Database Systems
Project Documentation

12/05/2017

John Nguyen
Gavithishan Ravechandran
Khanh Bui Trong

Chapter 1

Project Iteration 1

1.1 Introduction

1.1.1 Project Aim

The aim of this project to provide an interactive visualisation of tweet data collected prior to the american elections in 2016 from former presidential candidates Hillary Clinton and Donald J. Trump. It focuses primarily on the usage *hashtags* and aims to facilitate a user-oriented analysis on these hashtags. This includes providing visualizations of relationships between hashtags regarding the frequency of their single and pairwise occurrences; the development of their usages over time; and their inherent importance[1]. The final form of the project will be a web application that enables the user to interactively query the underlying database.

1.1.2 Requirements

The following list describes the type of requests that the web application must be able to respond to. Although this list is not exhaustive, the user should at least be able to retrieve the following information:

- which hashtags were used, by whom and in which tweets
- how often a single hashtag, or a pair of hashtags, was used
- who used the most hashtags
- which hashtags were used the most
- in which period hashtags (generally) used the most
- how did the usage of a hashtag (or pair) develop over time
- how the usage of a hashtag relates to the amount of times a tweet was retweet or favorited

1.1.3 The Team

This project is being developed by John Nguyen, Gavithishan Ravechandran and Khanh Bui Trong. We are all currently studying a Bachelor of Computer Science at the Freie Universität in Berlin, Germany.

[1] The importance of a hashtag is interpreted as the accumulative impact of the tweets that it belongs to.

1.2 Data Analysis

The data used in this project is a collation of tweet data produced by former american presidential candidates Hillary Clinton and Donald J. Trump prior to the 2016 election. The tweets were created between January and October and consist of the following components:

- `handle`: indicates the user who posted the tweet
- `text`: the actual tweet content
- `is retweet`: indicates if the tweet is a retweet
- `original author`: indicates the original author of the tweet (applicable only to retweets)
- `time`: indicates the date and time the tweet was created
- `in reply to screen name`: if the tweet is a reply, this indicates who the reply is directed to
- `is quote status`: indicates if the tweet was quoted tweet
- `retweet count`: indicates the number of times the tweet was retweeted (by other users)
- `favorite count`: indicates how many times the tweet was favorited (by other users)
- `source url`: indicates from which platform (device) the tweet was published
- `truncated`: indicates if the tweet content was truncated

It is important note that not all components are necessarily relevant for the purposes of this project. For this reason the components `in reply to screen name`, `is quote status`, `source url` and `truncated` will be excluded from the data model (section 3).

For the remaining components we will briefly describe their importance: `handle, text` and `time` are core components that identify and describe particular tweet and the hashtags contained within; `favorite count` and `retweet count` describes the impact of the tweet; and finally `is retweet` and `original author` may provide additional contextual information about the tweet.

1.3 Data Model

1.3.1 The Entity-Relationship Model

The Entity-Relationship model is depicted in Figure 1.1. It consists of a single tweet-entity with a single value attribute for each component with the exception of `ID` and `hashtags`. Hashtags contained within the tweet text are represented as a multivalued attribute.

The key `ID` attribute is a composed from `handle` and `time`. The reason for this is because no single simple attribute forms a key since it can not be guaranteed that the value of this attribute will be distinct among multiple tweets objects. In other words, a pair of tweets may share a common value for each of the attributes. In order to uniquely identify a tweet, one must consider a combination of simple attributes. On the assumption that a single user may not publish two distinct tweets simultaneously, the paired value of `handle` and `time` forms a key[2].

A hashtag is not represented as an entity (nor their relationship to a tweet through a corresponding relationship type) because it increases unnecessary complexity to the model. A hashtag entity would have only a single attribute (to specify the hashtag name) and participate in a single relationship with the tweet entity. This is more simply expressed using the multivalued approach as in Figure 1.1. One concern with the current model, however, is its capacity to express pairwise relationships between distinct hashtags. As we shall see in the next section, our ER-model does not limit us to fulfill this requirement, since its translation to the relational model expresses a hashtag as distinct relation[3].

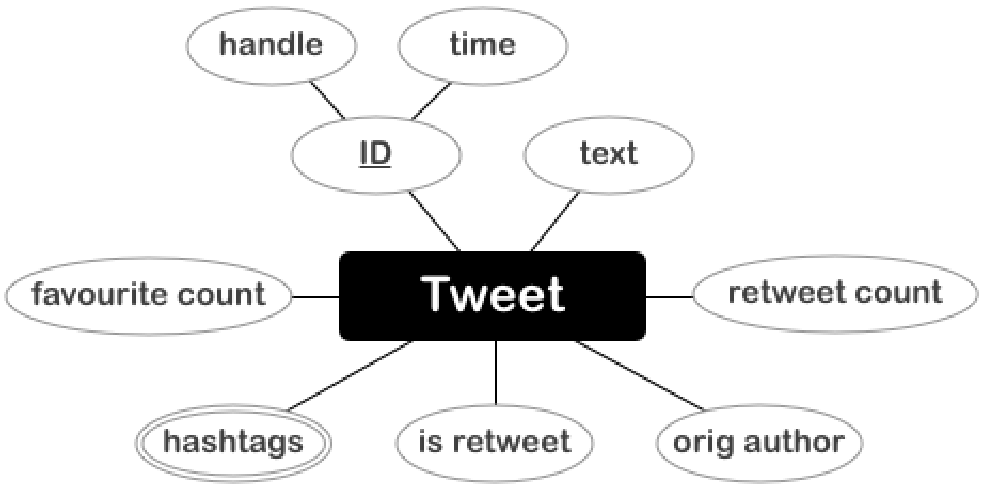

Figure 1.1: The Entity-Relationship model of the dataset

[2]Although the time attribute is only precise to the second, the key remains valid in our (static) dataset.

[3]When discussing relational models, we will use the terms *relation* and *table* interchangeably.

1.3.2 The Relational Model

The ER-model translates[4] into the relational model as illustrated in Figure 1.2. It consists of two relations, one corresponding to the a tweet and the other to a hashtag. The primary key of the Tweet-relation is the tuple `(handle, time)`, as defined in section 1.3.1.

The Hashtag-relation contains three attributes, the first of which, `tag`, stores the actual hashtag. The remaining two attributes `tweetHandle` and `tweetTime` form a foreign key referencing the primary key of the Tweet-relation. All three attributes then form the primary key of Hashtag-relation.

The following list specifies the domains and constraints of each attribute:

handle: The domain is all alphanumeric (including underscores '_') strings with maximum 15 characters[5]. This attribute must have a value, i.e, it is constrained be `NOT NULL`, because it is part of the primary key. These constraints are defined by Twitter[6].

text: The domain is all strings containing up to 140 characters. This attribute is constrained to be `NOT NULL`. These constrainst are also defined by Twitter[7].

time: The domain is all timestamps in the format `YYYY-MM-DD HH:MM:SS`. This attribute is constrained to be `NOT NULL` because it is part of the primary key.

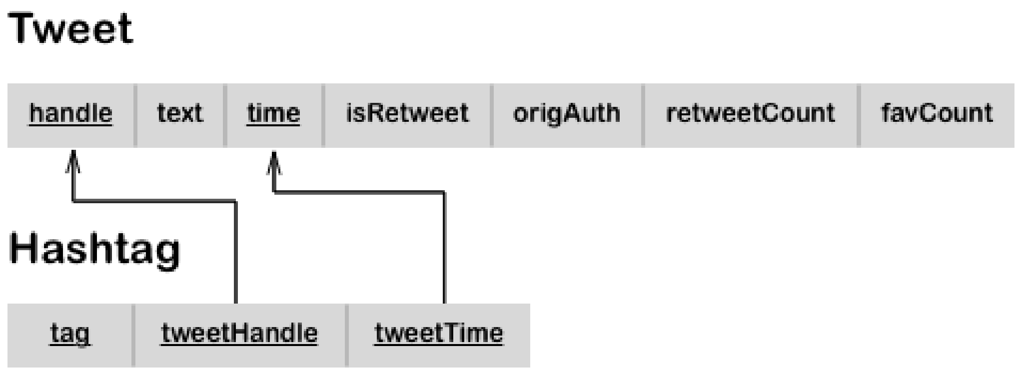

Figure 1.2: The relational model of the dataset

[4]The algorithm used is from: Fundamentals of Database Systems (7ED), Elmasri & Navathe, 2016, ch. 9

[5]More specifically, this is any string matching the regular expression `[a-zA-Z0-9_]{1,15}`.

[6]See `https://support.twitter.com/articles/101299#unavailable`

[7]See `https://support.twitter.com/articles/15367`

isRetweet: The domain is the set of boolean values $\{true, false\}$. This attribute is naturally constrained to be `NOT NULL`.

origAuthor: The domain is the same as the `handle` attribute since it refers to another twitter user. The attribute will contain a value only if `isRetweet` set to true.

retweetCount & favCount: The domain is the set of positive whole numbers and the attribute is constrainted to be `NOT NULL`, since a tweet must be retweeted 0 or more times.

tag: The domain is the set of all valid hashtags; strings containing alphanumeric characters (including underscores) but beginning with the `#` symbol and otherwise containing at least one alphabetic letter. Because the entire tag is part of the tweet text, it may not exceed 140 characters in length[8]. Due to this attribute being part of the primary key for the Hashtag-relation, it is constrained to be `NOT NULL`.

1.3.3 Important Considerations

The Hashtag-relation does not represent single hashtag objects, but rather the occurrences of hashtags. Thus each tuple in the Hashtag-relation pairs together the hashtag used, and the tweet it was used in.

Since the primary key of the Hashtag-relation is the set of the `tag`, `tweetHandle` and `tweetTime` attributes, each tuple within the relation must have a unique combination of values for these attributes. As a consequence, no single hashtag may occur multiple times in a single tweet because an attempt to insert two occurrences of the same hashtag from the same tweet leads to two tuples with equal attriubte values. This is a violation of the primary key constraint. To avoid this violation it is required that **the occurrence of a hashtag within a tweet is to be counted at most once**. This requirement does not adversely affect the goals of the projects since the functionality of a hashtag depends solely on its existence rather than its quantity.

[8] This is any string matching the regular expression `#\w*[a-zA-Z]\w*'`.

1.4 Constructing the Database

The database management system used in this project is *PostgreSQL* (version 9.6). It can be installed by the *Homebrew*[9] package manager via the command line.

The PostgreSQL installation includes the command line application `psql`, with which we can execute the SQL statement required to create the database. Entering the command `psql` starts the application. Once inside `psql`, we can execute the following SQL statement:

```
CREATE DATABASE election;
```

This command creates a empty database called `election` based on the default settings. At this point in the project this is sufficient for experimentation purposes, however it may be necessary later to adjust the database settings to specify user access privileges and server information. We can confirm the database was created by entering the command:

`\l`.

This lists all the exisitng databases. To quit `psql` simply enter the command:

`\q`

[9] for macOS see https://brew.sh, or alternatively for linux see http://linuxbrew.sh. For instructions on installing PostgreSQL see `https://gist.github.com/sgnl/609557ebacd3378f3b72`.

Chapter 2

Project Iteration 2

2.1 Database Schema

Section 1.3.2 outlines in detail the database schemas needed to be constructed. We must simply express the domains and constraints with SQL statements. Firstly we define two domains, `handle_type` and `natural_num`, which correspond to the set of all valid twitter handles and the set of all positive whole numbers. Note that PostgreSQL allows the use of *regular expressions* within `CHECK` constraints, which we will utilize to validate the twitter handles. The following SQL statement defines the domains:

```
CREATE DOMAIN HANDLE_TYPE AS VARCHAR(15) CHECK  (VALUE ~ '^\w{1,15}$');
CREATE DOMAIN NATURAL_NUM AS INTEGER NOT NULL CHECK (VALUE >= 0);
```

Next we define the schemas corresponding to the *Tweets* and *Hashtags* relations with the following SQL:

```
CREATE TABLE Tweets (
      handle          HANDLE_TYPE     NOT NULL,
      content         VARCHAR(140)    NOT NULL,
      t_stamp         TIMESTAMP       NOT NULL,
      is_retweet      BOOLEAN         NOT NULL,
      orig_author     HANDLE_TYPE,
      retweet_count   NATURAL_NUM,
      fav_count       NATURAL_NUM,
      PRIMARY KEY (handle, t_stamp)
  );

  CREATE TABLE Hashtags (
        tag           VARCHAR(140)  NOT NULL
        CHECK (tag ~ '^#\w*[a-zA-Z]\w*$'),
        tweet_handle  HANDLE_TYPE   NOT NULL,
        tweet_t_stamp TIMESTAMP     NOT NULL,
        PRIMARY KEY (tag, tweet_handle, tweet_t_stamp),
        FOREIGN KEY (tweet_handle, tweet_t_stamp)
        REFERENCES Tweets(handle, t_stamp)
        ON UPDATE CASCADE ON DELETE CASCADE
    );
```

We note that if a tweet is updated or deleted from the Tweets relation, then any hashtags from the Hashtag relation that references this tweet will also be updated or deleted.

The Python script in listing 2.1 creates the relation schemas by first connecting to the `election` databse and then sequentially executing each of the SQL statements above. Note that before the domains and tables are created, the script first checks to see if they exist, and if so, they are deleted. This allows the program to be run even if the database is not empty and is included as a convenience. The *psycopg2* Python package is used to communicate with PostgreSQL.

Listing 2.1: table_creator.py

```python
#!/usr/bin/python
import psycopg2
from config import db_config

table_commands = (
    """
    DROP TABLE IF EXISTS Tweets CASCADE;
    DROP TABLE IF EXISTS Hashtags;
    DROP DOMAIN IF EXISTS HANDLE_TYPE;
    DROP DOMAIN IF EXISTS NATURAL_NUM;
    """,
    """
    CREATE DOMAIN HANDLE_TYPE AS VARCHAR(15) CHECK (VALUE ~ '^\w{1,15}$');
    CREATE DOMAIN NATURAL_NUM AS INTEGER NOT NULL CHECK (VALUE >= 0);
    """,
    """
    CREATE TABLE Tweets (
        handle          HANDLE_TYPE     NOT NULL,
        content         VARCHAR(140)    NOT NULL,
        t_stamp         TIMESTAMP       NOT NULL,
        is_retweet      BOOLEAN         NOT NULL,
        orig_author     HANDLE_TYPE,
        retweet_count   NATURAL_NUM,
        fav_count       NATURAL_NUM,
        PRIMARY KEY (handle, t_stamp)
    );
    """,
    """
    CREATE TABLE Hashtags (
```

```
        tag           VARCHAR(140)  NOT NULL CHECK (tag ~ '^#\w*[a-zA-Z]\w*$'),
        tweet_handle  HANDLE_TYPE   NOT NULL,
        tweet_t_stamp TIMESTAMP     NOT NULL,
        PRIMARY KEY (tag, tweet_handle, tweet_t_stamp),
        FOREIGN KEY (tweet_handle, tweet_t_stamp)
        REFERENCES Tweets(handle, t_stamp)
        ON UPDATE CASCADE ON DELETE CASCADE
    );
    """
)

def create_tables():

    conn = None

    try:
        # connect to the PostgreSQL server
        print "\nconnecting to DB.."
        # get DB info
        params = db_config()
        conn = psycopg2.connect(**params)
        cur = conn.cursor()
        # execute commmands
        print "creating tables..."
        for command in table_commands:
            cur.execute(command)

        # commit the changes
        conn.commit()
        cur.close()
        print "DONE"

    except (Exception, psycopg2.DatabaseError) as error:
        print "ERROR:", error

    finally:
        if conn is not None:
            conn.close()
```

2.2 Data Cleaning

In order to correctly import the data into the database, we must first prepare and clean the raw data. This involves extracting only the necessary data, formatting the data components into the expected data types as defined by the corresponding relation schema, as well as removing or correcting any other anomalies.

Of all the attributes listed in the schemas of the previous section, only content and t_stamp attributes need to be reformatted.

content: In the raw data file, some tweet entries contain symbols encoded for web usages. These include &, <, and >, and appear in the raw data as &, <, and > respectively. Thus we must replace each occurrence of these encoded symbols with the decoded symbols they represent.

t_stamp: In the raw data file, this element has the form yyyy-mm-ddThh:mm:ss; for example 2017-01-01T00:00:00 represents midnight on the 1st of January 2017. However, the TIMESTAMP data type in PostgreSQL is expected to be in the form yyyy-mm-dd hh:mm:ss; the only difference is absence of the letter T. Therefore we must replace the letter T with a space for every entry of this data element in the raw data.

All the other data elements corresponding to the remaining attributes are already in the appropriate format and thus no additional conversions are required.

The Python script shown in listing 2.2 automates this process by reading the raw data, performing the conversions, and writing only the needed (and cleaned) data into a new file. The raw data has been provided in two different file formats: xlsx (excel spreadsheet) and csv (comma separated values). We have chosen to work with the xlsx file because it is better structured compare to the *csv* file. Each line of csv file does not always represent a complete tweet entry; multiline tweets (those with newline characters) are spread across multiple lines within the csv file and thus pose a problem when trying to parse this file.

This problem does not arise when parsing a spreadsheet; all tweet elements are contained within individual cells, including multiline tweets. Thus the program can easily iterate through each row and then through each cell to extract the elements accordingly. Furthermore, the standard Python csv parsing module does not support unicode strings (and therefore also the emojis contained in the tweets), whereas the spreadsheet does. The program utilizes the *openpyxl* package for reading and writing the xlsx file format.

Listing 2.2: data_cleaner.py

```python
#!/usr/bin/python
from openpyxl import Workbook
from openpyxl import load_workbook
from config import data_config

def clean_row(data):
    # convert encoded symbols
    data[1] = data[1].replace("&amp;", "&")
    data[1] = data[1].replace("&lt;", "<")
    data[1] = data[1].replace("&gt;", ">")
    # reformat time stamp
    data[4] = data[4].replace("T"," ")
    # return only needed columns
    return data[:2] + data[4:5] + data[2:4] + data[7:9]

def clean_data():
    # open raw data file & get the worksheet
    input_file = data_config['input_filename']
    print '\nopening file:', input_file, '...'
    input_wb = load_workbook(input_file, read_only = True)
    input_ws = input_wb[data_config['sheet_name']]

    # create new workbook with a new worksheet
    output_wb = Workbook()
    output_ws = output_wb.active
    output_ws.title = data_config['sheet_name']

    print 'cleaning data...'
    # for each row in input sheet
    for row in list(input_ws.rows):
        # extract cell data
        row_data = []
        for cell in row:
            row_data.append(cell.value)
        # clean the data & add to new sheet
        output_ws.append(clean_row(row_data))

    # save wb
    output_wb.save(filename = data_config['clean_filename'])
```

```
    print 'DONE: saved file:', data_config['clean_filename']
```

2.3 Importing the data

The Python script in listing 2.3 automates the insertion of the cleaned data into the database. It first makes a connection to the database, then iterates through each row of the data file (`xlsx` format) and inserts both the tweet data and the hashtag data contained in this row. The structure of the data file corresponds to the `Tweets` table and thus we can simply insert the whole row directly into this table. The hashtag data must be extracted from the current tweet content, which is achieved by using a regular expression to find any embedded hashtags. As noted in section 1.3.3, only distinct hashtags are extracted. Once a list of distinct hashtags is obtained, the program then inserts a new row into the `Hashtags` table for each hashtag with a reference to the current tweet.

Finally, the insertions are committed to the database and the number of insertions is reported back to the user.

Listing 2.3: table_populator.py

```
#!/usr/bin/python
import psycopg2
import re
from openpyxl import load_workbook
from config import db_config
from config import data_config

def populate_tables():

    conn = None
    tweet_count = 0
    hashtag_count = 0

    filename = data_config['clean_filename']
    print "\nopening data file:", filename
    wb = load_workbook(filename, read_only = True)
    ws = wb[data_config['sheet_name']]

    # exclude first row (column names)
```

```
    rows = list(ws.rows)[1:]
    print "Number of tweets to insert:", len(rows)

    try:
        # connect to the PostgreSQL server
        print "connecting to DB.."
        params = db_config()
        conn = psycopg2.connect(**params)
        cur = conn.cursor()

        print "populating tables..."
        for row in rows:
            # extract cell values from row
            row = map(lambda cell: cell.value, row)
            # insert tweet
            sql = "INSERT INTO Tweets VALUES (%s,%s,%s,%s,%s,%s,%s)"
            cur.execute(sql, row)
            tweet_count += 1

            # get hashtags from tweet
            hashtags = extract_hashtags(row)
            sql = "INSERT INTO Hashtags VALUES (%s,%s,%s)"
            # for each tag
            for tag in hashtags:
                # insert tag, handle, timestamp
                cur.execute(sql, (tag, row[0], row[2]))
                hashtag_count += 1

        # commit the changes
        conn.commit()
        cur.close()
        print "DONE:", tweet_count, "tweets and", hashtag_count, "hashtags inserted.\n"

    except (Exception, psycopg2.DatabaseError) as error:
        print "ERROR:", error

    finally:
        if conn is not None:
            conn.close()
```

```python


def extract_hashtags(tweet):
    # extract hashtags
    hashtags = re.findall("#\\w*[a-zA-Z]\\w*", tweet[1])
    # remove duplicate tags
    return list(set(hashtags))
```

2.4 Executing the code

For simplicity, the code presented in the previous three sections have been collated into a sinlge Python script called `dbBuilder.py`. Running this script within terminal from the working directory will clean the raw data, save it to a new file, connect to the database, create the schemas and finally import the cleaned data. It is assumed the database `election` already exists and is configured, however it is not required that the database is empty because the program will drop all schemas before recreating them.