[[1]](#footnote-1)

Aggregation of multiple data sources for integrated equity market analysis

Daniel Serna, Peter Flaming, Brandon de la Houssaye, James Vazquez

***Abstract: In this paper, we present a means of gathering disparate information from multiple, publicly available information sources in a reproducible format for subsequent organization into a relational database schema. The information being gathered (into a database schema) is centered around equity market output (i.e., stock pricing). The authors observe that the equity markets are influenced by a myriad of factors, and it is only once proper empirical analysis is performed on these factors that we can better understand the equity market response (to external stimuli). It is therefore endeavored by the authors to make the information available (within a database structure) so such subsequent analyses may be performed. The authors began the analysis by first considering which publicly available information to gather; then gathering the information; and finally structuring it in a relational database schema. The results and conclusion in this paper show the path to establishing automated means of gathering data from certain sources and the appropriate relational database schema given the nature of the data gathered.***

# INTRODUCTION

T

HE use of stock analysis tools are very abundant in today’s fast moving economy. Many of these tools have integrated views that allow users to observe relationships. Lacking in these tools is the ability to expand these relationships to external data sets outside of the market place, such as political climates, major national events, natural disasters, and the emergence of social media.

We set out to define a database schema that allows for introduction of new data sets when needed. Within the context of data science, the one constant variable is change. Designing a flexible schema that allows for natural growth will enhance the user experience and allow for observational and statistical analysis.

The group tackles these issues, to define a flexible and agile schema that allows for political data, major US events, natural disasters and Twitter tweets.

The team shows how to join data by datetime keys in our relational database schema. The team’s design allows for database reporting and business intelligent (BI) tool plotting.

The data that the team uses as part of the project are dynamic in nature and updated frequently.

# Tutorial

The authors are interested in utilizing the large amount of publicly available social media data to create a predictive model for stock prices. To efficiently correlate these large datasets (such as twitter data), an efficient database schema must be designed that allows for effective data mining while also providing reasonable query performance.

The first step in the analysis is to define data of interest. Of primary concern is external stimuli (to a firm) that would impact/influence (the firm’s) equity market pricing. There are potentially infinite number of variables that could influence equity market pricing, so the authors limit focus on information/data that first meet the following criteria:

* Non-specific to any particular firm (in terms of publicly reported financial results); and
* Publicly available.

The authors concluded the following types of data to be of interest:

* Social media reference(s) on twitter;
* Catastrophic events (e.g., weather);
* Major historical events (e.g., treaty execution); and
* Political landscape (as measured by U.S. President and associated party affiliation).

Once the data of interest is defined, the authors gather the data from information sources such as Wikipedia utilizing a number of tools available, such as code packages from Python or predefined API’s.

Once the data is gathered, the authors construct a relational database schema in an appropriate normalized form. This final step, the relational database schema constructed, is done so in a manner to allow for additional data variables to be included at future points in time. Simply, as the goal is to create a database schema that allows for the understanding of equity markets, and there is acknowledgement by the authors that meaningful variables are not yet known, it is important to allow for such flexibility.

# Data Set

Twitter is currently the 10th most popular website globally with over 300 million active monthly users. Twitter is updated hundreds of millions of times a day with content varying from individual daily life updates to worldwide news and events. These data points can be used for future and real-time predictions of the stock market. The focus of this project is to create a database with a flexible schema, that is highly scalable to manage the millions of tweets that may possibly reflect market-trends of the globalized market in today’s microblogging world. As a result, data comes from [www.twitter.com](http://www.twitter.com).

Another data source utilized is Wikipedia. After extensive searching and researching, the team found these Wikipedia sites were the most frequently updated and formatted in a way that programming languages could easily scrape. In particular, the following websites are utilized (on Wikipedia) to scrape data:

* <https://en.wikipedia.org/wiki/List_of_Presidents_of_the_United_State>.
* <https://en.wikipedia.org/wiki/Timeline_of_United_States_history>
* <https://en.wikipedia.org/wiki/List_of_natural_disasters_in_the_United_States>

In addition to Wikipedia, the authors also gather information from <https://api.iextrading.com> with respect to individual stock prices as well as overall indexes.

The authors also gather information from twitter utilizing the twitter api - <https://api.twitter.com>.

All information gathered/scraped is publicly available to all market participants.

# Methods and Experiments

As discussed in the preceding sections, the authors first identify key data of interest and gather this data from publicly available sources. The team primarily utilizes Python packages including:

* BeautifulSoup
* Pandas
* Requests
* OAuth1
* Parse

These Python packages are utilized to scrape information from Wikipedia. Additionally, the authors utilized API’s for iextrading.com and twitter in order to pull in the relevant information. Code for web scraping the president and political affiliation can be found in Appendix 1. Twitter code can be found in Appendix 2.

The process to retrieve stock information through the iextrading.com API is written in C# and utilizes the RestSharp and Newtonsoft.Json 3rd party libraries. Code for retrieving stock information can be found in Appendix 5.

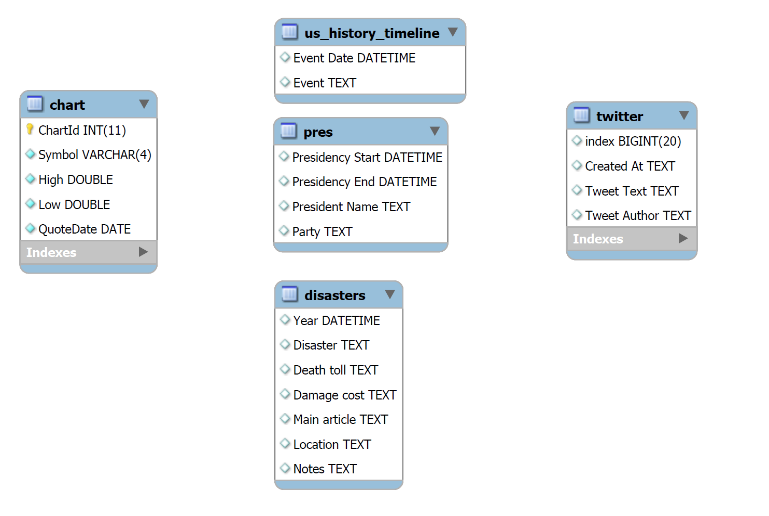
Once the various sources of data are gathered and formatted appropriately, the authors create a database schema using the MySQL Workbench tool.

In considering the data schema, the authors considered whether to use a SQL or NoSQL database schema.

# Results

The team was able to efficiently upload the data using the code provided in the appendices within this document. Once the data was uploaded, a SQL script was constructed (also provided in the index).

The SQL script was utilized within the ‘MySQL Workbench’ tool in order to establish a database schema. A pictorial of the final database schema is provided below.



# Analysis

Once the data was pulled the team explored creating a single, flat table within a database. This was not selected as an option because of the need to re-feed data. The team determined that it was possible and beneficial to automate the ‘refresh’ of data into the database schema from the original data sources. However, such automation comes with maintenance needs and certain data (e.g., stock prices) will need to be refreshed with more frequency.

For the reasons listed above, the team determined it better to maintain a schema with multiple tables.

The team next determined whether the database should be relational or non-relational. As stated previously, the purpose of the database is to consider the impact of non-firm controlled/generated data points to a firm’s stock price. This purpose by its very nature requires consideration of date(s). For this reason alone, the team determined that a relational database was necessary.

The next step, after the above decisions were made, was determining the level of normalization needed within the database. The information being pulled is interesting in that, with the combination of table schema choice by the team, arrives in third normalization form. That is, the data is stored in tables where there is only a single value; values are of the same domain (within a column); columns have unique names; order of the data stored does not matter; and there are not partial or transitive properties within the tables.

The last piece of analysis performed was creating new columns based on word placement of imported text. Within the new column events would be further categorized for ease in querying.

In particular, a new calculated column based off the ‘Event’ column in the ‘us\_history\_timeline’ table was created and titled ‘Event Short’. This column was populated from a case statement whereby the ‘Event’ column text string was analyzed for key words and then categorized into one of the following variables:

* Shooting
* International
* Equality
* World Mystery
* US High
* Politics
* Sports
* Disasters

As more data is added the case statements will group these events according to the defined logic.

The analysis described above was incorporated into the final product as described in the preceding sections of this document.

We were able to prove the feasibility of our final product through equity analysis of shooting event periods and natural disaster periods. Appendix 7 provides summary output of relevant data points.

# Ethics

At the core, this analysis is about determining if there is a feasible means of gathering significant amounts of data from multiple sources into a single relational database schema for purposes of analyzing, understanding, and possibly predicting standardized (i.e., over the counter products constructed for minority shareholder interests) corporate equity markets.

It may be concluded that with the construction of such a database schema the information available may in fact unlock asymmetric information to equity market participants. Simply put, with information comes advantage within market structures that are at times zero sum.

The authors of this document fully acknowledge that the markets at interest (corporate equity markets) are built and monitored under an almost singular premise of transparency to all participants. As a result, in performing this analysis, the authors focus on the gathering of information and data that would only be available to all market participants. In doing so, the authors avail themselves of any ethical concerns associated with providing an unfair trading advantage within the market of interest.

A last item of note is the notion of the unknown. An intended use of this analysis would be subsequent prediction (of equity market movement). Such prediction models, which have yet to be created, may in fact yield understandings between available information which can be further exploited according to variables which are not publicly distributed. In such cases, the authors rely upon the market governance structures currently in place along with the acceptance of the ‘unknown’.

# Conclusion and Future Work

Overall, a conclusion that can be reached from this work is as follows:

The team successfully demonstrated a path to downloading data from disparate information sources and in various states in order to consolidate into a single database schema.

The team explored bringing in additional forms of data (e.g., twitter) and determined that it was possible. However, not all of this data was included into the final database due to the lack of computational power required for data of such magnitude.

The final resulting database may be a useful tool to researches wishing to consider non-traditional forms of analysis when predicting or explaining changes in stock pricing.

In regard to future work and next steps, we have identified a number of options for future schema considerations. During the course of this project, our end state database resulted in a number of independent tables with no enforced relations. This is similar to a NoSQL structure. The authors are intrigued at how a NoSQL structure might better serve the business requirements.

Additionally, it could be worthwhile to investigate utilizing a denormalized structure. We performed most of our analysis with BI tools which are adept at pulling in large, flat data structures. Because of this, creating denormalized data structures would offset the complications and performance overhead of data refreshes. A pro/con analysis of this design could be intriguing.

A final piece of consideration with respect to future work may be exploring what additional types of information/data (again, fitting the non-traditional mold) could be included and whether such inclusion would require the construct of an intermediary table.

# Appendix 1

##############

#President Code

##############

import requests

import pandas as pd

from bs4 import BeautifulSoup

import re

def generate\_raw\_table(html\_table):

table = []

for row in html\_table.find\_all('tr'):

r=[]

for pos, col in enumerate(row.find\_all('td')):

if pos == 1:

for pos, span in enumerate(col.find\_all('span')):

if re.search(r'\]$', span.text):

r.append(span.text[:-3])

else:

r.append(span.text)

elif pos == 3:

r.append(col.find\_all('a')[0].text)

elif pos == 6:

try:

r.append(col.find('i').text)

except AttributeError:

r.append(col.find\_all('a')[0].text)

if r:

table.append(r)

return table

def create\_dataframe\_from\_raw\_table(raw\_table):

df = pd.DataFrame(raw\_table, columns=['Presidency Start', 'Presidency End', 'President Name', 'Party'])

df=df.dropna()

df['Presidency End'] =pd.to\_datetime(df['Presidency End'])

df['Presidency Start'] =pd.to\_datetime(df['Presidency Start'])

return df

def generate\_dates(start\_date, days\_count):

datelist = pd.date\_range(start\_date, periods=days\_count).tolist()

return datelist

def append\_presidency\_rows(df):

columns=['Presidency Start', 'Presidency End', 'President Name', 'Party']

lst = []

for index, row in df.iterrows():

start\_date = row['Presidency Start']

diff\_days = (row['Presidency End'] - start\_date).days

datelist = generate\_dates(start\_date=start\_date, days\_count=diff\_days)

for d in datelist:

r = [d, row['Presidency End'], row['President Name'],row['Party']]

lst.append(r)

new\_df = pd.DataFrame(lst, columns=columns)

return new\_df

# pick the last row's presidency End date

# get difference between presidency End date & today

# add diff rows in df, with current president name.

def add\_current\_president(df):

columns=['Presidency Start', 'Presidency End', 'President Name', 'Party']

last\_row = df.tail(1)

start\_date = last\_row['Presidency End'].iloc[0]

cur\_date = pd.to\_datetime("today")

diff= (cur\_date - start\_date).days

datelist = generate\_dates(start\_date=start\_date, days\_count=diff)

lst = []

for d in datelist:

r = [d, cur\_date.strftime("%Y-%m-%d"), 'Donald Trump','Republican Party']

lst.append(r)

ldf = pd.DataFrame(lst, columns=columns)

ldf['Presidency End'] =pd.to\_datetime(ldf['Presidency End'])

ldf['Presidency Start'] =pd.to\_datetime(ldf['Presidency Start'])

df = df.append(ldf, ignore\_index=True)

return df

def main():

url = 'https://en.wikipedia.org/wiki/List\_of\_Presidents\_of\_the\_United\_States'

res = requests.get(url).text

soup = BeautifulSoup(res,'lxml')

html\_table = soup.find('table',{'class':'wikitable'})

raw\_table = generate\_raw\_table(html\_table)

df = create\_dataframe\_from\_raw\_table(raw\_table)

df = append\_presidency\_rows(df)

df = add\_current\_president(df)

return df

#return main() function

main()

# Appendix 2

##############

#TWITTER CODE

##############

import requests

from requests\_oauthlib import OAuth1

import pandas as pd

from dateutil.parser import parse

consumer\_key = ‘’

consumer\_secret = ‘’

access\_token = ‘’

access\_secret = ‘’

def search\_tweets(author\_id, tweet\_count = 25):

search\_url = 'https://api.twitter.com/1.1/statuses/user\_timeline.json'

auth = OAuth1(consumer\_key, consumer\_secret, access\_token, access\_secret)

params = {

'screen\_name': author\_id,

'count' : tweet\_count

}

res = requests.get(search\_url, auth=auth, params=params)

if res.status\_code == 200:

tweets = res.json()

l = []

for tweet in tweets:

dt = parse(tweet['created\_at'])

t = (dt.strftime('%m/%d/%y %H:%M:%S'), tweet['text'], author\_id)

l.append(t)

else:

print("Something went wrong, Status Code ", res.status\_code)

return l

def main():

pattern = "@ScienceNews"

tweet\_count = 100

l = search\_tweets(author\_id=pattern, tweet\_count= tweet\_count)

df = pd.DataFrame(l, columns=['Created At', 'Tweet Text', 'Tweet Author'])

return df

main()

# Index 3

##############

#TIMELINE CODE

##############

# Imports of the respective libraries

import requests

import pandas as pd

from bs4 import BeautifulSoup

# Convert HTML table to a list of lists for each row

# so as to easily load the same pandas dataframe.

def generate\_raw\_table(html\_table):

table = []

year, date, event = None, None, None

# find all rows from the table

for row in html\_table.find\_all('tr'):

r=[]

# find all columns from the row

for col in row.find\_all('td'):

# remove extra spaces from cell value

cell = col.text.strip()

# if the text is of 4 chars, store it as year if it's an integer

if len(cell) == 4:

try:

# distinguish between 1901 and July

year = int(cell)

except:

pass

# HTML date column has values like July 21, May–June, Mid-October

# so split and convert the same in correct format

elif 4 < len(cell) < 13:

if cell.startswith('Mid'):

date = cell.split('-')[1] + ' 15'

else:

date = cell.split('–')[0]

# to pick up the description (Event Col)

elif len(cell) > 13:

# To remove text like [12][33][87] from the end.

event = '.'.join(s for s in cell.split('.')[:-1])

else:

continue

# if we have got all the values then convert it into a list of 2 values

if all([year, date, event]):

event\_date = str(year) +' '+ date + ' 00:00:00'

r.append([event\_date, event])

# extend parent table list with above rows.

if r:

table.extend(r)

return table

# Create dataframe with Event Date and Event as 2 coloumns

def create\_dataframe\_from\_raw\_table(raw\_table):

df = pd.DataFrame(raw\_table, columns=['Event Date', 'Event'])

return df

# Combine the dataframes at a time

# and convert datetime 'string' to 'datetime' type

def combine\_dataframes(df0, df1):

df = df0.append(df1, ignore\_index=True)

df['Event Date'] = pd.to\_datetime(df['Event Date'])

return df

# main(): the first function and it holds the flow of the script.

def main():

url = 'https://en.wikipedia.org/wiki/Timeline\_of\_United\_States\_history'

# To get the (HTML) text of the static webpage.

res = requests.get(url).text

soup = BeautifulSoup(res,'lxml')

# Get all tables from the HTML text

html\_table = soup.find\_all('table',{'class':'wikitable'})

# Since all tables follow common structure, so we selected our table of

# interest by giving the index of that table.

# 11- for table for 20th century

#raw\_table = generate\_raw\_table(html\_table[11])

# convert above raw table to a pandas dataframe

#df\_20 = create\_dataframe\_from\_raw\_table(raw\_table)

# 12- for table for 21st century

#raw\_table = generate\_raw\_table(html\_table[12])

#13 - for table for rest of 21st century

raw\_table = generate\_raw\_table(html\_table[13])

# convert above raw table to a pandas dataframe

df\_21 = create\_dataframe\_from\_raw\_table(raw\_table)

# return both the dataframes

#return df\_20, df\_21

return df\_21

df0, df1 = main()

# Combine the dataframes for 20th and 21st century to one.

#df = pd.DataFrame(combine\_dataframes(df0, df1))

df['Event Date'] =pd.to\_datetime(df['Event Date'])

#df.tail(10)

df=pd.DataFrame(main())

df.tail(5)

df['Event Date'] =pd.to\_datetime(df['Event Date'])

df.head(5)

# Import dataframe into MySQL

import sqlalchemy

from sqlalchemy import create\_engine

kwargs = dict(

username = 'root',

password = '1942bdla',

database\_ip = 'localhost',

database\_name = 'history',

)

from sqlalchemy import create\_engine

#engine = create\_engine("mysql+pymysql://root:"+'password'+"@localhost/ecommercedb")

conn\_string = "mysql+pymysql://{username}:{password}@{database\_ip}/{database\_name}".format(\*\*kwargs)

engine = create\_engine(conn\_string)

df.to\_sql(con=engine, if\_exists='replace', index=False,name='us\_history\_timeline')

# Appendix 4

##############

#DISASTERS CODE

##############

from requests import get

from requests.exceptions import RequestException

from contextlib import closing

from bs4 import BeautifulSoup

def simple\_get(url):

"""

Attempts to get the content at `url` by making an HTTP GET request.

If the content-type of response is some kind of HTML/XML, return the

text content, otherwise return None.

"""

try:

with closing(get(url, stream=True)) as resp:

if is\_good\_response(resp):

return resp.content

else:

return None

except RequestException as e:

log\_error('Error during requests to {0} : {1}'.format(url, str(e)))

return None

def is\_good\_response(resp):

"""

Returns True if the response seems to be HTML, False otherwise.

"""

content\_type = resp.headers['Content-Type'].lower()

return (resp.status\_code == 200

and content\_type is not None

and content\_type.find('html') > -1)

def log\_error(e):

"""

It is always a good idea to log errors.

This function just prints them, but you can

make it do anything.

"""

print(e)

raw\_html = simple\_get('https://en.wikipedia.org/wiki/List\_of\_natural\_disasters\_in\_the\_United\_States')

len(raw\_html)

no\_html = simple\_get('https://en.wikipedia.org/wiki/List\_of\_natural\_disasters\_in\_the\_United\_States')

no\_html is None

True

from bs4 import BeautifulSoup

raw\_html = open('contrived.html').read()

html = BeautifulSoup(raw\_html, 'html.parser')

for tr in html.select('tr'):

if tr['a'] == 'disaster':

print(tr.text)

raw\_html = simple\_get('https://en.wikipedia.org/wiki/List\_of\_natural\_disasters\_in\_the\_United\_States')

html = BeautifulSoup(raw\_html, 'html.parser')

for i, tr in enumerate(html.select('tr')):

print(i, tr.text)

def get\_names():

"""

Downloads the page where the list of disasters is found

and returns a list of strings, one per disaster

"""

url = 'https://en.wikipedia.org/wiki/List\_of\_natural\_disasters\_in\_the\_United\_States'

response = simple\_get(url)

if response is not None:

html = BeautifulSoup(response, 'html.parser')

names = set()

for tr in html.select('tr'):

for name in tr.text.split('\n'):

if len(name) > 0:

names.add(name.strip())

return list(names)

raise Exception('Error retrieving contents at {}'.format(url))

def get\_hits\_on\_name(name):

"""

Accepts a `name` of a disaster and returns the number

of hits that disaster's Wikipedia page received in the

last 60 days, as an `int`

"""

# url\_root is a template string that is used to build a URL.

url\_root = 'https://xtools.wmflabs.org/articleinfo/en.wikipedia.org/{}'

response = simple\_get(url\_root.format(name))

if response is not None:

html = BeautifulSoup(response, 'html.parser')

hit\_link = [a for a in html.select('tr')

if a['href'].find('latest-60') > -1]

if len(hit\_link) > 0:

# Strip commas

link\_text = hit\_link[0].text.replace(',', '')

try:

# Convert to integer

return int(link\_text)

except:

log\_error("couldn't parse {} as an `int`".format(link\_text))

log\_error('No pageviews found for {}'.format(name))

return None

if \_\_name\_\_ == '\_\_main\_\_':

print('Getting the list of names....')

names = get\_names()

print('... done.n')

results = []

print('Getting stats for each name....')

for name in names:

try:

hits = get\_hits\_on\_name(name)

if hits is None:

hits = -1

results.append((hits, name))

except:

results.append((-1, name))

log\_error('error encountered while processing '

'{}, skipping'.format(name))

print('... done.n')

results.sort()

results.reverse()

if len(results) > 5:

top\_marks = results[:5]

else:

top\_marks = results

print('nThe most popular disasters are:n')

for (mark, disaster) in top\_marks:

print('{} with {} page views'.format(disaster, mark))

no\_results = len([res for res in results if res[0] == -1])

print('nBut we did not find results for '

'{} disasters on the list'.format(no\_results))

data = []

table = soup.find('table', attrs={'class':'lineItemsTable'})

table\_body = table.find('tbody')

rows = table\_body.find\_all('tr')

for row in rows:

cols = row.find\_all('a')

cols = [ele.text.strip() for ele in cols]

data.append([ele for ele in cols if ele]) # Get rid of empty values

# Appendix 5

using System;

using System.Collections.Generic;

using System.Linq;

using System.Net.Http;

using System.Text;

using System.Threading.Tasks;

using Newtonsoft.Json;

using RestSharp;

using MySql.Data.MySqlClient;

using System.Configuration;

namespace GP1.App

{

class Program

{

static string IEXTrading\_API\_PATH = "https://api.iextrading.com/1.0/stock/{symbol}/chart/{years}y";

static string[] symbols = { "aobc", "rgr", "hd", "kr", "swy", "winn", "msft", "wmt", "dji", "ge" };

static void Main(string[] args)

{

DeleteExistingStockData();

foreach (string symbol in symbols)

{

var historicalDataList = GetChart(symbol, 5);

WriteStockDataToDatabase(historicalDataList, symbol);

}

}

public static List<HistoricalDataResponse> GetChart(string symbol, int years)

{

RestClient client = new RestClient(IEXTrading\_API\_PATH);

RestRequest request = new RestRequest(Method.GET);

request.AddHeader("Accept", "application/json");

request.AddUrlSegment("symbol", symbol);

request.AddUrlSegment("years", years);

var historicalDataList = new List<HistoricalDataResponse>();

try

{

var response = client.Execute(request);

if (response.StatusCode == System.Net.HttpStatusCode.OK)

{

historicalDataList = JsonConvert.DeserializeObject<List<HistoricalDataResponse>>(response.Content);

}

}

catch (Exception ex)

{

Console.WriteLine($"Error calling api. Message:{ex.Message}");

}

return historicalDataList;

}

private static void DeleteExistingStockData()

{

using (var connection = new MySqlConnection(ConfigurationManager.ConnectionStrings["StockData"].ToString()))

{

var command = new MySqlCommand("usp\_DeleteStockData", connection);

command.CommandType = System.Data.CommandType.StoredProcedure;

try

{

connection.Open();

command.ExecuteNonQuery();

}

catch (Exception ex)

{

Console.WriteLine($"Error deleting stock data from database. Message:{ex.Message}");

}

}

}

private static void WriteStockDataToDatabase(List<HistoricalDataResponse> stockDataList, string symbol)

{

Console.WriteLine($"Writing {stockDataList.Count} records to the database");

int counter = stockDataList.Count;

using (var connection = new MySqlConnection(ConfigurationManager.ConnectionStrings["StockData"].ToString()))

{

try

{

connection.Open();

foreach (var stockData in stockDataList)

{

var command = new MySqlCommand("usp\_InsertStockData", connection);

command.CommandType = System.Data.CommandType.StoredProcedure;

command.Parameters.AddWithValue("@Symbol", symbol);

command.Parameters.AddWithValue("@High", stockData.high);

command.Parameters.AddWithValue("@Low", stockData.low);

command.Parameters.AddWithValue("@QuoteDate", stockData.date);

command.Parameters.AddWithValue("@Close", stockData.close);

command.ExecuteNonQuery();

counter--;

if (counter % 100 == 0)

{

Console.WriteLine($"{counter} records remaining.");

}

}

}

catch (Exception ex)

{

Console.WriteLine($"Error writing stock data to database. Message:{ex.Message}");

}

}

}

}

}

# Appendix 6

\*\*\*\*\*\*\*\*\*\*\*\*\*

EVENT SHORT CASE STATEMENT

\*\*\*\*\*\*\*\*\*\*\*\*\*

# Appendix 7





1. [↑](#footnote-ref-1)