Twitter Analytics Documentation

Release 0.1

Deepak Saini, Abhishek Gupta

Jun 11, 2018

CONTENTS:

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 comprehenstive documentaion of each component of the system along with a documentation of the code.

CHAPTER

ONE

INTRODUCTION TO TWITTER ANALYTICS SYSTEM

CHAPTER

TWO

READ DATA FROM TWITTER STREAMING API

CHAPTER

THREE

INGESTING DATA INTO NEO4J

3.1 Code Documentation

CHAPTER

FOUR

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 unnecesarily 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 infomration:**
    - 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 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 benefitial to store much more condensed data in mongoDB and use it to answer more complex queries.

4.4 Ingesting the data into database

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 tweeets are being collected in memory.[scheme 2]

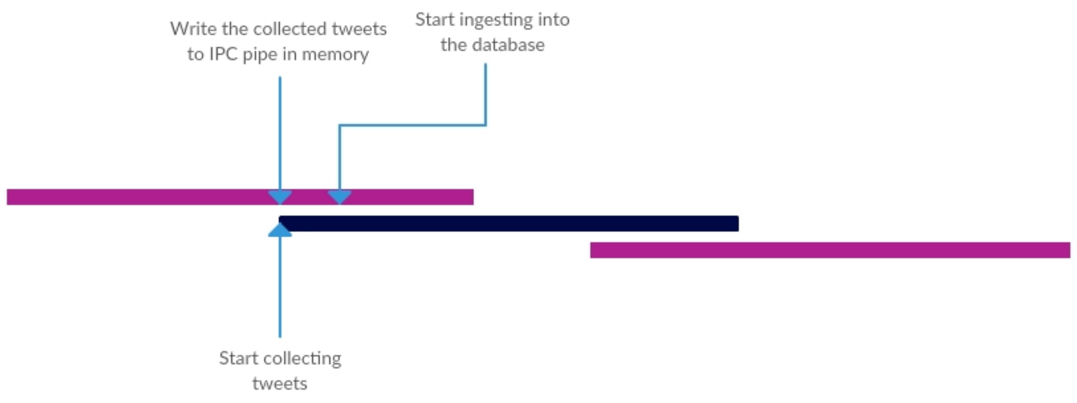
So finally we the approach of utilising multiple processes to write data to mongoDB.[scheme 3]

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 wriing to database and collecitng 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 begining 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:


Write the collected tweets to IPC pipe in memory
Start ingesting into the database
Start collecting tweets

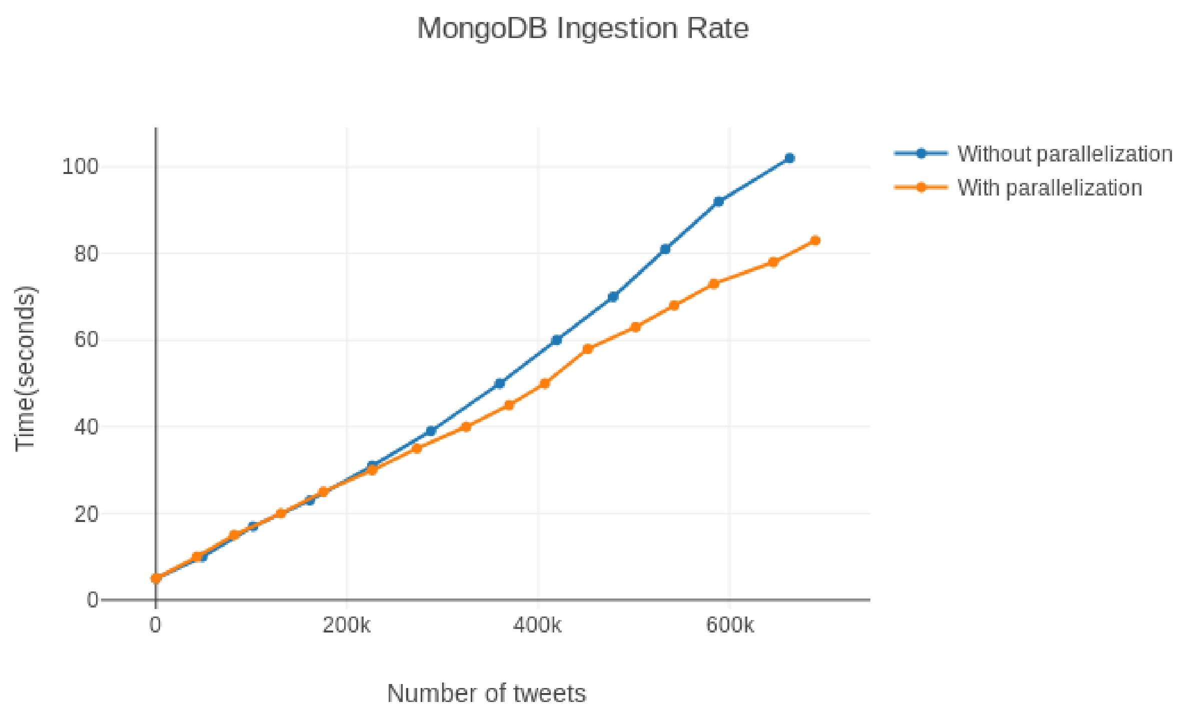
So, the timer process in the accumulator process genertes an interrupt after every T seconds, at this instant, the accumulator stops collecting tweets and writes those to Inter process communication(IPC) pipe. This is generally fast as IPC pipe are implemented in memory. Now, the other end of the pipe is in the connector process. After the writing process has been complete, it recieves 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 acuumulator 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 accummulator 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 hastags in total, or most popular hashtags in a large interval. It would be benefitial 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.

4.5 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.

4.6 Code Documentation

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

Bases: `object`

`aggregate`()

`exit`()

`insert_tweet`(*tweet*)

update the in memory dictionaries

`populate`()

write to the mongoDB

`worker`(*q*)

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()
Method to be run in sub-process; can be overridden in sub-class

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**(*positive_words*, *negative_words*, *tweet_text*)

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

ingest_raw.**read_tweets**(*path*, *filename*)

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

CHAPTER

FIVE

NEO4J: API TO GENERATE CYPHER QUERIES

Here we expalin 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.

5.3 Code Documentation

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

SIX

GENERATING QUERIES IN MONGODB

CHAPTER

SEVEN

ABOUT POSTPROCESSING FUNCTIONS

CHAPTER

EIGHT

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 outups of query(ies) as inputs and transforms them to give the output

DAG stands for directed acylic graph. Thus it a directed graph with no cycles. The idea behind a DAG is to compose mutiple queries to build a complex queries. A DAG has nodes and has directed connections connections between the nodes. The nodes represent queries.

8.2 Idea behind a DAG

As mentioned above, our main idea is to provie 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 queires from simple queries.
- A particular database may be suited to answer particular type of queries. In fact this is the root 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 different queries in sake of efficiency.
- 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. Thus, given the DAG abstraction, the user can feed te 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 benefitial from the end user point of view because by doing so we can show the incremental results of the smaller parts to the user instead of waiting for the entire big query to execute.

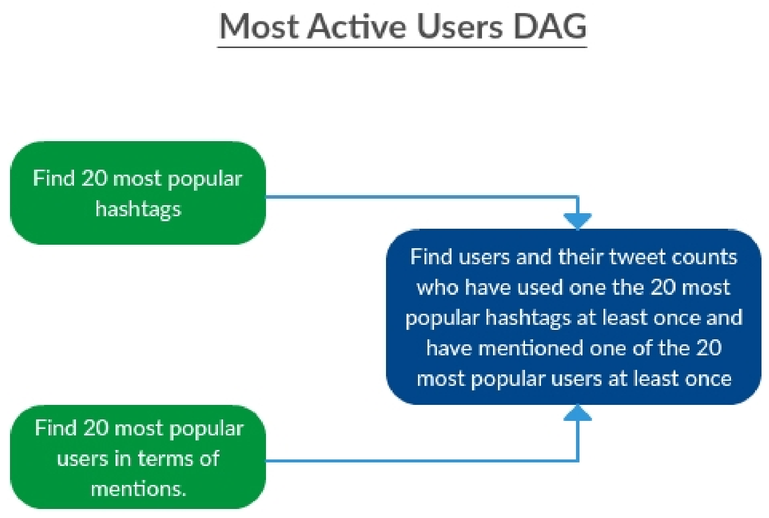
In this abstraction, a single query can also be treated as a DAG, one having a single node and no connections.

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 ouputs 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 downstrea feed into the inputs of the upstream ones.

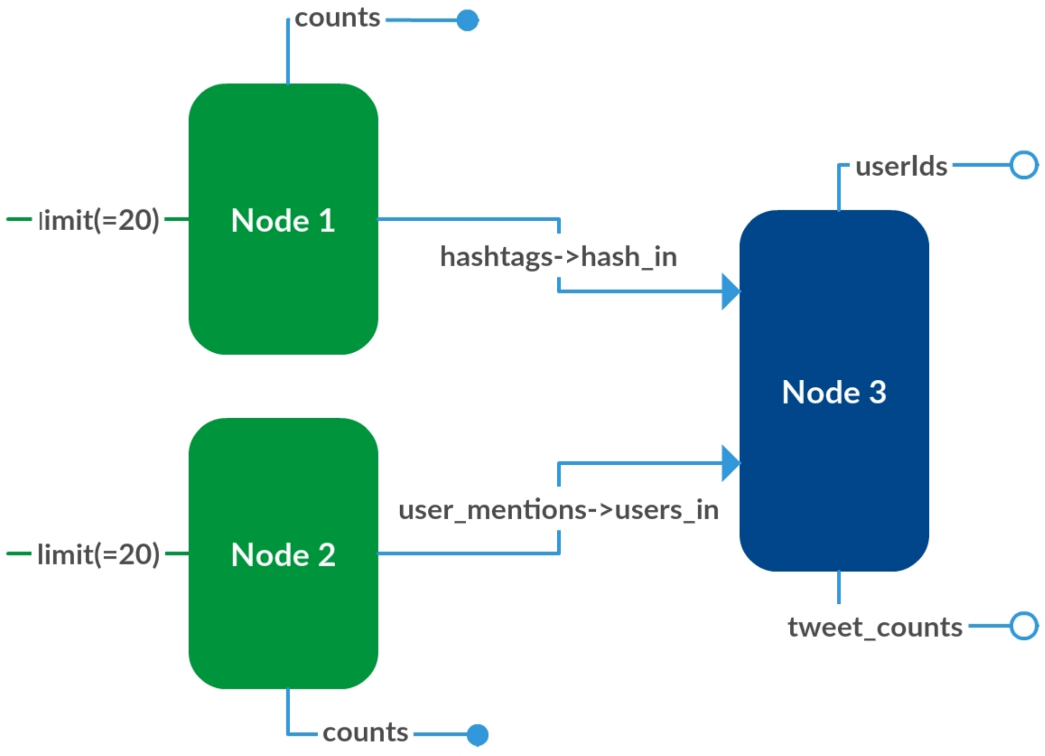
We explain how to build the queries with the help on an example. Let us build a DAG to get the most active users. Refer to this image(the green queries represent mongoDB queries and blue ones represent neo4j queries):


Most Active Users DAG
Find 20 most popular hashtags
Find users and their tweet counts who have used one the 20 most popular hashtags at least once and have mentioned one of the 20 most popular users at least once
Find 20 most popular users in terms of mentions.

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 keys are lists and similarly returns a dictionary with list values. The keys in both dictioanry is the name of hte 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
```

8.4 DAG in airflow

Similary 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)
```

8.5 Creating custom metric

Custom metric can be created on top of the DAG.

CHAPTER

NINE

GENERATING ALERTS USING FLINK AND KAFKA

CHAPTER

TEN

BENCHMARKING THE QUERY ANSWERING

CHAPTER

ELEVEN

DASHBOARD WEBSITE

CHAPTER TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX