Big Data Tools and Techniques

Analysis of Event Data

Submitted by

Mohammad Bilal Iqbal

Contents

Introduction

This project analyses twitter data over the course of 48 Premier League matches. Tweets with official hashtags were primarily monitored and the entire data was made available for analysis.

Tables & Structure

The data was provided in the form of SQL dump. It consists of the following six tables:

FC

This table consists of 22 rows and is basically a list of all the 20 football clubs that are played in the Premiere League. The table also contains two additional team names from La Liga: Barcelona and Sevilla. These two teams did not play any of the 48 games that were monitored for this analysis project, as revealed in the answer to question 3.

Game

This table consists of 48 rows — the 48 matches that were monitored under this analysis project. This table consists of the following columns:

ID: Ranges from 1-48, representing the game ID of the match.

FC1: This column lists the team hashtag ID of the first of the two teams that played in the match.

FC2: This column lists the team hashtag ID of the second of the two teams that played in the match.

OfficialStart: The official start time of the game, captured in SQL datetime format.

OfficialEnd: The official end time of the game, captured in SQL datetime format.

HalfTimeStart: Start of the half time in the game, captured in SQL datetime format.

HalfTimeEnd: End of the half time in the game, captured in SQL datetime format.

Hashtag FC

This table consists of 22 rows and two columns: Hashtag ID and FC ID. This table links the Hashtag IDs to the teams, as they are numbered in the FC table.

Tweet

This table consists of 984001 rows and consists of the following three columns:

ID: Each tweet has a unique ID that is represented in this column.

Created: This column captures the time each tweet was created.

UserID: This column captures the ID of the user who generated the tweet.

Tweet Hashtag

This table consists of 1338097 rows and consists of the following two columns:

TweetID: This is the ID used to identify a tweet and is the same as the ID column in the Tweet table. This column is what links this table to the Tweet table. Please note that the number of rows in this table is greater than Tweet because a single tweet can contain more than 1 hashtags in it.

HashtagID: The Hashtag ID numbers the hashtag that was used in a particular tweet and links this table to the to the Hashtag table.

Game Goals

This table consists of 136 rows and consists of the following columns:

GameID: Identifies the game number in which the goal was scored.

Time: The time at which the goal was scored.

FC: This column identifies the team that scored the goal.

The number of rows indicates the total number of goals that were scored in the 48 matches that were monitored.

Hashtag

This table consists of 15070 rows and is basically a list of Hashtag ID along with the hashtag that the ID represents.

Data Import and Correctness

All six tables in the SQL dump were loaded into the supplied machine's MySQL server, from where they were imported into HDFS and Hive using sqoop. The following command was used for Hive:

```
sqoop import-all-tables --connect jdbc:mysql://localhost/bigdata --username training --password training --hive-import
```

Please note that the username and password supplied here (training) works specifically for the Virtual Machine (VM) copy that was made available for the Big Data Tools and Techniques course. A different VM will have different username and passwords (the username for Cloudera Quickstart VM is root, password is cloudera, for instance).
Once imported, a few basic checks were performed on the data to check for its correctness. One of these checks revealed that the time stamp for one of the games was off by a day. This was discovered when the official start timestamps were subtracted from the official end timestamps to calculate the game duration of each game using the following code:

```
select unix_timestamp(game.officialend)-unix_timestamp(game.officialstart) from game;
```

An error was discovered in game 17. Since the supplied version of the Cloudera VM (5.4) does not support Hive table updates[1], the necessary correction was made in the VM's MySQL local instance and the data was imported again. Other checks did not reveal anything that would interfere with the answers to the questions.

Data Visualization

All data processing was done using Hive/Impala and Spark RDDs. The results were exported out to text files with comma separated values. These results were then brought into Apache Open Office Calc spreadsheet processor. Calc was used to generate all the graphs below.

Implementation of solutions in Hive/Impala

Data Preparation

All problem statements pertaining to the dataset deal with tweets that were created during game times. All tweets outside of game times were therefore filtered out to improve query speed time and reduce CPU and memory load. This reduced the size of tweets table from 984001 to 678232 (31% reduction). The following query was used:

```
create table if not exists tweets_during_game_table

as

select distinct tweet.id, tweet.created, tweet.user_id from tweet, game

where tweet.created between game.officialstart and game.officialend;
```

Moreover, since five of the six problem statements pertain to tweets with official hashtags, all non-official hashtags were also filtered out, using the following query:

```
create table if not exists tweet_hashtag_official_table

as

select tweet_hashtag.tweet_id, tweet_hashtag.hashtag_id from tweet_hashtag

where tweet_hashtag.hashtag_id > 0 and tweet_hashtag.hashtag_id<23;
```

[1] This functionality has been added in Cloudera VMs from version 5.10 onwards

These two tables were then combined to form a consolidated tweets table, which was used to answer five of the six problem statements.

```
create table if not exists tweets_consolidated

as

select a.tweet_id, b.user_id, b.created, a.hashtag_id


from tweets_during_game_table b


join tweet_hashtag_official_table a


where a.tweet_id =b.id;
```

Question 1

The average number of games was achieved in two parts. The first step involved creating a view with the total number of games, i.e.:

```
count(distinct tweets_consolidated.tweet_id)
```

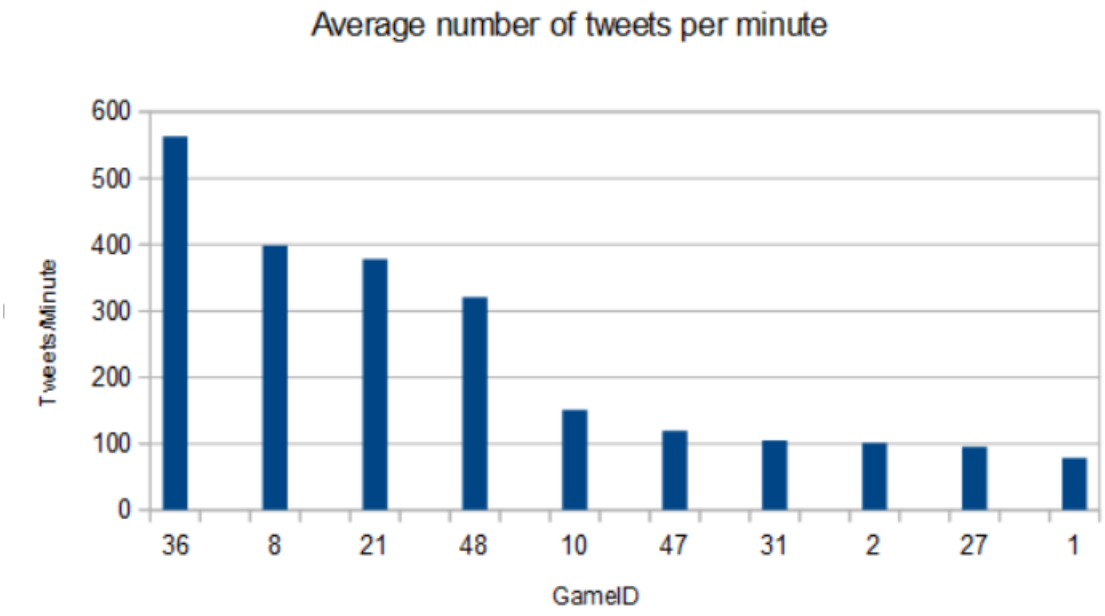
from game, tweets_consolidated such that:

```
(tweets_consolidated.hashtag_id = game.fc1 or tweets_consolidated.hashtag_id = game.fc2) and
(tweets_consolidated.created between game.officialstart and game.officialend)
```

This view lists the total count for each game. To get average, calculate the game duration for each game and divide count by the game duration to get the average number of tweets per minute (calculated average is per second, so multiply by 60 to get per-minute average):

```
select id, (num_tweets/(unix_timestamp(officialend)-unix_timestamp(officialstart)))*60 average
from games_with_tweet_counts order by average desc limit 10;
```

The game between Arsenal and Manchester United generated the most buzz on Twitter — an average of 562 tweets per minute. A total of four goals were scored in this game, with Manchester United winning 3-1. In fact, the top five most popular matches in terms of tweets per minute featured Manchester United, indicating that this team's fans are the most vocal on Twitter.

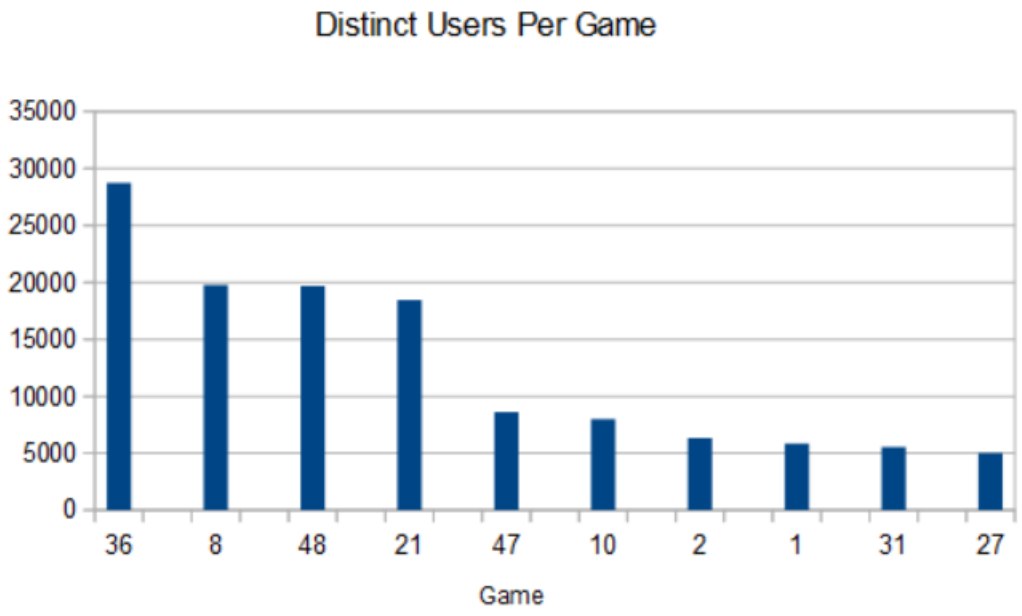


Question 2

The solution to this problem statement is similar to Question 1. Instead of distinct tweets, we are now calculating distinct users per game, so:

```
select  count(distinct user_id) UserCount

from tweets_consolidated join game where

(tweets_consolidated.created between game.officialstart and game.officialend)

and

(tweets_consolidated.hashtag_id = game.fc1 or tweets_consolidated.hashtag_id = game.fc2)
```

Immediately, we see similarities between graphs for Q1 and Q2. The same ten games appear in both tables, albeit in somewhat different order. Games 36, 8, 48 and 21 continue to be the most popular in both graphs.

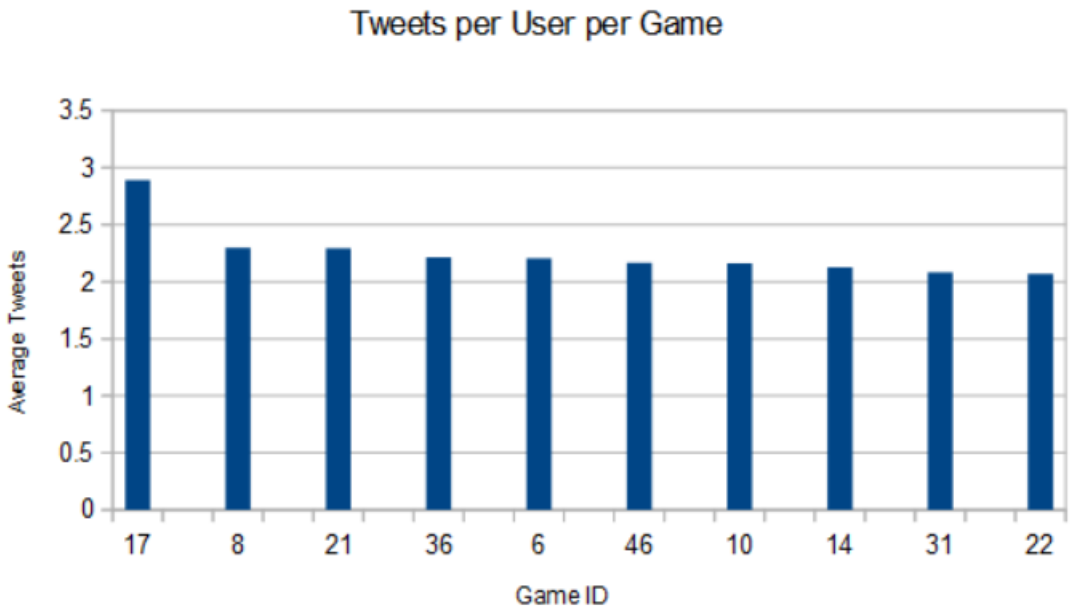


Q1&Q2 Extension

The first two problem statements lead to another consideration: which games have the most engaged audience. One way of determining this is to calculate the average number of tweets that each user creates during the game. Turning the query that answers Q2 into a view, the following SQL query generates the required output:

```
select distinct_users.id, num_tweets/usercount perUser
from games_with_tweet_counts, distinct_users
where
distinct_users.id=games_with_tweet_counts.id
group by distinct_users.id, perUser order by perUser desc limit 10;
```

Even though some of the games that appeared in the top 10 list for Q1 and Q2 appear in this graph, we see that the game with the most user engagement was played between Arsenal and Burnley, two closely matched teams in the Premier League table. The game went goalless until the last minute, where Arsenal managed to score and beat Burnley 1-0. The final score line, when the goal was scored and the higher average tweets per user indicates that this may have been a closely contested game, generating greater user engagement than average. Apart from this game, we see that the average number of tweets per users per game for other top games is fairly the same and averages at slightly over 2 tweets per game per user. This confirms what the graphs for Q1 and Q2 indicate when looked at together: that the games with the highest number of tweets also had the most number of users tweeting.



Question 3

This question has two parts. First, we need to calculate the teams that played the highest number of games. So:

select fc.id, count (*) NumberofGames from fc, game

where fc.id=game.fc1 or fc.id=game.fc2

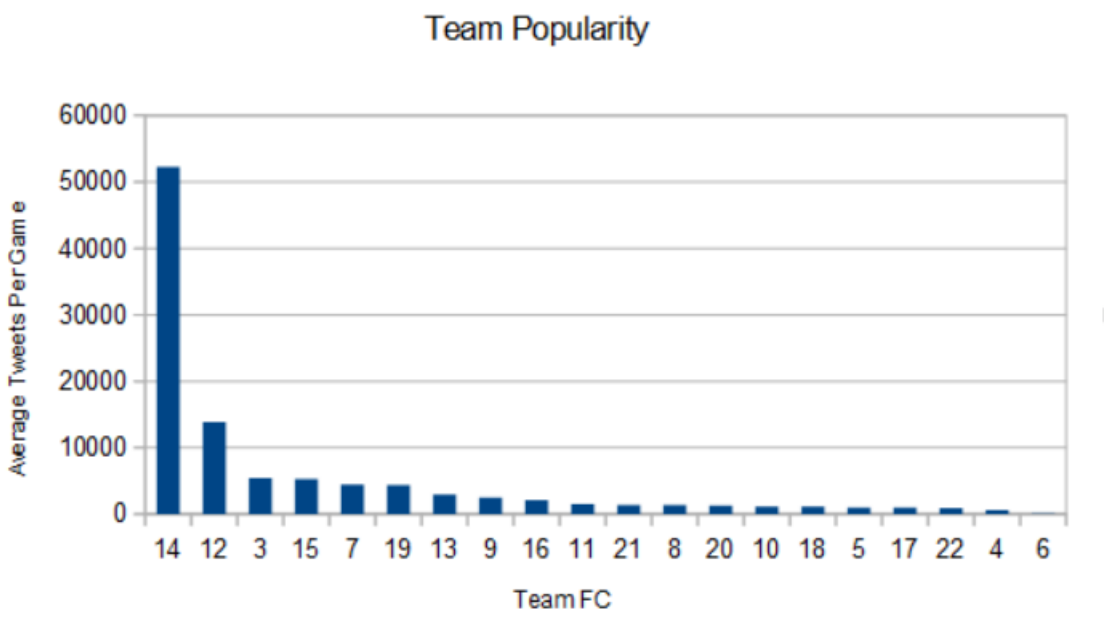
This reveals that most of the teams have played the same number of games, i.e. 5. Only Leicester (4 games), Watford (4 games) and West Ham (3 games) played fewer games. Barcelona (FC1) and Sevilla (FC2), not part of the Premier League, did not play any games. Since most teams played the same amount of games, it seemed pertinent to chart the averages for all teams. So:

count(distinct tweet_id)/teams_games_played.numberofgames

where

tweets_consolidated.hashtag_id=teams_games_played.id and (unix_timestamp(tweets_consolidated.created) between unix_timestamp(game.officialstart) and unix_timestamp(game.officialend))

Immediately, it is clear that Manchester United (Team ID 14) commands the most vocal supporters in Premier League on Twitter. There average of just over 52,000 tweets per game is nearly four times then second most popular team, Liverpool (about 14,000 tweets per game). Liverpool in turn is mentioned over 2.5 times more often than Arsenal (5,457 tweets per game) and other teams.



Question 4

One way to calculate spike for a game in the last 10 minutes is the ratio of average number of tweets in the last ten minutes to the average number of tweets in the game time leading up to the last 10 minutes. Therefore, count the total number of tweets for the last ten minutes for each, and for the interval before that. Calculate averages for both time periods (divide by 10 in the case of last ten minutes, and divide by (OfficialEnd minus ten minutes) minus OfficialStart for the other time period). Divide the average number of tweets per minute in the last ten minutes with the average number of tweets for the first interval to calculate the spike for that game.

This approach eliminates any bias in favor of more popular games in terms of total tweet counts, as the final spike being calculated depends on a ratio of two averages and not their absolute numbers.

So:

```
create view if not exists ten_with_tweet_counts

as

select ten.id, (count(distinct tweets_consolidated.tweet_id))/10 num_tweets from ten join
tweets_consolidated
```

And:

```
select until_ten.id, ((count(distinct tweets_consolidated.tweet_id))/(unix_timestamp(t10)-
unix_timestamp(officialstart)))*60 num_tweets from until_ten join tweets_consolidated
```

Then bring the two together:

```
select until_ten_with_tweet_counts.id, ten_with_tweet_counts.num_tweets ten,
until_ten_with_tweet_counts.num_tweets untilten
```

from until_ten_with_tweet_counts join ten_with_tweet_counts

where

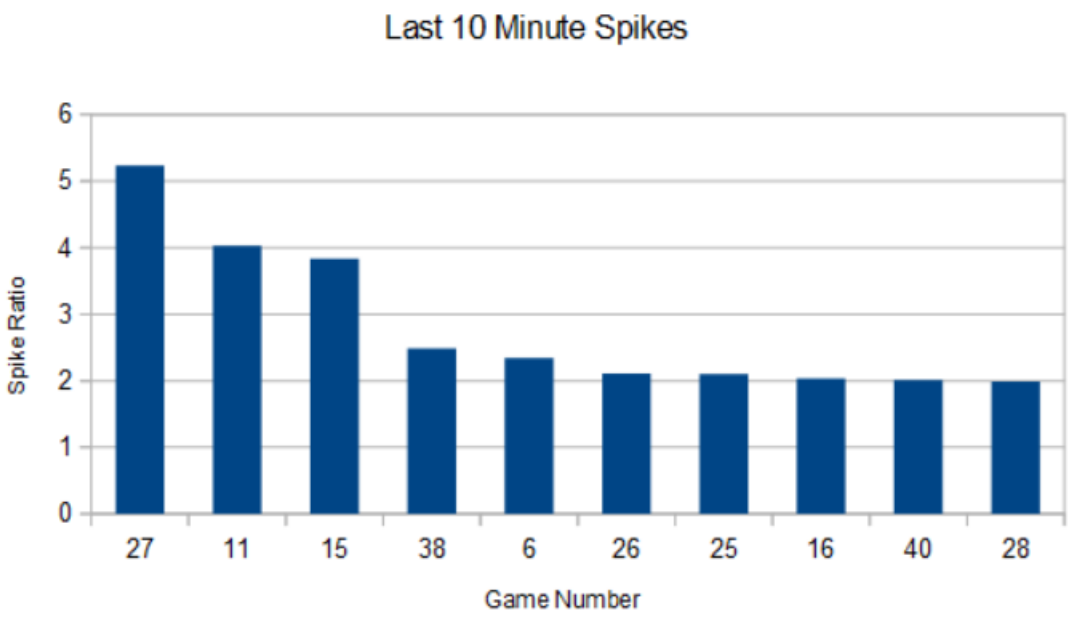
until_ten_with_tweet_counts.id=ten_with_tweet_counts.id;

And Finally:

select id, ten/untilten spike from prespike_q4 group by id, spike order by spike desc limit 10;

The graph reveals that Game 27 saw the highest spike in the last 10 minutes, with the average number of tweets over five times as high as the average number of tweets in the interval preceding it. This game was played between Liverpool and Swansea, with Liverpool winning 3-0. The spike was likely because 2 of those 3 goals came in the last 10 minutes. Stoke and Crystal Palace in game 11 were tied up 1-1 until Crystal Palace took the lead in the last 5 minutes to secure the win. In game 15 between Chelsea and Liverpool, Liverpool was winning 1-0 until Chelsea scored an equalizer in the last 10 minutes.

The spikes in the last 10 minutes for each of these games were prompted by the scoring of goals, which makes sense as fans of both teams would be eager to let their feelings known online.

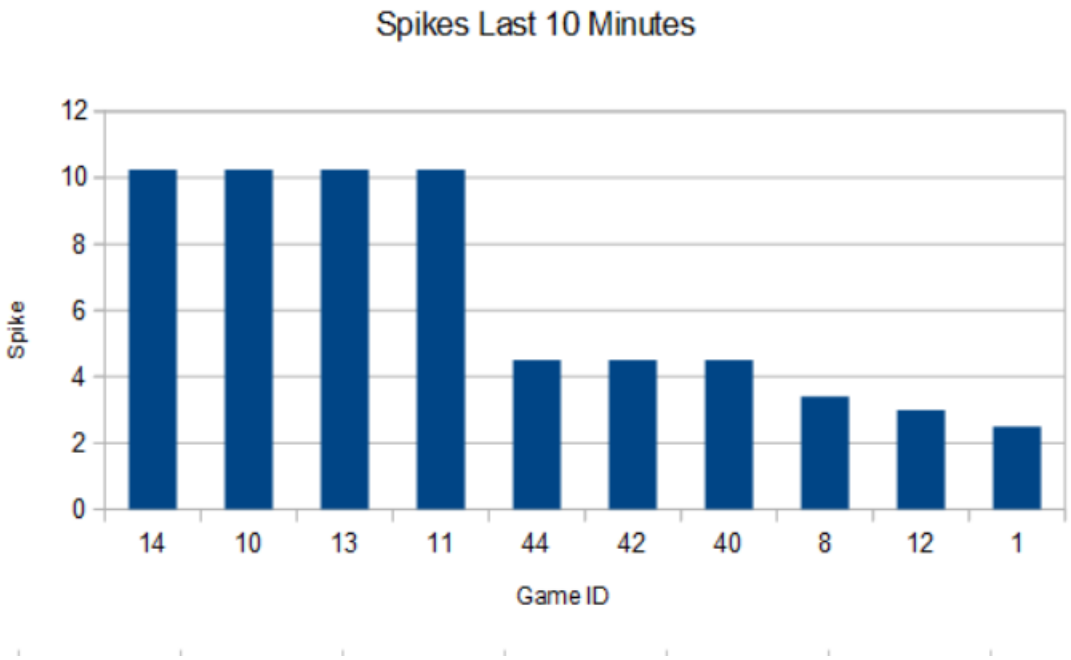


Q4 Alternative

Another way to calculate spike in the last ten minutes of a is to calculate tweet counts for each of the last 11 minutes of a game. These are calculated the same way as the original Q4 answer. Then use those counts to generate a moving average. The greatest number of these gives the maximum value of a spike at any given minute. So:

greatest((T11+T10)/2, (T10+T9)/2, (T9+T8)/2, (T8+T7)/2, (T7+T6)/2, (T6+T5)/2, (T5+T4)/2, (T4+T3)/2, (T3+T2)/2, (T2+T1)/2, (T1+T1)/2)

Trying to correlate this graph with what was happening during the games in the last 10 minutes reveals little useful information. This approach would be more helpful if the spikes are calculated over the duration of the game for the entire game and then correlated to specific events as and when they happened (goals, cards, fights etcetera). The solution is nevertheless submitted along with this report for perusal.



Question 5

Since we are considering un-official hashtags for this question, we need to make a tweets_consolidated2 table that would keep tweets with unofficial hashtags. So:

```
select a.tweet_id, b.user_id, b.created, a.hashtag_id
from tweet_hashtag as a full join tweets_during_game_table as b
on a.tweet_id =b.id;
```

Filter out the resulting dataset to contain only tweets that were created during the game times:

```
select tweet_id, hashtag_id from
tweets_consolidated2, game
where tweets_consolidated2.created between game.officialstart and game.officialend;
```

Filter out the official hashtags from this dataset, so tweets from 1-22. Count the occurrence of each hashtag in the dataset and finally combine with the hashtag table to get the names of the most common 10 non-official hashtags that were used using the following query:

```
select hashtag.id, hashtag.hashtag, counts
```

from hashtag, top_nonofficial_hashtags

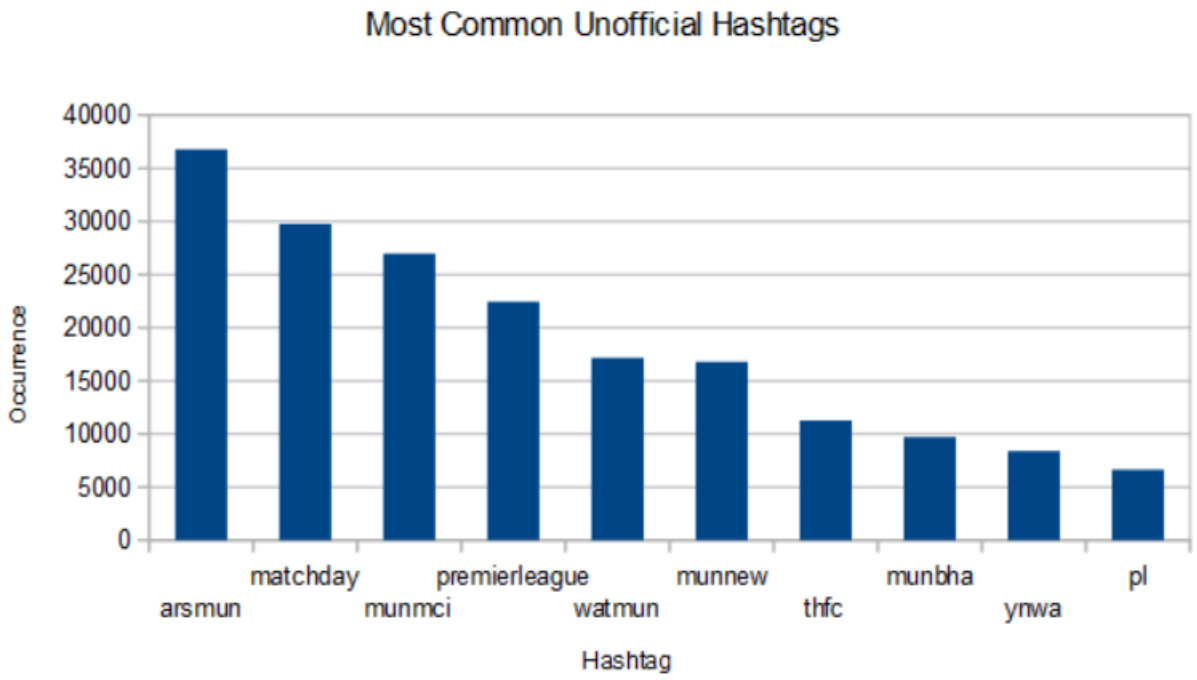
where hashtag.id=hashtagid group by hashtag.id, hashtag.hashtag, counts

order by counts desc;

Manchester United dominates the unofficial hashtags in some form or another. The most common unofficial hashtag was arsmun (36,770 mentions), which corresponds well with the most popular game in terms of tweets (Game 36, played between Arsenal and Manchester United).

Manchester United's hashtag mun, is found in five of the 10 most popular unofficial hashtags that were tabulated for the 48 games under consideration.

The general hashtag matchday was the second most popular, with 29,776 mentions, followed by munmci (Manchester United vs Manchester City) with 26,999 mentions.



Question 6

The total number of tweets at each minute of game 36 were graphed out, as this was the most popular game in terms of the total number of tweets. The time stamps for all tweets were converted to unix_timestamp. These were divided by 60 and converted to bigints to collapse the seconds part of each timestamp to minutes. This had the result of sectioning all tweets in minute-long partitions, which were then used to count total number of tweets at each minute and plot the graph. The game starts at 25203931 (first minute) and ends at 25204042 (last minute) for a total duration of 112 minutes (the accompanying graph is marked in half-second increments). The following query was used:

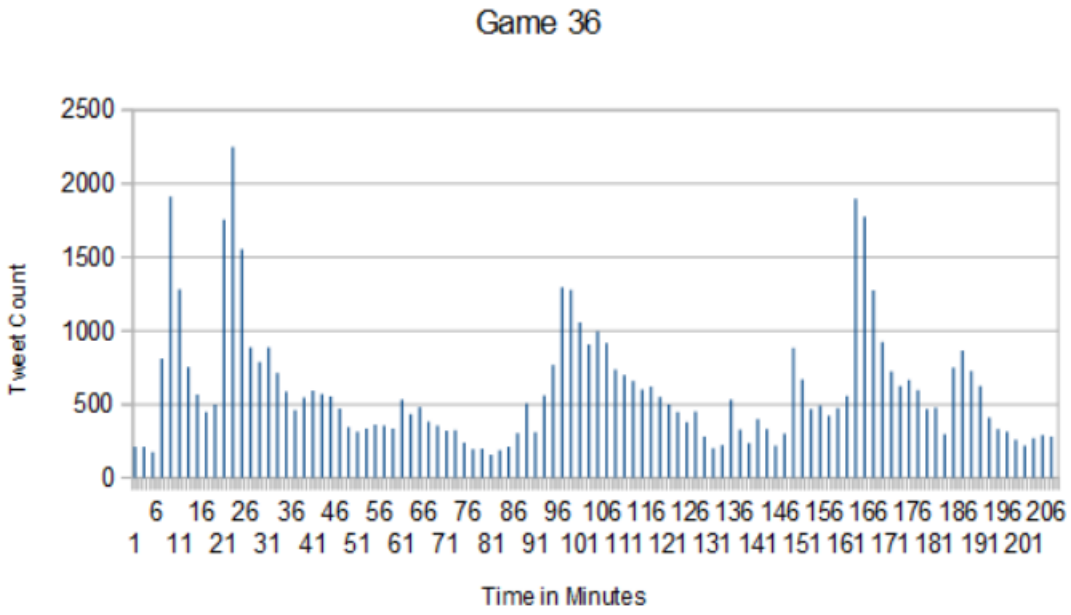
select

cast(unix_timestamp(created)/60 as bigint), count(tweet_id)

```
from tweets_consolidated
where
cast(unix_timestamp(tweets_consolidated.created)/60 as bigint)
>
cast(unix_timestamp('2017-12-02 17:30:00.0')/60 as bigint)
and
cast(unix_timestamp(created)/60 as bigint)
<
cast (unix_timestamp('2017-12-02 19:23:00.0')/60 as bigint)
and (hashtag_id=14 or hashtag_id=3)
group by
cast(unix_timestamp(created)/60 as bigint)
order by
cast(unix_timestamp(created)/60 as bigint);
```

From the graph, we four major spikes. Two happen at the start: the fifth minute (from 813 to 1913 tweets) and the 11th minute (from 500 to 1756). Another milder spike occurs at the 49th minute mark (from 766 to 1294), probably due to half time. The last major spike happens at 82nd minute (from 556 to 1900).

Four games were scored in this match, and the spikes correspond to when the goals were scored. The first goal was scored 4 minutes into the match and the second goal was scored 10^{th} minute into the match (both by Manchester United). The third goal was scored by Arsenal in 57th minute and was also marked by spike (from 222 to 532). The last goal, was scored by Manchester United in the 81st minute and directly corresponds to the last major spike in the game. As expected, people tend to swarm Twitter with Tweets when a goal is scored. Half-time is another point where fans become active, probably because of the break in the game.



[2]

Implementation of solutions in Scala

Data Preparation

There are three ways to work with data in Spark: Java, Python and Scala. Scala was chosen for the implementation of the solutions to the six problem statements because it is 10 times as fast as Python for Spark applications.[3]

Additionally, there are three ways to deal with the data itself once it is imported: RDDs, Dataframes and Datasets. RDDs were chosen for the implementation of the solutions because the version of the supplied VM (with Spark 1.3) is quite old and has limited support for Dataframes and no support for Datasets (these were introduced in Spark 1.6)[4].

Where possible, the logic of all the solutions was kept as close to the Hive solution as possible. The original tables were imported into HDFS using the following command:

```
sqoop import-all-tables --connect jdbc:mysql://localhost/bigdata --username training --password training --warehouse-dir /user/training/bigdata
```

These tables were then individually read into RDDs in the Spark-Shell for further processing. The first step was to get a consolidated tweets RDD, which would hold information pertaining to TweetID, Tweet Hashtags, UserID and Time Created.

Like the solution for Hive/Impala, the Tweet Hashtag RDD was filtered to keep only the official hashtags (1 to 22):

---

[2] Please note that the X-Axis is marked in half-second increments

[3] https://www.dezyre.com/article/scala-vs-python-for-apache-spark/213

[4] https://www.linkedin.com/pulse/apache-spark-rdd-vs-dataframe-dataset-chandan-prakash

```
val tweet_during_game= tweet_hashtag3.filter(x=> {x(1).toInt>0 && x(1).toInt<23})
```

Like the solution for Hive/Impala, the Tweet Hashtag RDD was filtered to keep only the official hashtags. The Tweet and Tweet_Hashtag RDDs were then keyed by the TweetID and joined to get a consolidated RDD:

```
tweet_during_game2.join(tweet4)
```

This RDD would then need to be joined to the Game RDD, and the only common elements (columns) between the tweet_consolidated and Game RDDs are the hashtags. Since each Game RDD has two of these, it was necessary to divide the Game RDD into two, each keyed by one of the FCs, i.e. FC1 and FC2. The halftime start and end times were left out because they do not feature in the solutions to the questions.

```
var game5= game4.map(x=> (x._2, (x._1, x._4, x._5)))

var game6= game4.map(x=> (x._3, (x._1, x._4, x._5)))
```

The mapped structure of game5 is: (FC1, GameID, OfficialStart, OfficialEnd).

The mapped structure of game6 is: (FC2, GameID, OfficialStart, Official End).

This mapping made it possible for them to be joined to the tweet_consolidated RDDs:

```
var bigjoin= game5.join(tweets_consolidated)

var bigjoin2=game6.join(tweets_consolidated)
```

These two RDDs would now allow us to get the final results. The first step in doing this was to convert the timestamps into unix_timestamp, using the following code:

```
import java.util.Date

import java.text.SimpleDateFormat

val formatter = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.S")

var bigjoinunix =.map(x=> (x._1, x._2, formatter.parse(x._3.toString()).getTime()/1000,
formatter.parse(x._4.toString()).getTime()/1000, x._5,
formatter.parse(x._6.toString()).getTime()/1000, x._7))

var bigjoinunix2 =.map(x=> (x._1, x._2, formatter.parse(x._3.toString()).getTime()/1000,
formatter.parse(x._4.toString()).getTime()/1000, x._5,
formatter.parse(x._6.toString()).getTime()/1000, x._7))
```
[5]

With the time stamps converted to unix, it is easy to filter out the tweets based on whether they were created during the game times or not:

```
val during_game= bigjoinunix.filter(x=> {x._6>x._3 && x._6<x._4 })
```

[5] Please note that the supplied code with this submission has been modified since to reduce the total number intermediate variables used. However, they have been left unaltered here to aid with code readability.

```
val during_game2= bigjoinunix2.filter(x=> {x._6 > x._3 && x._6<x._4})
```

The data was therefore in the correct format to answer the questions.

Question 1

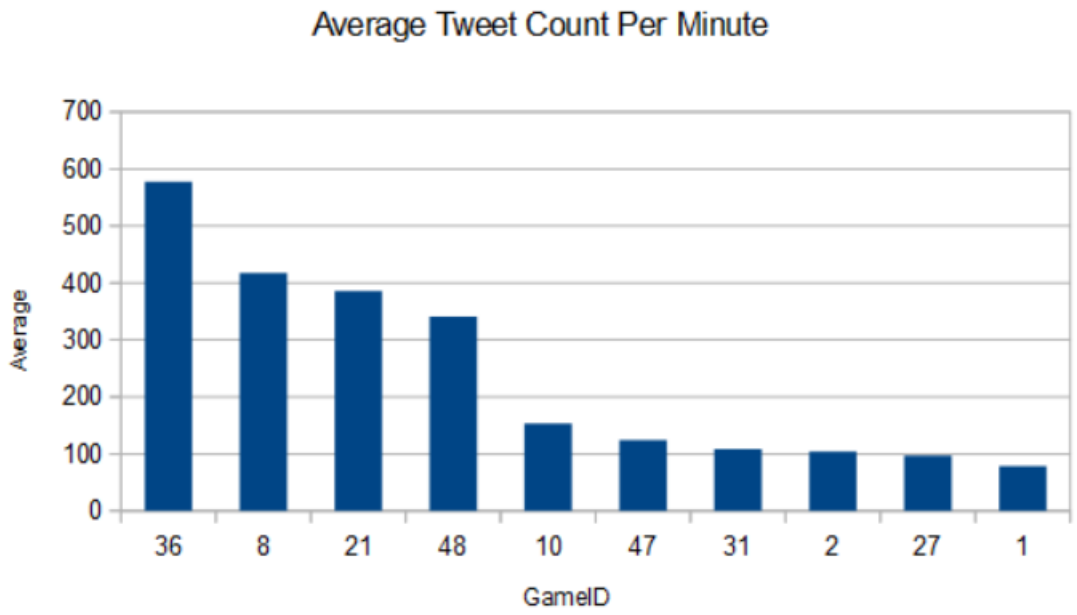
We are working with during_game and during_game2 RDDs for Q1. Since each element of the RDD represents a Tweet that was created between game start and game end, and contained the hashtag of at least one of the two teams playing, counting the number of times game ID appears will give us the total count of tweets for that particular game. We do this separately for both RDDs and then add them up to get the final count. Thus:

```
val precount= during_game.map (x=> (x._2, 1))

val y = precount.reduceByKey((accum, n) => (accum + n))

val precount2= during_game2.map (x=> (x._2, 1))

val y2 = precount2.reduceByKey((accum, n) => (accum + n))

val countjoin= y.join(y2)

val totalcount= countjoin.map(x=> (x._1, (x._2._1+x._2._2)))
```

To calculate the average, calculate the game time, join with total count, divide for average per second and finally multiply by 60 to get the average per minute. Sort and display to get the final answer. Thus:

```
val gameduration= gametimeunix.map(x=> (x._1, x._3-x._2))

val prefinal= totalcount.join(gameduration)

val almostfinal= prefinal.map(x=> (x._1, x._2._1.toDouble/x._2._2.toDouble))

val afinal= almostfinal.map(x=> (x._1, x._2*60))

afinal.sortBy(_._2, false).take(10)
```

The averages are slightly higher in this case (for instance, we get 577 instead of 562 for game 36). This is probably due to some minor variations in the way the solution was implemented and the way the timestamps were converted. The overall solution remains the same, however, and the analysis for Q1 Hive implementation stands for Q1 Scala implementation.



Question 2

The solution to Q2 is similar to Q1 and builds on during_game and during_game2 RDDs that have been prepared in the data prep section. Instead of just counting the occurrence of Game ID in the two RDDs, this time we need to filter out multiple tweets by a single user during a game. One way to do this is to pair GameID and Tweet User ID as a key with 1, add them up. Then you discard the counts and map GameID as question 1 to get the final answer. So:

```
during_game.map (x=> ((x._2, x._7), 1)).reduceByKey((accum, n) => (accum + n))
```

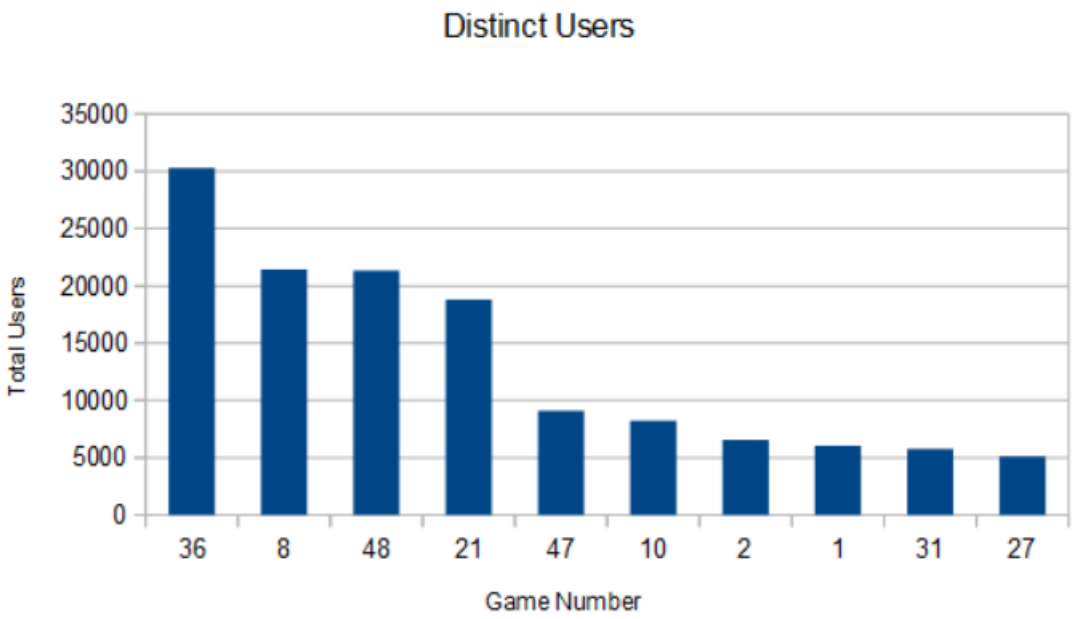
```
during_game2.map (x=> ((x._2, x._7), 1)).reduceByKey((accum, n) => (accum + n))
```

And then:

```
y.map (x=> (x._1._1, 1)).reduceByKey((accum, n) => (accum + n))
```

```
y2.map (x=> (x._1._1, 1)).reduceByKey((accum, n) => (accum + n))
```

After this step it is a simple step of joining the two RDDs and adding up their accumulated values to get the total number of distinct users per game, same as was done to get the answer for question 1. Again the individual counts for each game is slightly greater than the solution achieved in Hive, but the overall analysis remains the same because the structure remains the same.



Question 3

The first step here is to count the total number of games that each team played. This can be done with the game RDD alone. Pair FC1 and FC2 from the game RDD with 1 into two different RDDs. Reduce both RDDs by key, join and add to get the total count for each team. So:

```
gameteams.map (x=>(x._2, 1)).reduceByKey((accum, n) => (accum + n))
```

And then:

```
.join(test2).map(x=> (x._1, x._2._1+x._2._2))
```

This gives the same answer as the solution for this question in Hive/Impala. Most teams played five games each. Barcelona and Sevilla played no games. It makes sense then not restrict the averages to just the top three teams at this point. Instead we will count the total tweets for each team, calculate their averages and order them by top 10 to get the final answers.

Like Q1 and Q2, we build on during_game and during_game2 RDDs. To get the total count for each team, pair the FC number with 1 for both RDDs and add the values up. So:

```
during_game.map (x=> (x._1, 1)).reduceByKey((accum, n) => (accum + n))
```

```
during_game2.map (x=> (x._1, 1)).reduceByKey((accum, n) => (accum + n))
```

Join the two RDDs, add up and get total count. The final step involves joining the total counts RDD with the game count RDD. Dividing gives the average tweets per game for each time. So:

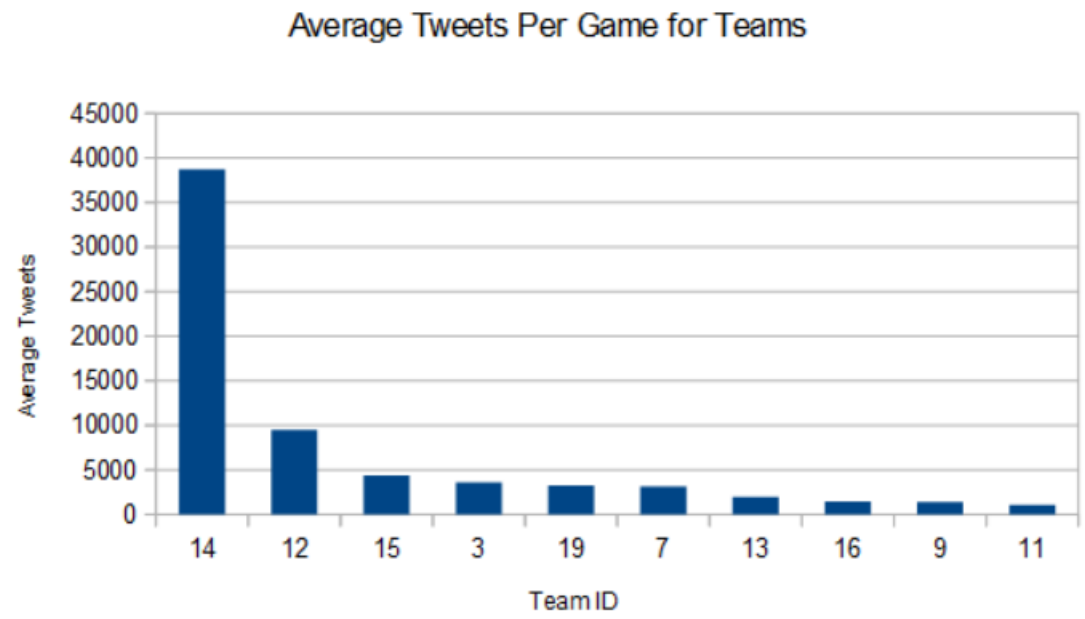
```
val almostfinal= totalcount.join(totalgames)
```

```
val afinal= almostfinal.map(x=> (x._1, x._2._1/x._2._2))
```

The accompanying graph shows the top 10 teams when ranked by average number of tweets per match. It is pertinent to note that the averages here are significantly different to the solution

implemented in Hive/Impala. Manchester United remains the most popular team by far, followed by Liverpool (9491 tweets). Arsenal (ID 3 with 3594 average tweets) and Newcastle (ID 15 with 4370) swap places. Other teams have also swapped places, though the teams generally maintain their positions.

It is not exactly clear why the discrepancy between the two solutions is occurring. The best guess at this point for this discrepancy is that the Scala solution is counting tweets by unique users, while the Hive implementation is looking at individual tweets.

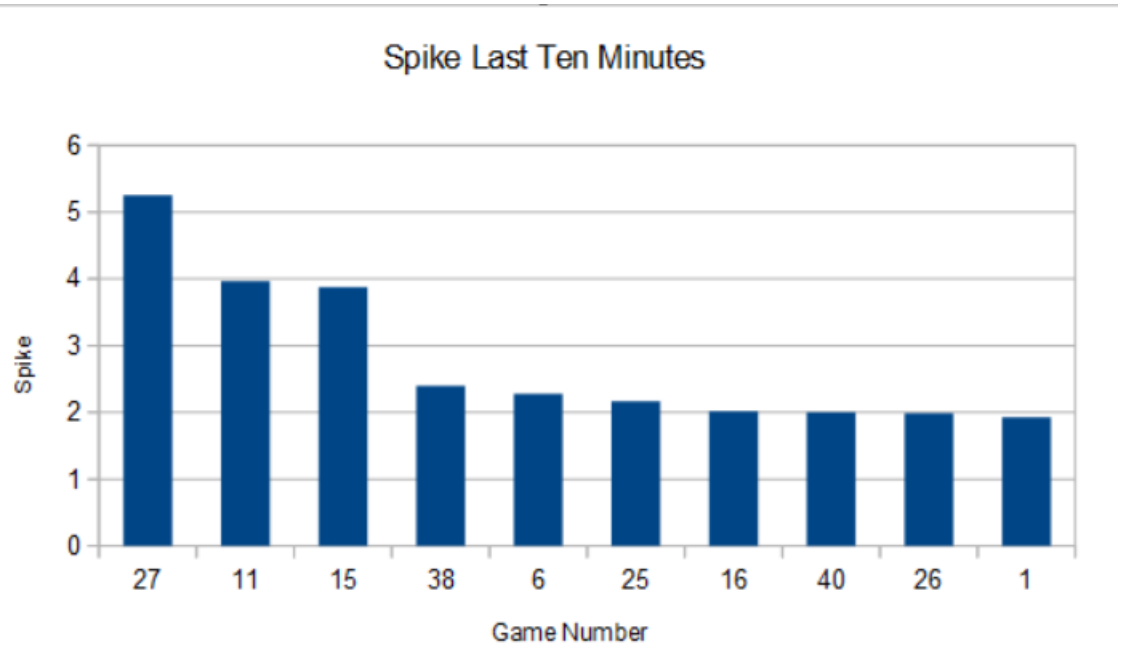


Question 4

The solution builds on during_game and during_game2 RDDs as before, but the preceding RDDs are mapped and then grouped by TweetID to get rid of repetitions:

```
.map(x=> (x._5, (x._1, x._2, x._3, x._4, x._6, x._7))).groupByKey().map(x=>(x._1,
x._2.toList)).flatMap{case(key, list) => list.map(x => (key, x._1, x._2, x._3, x._4, x._5, x._6))}
```

Here on, the solutiom is like the Hive implementation. Divide the analysis into two time periods: (officialend-10minutes to officialend) and then (officialstart to officialend-10minutes). Calculate the total number of tweets for each game for both durations. Calculate averages for the two. Then it is a matter of dividing the averages calculated for the last 10 minutes by the averages for the preceding time interval. Sorting gives the answer, which is similar to the graph generated in the Hive section.



Question 4 Alternative

The alternative solution for Question 4 implemented in hive was attempted in Scala as well. A similar approach as Hive was used, by generating RDDs for each of the last 10 minutes and calculating the number of tweets during that point. Even though the counts were calculated and the RDDs unioned (instead of join) successfully, the final step was abandoned due to the complexity of unpacking the final RDD for each game as well as a quick calculation showing that the results were different to that in the Hive implementation. That said, given more time, further exploration into the solution may yield the required results. The incomplete solution is provided with the submission.

Question 5

Similar to the implementation for Q5 in Hive/Impala, we need a tweet_consolidated RDD where non-official hashtags have not been filtered out. We have to keep tweets with official hashtags around because this is the only way to join the tweets_consolidated table with the game RDD (using FC1 and FC2). Filter tweets to keep them within game start and end times, like the solutions for other questions.

We can now filter out the official hashtags:

```
map(x=>(x._1, x._2._2._1)).filter(x=>{x._2.toInt>22})
```

Sum up all the non-official hashtags that are left behind and sort to get the most popular non-official hashtags that were used by fans during games:

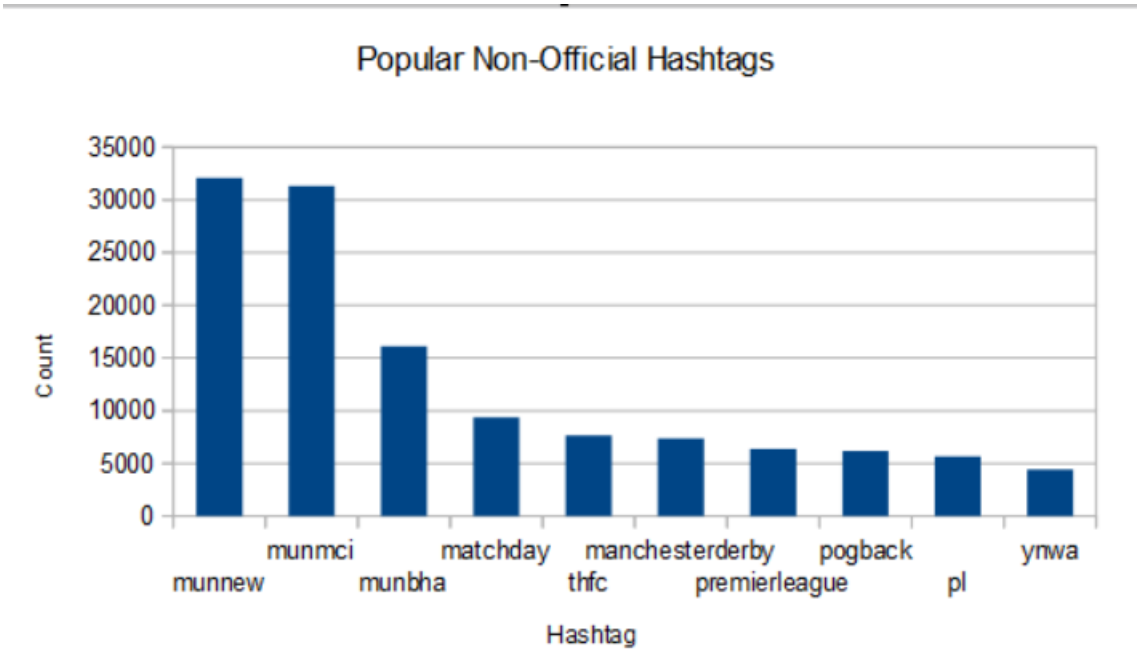
```
.map(x=>(x._2, (1))).reduceByKey((accum, n) => (accum + n))
```

Read in the hashtags table from HDFS, join with the summed results to get the names that are associated with each Hashtag ID (the Hashtags Table contains uncleaned data and one of the hashtag values had to be filtered out for the join operation to work):

```
prefinal2.join(hashtag4).map(x=> (x._1, x._2._1, x._2._2))
```

Please note that the results here are different because the implementation of solution in Scala restricts the analysis to only tweets that also contain official hashtags. The Hive solution, on the other hand, looks at all tweets that were created during the games.

The Manchester United-Newcastle (munnew) and Manchester United-Manchester City (munmci) matches were the most popular nonofficial hashtags. These hashtags continue to indicate that Manchester United is the most popular team in the Premier League on Twitter. Other most popular unofficial hashtags included matchday, premierleague, pogback and pl.



Question 6

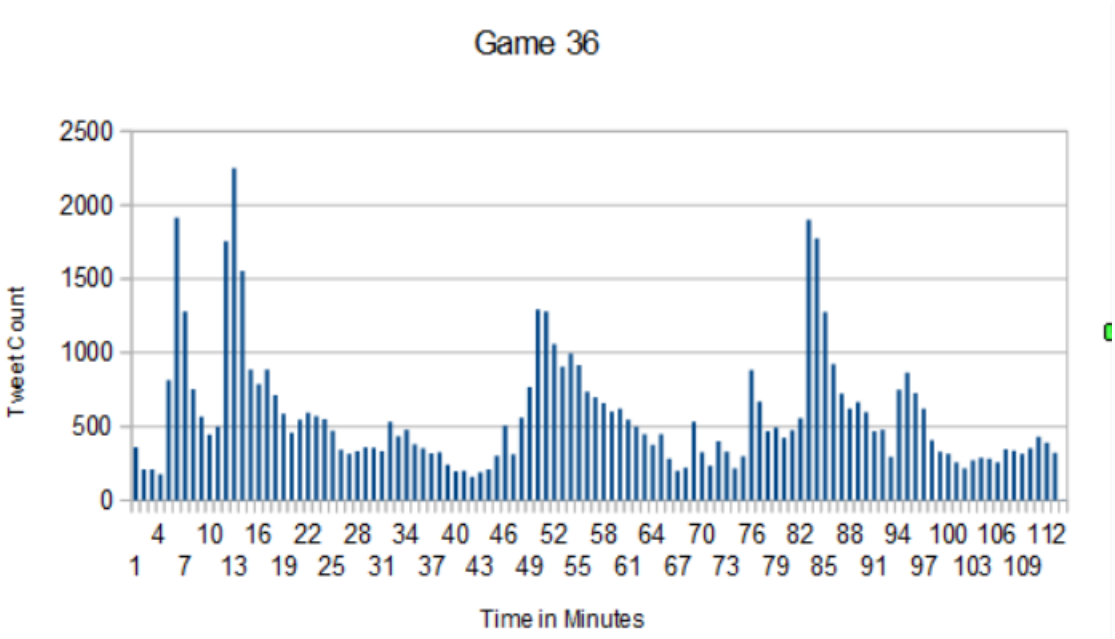
We continue from the tweet_consolidated RDD described in the data preparation section. Game 36, the most popular game, was chosen for the plot. This would also help check the correctness of the graph, as a quick visual comparison should tell if the two graphs are similar or not. The tweets were filtered out based on the game start and end times. So:

```
.filter(x=>x._3>=formatter.parse(starttime).getTime().toLong/1000 &&
x._3<=formatter.parse(endtime).getTime().toLong/1000)
```

The tweets were then again filtered so that they only contained hashtag 3 or 14, the two teams that played in game 36. The time duration was then collapsed down to minutes:

```
.map(x=> (x._3/60.toInt, 1))
```

These key-value pairs were then added up to get the total number of tweets at each minute. The graph below is very similar to the graph generated in Hive and the analysis for that graph holds here as well.



Project Extension

This project can be extended in several ways. The existing analysis already indicates that goals have a significant impact on how Twitter users respond and exploring how this relates to the rest of the data may yield interesting results.

Since the data is being analyzed to inform security decisions, it may also be pertinent to get the actual Tweets in for analysis using Machine Learning and other tools to try and judge tweets based on their content than just numbers. This line of exploration may have some privacy implications and concerns that will have to be looked into, even though Twitter data is generally publicly accessible.

It would also be useful to find information pertaining to security incidents around the time the games were played and correlate them with the game results and Twitter activity. Of particular interest are the times leading up to the start of a game and the times following the end of a game.

References

1. DeZyre. (2016). *Scala vs. Python for Apache Spark*. [online] Available at: https://www.dezyre.com/article/scala-vs-python-for-apache-spark/213 [Accessed 12 May 2018].
2. Prakash, C. (2016). *Apache Spark : RDD vs DataFrame vs Dataset*. [online] LinkedIn. Available at: https://www.linkedin.com/pulse/apache-spark-rdd-vs-dataframe-dataset-chandan-prakash [Accessed 12 May 2018].
3. Stack Overflow. (2016). *how to merge two RDD to one RDD*. [online] Available at: https://stackoverflow.com/questions/41120341/how-to-merge-two-rdd-to-one-rdd [Accessed 12 Apr. 2018].
4. Itversity, T. (2016). *Sqoop Import into Hadoop – Cloudera QuickStart VM or Single Node lab*. [online] IT Versity. Available at: http://itversity.com/topic/sqoop-import-data-ingestion-into-hadoop-cloudera-vm/ [Accessed 12 Apr. 2018].
5. Pandya, H. (2016). *Sqoop: Import Data From MySQL to Hive - DZone Big Data*. [online] dzone.com. Available at: https://dzone.com/articles/sqoop-import-data-from-mysql-tohive [Accessed 12 Apr. 2018].
6. Back To Bazics. (2015). *Apache Spark reduceByKey Example - Back To Bazics*. [online] Available at: http://backtobazics.com/big-data/spark/apache-spark-reducebykey-example/ [Accessed 13 Apr. 2018].
7. Preiss, J 2018, Big Data Tools & Techniques, lecture notes, BDTT, University of Salford, delivered January 2018 to May 2018.