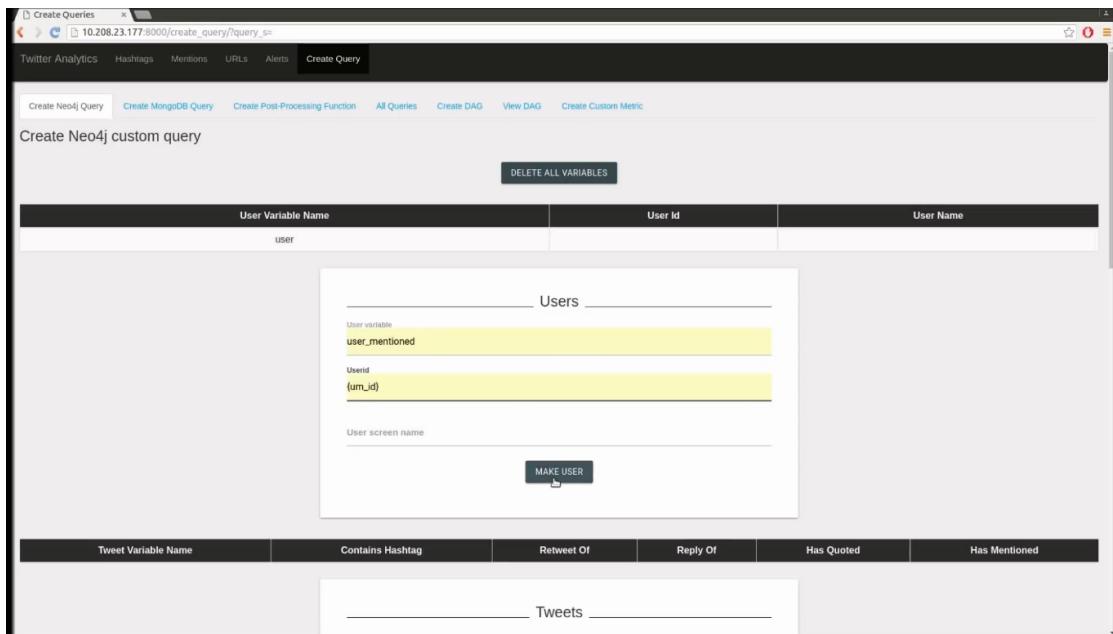
The screenshot shows the 'Create MongoDB Query' page. At the top, there are tabs for 'Create Neo4j Query', 'Create MongoDB Query' (which is selected), 'Create Post-Processing Function', 'All Queries', 'Create DAG', 'View DAG', and 'Create Custom Metric'. Below the tabs, the title 'Create MongoDB custom query' is displayed. The main area contains two sections: 'Most Popular hashtags' and 'Most popular hashtags in interval'. In the 'Most Popular hashtags' section, the 'Query name' field is set to 'most_popular_hashtags_20', the 'Number' field is set to '20', and a 'CREATE' button is visible. In the 'Most popular hashtags in interval' section, fields for 'Query name', 'Number', 'Begin time', and 'End time' are present but empty.

Similarly other mongoDB queries can be created.

11.2.5 Creating neo4j queries

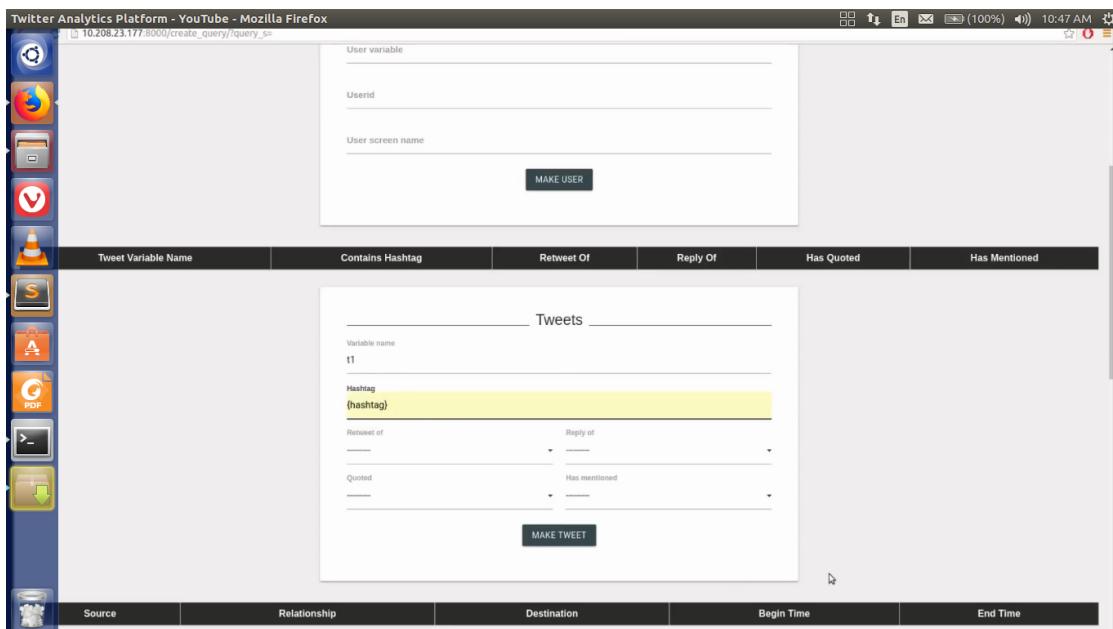
To create a neo4j query we need to create user and tweet entities and relationships between them. Here we show how to create the neo4j to get userIds and their tweet counts who have used one of the hashtags from a list of hashtags atleast once and have tweeted with one of the popular user mentions from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



The screenshot shows the 'Create Queries' interface. At the top, there are tabs for 'Create Neo4j Query', 'Create MongoDB Query', 'Create Post-Processing Function', 'All Queries', 'Create DAG', 'View DAG', and 'Create Custom Metric'. The 'Create Neo4j Query' tab is selected. Below it, the title 'Create Neo4j custom query' is displayed. A 'DELETE ALL VARIABLES' button is at the top right of the main form area. The form has two main sections: 'User Variable Name' and 'Tweet Variable Name'. In the 'User Variable Name' section, there is a table with columns 'User Variable Name', 'User Id', and 'User Name'. The 'User Variable Name' column contains 'user'. The 'User Id' column contains 'User' and 'User variable' fields, with 'user_mentioned' highlighted. The 'User Name' column contains '(um_id)' and 'User screen name' fields. A 'MAKE USER' button is located below these fields. In the 'Tweet Variable Name' section, there is a table with columns 'Contains Hashtag', 'Retweet Of', 'Reply Of', 'Has Quoted', and 'Has Mentioned'. The 'Contains Hashtag' column contains 'Tweets'. Below this table, there is another table for 'Tweets' with columns 'Variable name', 'Hashtag', 'Retweet of', 'Reply of', 'Quoted', and 'Has mentioned'. The 'Hashtag' column contains '(hashtag)' and is highlighted.

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.



This screenshot shows the same interface as the previous one, but with different input values. In the 'User Variable Name' section, the 'User Variable Name' field contains 'User' and the 'User Id' field contains 'User variable' with 'user_mentioned' highlighted. In the 'Tweet Variable Name' section, the 'Contains Hashtag' column contains 'Tweets'. Below this table, there is another table for 'Tweets' with columns 'Variable name', 'Hashtag', 'Retweet of', 'Reply of', 'Quoted', and 'Has mentioned'. The 'Hashtag' column contains '(hashtag)' and is highlighted. A 'MAKE TWEET' button is located at the bottom of the 'Tweets' table.

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

Relations _____

Source: user

Type: None

Destination: user

Begin Time _____ End Time _____

Choose the Tweet Relationships

Type: TWEETED

Destination: t1

Begin Time _____ End Time _____

INTRODUCE RELATION

Create Query _____

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.

Tweet Variable Name	Contains Hashtag	Retweet Of	Reply Of	Has Quoted	Has Mentioned
t1	{hashtag}				
t2					
t3					user_mentioned

Tweets _____

Variable name _____

Hashtag _____

Retweet of _____ Reply of _____

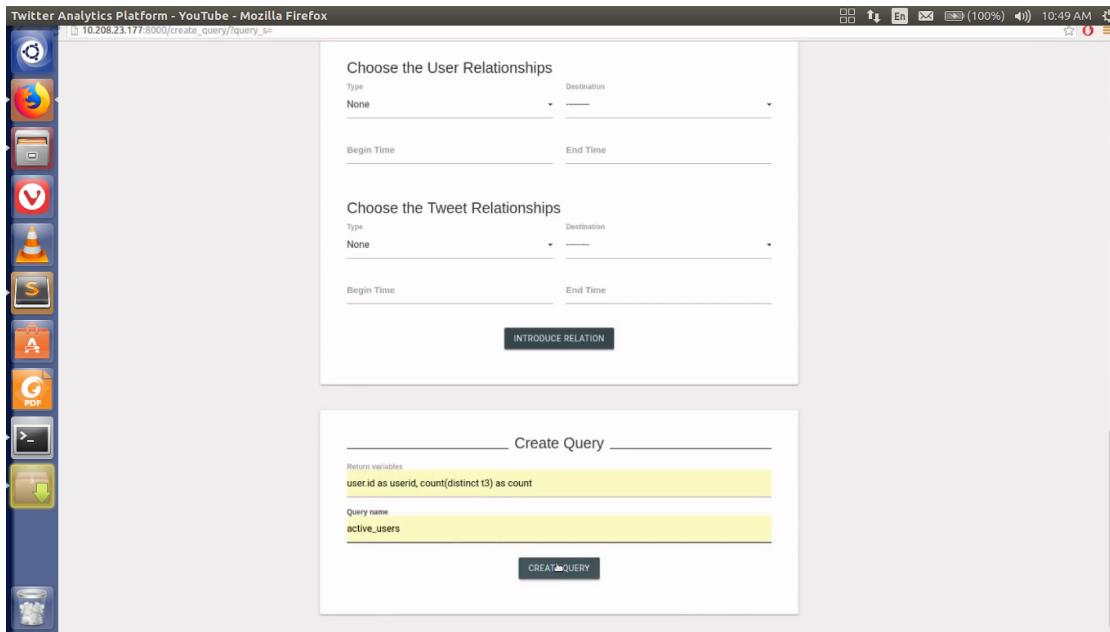
Quoted _____ Has mentioned _____

MAKE TWEET

Source	Relationship	Destination	Begin Time	End Time
user	TWEETED	t1		
user	TWEETED	t2		
user	TWEETED	t3		

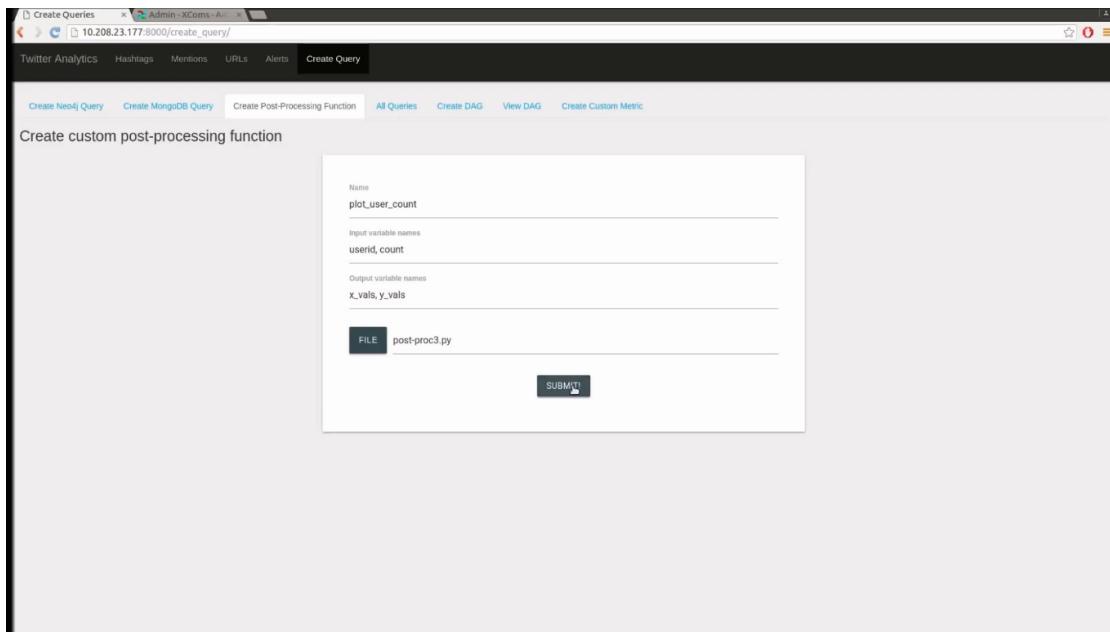
Relations _____

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



11.2.6 Create Post processing function

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



11.2.7 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:

The screenshot shows a web-based interface for managing Twitter Analytics queries. At the top, there are tabs for 'Create Queries', 'Admin - XGears - A...', 'Twitter Analytics', 'Hashtags', 'Mentions', 'URLs', 'Alerts', and 'Create Query'. Below these are sub-tabs: 'Create Neo4j Query', 'Create MongoDB Query', 'Create Post-Processing Function', 'All Queries', 'Create DAG', 'View DAG', and 'Create Custom Metric'. The main area displays a table with four rows:

Query Name	Code
most_popular_hashtags_20	Give the most popular hashtags in total
most_popular_mentions_20	Give the most popular users in total
active_users	<pre>UNWIND (um_id) AS um_id value UNWIND (hashtag) AS hashtag value MATCH (user :USER) -[:TWEETED]-> (t1 :TWEET), (user) -[:TWEETED]-> (t2 :TWEET), (user) -[:TWEETED]-> (t3 :TWEET), (t1) -[:HAS_HASHTAG]-> (:HASHTAG {text:hashtag}) RETURN user.id as userid, count(distinct t3) as count</pre>
plot_user_count	<pre>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</pre>

11.2.8 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 “active_users_dag”, optionally the description and the file containing the structure specification of the DAG.

The screenshot shows the 'Create DAG' subtab in the Twitter Analytics interface. The form fields are as follows:

- Name: active_users_dag
- Description: Finding users and their tweet count who have tweeted with at least one of 20 most popular hashtag and at least one of 20 most mentioned users
- File: active_users_dag.txt
- Submit button

11.2.9 View DAGs

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

Select a DAG and the action you want to apply on the DAG

DAG Name	DAG Description	Actions
top10	Finding top10 users of GOT7 hashtag	<button>DELETE</button> <button>VIEW</button>
active_users_dag	Finding users and their tweet count who have tweeted with at least one of 20 most popular hashtag and at least one of 20 most mentioned users	<button>DELETE</button> <button>VIEW</button>

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.

View the DAG and its source

Your Query DAG

```

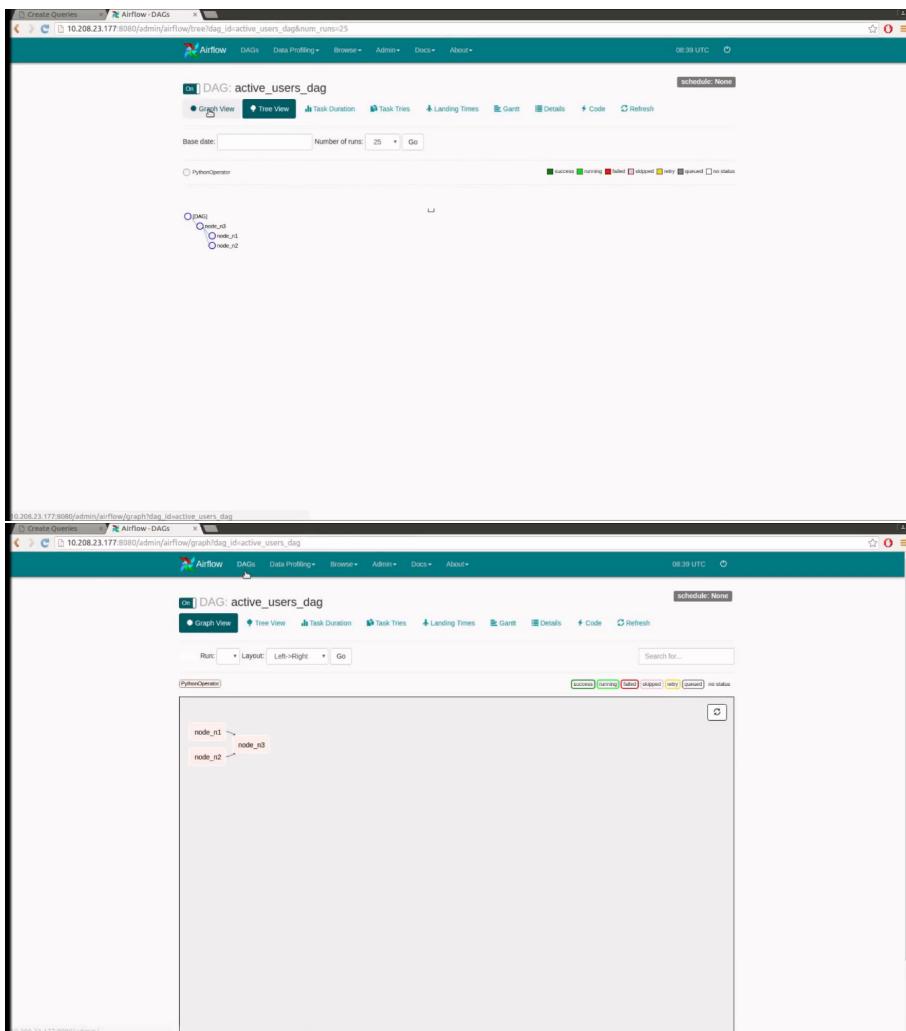
graph TD
    n1[most_popular_hashtags_20] --> n2[most_popular_mentions_20]
    n2 --> n3[active_users]
    n3 --> n4[most_popular_hashtags_20]
    n4 --> n5[most_popular_mentions_20]
    n5 --> n6[active_users]
    
```

Export to plotly

```

3
n1 most_popular_hashtags_20
n2 most_popular_mentions_20
n3 active_users
IN
OPTIONS:
  CONNEXIONS:
    n1.hashtag n3.hashtag
    n2.userId n3.um_id
  RETURNS:
    n3.userId
    n3.count
  
```

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 screenshot of the XComs list is provided here.

11.2.10 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:

Create Custom Metric here to plot the graphs

Choose the DAG you want to execute

Dag
active_users_dag

Choose the Post Processing function to get plot values

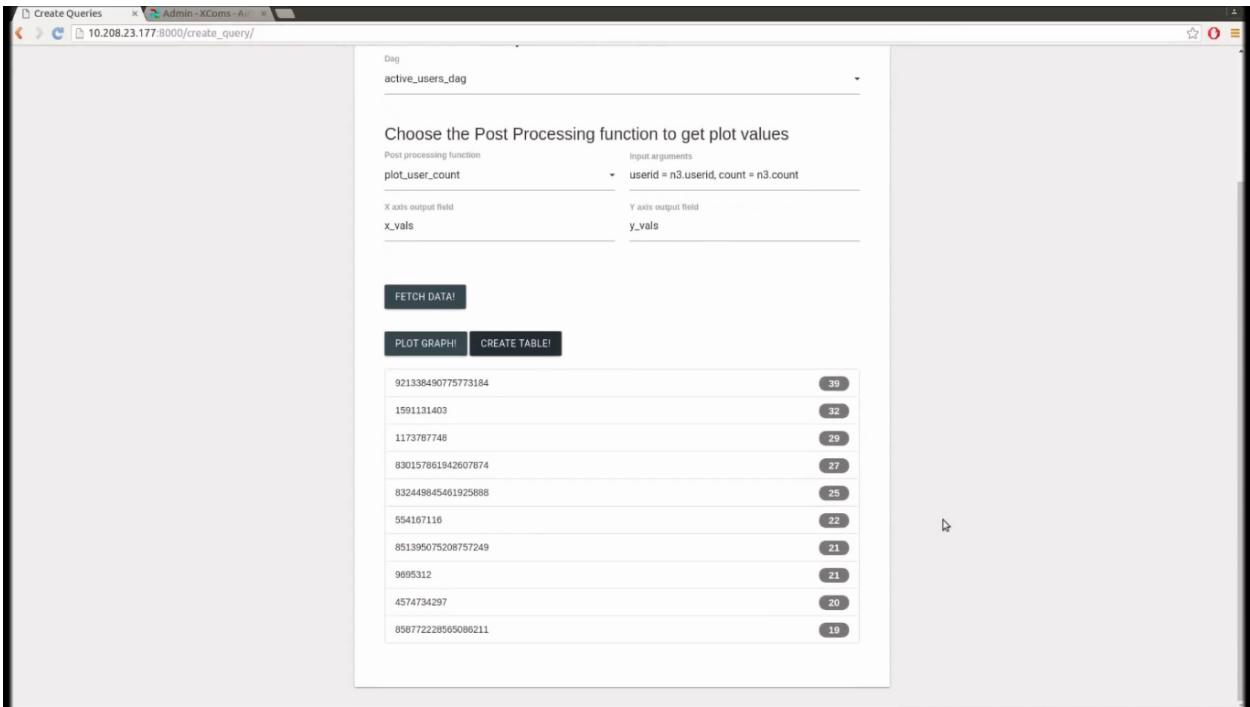
Post processing function plot_user_count	Input arguments userid = n3.userid, count = n3.count
X axis output field x_vals	Y axis output field y_vals

FETC DATA!

PLOT GRAPH! **CREATE TABLE!**

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:



11.2.11 Create Alert

To create an alert on the tweet stream, we need to specify the alert name, the filter, choose keys on which to generate the filter, 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.

The screenshot shows the 'Alerts' section of the Twitter Analytics interface. At the top, there are tabs for 'Create Alert' (which is selected), 'Manage Alerts', and 'Live Alerts'. Below the tabs, it says 'Create a new alert'. The form fields are as follows:

- Alert name:** viral_hashtags
- Filter:** (empty field)
- Keys:** Hashtag (checkbox checked)
- Window length:** 60
- Window length in seconds:** (empty field)
- Window slide:** 30
- Window slide in seconds:** (empty field)
- Count threshold:** 3
- Alert threshold of tweets in above window:** (empty field)

At the bottom of the form is a 'CREATE ALERT!' button.

11.2.12 View Alerts

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

Alert Name	Window Description				Delete Alert
	Keys	Start Time	End Time	Current Tweet Count	
viral_hashtags	{"hashtag":"PremiosMTVMiaw"}	2018-05-08 20:55:30	2018-05-08 20:56:30	12	<button>DISMISS</button>
viral_hashtags	{"hashtag":"PremiosMTVMiaw"}	2018-05-08 20:55:30	2018-05-08 20:56:30	3	<button>DISMISS</button>
viral_hashtags	{"hashtag":"PremiosMTVMiaw"}	2018-05-08 20:55:00	2018-05-08 20:56:00	9	<button>DISMISS</button>
viral_hashtags	{"hashtag":"BTS"}	2018-05-08 20:55:30	2018-05-08 20:56:30	21	<button>DISMISS</button>
viral_hashtags	{"hashtag":"BTS"}	2018-05-08 20:55:00	2018-05-08 20:56:00	12	<button>DISMISS</button>
viral_hashtags	{"hashtag":"#방탄소년단"}	2018-05-08 20:55:30	2018-05-08 20:56:30	18	<button>DISMISS</button>
viral_hashtags	{"hashtag":"#방탄소년단"}	2018-05-08 20:55:00	2018-05-08 20:56:00	9	<button>DISMISS</button>
viral_hashtags	{"hashtag":"#LOVE_YOURSELF"}	2018-05-08 20:55:30	2018-05-08 20:56:30	27	<button>DISMISS</button>
viral_hashtags	{"hashtag":"#LOVE_YOURSELF"}	2018-05-08 20:55:00	2018-05-08 20:56:00	15	<button>DISMISS</button>
viral_hashtags	{"hashtag":"GOT7"}	2018-05-08 20:55:30	2018-05-08 20:56:30	3	<button>DISMISS</button>
viral_hashtags	{"hashtag":"GOT7"}	2018-05-08 20:55:00	2018-05-08 20:56:00	3	<button>DISMISS</button>
viral_hashtags	{"hashtag":"#SMTMThailand"}	2018-05-08 20:55:30	2018-05-08 20:56:30	6	<button>DISMISS</button>
viral_hashtags	{"hashtag":"#SMTMThailand"}	2018-05-08 20:55:00	2018-05-08 20:56:00	6	<button>DISMISS</button>
viral_hashtags	{"hashtag":"#MTVLAFANARMYBTS"}	2018-05-08 20:55:30	2018-05-08 20:56:30	3	<button>DISMISS</button>
viral_hashtags	{"hashtag":"#MTVLAFANARMYBTS"}	2018-05-08 20:55:30	2018-05-08 20:56:30	3	<button>DISMISS</button>

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](#)

GETTING THE SYSTEM RUNNING

12.1 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
// 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.txt
.
.
// deactivate the virtual environment
source deactivate
```

Apart from these, the user also needs to install mongoDB, neo4j databases and The Apache flink-kafka framework.

12.2 Running the dashboards

To run the website on a local server on your machine, navigate to Dashboard Website/ and run `python manage.py runserver`. Presently the databases will be empty, to insert the new data into the databases see the *Ingesting data into MongoDB* and the *Ingesting data into Neo4j* sections.

CHAPTER
THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search