Author's Note

This document was created as a reference for how Project 3's code is laid out. A high level overview is covered in slides 4-6. In slides 8-23, the algorithm to generate, plot and show Task 3 is explained step by step.

Revision 2 – Last Updated 05/15/2017

Although built with PowerPoint, it is not meant to be used for a presentation as is. (Too much text)

ATTENTION: BUG FIX

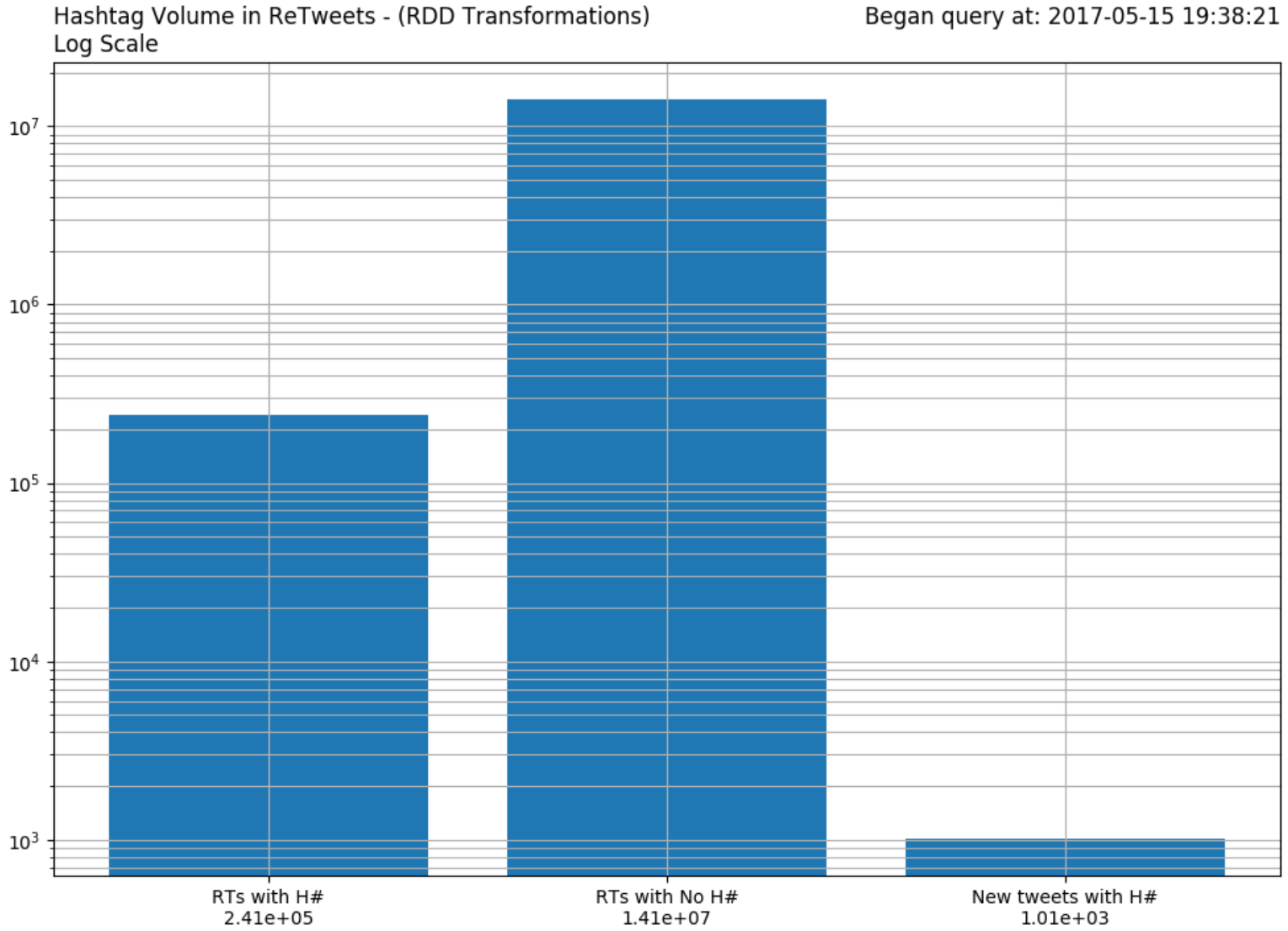
A major bug was identified in Task 2. It did not properly aggregate by the rt_rt_cnt, instead just doing a simple count.

Furthermore duplicates were being counted, further polluting the data. The code has been rebuilt and the new results are shown on the next page.

Changes were made to:

- t2_csv.py: Added a new field – the rt rt id, so as to remove duplicates
- t2_csv.txt: Added a new column for the rt rt id (called "id")
- query.py: Replaced qRDD_t2 with new code. Scrapped t2_map function in favor of splitting into one RDD for each bar on the chart.
- graph.py: Modified graph_t2. Now uses a log scale to show results. Also added raw numbers, below bar labels.

ATTENTION: BUG FIX



How important are tweets for hashtag trending? A hashtag can be used in either a new tweet or in a retweet.
How much hashtag volume comes from retweets compared to new tweets?And out of all the retweet volume, how much of it is for a hashtag?

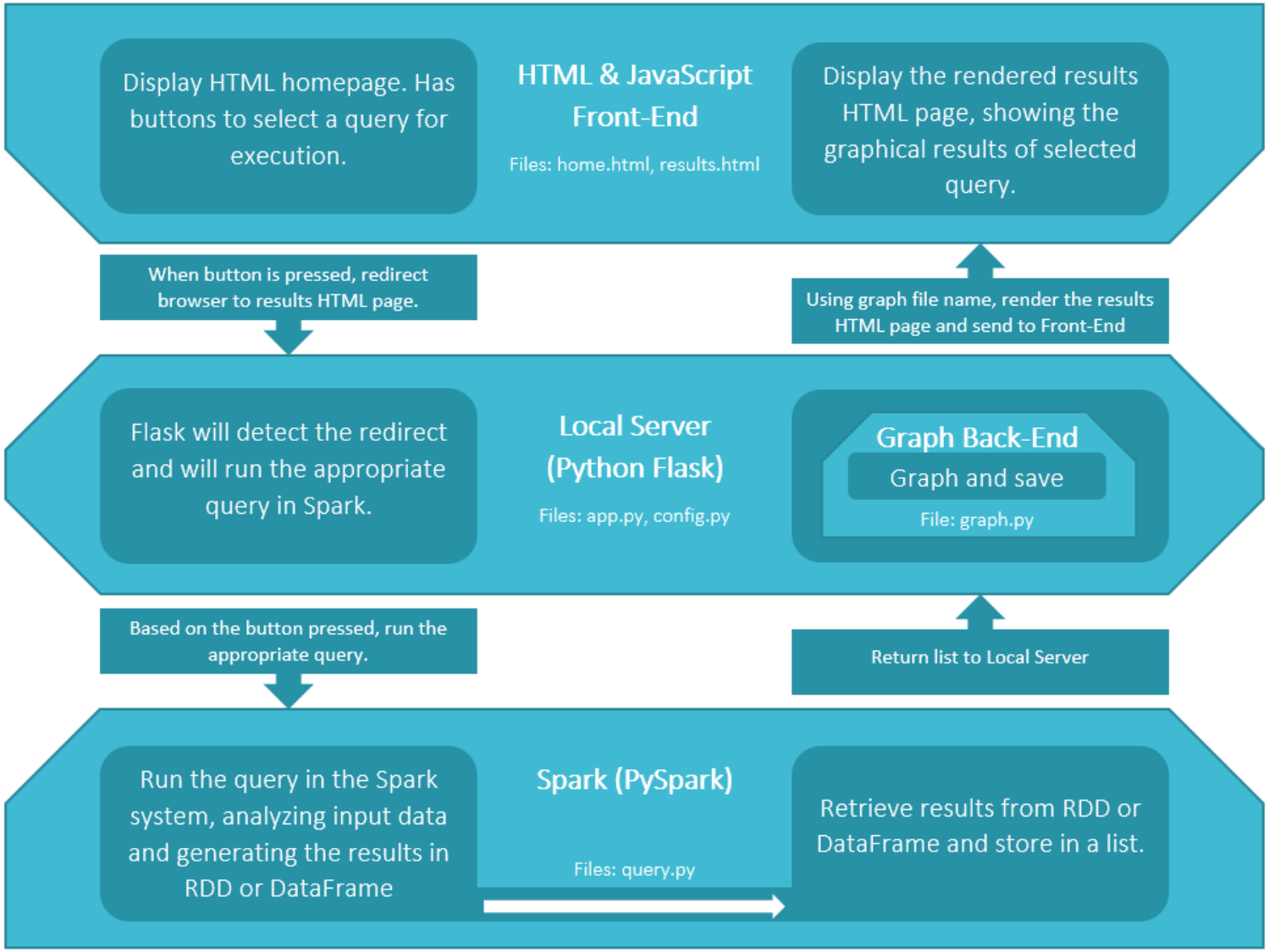
Source Code

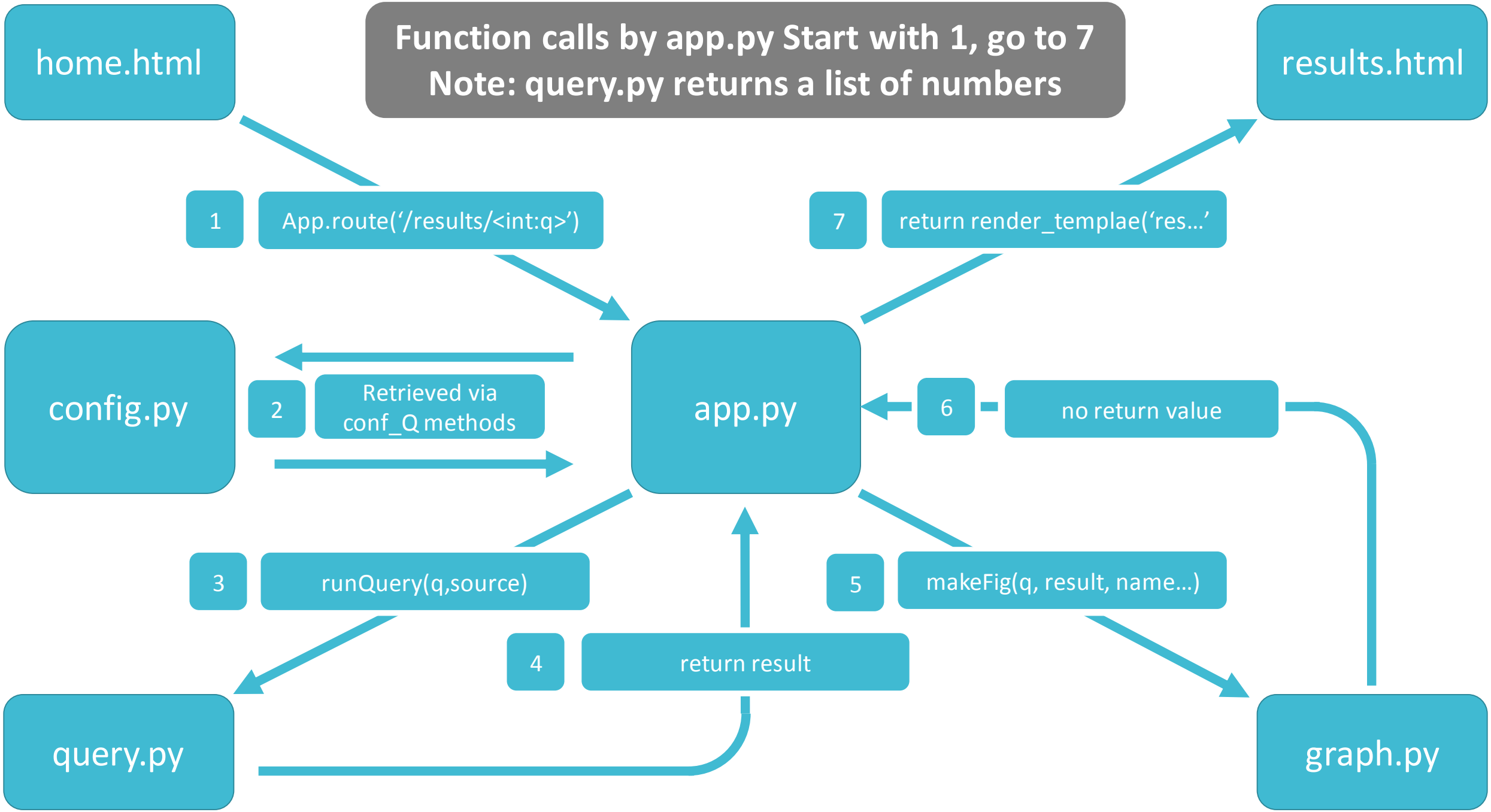
Files		
Front-End	Server	Back-End
home.html	app.py	query.py
results.html	config.py	graph.py

Dependencies (Library/Module)	
Python 2.7 Modules	Python numpy
Apache Spark (PySpark)	Python matplotlib
Python Flask + Depend.	

Layer-View

Front-End
Server
Back-End




home.html
Function calls by app.py Start with 1, go to 7
Note: query.py returns a list of numbers
results.html
1
App.route('/results/<int:q>')
7
return render_templae('res…'
config.py
app.py
2
Retrieved via conf_Q methods
6
no return value
3
runQuery(q,source)
5
makeFig(q, result, name…)
4
return result
query.py
graph.py

query.py

After the server has retrieved the name of the input data file, it calls the **runQuery** function, passing arguments **q** and **source**.

- **source** is the absolute path and name of the file holding the input data.
- **q** is a number representing the task (1,2 or 3)

Based on q, the arguments are passed into one of three functions:

- **qSQl_t1**, **qRDD_t2**, or **qSQL_t3**

Each function will run an analysis on Spark using the input data and will return a list object. The list object contains the results of the analysis.

Let's walk through Task 3, from generating results, to plotting the figure and showing it on the GUI.

NOTE: MS Access is used for the visualizing Task 3's SQL queries.

query.py
Task 3

Task 3 uses Spark SQL & DataFrames to conduct the analysis. The input data in "t3_csv.txt" is read into the DataFrame **df_t3**, and is labeled as "t3_csv". Starting from there, the analysis will the desired results.

t3_csv

time	trend	volume
2017-04-06 20:48:18	#LEmissionPolitique	92636
2017-04-06 20:48:18	Don Rickles	112258
2017-04-06 20:48:18	#Gala15GHVIP5	17868
2017-04-06 20:48:18	#Fast8EH	
2017-04-06 20:48:18	#VzlaTrancaContraElGolpe	152349
2017-04-06 20:48:18	Dustin Johnson	53490
2017-04-06 20:48:18	French Montana	93712
2017-04-06 20:48:18	Mari Palma	
2017-04-06 20:48:18	#twepsv	
2017-04-06 20:48:18	#NuclearOption	95342
2017-04-06 20:48:18	#PuebloYFANBLealtadAbsoluta	67205
2017-04-06 20:48:18	#gntm	
2017-04-06 20:48:18	#CelebrityMasterChefIt	
2017-04-06 20:48:18	#TPMPlesparis	
2017-04-06 20:48:18	#survirorgr	

t3_csv | time

time	total
2017-04-06 20:48:18	1210685
2017-04-06 20:53:10	1211454
2017-04-06 21:03:16	891431
2017-04-06 21:18:16	876058
2017-04-06 21:28:08	926815
2017-04-06 21:33:13	876452
2017-04-06 21:43:08	907272
2017-04-06 21:53:04	957933
2017-04-06 22:03:06	978410
2017-04-06 22:17:58	971845
2017-04-06 22:27:59	1030493
2017-04-06 22:37:56	1067197
2017-04-06 22:42:52	1083190
2017-04-06 22:52:50	1110049
2017-04-06 23:02:56	1112329

query.py
Task 3

Using input, task 3 will generate two DataFrames:

- **df_time**, which holds the aggregate sum of tweet volume by time across all trends
- **df_trend_time**, which holds the top 3 trends, when they appear, and how much tweet volume at that time

t3_csv | time

time	total
2017-04-06 20:48:18	1210685
2017-04-06 20:53:10	1211454
2017-04-06 21:03:16	891431
2017-04-06 21:18:16	876058
2017-04-06 21:28:08	926815
2017-04-06 21:33:13	876452
2017-04-06 21:43:08	907272
2017-04-06 21:53:04	957933
2017-04-06 22:03:06	978410
2017-04-06 22:17:58	971845
2017-04-06 22:27:59	1030493
2017-04-06 22:37:56	1067197
2017-04-06 22:42:52	1083190

t3_csv | time | trend_time

trend	Time	volume
Syria	2017-04-07 01:32:42	1048749
Syria	2017-04-07 01:42:53	1125025
Syria	2017-04-07 01:52:49	1211343
Syria	2017-04-07 02:07:45	1299043
Syria	2017-04-07 02:17:50	1383853
Syria	2017-04-07 02:22:46	1425476
Syria	2017-04-07 02:37:49	1552001
Syria	2017-04-07 02:47:46	1635870
Syria	2017-04-07 02:52:45	1675279
Syria	2017-04-07 03:02:46	1724803
Syria	2017-04-07 03:12:40	1806824
Syria	2017-04-07 03:22:33	1889523
Syria	2017-04-07 03:32:26	1968216

query.py
Task 3 – **df_time**

Starting with "t3_csv", run an SQL query and store the results in **df_time**.

- q_time = "SELECT time, Sum(volume) AS `total` FROM t3_csv GROUP BY time ORDER BY time"
- df_time = spark.sql(q_time)

t3_csv

time	trend	volume
2017-04-06 20:48:18	#LEmissionPolitique	92636
2017-04-06 20:48:18	Don Rickles	112258
2017-04-06 20:48:18	#Gala15GHVIP5	17868
2017-04-06 20:48:18	#Fast8EH	
2017-04-06 20:48:18	#VzlaTrancaContraElGolpe	152349
2017-04-06 20:48:18	Dustin Johnson	53490
2017-04-06 20:48:18	French Montana	93712
2017-04-06 20:48:18	Mari Palma	
2017-04-06 20:48:18	#twepsv	
2017-04-06 20:48:18	#NuclearOption	95342
2017-04-06 20:48:18	#PuebloYFANBLealtadAbsoluta	67205
2017-04-06 20:48:18	#gntm	
2017-04-06 20:48:18	#CelebrityMasterChefIt	
2017-04-06 20:48:18	#TPMPlesparis	
2017-04-06 20:48:18	#survirorgr	

t3_csv | time

time	total
2017-04-06 20:48:18	1210685
2017-04-06 20:53:10	1211454
2017-04-06 21:03:16	891431
2017-04-06 21:18:16	876058
2017-04-06 21:28:08	926815
2017-04-06 21:33:13	876452
2017-04-06 21:43:08	907272
2017-04-06 21:53:04	957933
2017-04-06 22:03:06	978410
2017-04-06 22:17:58	971845
2017-04-06 22:27:59	1030493
2017-04-06 22:37:56	1067197
2017-04-06 22:42:52	1083190
2017-04-06 22:52:50	1110049
2017-04-06 23:02:56	1112329

query.py
Task 3 – **df_trend**

Starting with "t3_csv", run an SQL query and store the results in **df_trend**. Label DataFrame as "trend"

- q_time = "SELECT trend, Sum(volume) AS `total` FROM t3_csv GROUP BY trend ORDER BY Sum(volume) DESC LIMIT 3"
- df_trend = spark.sql(q_trend)

t3_csv

time	trend	volume
2017-04-06 20:48:18	#LEmissionPolitique	92636
2017-04-06 20:48:18	Don Rickles	112258
2017-04-06 20:48:18	#Gala15GHVIP5	17868
2017-04-06 20:48:18	#Fast8EH	
2017-04-06 20:48:18	#VzlaTrancaContraElGolpe	152349
2017-04-06 20:48:18	Dustin Johnson	53490
2017-04-06 20:48:18	French Montana	93712
2017-04-06 20:48:18	Mari Palma	
2017-04-06 20:48:18	#twepsv	
2017-04-06 20:48:18	#NuclearOption	95342
2017-04-06 20:48:18	#PuebloYFANBLealtadAbsoluta	67205
2017-04-06 20:48:18	#gntm	
2017-04-06 20:48:18	#CelebrityMasterChefIt	
2017-04-06 20:48:18	#TPMPlesparis	
2017-04-06 20:48:18	#survirorgr	

t3_csv | top_trend

trend	total
Syria	89464572
Happy Easter	70626740
#PREMIOSMTVMIAW	70564474

query.py
Task 3 – **df_trend_time**

Using "trend" & "t3_csv, run an SQL query and store the results in **df_trend_time**.

- q_trend_time = "SELECT trend.trend, t3_csv.time, t3_csv.volume FROM trend INNER JOIN t3_csv ON trend.trend = t3_csv.trend"
- df_trend_time = spark.sql(q_trend_time)

t3_csv

time	trend
2017-04-06 20:48:18	#LEmissionPolitique
2017-04-06 20:48:18	Don Rickles
2017-04-06 20:48:18	#Gala15GHVIP5
2017-04-06 20:48:18	#Fast8EH
2017-04-06 20:48:18	#VzlaTrancaContraElGolpe
2017-04-06 20:48:18	Dustin Johnson
2017-04-06 20:48:18	French Montana
2017-04-06 20:48:18	Mari Palma

t3_csv | top_trend

trend	total
Syria	89464572
Happy Easter	70626740
#PREMIOSMTVMIAW	70564474

t3_csv | trend | trend_time

trend	Time	volume
Syria	2017-04-07 01:32:42	1048749
Syria	2017-04-07 01:42:53	1125025
Syria	2017-04-07 01:52:49	1211343
Syria	2017-04-07 02:07:45	1299043
Syria	2017-04-07 02:17:50	1383853
Syria	2017-04-07 02:22:46	1425476
Syria	2017-04-07 02:37:49	1552001
Syria	2017-04-07 02:47:46	1635870
Syria	2017-04-07 02:52:45	1675279
Syria	2017-04-07 03:02:46	1724803
Syria	2017-04-07 03:12:40	1806824
Syria	2017-04-07 03:22:33	1889523
Syria	2017-04-07 03:32:26	1968216
Syria	2017-04-07 03:42:17	2042145

query.py
Task 3

Both of the final DataFrames, **df_time** & **df_time_trend**, are now complete. The results will now be passed to graph.py to be plotted.

The graph.py needs to be able to access the results via an index, so both **df_time** & **df_time_trend** are converted via the collect method and returned in a list.

- result = [df_time.collect(),df_trend_time.collect()]
- return result

This concludes the **qSQL_t3** function. The result will be returned to runQuery, which in turn returns the result to the flask server. The flask server will pass the result, along with other arguments, to the graph.py

graph.py

After the server has retrieved the result from query.py, the server will call **makeFig** from graph.py to plot the results.

Along with the results, figure save name, figure attributes and time stamp are passed in. Figure attributes include information such as title, axis and description.

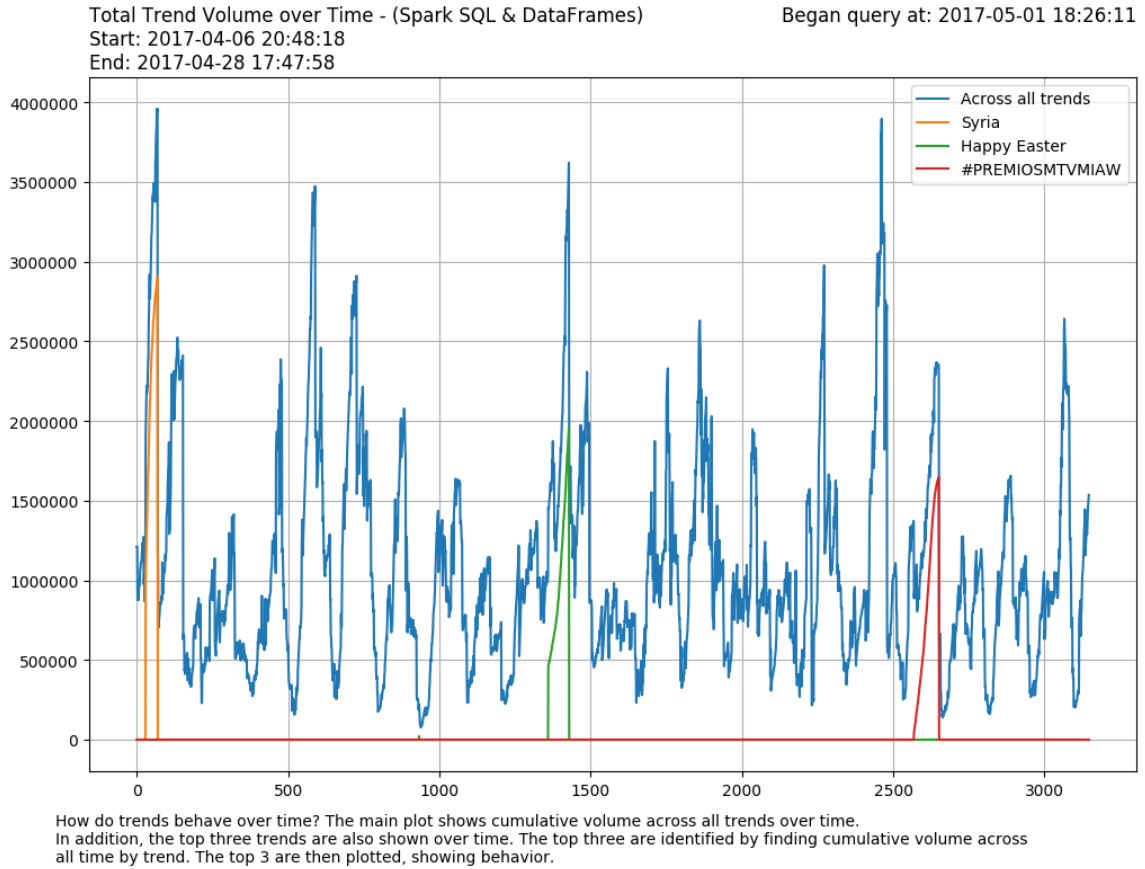
Each task has a specific function in graph.py. Depending on what **q** is, makeFig will call the appropriate one. The figure is plotted using the matplotlib module and is stored in a pyplot object. Using the pyplot object, the figure is saved using **saveGraph**.

```
def makeFig(q, result, nameSave, fig_attrib, time_stamp):
    #Each query needs to be graphed a bit differently
    if q == 1:
        pyplotfig = graph_t1(q, result, fig_attrib, time_stamp)
    elif q == 2:
        pyplotfig = graph_t2(q, result, fig_attrib, time_stamp)
    elif q == 3:
        pyplotfig = graph_t3(q, result, fig_attrib, time_stamp)
    #Save graph
    saveGraph(q, pyplotfig, nameSave)
```

graph.py
Task 3

For task 3, four sets of data need to be plotted:

- **The main plot**, which shows cumulative tweet volume across all trends over time.
- **Three secondary plots**, which show how the top 3 trends' tweet volume vary over time.



Main plot in blue. Three Secondary plots in orange, green and red.

graph.py
Task 3

The raw input, the results from query.py, is a list containing two tables. The main plot, using the collected results from **df_time**, can more or less be used as is; just need to split the time and total fields into separate lists.

The secondary plots will need to be prepared before plotting. Each secondary plot may be shorter than the main plot, as they only include the times at which trend volume was significant. (Data collection limitation).

If the main plot has N data points but a secondary plot has M data points, where M < N, then N - M additional data points need to be added to enable plotting. The additional N – M data points will all be zero, as these are times when the trend did not have significant volume.

graph.py
Task 3

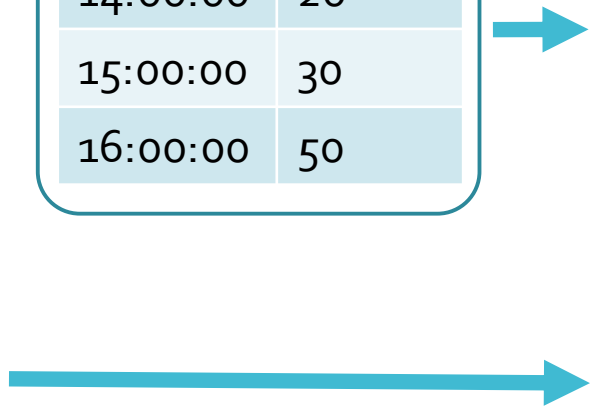
To illustrate, imagine a toy example. Say the main plot has 7 data points, starting at 11 AM and ending at 5 PM as shown to the left. A secondary plot has 3 data points, as shown in the middle. So M = 7, N = 3. Since 7 > 3, M – N = 4 data points need to be added to enable plotting. The new secondary plot's 7 data points are shown in the rightmost table.

Main

Time	Total
11:00:00	5
12:00:00	10
13:00:00	15
14:00:00	30
15:00:00	40
16:00:00	60
17:00:00	10

Secondary

Time	Total
14:00:00	20
15:00:00	30
16:00:00	50



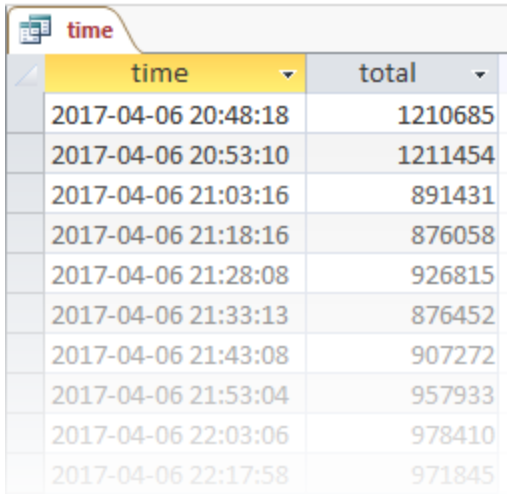
New Secondary

Time	Total
11:00:00	0
12:00:00	0
13:00:00	0
14:00:00	20
15:00:00	30
16:00:00	50
17:00:00	0

graph.py
Task 3

Before discussing the implementation, it is important understand how the plot's data is made. Data from **df_time** is used for the main plot. Only the "total" field's values are used, with the "time" field's values mapped to the list indexes. The mapping of time to index is shown on the right. The list of data for plotting is shown in the lower right.

Time	Index
2017-04-06 20:48:18	0
2017-04-06 20:53:10	1
2017-04-06 21:03:16	2
…	4 – 3146
2017-04-28 17:47:58	3148


time

time	total
2017-04-06 20:48:18	1210685
2017-04-06 20:53:10	1211454
2017-04-06 21:03:16	891431
2017-04-06 21:18:16	876058
2017-04-06 21:28:08	926815
2017-04-06 21:33:13	876452
2017-04-06 21:43:08	907272
2017-04-06 21:53:04	957933
2017-04-06 22:03:06	978410
2017-04-06 22:17:58	971845

Index	Total
0	1210685
1	1211454
2	891431
3-3146	…
3147	1535997

graph.py
Task 3

We'll use the toy problem to explain the implementation.

Since time is mapped to index, let's update both the primary (below) and secondary (right) examples.

Time	Total
11:00:00	5
12:00:00	10
13:00:00	15
14:00:00	30
15:00:00	40
16:00:00	60
17:00:00	10

→

Idx	Total
0	5
1	10
2	15
3	30
4	40
5	60
6	10

Time	Total
11:00:00	0
12:00:00	0
13:00:00	0
14:00:00	20
15:00:00	30
16:00:00	50
17:00:00	0

→

Idx	Total
0	0
1	0
2	0
3	20
4	30
5	50
6	0

graph.py
Task 3

In the implementation, the new secondary plots' data will be stored in **trend1_val**, **trend2_val** & **trend3_val**; all are initialized as N long lists full of zeroes. The assigning of non-zero values is done in the for loop.

```
trend1_val = [0]*len(time)
trend2_val = [0]*len(time)
trend3_val = [0]*len(time)
for row in trend:
    if row[0] == trend1 and row[1] in main_axis:
        idx = main_axis.index(row[1])
        trend1_val[idx] = float(row[2])
    elif row[0] == trend2 and row[1] in main_axis:
        idx = main_axis.index(row[1])
        trend2_val[idx] = float(row[2])
    elif row[0] == trend3 and row[1] in main_axis:
        idx = main_axis.index(row[1])
        trend3_val[idx] = float(row[2])
```

Below the for loop, the first conditional (e.g. row[0] == trend1) sorts the data from trend (which holds all 3 trends) into the appropriate bucket. The second conditional (e.g. row[1] in main_axis) checks if the trend's time appears in the main plot's data.

trend_time

trend	Time	volume
Syria	2017-04-07 07:34:08	2864677
Syria	2017-04-07 07:49:11	2895475
Syria	2017-04-07 07:54:15	2905677
Syria	2017-04-07 08:04:13	2904702
Happy Easter	2017-04-13 08:16:03	21975
Happy Easter	2017-04-16 07:26:46	467223
Happy Easter	2017-04-16 07:31:44	471602
Happy Easter	2017-04-16 07:41:42	479551
Happy Easter	2017-04-16 07:56:40	492993
Happy Easter	2017-04-16 08:01:42	491979

If true, use the time to locate what index the volume should be mapped to. (e.g. idx = main_axis.index(row[1])).
Using that index, assign the trend volume to the new secondary (e.g. trend1_val[idx] = float(row[2]))

graph.py
Task 3

To illustrate, let's use the toy problem again. To simplify, we'll only walk through the algorithm using a single trend.

Going through the rows in the Secondary, the first time is 14:00:00.

- 1st Conditional: N/A, since only 1 trend
- 2nd Conditional: yes 14:00:00 appears in Main
  - What is index of 14:00:00 in Main? 3
  - Assign value of 20 to the new Secondary at index 3

The next two rows, 15:00:00 and 16:00:00 appears in Main and correspond to indexes of 4 and 5. Hence values 30 and 50 are assigned to the new Secondary at indexes of 4 and 5 respectively.

Idx	0	1	2	3	4	5	6	7
Total	0	0	0	20	40	50	0	0

Main

Time	Total
11:00:00	5
12:00:00	10
13:00:00	15
14:00:00	30
15:00:00	40
16:00:00	60
17:00:00	10

Secondary

Time	Total
14:00:00	20
15:00:00	30
16:00:00	50

graph.py
Task 3

With the plot data read, the main plot and three secondary plots are ready to be plotted to a figure.

- plt.plot(x, main_val) and subsequent plt.plot() will plot the data.
- x is a list of numbers from 0 to N-1. (x = range(len(time)))

The rest of the code manages attributes such as description, title, legend, etc. When complete, the plot is returned as a pyplot object.

```
legend = ['Across all trends',trend1, trend2, trend3]
fig_title, fig_desc = fig_attrib
#Create figure and configure it
plt.figure(3,figsize=(12,8))
plt.plot(x,main_val)
plt.plot(x,trend1_val)
plt.plot(x,trend2_val)
plt.plot(x,trend3_val)

plt.tick_params(axis = 'x', which = 'minor', bottom = 'off', top = 'off', labelbottom = 'off' )
fig_timestamp = 'Began query at: ' + time_stamp
plt.title(fig_title + '\nStart: ' + main_axis[0] +'\nEnd: '+ main_axis[-1], loc='left')
plt.title(fig_timestamp + '\n\n', loc='right')
plt.figtext(.1,0.01,fig_desc)
plt.legend(legend)
plt.grid()
return plt
```

results.html

After the figure is saved, the flask server renders the result page, passing on which task was selected and what the file's save name is.

Flask renders a jinja2 template, which is an HTML file but some of the content are variables that are decided at render time. (e.g. {{**q**}})

A jinja2 template also supports conditional statements. If the q is equal to zero (not valid query) an error message is shown. Otherwise figure is shown. The figure's path is stored in the variable {{**figName**}}

```
<!doctype html>
<html>
  <head>
    <title>Project 3 Results</title>
    <link rel="stylesheet" type="text/css" href="/static/style.css" />
    <link rel="stylesheet" type="text/css" href="/static/result.css" />
  </head>
  <body>
    <div id = "result_container">
      <div id ="header">
        <h1>Intro to Big Data - Project 3 GUI</h1>
      </div>
      <div id = "result_content">
        <div id = "result_title"><h1>Results From Task {{q}}</h1></div>

        <button id="result_return" onclick="return_home()"><h1>Run Another Task</h1></button>

        <div id ="result_frame">
          {% if q == 0 %}
          <img id = "result_figure" src="/static/results_error.jpg" />
          {% else %}
          <img id = "result_figure" src={{figName}} />
          {% endif %}
        </div>
      </div>
      <div id = "footer">
        UMKC - SP2017 - GO ROOS!
      </div>
    </div>
  </body>
</html>

<script  type="text/javascript">
  function return_home(){
    home = "http://localhost:5555/"
    window.location.assign(home);
  }
```