# **Graph Database and its Application: Twitter Graph Search**

*Project Report*

**Delhi Technological University**

Amanjeet Singh Bhatia(2K11/CO/020)

Aviral Takkar(2K11/CO/031)

Ayush Choudhary(2K11/CO/032)

Ayush Ravi Rai(2K11/CO/033)

# Abstract

We have made graph search web interface for the Twitter social network using neo4j and its existing libraries for python. We have chosen python to implement our project in. The main idea is that the user will enter a query, much like the one as follows

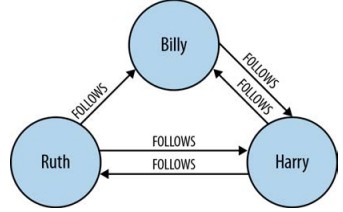
My friends living in India who have followers less than 1000

And we would parse this query to generate a relevant result from the database that would have been created once the user had logged into his Twitter account.

# Introduction

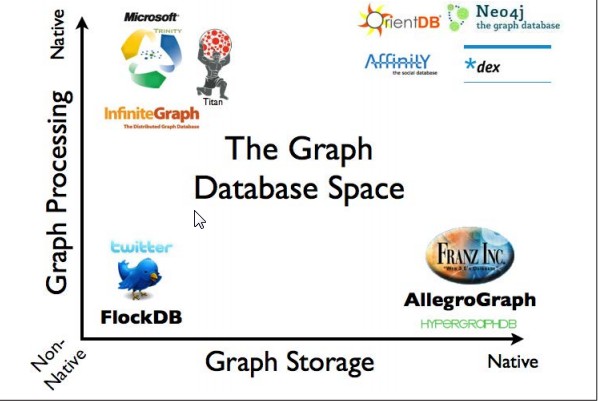
# **Graphs Are Everywhere**

* A graph is a collection of vertices and edges—or, a set of nodes and the relationships that connect them.
* Graphs represent entities as nodes and the ways in which those entities relate to the world as relationships.
* For data of any significant size or value, graph databases are the best way to represent and query connected data. Connected data is data whose interpretation and value requires us first to understand the ways in which its constituent elements are related.



# GRAPH DATABASES

A **graph database** is a [database](http://en.wikipedia.org/wiki/Database) that uses [graph structures](http://en.wikipedia.org/wiki/Graph_(data_structure)) with nodes, edges, and properties to represent and store data. A graph database is any [storage system](http://en.wikipedia.org/wiki/Storage_system) that provides index-free adjacency. This means that every element contains a direct [pointer](http://en.wikipedia.org/wiki/Pointer_(computer_programming)) to its adjacent elements and no index [lookups](http://en.wikipedia.org/wiki/Lookup) are necessary. General graph databases that can store any graph are distinct from specialized graph databases such as [triplestores](http://en.wikipedia.org/wiki/Triplestore" \o "Triplestore) and [network databases](http://en.wikipedia.org/wiki/Network_database_model).



# ADVANTAGES AND CHARACTERSTICS

* **The sheer performance increase when dealing with connected data versus relational databases and NOSQL stores**. In contrast to relational databases, where join-intensive query performance deteriorates as the dataset gets bigger, with a graph database performance tends to remain relatively constant, even as the dataset grows. This is because queries are localized to a portion of the graph, hence, the execution time for each query is proportional only to the size of the part of the graph traversed to satisfy that query, rather than the size of the overall graph.
* **Flexibility.** We can add new kinds of relationships, new nodes, and new sub-graphs to an existing structure without disturbing existing queries and application functionality.
* **Agility.** The schema-free nature of the graph data model, coupled with the testable nature of a graph database’s application programming interface (API) and query language, empower us to evolve an application in a controlled manner.
* **Complexity.** The increasing interconnectivity of data as well as systems has led to denser data sets that cannot be scaled by using relational databases.

# NEO4j

Neo4j is an open-source [**graph database**](http://www.neo4j.org/learn/graphdatabase) supported by [**Neo Technology**](http://neotechnology.com/).

Neo4j stores data in nodes connected by directed, typed relationships with properties on both, also known as a [**Property Graph**](http://www.neo4j.org/learn/graphdatabase).

**Main Features:**

* *Intuitive*, using a graph model for data representation.
* *Reliable*, with full ACID transactions.
* *Durable and fast*, using a custom disk-based, native storage engine.
* *Massively scalable*, up to several billion nodes/relationships/properties.
* *Highly-available*, when distributed across multiple machines.
* *Expressive*, with a powerful, human readable **graph query language (Cypher).**
* *Fast*, with a powerful traversal framework for high-speed graph queries.
* *Embeddable*, with a few small jars.
* *simple*, accessible by a convenient **REST interface** or an object-oriented Java **API**

# CYPHER

Cypher is a declarative graph query language for the [**graph database**](http://en.wikipedia.org/wiki/Graph_database)**,**[**Neo4j**](http://en.wikipedia.org/wiki/Neo4j) that allows for expressive and efficient querying and updating of the graph store. Cypher is a relatively simple but still very powerful language. Very complicated database queries can easily be expressed through Cypher. This allows you to focus on your domain instead of getting lost in database access.

Cypher is designed to be a humane query language, suitable for both developers and (importantly, we think) operations professionals. Our guiding goal is to make the simple things easy, and the complex things possible. Its constructs are based on English prose and neat iconography which helps to make queries more self-explanatory.

**Structure**

Cypher borrows it structure from SQL — queries are built up using various clauses.

Clauses are chained together, and they feed intermediate result sets between each other. For example, the matching identifiers from one MATCH clause will be the context that the next clause exists in.

The query language is comprised of several distinct clauses.

Here are a few clauses used to read from the graph:

* **MATCH:** The graph pattern to match. This is the most common way to get data from the graph.
* **WHERE:** Not a clause in its own right, but rather part of **MATCH**, **OPTIONAL** **MATCH** and **WITH**. Adds constraints to a pattern, or filters the intermediate result passing through**WITH**.
* **RETURN:** What to return.

Example:

MATCH (node {name: 'AyushRaviRai'})-[:follows]->()<-[:follows]-(n)

RETURN node.name , n.name

And here are examples of clauses that are used to update the graph:

* CREATE (and DELETE): Create (and delete) nodes and relationships.
* SET (and REMOVE): Set values to properties and add labels on nodes using SET and use REMOVE to remove them.
* MERGE: Match existing or create new nodes and patterns. This is especially useful together with uniqueness constraints.

Method

Our project is divided into 3 main parts:

1. *Database Creation*
2. *Parsing Query & Cypher Generation*
3. *Cypher Execution & Result Display*

# CREATING THE DATABASE

The user is first asked to log into his Twitter account using his Twitter Login ID and Password. This information is then used to fetch data about the user from Twitter. For this purpose, we have made use of the Twitter API. We obtain the following data about the user from Twitter

* Friends count
* Followers count
* Screen name(name that appears on our twitter profile)
* Location
* Followers (who follows user)

Using this data, we now proceed to create the database in Neo4j. Every other user following the current user is related to him by the relation ‘FOLLOWS’.

We further obtain information about the users followers in a recursive manner to further build the database.

// no idea what to write

We have restricted the number of levels in this recursion to two, due to our limited computational capabilities.

// please suggest chd

# PARSING QUERY & CYPHER GENERATION

Our next task is to parse the query entered by the user and transform it into a Cypher query.

For this purpose, we have created a grammar, and made use of libraries available in Python to perform syntactic analysis.

Parsing is done in python using LEX/YACC parser libraries. The following two source files have been written for the same purpose

* Parser.py
* Tokens.py

Tokens.py is the file given as input to the LEX program, which generates a lexical analyzers for the patterns specified in the file. Tokens may be simple patterns, or have some corresponding action associated with the patterns to be performed by the lexer once it has recognized the pattern.

Parser.py contains the specification of grammar rules. This file is used by the YACC utility to generate a parser according to the rules specified. The parser uses the lexer as a subroutine to get the next token. Parsing is performed by creating a LALR(1) parsing table for the grammar, and performing bottom up shift reduce parsing.

Finally parser.py has the ability to present us with a parse tree of the given input.

|  |  |
| --- | --- |
| **Start🡪** | *friendsWithCond*  *| friendsWithoutCond*  *| followersWithCond*  *| followersWithoutCond* |
| **friendsWithCond 🡪** | *FRIENDS OF friendsWithCond WHO cond ENDENTITY*  *| FRIENDS OF friendsWithoutCond WHO cond ENDENTITY*  *| FRIENDS OF followersWithCond WHO cond ENDENTITY*  *| FRIENDS OF followersWithoutCond WHO cond ENDENTITY*  *| myFriends WHO cond ENDENTITY* |
| **friendsWithoutCond 🡪** | *FRIENDS OF friendsWithoutCond ENDENTITY*  *| FRIENDS OF followersWithoutCond ENDENTITY*  *| myFriends ENDENTITY* |
| **followersWithCond 🡪** | *FOLLOWERS OF followersWithCond WHO cond ENDENTITY*  *| FOLLOWERS OF followersWithoutCond WHO cond ENDENTITY*  *| FOLLOWERS OF friendsWithCond WHO cond ENDENTITY*  *| FOLLOWERS OF friendsWithoutCond WHO cond ENDENTITY*  *| myFollowers WHO cond ENDENTITY* |
| **followersWithoutCond 🡪** | *FOLLOWERS OF followersWithoutCond ENDENTITY*  *| FOLLOWERS OF friendsWithoutCond ENDENTITY*  *| myFollowers ENDENTITY* |
| **Cond 🡪** | *specificCond AND cond*  *| specificCond* |
| **specificCond 🡪** | *placeCond*  *| followCond*  *| whoFollow* |
| **placeCond 🡪** | *LIVING\_IN\_PLACE* |
| **followCond 🡪** | *HAVING\_FOLLOWERS\_GREATER\_THAN\_NUM*  *| HAVING\_FOLLOWERS\_LESS\_THAN\_NUM* |
| **whoFollow 🡪** | *FOLLOW followersWithoutCond*  *| FOLLOW friendsWithoutCond* |
| **myFriends 🡪** | *MY FRIENDS* |
| **myFollowers 🡪** | MY FOLLOWERS |
|  |  |

# The Grammar used

The query is initially sent to a preprocessor program which basically counts number of “who” in the user query and puts a constant “ENDENTITY” before “who “. This step is done to make the process of generating cypher query for multilevel user query easy as the parser has a lookahead of 1.

Example of multilevel query:

*Friends of friends who live in India who live in USA*

After pre-processing, the query is parsed and a parse tree is formed. A parse tree is a rooted tree that represents the syntactical structure of the string according the previously defined context-free grammar.

The parse tree, along with the start symbol of the grammar is passed to “start” function of the query generator class. For each Non-Terminal in the grammar, we have created a function with name same as the symbol and each function takes arguments the symbol and the parse sub tree rooted at this symbol.

The above procedure generates the Cypher query equivalent to the user submitted input string.

Example of the transformation:

*Input Query : my friends who live in India and having followers less than 1000*

*Cypher Query : (node{screen\_name : username})-[:FOLLOWS]->(n)*

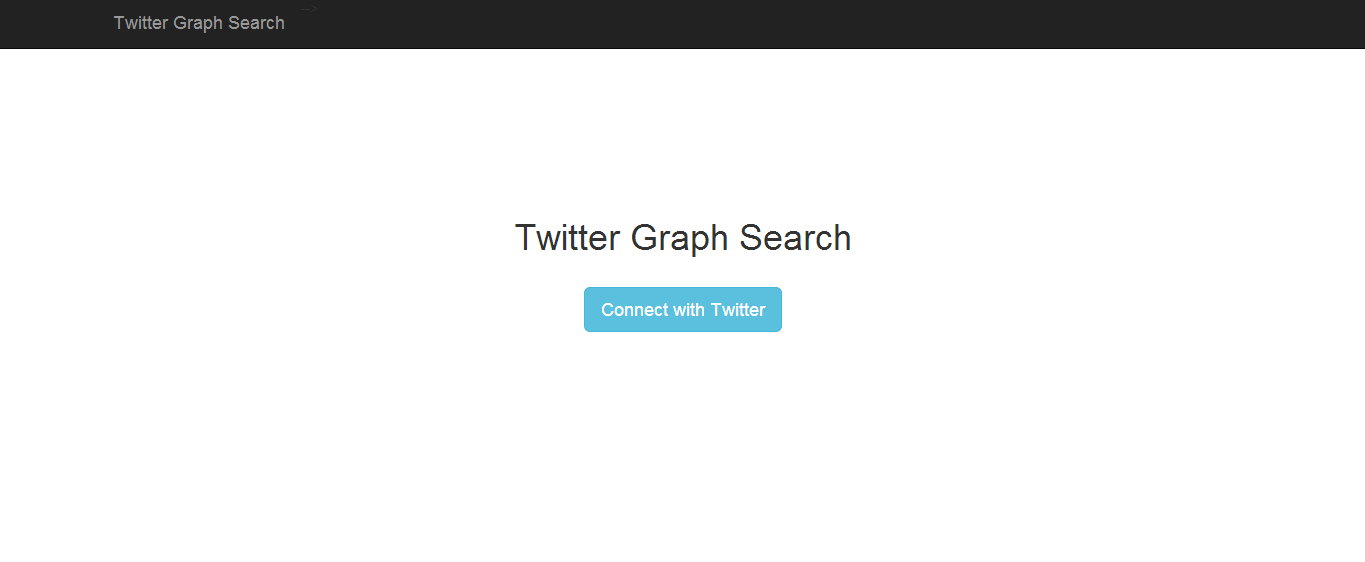
*WHERE n.location = “India” AND n.followers\_count < 1000*

*Return n*

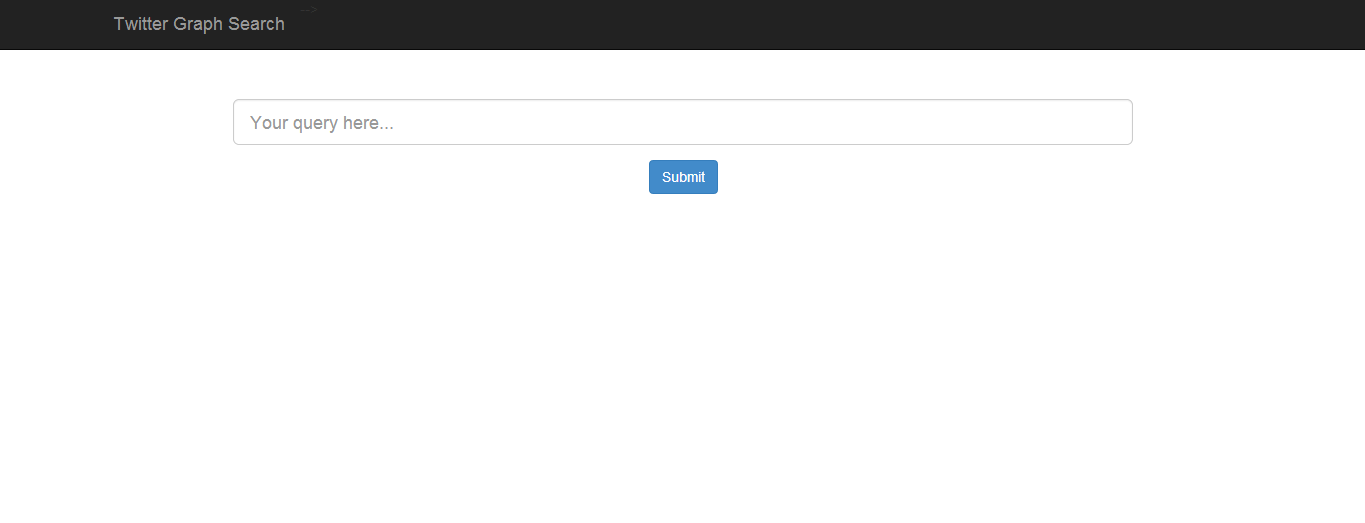
# Results

The next step involves running this query in Neo4j and displaying the results obtained. py2neo is library available in python to provide wrappers for the Neo4j database from python. The Django framework is used to publish the results obtained on the web page.

The home page of the website looks as follows



Once the user clicks “Connect with twitter”, a new page is displayed



Query Execution

The generated Cypher query is passed to a procedure in the file “views.py”.

Views.py script initially connects to the Neo4j Database, and then calls the parser to parse the query.

The “cypherQuery” variable is used to hold the generated Cypher query.

The query is now executed on the database using the py2neo wrappers as sown below.

*Print request.POST[‘userQuery’].lower()*

*cypherQuery = parser.parseQuery(request.POST[‘userQuery’].lower(),request.session [‘current\_user’])*

*graph\_db = getNeo4jService()*

*l = graph\_db.query(q = cypherQuery)*

The result of the executed query is stored in the variable “L” which has now become a dictionary containing all the IDs returned from the database

Displaying the result

The obtained information must now be displayed on the web page.

As explained above that the results from the query is stored in the variable “L”, so now we have to traverse the list/dictionary formed by “L”.

We run a while loop over the list “L” , for each item in the list we display that item on the webpage using Django by defining a series of “div” layers.

Also there may be the case where the result of the query is empty and there are no results to display.

If the results are not empty then following details are displayed on the screen:

Screen\_name(name used by the user)

Image of the user that he/she has set

Friends\_count

Followers\_count

Location

The following code snippet performs the above operations

*for record in l:*

*r = record[0]['data']*

*resp += """<a class='profile center-block' target='\_blank' href='https://twitter.com/""" + r['screen\_name'] +"""'>*

*<div class='img'>*

*<img src='""" + r['profile\_image\_url'] + """' class='img-circle' />*

*</div>*

*<div class='profileData'>*

*<div class='profileDataItem'>""" + r['name'] + """</div>*

*<div class='profileDataItem text-info'>""" + r['screen\_name'] + """</div>*

*<div class='profileDataItem text-muted'>""" + str(r['friends\_count']) + """ friends, """ + str(r['followers\_count']) + """ followers</div>*

*<div class='profileDataItem text-muted'>""" + r['location'] + """</div>*

*</div>*

*</a>"""*

*if resp == "":*

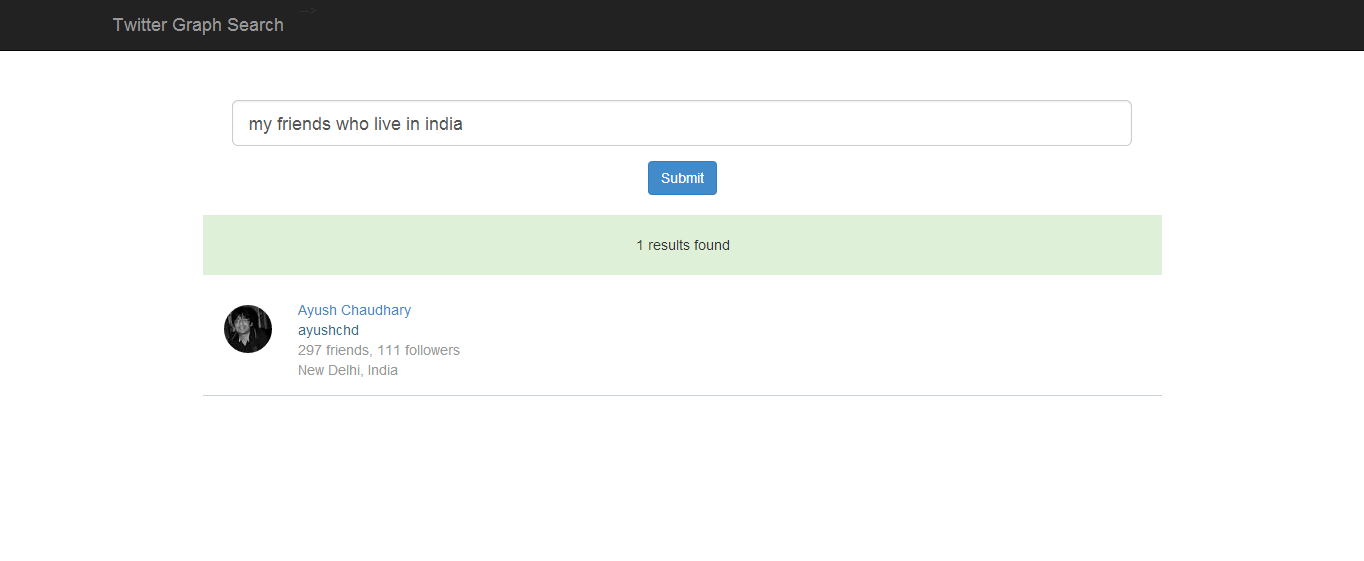
*resp = "<p class='bg-danger pad'>No matching nodes found</p>"*

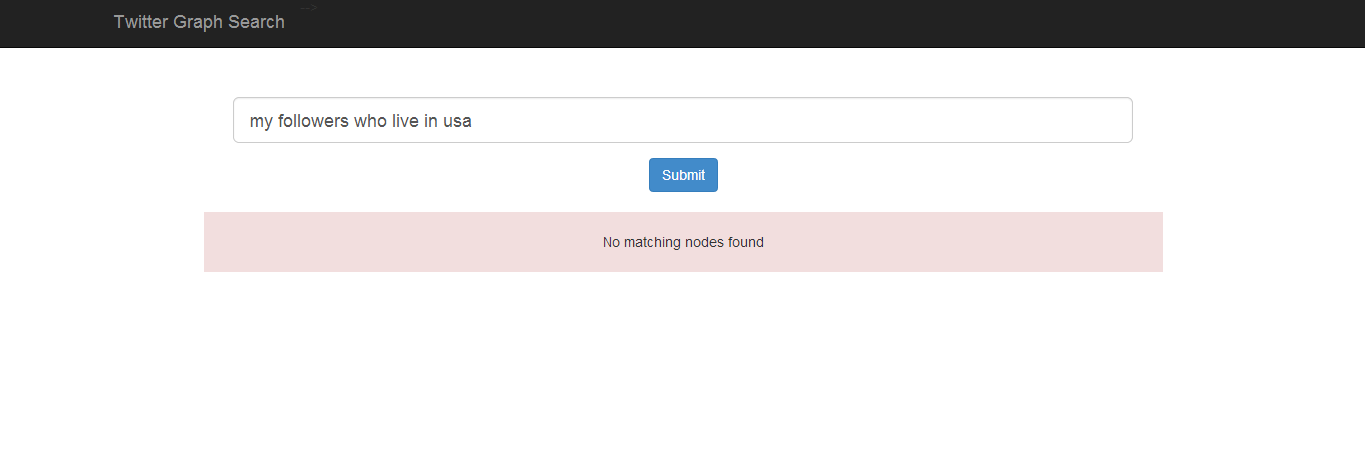
*else:*

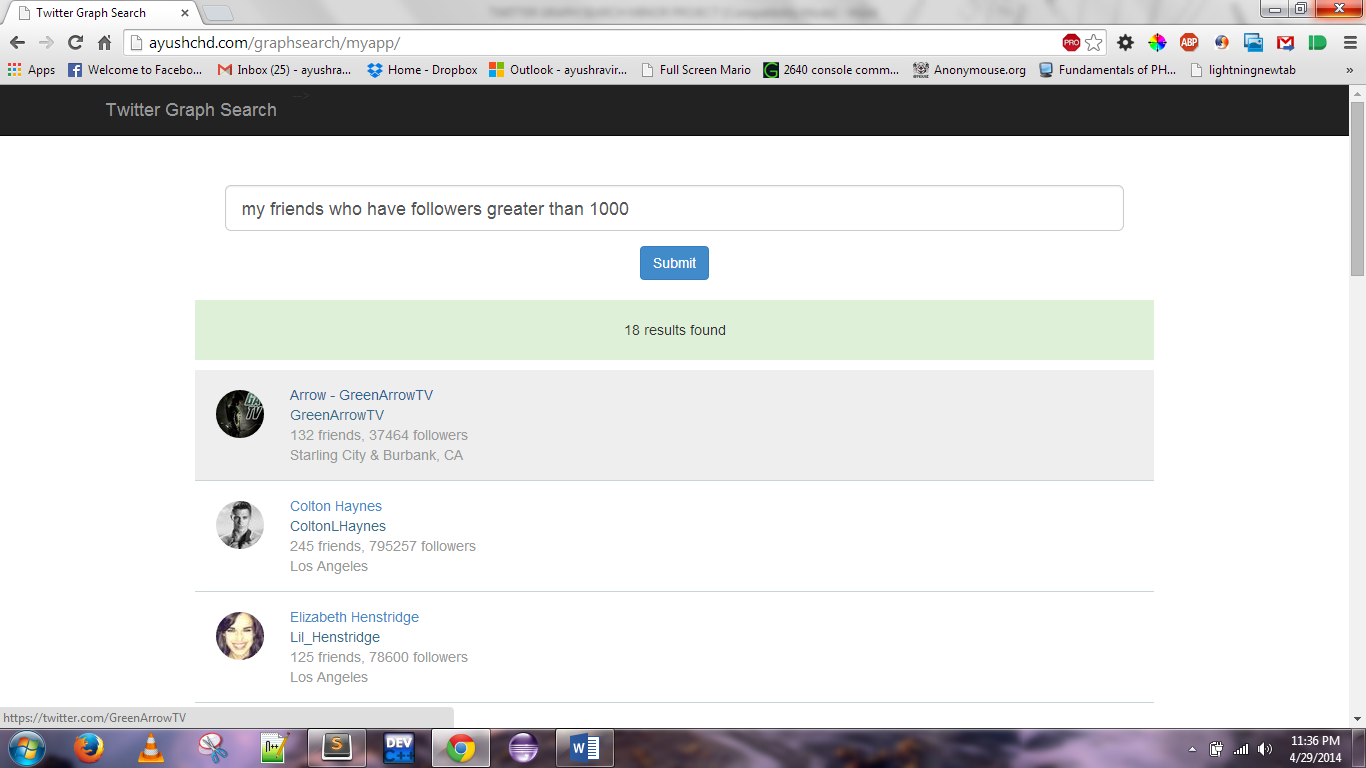
*resp = "<p class='bg-success pad'>" + str(len(l)) + " results found </p>" + resp*

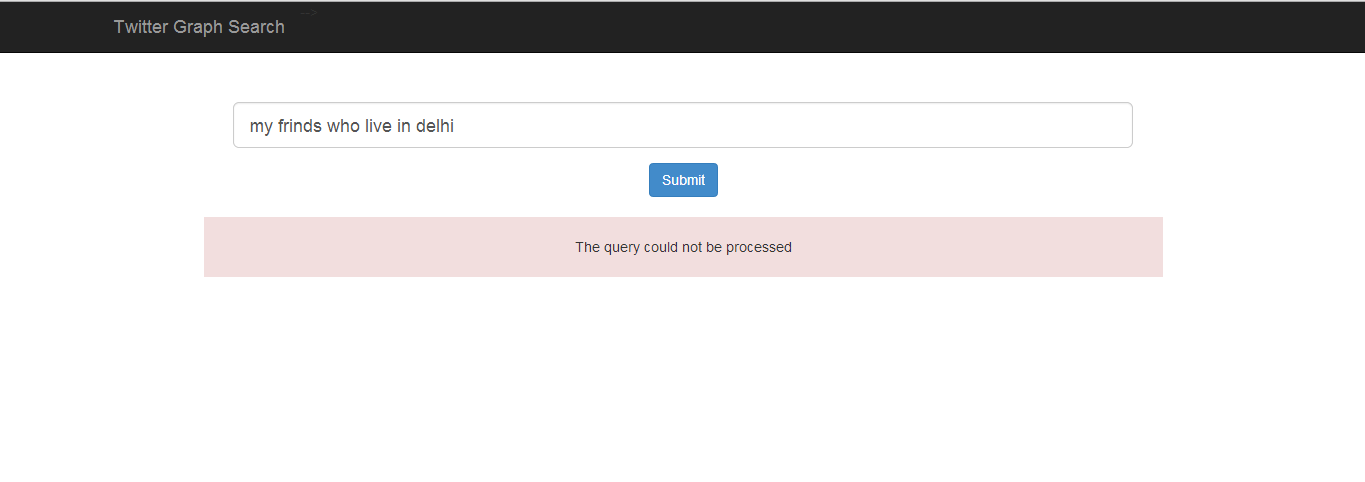
*return HttpResponse(resp)*

Here are some snapshots of the web page:









# References

## Citing Internet Sources