Table of Contents

FiveThirtyNine

https://cis550project-api-heroku.herokuapp.com/

1. Introduction and Project Goals

2018 was an exhausting year in America. The two major political parties clashed over issues large and small, including climate change, tax policy, President Trump's conduct, a heated midterm election, and the Supreme Court. With so many important news events and political developments occurring at warp-speed throughout the year, it was difficult to parse through what truly mattered - and why. By studying larger trends in politics, the economy, and the media, one can gain a greater understanding for the the complicated interplay between these facets of American society. Looking back through the year that was, we can gain a greater understanding for where we've been, and perhaps more importantly, where we're going.

Our project, FiveThirtyNine, aimed to organize economic data, national political indicators (including President Trump's approval rating and generic ballot party preferences), and top news headlines all in one easy-to-use to use platform. Meant for political junkies and casual observers alike, FiveThirtyNine has a variety of features that allow the user to study the national trends that defined 2018. The calendar feature provides data on stock indices, top news headlines, twitter trends, and political measurements for each day in 2018, allowing the user to explore at their own pace. We also wanted users to be able to dive into the data more deeply, so we created a functionality that allows users to search keywords such as "Hurricane" or "Saudi Arabia" to see how President Trump's approval rating changed the last time these topics were in the national discussion. Users can also pick a range of dates for which they can see the dataset of their choice visualized graphically, or see a wordcloud composed of the news headlines and Twitter trends in that timeframe. Finally, users can explore what occurred during the most volatile days of Donald Trump's presidency, and directly see how national news events impact the political players in our democratic system.

Our hope is that with the benefit of perspective, our users can start to make sense of the year that was. Why did President Trump's approval rating improve in September? How were the Democrats able to reclaim the House of Representatives? These questions don't have easy answers; nonetheless, the FiveThirtyNine team aimed to give users the tools to try. We're all political junkies and we wanted to combine our love of politics with the skills and tools developed through a semester of CIS 550. Political involvement is at an all-time high, and FiveThirtyNine matches user's desire to be politically connected with empirical data so that they can best understand the beautifully complicated system that is the U.S government.

2. Data Sources and Technologies Used

In order to provide perspective on political sentiment using insights from polls, media outlets and economic indicators, we needed to cull data from a diverse array of resources. The underlying relations that define our project fall broadly into two categories: quantitative information that was available in flat (.CSV) files and plain-English media topics that we retrieved via web scraping. The former category includes polling data from **FiveThirtyEight** (both Presidential Approval and Generic Pollist estimates) as well as daily close data from the three major U.S. stock market indices; all stock data is published by the **Federal Reserve Bank of St. Louis**. We also used a **Holidays Dataset csv,** extracted from Github.

Given that there is no firm definition for what constitutes a major media headline, there was also no pre-packaged dataset that we could download in order to incorporate news stories and significant Twitter topics into our data. However, the **New York Times** provides an archive page that lists its four front-page headlines for a user-selected date. Using the beautiful soup Python web scraping package, we were able to select each date within the last calendar year and pull the corresponding headlines. The resulting table contains the headlines themselves, their more verbose sub-headlines, and the date on which the headline was published. We also wanted to capture online sentiments in a similar manner, so we focused on pulling Twitter data from the web. Although the Twitter API did not help us with this task, **Trendogate.com** archives Trending topics by date. Using logic similar to that employed for the New York Times scrape, we collected trending Twitter topics for each date within the last year. In both cases, we placed our scraped data into CSVs before importing to the database.

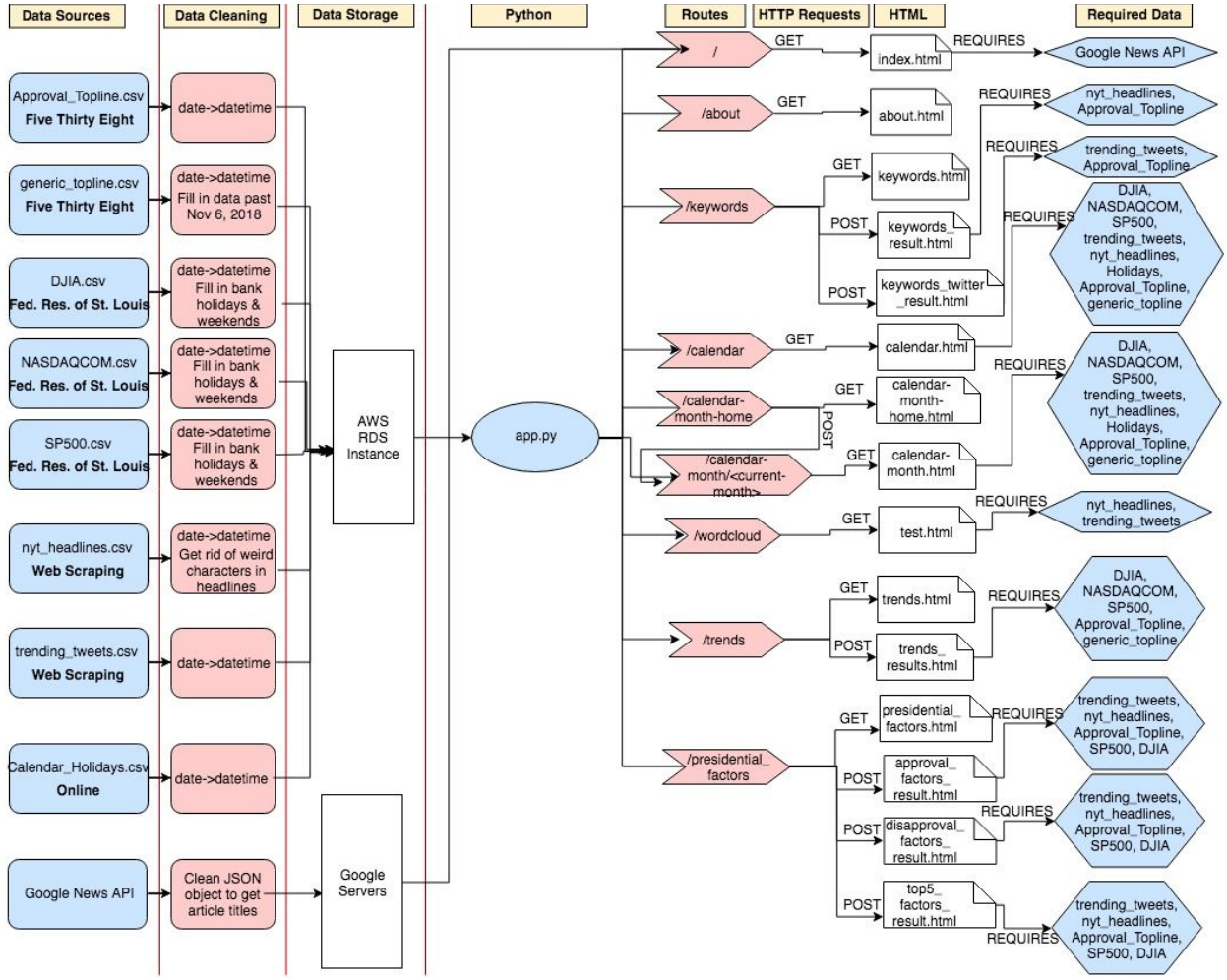
The technology used to implement our application can be divided into three categories: tools used to construct and host the database, our application infrastructure (how we built pages and connected them to the database), and front-end functionality. Our first priority was to ensure that we could construct a database that would be easily accessible to all members of the group (i.e. not a local database on one member's laptop). We experimented with multiple implementations on **AWS Relational Database Service**; our **MySQL** instance was the first for which we successfully configured security and confirmed that all members could query successfully. AWS also provided the indexing functionality that allowed us to apply BTREE indexing over some of our attributes.

In order to import our flat files, we used the **MySQLWorkbench desktop application**. It provides a GUI that allowed us to easily diagnose issues with attribute type, key declarations and table importation. The Workbench application was most valuable as a tool for running and modifying queries without intermediate steps (e.g. writing routes in our application and displaying data via the webpage).

The FiveThirtyNine application itself is written primarily in Python via the **Flask** development framework. Flask's file structure allowed us to easily organize our Python code, .HTML pages, and underlying CSS while providing access to a variety of packages related to querying, data manipulation and visualization. In particular, the **flask-MySQL extension** was key in allowing us to build queries into the app and incorporate user inputs as selection criteria. Another helpful package was **JSON,** which worked in conjunction with SQL aggregation commands and enabled the placement of multiple strings (e.g. headlines and topics) into a single result tuple. Its value becomes clear when our calendar and extreme event pages return multiple stories for a given date. Finally, the **time** package enabled us to monitor application performance.

A demonstration of the FiveThirtyNine application reveals a few more Python packages at work. In particular, **Bokeh** enabled all of the visualization of our query results, including both calendar pages and the trending graphs.

3. Diagram and Description of System Architecture



To give context to the technology descriptions in Part 2, the above diagram details the overall design of our application as well as the flow of data between its constituent parts. The **Data Sources** column at far left begins with flat-file versions of all the data that we collected, and **Data Cleaning** offers a concise overview of the data preparation tasks detailed in Section 4 below.

Live contents that appear within the application itself begin with the **Data Storage** column. We imported each cleaned .csv file into our AWS RDS MySQL database *as a table of the same name*, whose schema are detailed in section 4. As the diagram suggests, the Python script **app.py** functioned as the lynchpin of our entire web project. app.py contains our database connection parameters (handled using flask-mysql syntax), routes for each of our functionality pages (generating user forms and the corresponding results pages) and, within those routes, the MySQL queries themselves including wildcards for user input integration.

While using our application or learning about its functionality in Section 4, please refer to the **HTML** and **Required Data** columns for a better understanding of the underlying relations that we query to show the user each type of result.

4. How you addressed the technical features

We addressed all the required technical features in building FiveThirtyNine.

The first step of our project was drawing Information from at least two large datasets; to give holistic insights on politics we had to provide economic, media and political data. The media information was acquired using web scraping (HTML), the economic data came from CSV files downloaded from the web, and we downloaded the FiveThirtyEight political data in CSV format. Finally, a Holidays database was extracted from Github. More detailed descriptions can be found in Section 2 (Data Sources).

We also engaged in comprehensive data cleaning. The Twitter and New York Times data required more cleaning than the other tables, as parsing web-hosted text into a CSV file generated incomprehensible special characters. To correct for this, our team created a map of key-value pairs used to substitute the wrong characters for the correct ones. Additionally, there were missing date values in the economic tables - stock indices are not traded on non working days - which could have ultimately lead to data loss when joining relations. In response, we imputed stock market data for every weekend using the closing stock price of the previous. Finally, to store each column's data correctly and facilitate the entity resolution step, we used Python Pandas package to convert all dates to datetime variables, enabling cleaner entity resolution.

To handle Entity Resolution, we focused on connecting the data using the Date attribute. Many of our functionalities were based on joining multiple tables on the date column, most notably the calendar function, which showcased a wide variety of information about every day of the year. That being said, the news tables did not originally have 'Date' as a single primary key. We amalgamated every headline from a single date together into one single tuple using JSON aggregation command in our SQL queries.

FiveThirtyNine developed a robust normalized relational schema, containing the following eight tables, each listed with the primary key underlined:

Politics relations

a. Approval_Topline (president, subgroup, date, approve_estimate, disapprove_estimate)
b. generic_topline(subgroup, date, dem_estimate, rep_estimate)

Economics relations

c. DJIA (DJIA, date)
d. NASDAQCOM(NASDAQCOM, date)
e. SP500(SP500, date)

Media relations

f. nyt_headlines(headline, subheadline, date)
g. trending_tweets(topic, date)

Calendar relation

h. Calendar_Holidays (Holiday, date)

FiveThirtyNine features a total of six webpages interacting with the database. Each provided a different functionality for the user to explore the political environment. Some examples include: the 2018 Yearly Calendar, where the user can see an interactive calendar which displays political, media and economic data in a calendar format colored with a heatmap of political sentiment, the trends graph feature, which allows users to plot charts of economic and political variables over time. We also created a keyword search interface that allows users to search for a topic of interest, for instance "Gun control", and see all of the times that topic has been in the headlines and the accompanying changes in President Trump's approval. Finally, the presidential approval drivers page allowed the user to see the maximum approval/disapproval and highest absolute changes in presidential popularity within a date range.

FiveThirtyEight design was implemented with HTML and CSS. We tried to take a clean approach to design, using a centralized home page and easy parallax scrolling, in order to keep the user focused on understanding the political environment. The calendar application was created using Bokeh, a python wrapper that created good-looking calendars that allowed for a hovering feature. Furthermore, our queries used temporary tables, joins, aggregation (GROUP and JSON), ordering and included user input data for selections. More details can be found below on Table I: Query Lookup Table.

In an effort to improve the run time of our queries, we implemented B-Tree indexing in Date for every relation. We chose B-tree indexing due to its ability to support both range and equality searches, which we used frequently. Also, for more complex queries, it was important to push selections to the moment of the temporary table creation. In fact, indexing performance improvements can be found on tables II, III and IV. One major query optimization win via pushing selections in temporary tables actually improved run time by 500%.

6. Technical Challenges

One major technical challenge we faced was quite basic - how to store our varied datasets together in an RDBMS. Since data organization by date was a key functionality of FiveThirtyNine, it was extremely important that we be able to join, query, and select tables on the date attribute, but our incoming data all had dates that were formatted slightly differently. To correct for this, we created a datetime parser function that iterated through the table, grabbed each date, and converted it to pd.datetime(). However, the type of the date column was still stored as 'text', so we had to create a new column called 'date' (whose type was set to 'datetime'), and copy the updated column to this new 'date' column. This put every date column in the entire project on equal footing and allowed us to filter information based on date.

Query optimization also posed a significant challenge for our team. For instance, an original temp table we created for headlines grabbed all tuples in the relation. It was only filtered on date as part of the join to a second table including economic and approval indicators. However, we were able to avoid retrieving all tuples by imposing the BTREE index on the headlines' dates and pushing our date range selection to the temp table, RATHER than evaluating for all headlines as part of a join (the most expensive component of query execution).

It was also very important to limit queries to the relevant data for the end user, push projections to the base relations when creating temporary tables; this improved our query speed by up to 80%. Moreover, we selected on subgroup (='all polls') in subqueries without actually returning that column.

A final challenge occurred through our use of Bokeh, a visual plotting package that fits into Python. For multiple functionalities, including monthly calendar display and trend visualization, the user was able to quickly alter which data they would like to see graphed. However, even after refreshing the page, we noticed that the graphs were not reloading - they remained the same graphs that the user initially picked. To remedy

this, we used the bokeh command “components”, which broke up the script and div tags that could be transferred to the appropriate html page through flask. This fix allowed us to show graphs that updated quickly and easily.

5. Performance Evaluation

i. Indexing

We used python’s ‘time’ feature to time our queries before and after creating indexes on our tables. Since every query that we run relies on the ‘date’ column, we decided to create an index on ‘date’ for each of our 8 tables. We chose B Tree indexes because we have both equality and range searches on date, so we need our index type to support both operations efficiently.

To perform the tests, we ran each of our 26 queries 5 times and took the average of these times in seconds (per query) to obtain an average query runtime.

We then created B Tree indexes and ran the same tests again. We saw an average percent time decrease of **14.29%** after indexing, though the results varied significantly among queries.

These tests were impacted by the speed of WiFi connection at time of testing and load on AWS RDS system at time of testing, among other factors. Despite this, a >10% decrease in query execution time is notable and significant.

Table I: Query Lookup Table

Route	Query	Query #

/keywords	cur1.execute("DROP TABLE IF EXISTS NYT_STUFF");
	cur1.execute("CREATE TEMPORARY TABLE NYT_STUFF as (SELECT date, headline, `sub-headline` FROM cis550hpps.nyt_headlines WHERE headline like '%s' ORDER BY date DESC LIMIT 20);" %("%" + user_keyword + "%"));
	cur1.execute("SELECT nyt.date, headline, `sub-headline`,t1.approve_estimate - (SELECT t2.approve_estimate FROM cis550hpps.APPROVAL_TOPLINE t2 WHERE t2.date < t1.date AND t2.subgroup = 'All polls' ORDER BY t2.date DESC LIMIT 1) AS DAY_CHANGE_IN_APPROVAL FROM NYT_STUFF nyt LEFT JOIN cis550hpps.APPROVAL_TOPLINE t1 ON nyt.date = t1.date WHERE t1.subgroup = 'All polls' ORDER BY nyt.date desc;");
	cur1.execute("DROP TEMPORARY TABLE NYT_STUFF;");
		1

/keywords	cur1.execute("DROP TABLE IF EXISTS TWITTER_STUFF");
	cur1.execute("CREATE TEMPORARY TABLE TWITTER_STUFF as(SELECT date, topic FROM cis550hpps.trending_tweets WHERE topic like '%s'ORDER BY date DESC LIMIT 20);" %("%" + user_keyword + "%"));
	cur1.execute("SELECT ts.date, ts.topic,t1.approve_estimate - (SELECT t2.approve_estimate FROM cis550hpps.APPROVAL_TOPLINE t2 WHERE t2.date < t1.date AND t2.subgroup = 'All polls'ORDER BY t2.date DESC LIMIT 1) AS DAY_CHANGE_IN_APPROVAL FROM TWITTER_STUFF ts LEFT JOIN cis550hpps.APPROVAL_TOPLINE t1 ON ts.date = t1.date WHERE t1.subgroup = 'All polls';");
	cur1.execute("DROP TEMPORARY TABLE TWITTER_STUFF;");
		2

/calendar	cur.execute("SELECT date AS dj_date, DJIA FROM DJIA");	3
/calendar	cur2.execute("SELECT date AS nas_date, NASDAQCOM FROM NASDAQCOM");	4
/calendar	cur3.execute("SELECT date AS sp500_date, SP500 FROM SP500");	5
/calendar	cur4.execute("SELECT date AS twitter_date, JSON_ARRAYAGG(topic) AS topics FROM trending_tweets GROUP BY date")	6
/calendar	cur5.execute("SELECT date AS holiday_date, Holiday FROM CalendarHolidays WHERE date <> '0000-00-00 00:00:00'");	7
/calendar	cur6.execute("SELECT date AS nyt_date, JSON_ARRAYAGG(headline) AS headlines FROM nyt_headlines GROUP BY date")	8
/calendar	cur7.execute("SELECT date AS approval_date, approve_estimate FROM APPROVAL_TOPLINE WHERE subgroup='All polls'")	9
/calendar	cur8.execute("SELECT date AS generic_date, dem_estimate FROM generic_topline WHERE subgroup='All polls'")	10
/calendar	cur9.execute("SELECT date AS generic_date, rep_estimate FROM generic_topline WHERE subgroup='All polls'")	11
/calendar -month	cur.execute("SELECT date AS dj_date, DJIA FROM DJIA WHERE date BETWEEN '%s' AND '%s'" %(start_date, end_date));	12
/calendar -month	cur2.execute("SELECT date AS nas_date, NASDAQCOM FROM NASDAQCOM WHERE date BETWEEN '%s' AND '%s'" %(start_date, end_date));	13
/calendar -month	cur3.execute("SELECT date AS sp500_date, SP500 FROM SP500 WHERE date BETWEEN '%s' AND '%s'" %(start_date, end_date));	14
/calendar -month	cur4.execute("SELECT date AS twitter_date, JSON_ARRAYAGG(topic) AS topics FROM trending_tweets WHERE date BETWEEN '%s' AND '%s' GROUP BY date" %(start_date, end_date))	15
/calendar -month	cur5.execute("SELECT date AS holiday_date, Holiday FROM CalendarHolidays WHERE date <> '0000-00-00 00:00:00' AND date BETWEEN '%s' AND '%s'" %(start_date, end_date));	16
/calendar -month	cur6.execute("SELECT date AS nyt_date, JSON_ARRAYAGG(headline) AS headlines FROM nyt_headlines WHERE date BETWEEN '%s' AND '%s' GROUP BY date" %(start_date, end_date))	17
/calendar -month	cur7.execute("SELECT date AS approval_date, approve_estimate FROM APPROVAL_TOPLINE WHERE subgroup='All polls' AND date BETWEEN '%s' AND '%s'" %(start_date, end_date))	18
/calendar -month	cur8.execute("SELECT date AS generic_date, dem_estimate FROM generic_topline WHERE subgroup='All polls' AND date BETWEEN '%s' AND '%s'" %(start_date, end_date))	19
/calendar -month	cur9.execute("SELECT date AS generic_date, rep_estimate FROM generic_topline WHERE subgroup='All polls' AND date BETWEEN '%s' AND '%s'" %(start_date, end_date))	20
/wordcloud	cur.execute("SELECT headline FROM nyt_headlines")	21

/wordcloud	cur4.execute("SELECT topic FROM trending_tweets")	22

/presidential_factors	curP.execute("DROP TABLE IF EXISTS daily_tweets;");
	curP.execute("CREATE TEMPORARY TABLE daily_tweets AS (SELECT date, JSON_ARRAYAGG(topic) AS topics FROM trending_tweets WHERE date > '%s' AND date < '%s' GROUP BY date);" %(first_date_datetime, second_date_datetime));
	curP.execute("DROP TABLE IF EXISTS nyt_grouped_headlines;");
	curP.execute("CREATE TEMPORARY TABLE nyt_grouped_headlines AS (SELECT date AS date, JSON_ARRAYAGG(headline) AS headlines FROM nyt_headlines WHERE date > '%s' AND date < '%s' GROUP BY date);"%(first_date_datetime, second_date_datetime));
	curP.execute("DROP TABLE IF EXISTS SP500_Daily_Changes;");
	curP.execute("CREATE TEMPORARY TABLE SP500_Daily_Changes AS (SELECT sp1.date, sp1.SP500, TRUNCATE(((sp1.SP500 / (SELECT sp2.SP500 FROM cis550hpps.SP500 AS sp2 WHERE sp2.date < sp1.date ORDER BY sp2.date DESC LIMIT 1)) - 1)*100, 3) AS DAYLY_PERC_CHANGE_SP500 FROM cis550hpps.SP500 AS sp1 WHERE sp1.date > '%s' AND sp1.date < '%s');"%(first_date_datetime, second_date_datetime));
	curP.execute("SELECT at.date, TRUNCATE(MAX(at.approve_estimate),2), TRUNCATE(spd.DAYLY_PERC_CHANGE_SP500,2) , dt.topics, nyt.headlines FROM APPROVAL_TOPLINE AS at LEFT JOIN SP500_Daily_Changes AS spd ON at.date = spd.date LEFT JOIN daily_tweets AS dt ON at.date = dt.date LEFT JOIN nyt_grouped_headlines AS nyt ON at.date = nyt.date WHERE at.date > '%s' AND at.date < '%s';" %(first_date_datetime, second_date_datetime));
		23

/presidential_factors	curP.execute("DROP TABLE IF EXISTS daily_tweets;");
	curP.execute("CREATE TEMPORARY TABLE daily_tweets AS (SELECT date, JSON_ARRAYAGG(topic) AS topics FROM trending_tweets WHERE date > '%s' AND date < '%s' GROUP BY date);" %(first_date_datetime, second_date_datetime));
	curP.execute("DROP TABLE IF EXISTS nyt_grouped_headlines;");
	curP.execute("CREATE TEMPORARY TABLE nyt_grouped_headlines AS (SELECT date AS date, JSON_ARRAYAGG(headline) AS headlines FROM nyt_headlines WHERE date > '%s' AND date < '%s' GROUP BY date);"%(first_date_datetime, second_date_datetime));
	curP.execute("DROP TABLE IF EXISTS SP500_Daily_Changes;");
	curP.execute("CREATE TEMPORARY TABLE SP500_Daily_Changes AS (SELECT sp1.date, sp1.SP500, TRUNCATE(((sp1.SP500 / (SELECT sp2.SP500 FROM cis550hpps.SP500 AS sp2 WHERE sp2.date < sp1.date ORDER BY sp2.date DESC LIMIT 1)) - 1)*100, 3) AS DAYLY_PERC_CHANGE_SP500 FROM cis550hpps.SP500 AS sp1 WHERE sp1.date > '%s' AND sp1.date < '%s');"%(first_date_datetime, second_date_datetime));
	curP.execute("SELECT at.date, TRUNCATE(MAX(at.disapprove_estimate),2), TRUNCATE(spd.DAYLY_PERC_CHANGE_SP500,2), dt.topics, nyt.headlines FROM APPROVAL_TOPLINE AS at LEFT JOIN SP500_Daily_Changes AS spd ON at.date = spd.date LEFT JOIN daily_tweets AS dt ON at.date = dt.date LEFT JOIN nyt_grouped_headlines AS nyt ON at.date = nyt.date WHERE at.date > '%s' AND at.date < '%s';" %(first_date_datetime, second_date_datetime));
		24

/presidential_factors	curP.execute("DROP TABLE IF EXISTS PA_Changes;");
	curP.execute("CREATE TEMPORARY TABLE PA_Changes AS (SELECT t1.date, (((t1.approve_estimate - (SELECT t2.approve_estimate FROM cis550hpps.APPROVAL_TOPLINE AS t2 WHERE t2.date < t1.date AND t2.subgroup = 'All polls' ORDER BY t2.date DESC LIMIT 1)))) AS Day_Change_Approval FROM cis550hpps.APPROVAL_TOPLINE AS t1 WHERE t1.subgroup = 'All polls');");
	curP.execute("DROP TABLE IF EXISTS daily_tweets;");
	curP.execute("CREATE TEMPORARY TABLE daily_tweets AS (SELECT date, JSON_ARRAYAGG(topic) AS topics FROM trending_tweets WHERE date > '%s' AND date < '%s' GROUP BY date);" %(first_date_datetime, second_date_datetime));
	curP.execute("DROP TABLE IF EXISTS nyt_grouped_headlines;");
	curP.execute("CREATE TEMPORARY TABLE nyt_grouped_headlines AS (SELECT date AS date, JSON_ARRAYAGG(headline) AS headlines FROM nyt_headlines WHERE date > '%s' AND date < '%s' GROUP BY date);"%(first_date_datetime, second_date_datetime));
	curP.execute("DROP TABLE IF EXISTS SP500_Daily_Changes;");
		25

	curP.execute("CREATE TEMPORARY TABLE SP500_Daily_Changes AS (SELECT sp1.date, sp1.SP500, TRUNCATE(((sp1.SP500 / (SELECT sp2.SP500 FROM cis550hpps.SP500 AS sp2 WHERE sp2.date < sp1.date ORDER BY sp2.date DESC LIMIT 1)) - 1)*100, 3) AS DAYLY_PERC_CHANGE_SP500 FROM cis550hpps.SP500 AS sp1 WHERE sp1.date > '%s' AND sp1.date < '%s');"%(first_date_datetime, second_date_datetime)); curP.execute("SELECT DISTINCT(pa.date), TRUNCATE(pa.Day_Change_Approval,2), TRUNCATE(spd.DAYLY_PERC_CHANGE_SP500,2), dt.topics, nyt.headlines FROM PA_Changes AS pa LEFT JOIN SP500_Daily_Changes AS spd ON pa.date = spd.date LEFT JOIN daily_tweets AS dt ON pa.date = dt.date LEFT JOIN nyt_grouped_headlines AS nyt ON pa.date = nyt.date WHERE pa.date > '%s' AND pa.date < '%s' ORDER BY ABS(pa.Day_Change_Approval) DESC LIMIT 5;" %(first_date_datetime, second_date_datetime));	
/trends	cur.execute("SELECT DJIA.date AS date, SP500, NASDAQCOM, DJIA, approve_estimate, dem_estimate, rep_estimate FROM DJIA JOIN SP500 ON DJIA.date = SP500.date JOIN NASDAQCOM ON DJIA.date = NASDAQCOM.date JOIN APPROVAL_TOPLINE ON DJIA.date = APPROVAL_TOPLINE.date JOIN generic_topline ON generic_topline.date = DJIA.date WHERE APPROVAL_TOPLINE.subgroup = 'All polls' AND generic_topline.subgroup = 'All polls' AND DJIA.date BETWEEN '%s' AND '%s' GROUP BY DJIA.date" %(first_date_datetime, second_date_datetime));	26

Table II: Query Trial Results- Before Indexing

Route	Query #	Test 1	Test 2	Test 3	Test 4	Test 5
/keywords	1	0.09795	0.09974	0.09066	0.09189	0.0905
/keywords	2	0.14529	0.11673	0.12606	0.10912	0.10698
/calendar	3	0.02098	0.02049	0.02227	0.03114	0.02201
/calendar	4	0.07962	0.02153	0.02113	0.02131	0.03169
/calendar	5	0.0223	0.02202	0.02057	0.02098	0.02074
/calendar	6	0.04444	0.04551	0.03641	0.03514	0.03579
/calendar	7	0.02083	0.02084	0.07028	0.02057	0.02071
/calendar	8	0.03068	0.03102	0.03069	0.03095	0.02187
/calendar	9	0.02298	0.02211	0.02182	0.02082	0.0209
/calendar	10	0.03093	0.03123	0.03186	0.03146	0.03147
/calendar	11	0.03311	0.02315	0.0313	0.03109	0.03139
/calendar-month	12	0.03091	0.02177	0.02128	0.02105	0.02109
/calendar-month	13	0.03137	0.03229	0.03095	0.03099	0.03008
/calendar-month	14	0.02102	0.02126	0.02132	0.02114	0.02096

/calendar-month	15	0.04623	0.03572	0.0447	0.15338	0.04477
/calendar-month	16	0.02144	0.0212	0.0212	0.02151	0.02137
/calendar-month	17	0.02157	0.02148	0.0298	0.03073	0.02232
/calendar-month	18	0.0307	0.02113	0.02996	0.03057	0.03037
/calendar-month	19	0.03133	0.03136	0.0319	0.03081	0.03176
/calendar-month	20	0.03123	0.03276	0.03068	0.09173	0.08295
/wordcloud	21	0.02935	0.0291	0.03412	0.03434	0.02975
/wordcloud	22	0.02961	0.02982	0.02991	0.02982	0.03415
/presidential_factors	23	0.45781	0.4543	0.44676	0.44388	0.43899
/presidential_factors	24	0.35908	0.345	0.37172	0.35065	0.35227
/presidential_factors	25	0.72355	0.70611	0.70748	0.72267	0.71259
/trends	26	0.05881	0.06031	0.05877	0.06079	0.06014

Table III: Query Trial Results- After Indexing

Route	Query #	Test 1	Test 2	Test 3	Test 4	Test 5
/keywords	1	0.08333	0.08126	0.08229	0.0808	0.08932
/keywords	2	0.10256	0.10145	0.10162	0.10203	0.11628
/calendar	3	0.02098	0.02066	0.02046	0.02046	0.02016
/calendar	4	0.02939	0.02994	0.02964	0.02977	0.02952
/calendar	5	0.01939	0.02021	0.01993	0.01807	0.01832
/calendar	6	0.04065	0.03717	0.03768	0.03816	0.03582
/calendar	7	0.03091	0.02516	0.02605	0.02562	0.02669
/calendar	8	0.03004	0.02615	0.0243	0.02574	0.02545
/calendar	9	0.02065	0.02092	0.01785	0.01865	0.01567

/calendar	10	0.02626	0.02554	0.02658	0.02471	0.02549
/calendar	11	0.02912	0.02978	0.0302	0.02702	0.0295
/calendar-month	12	0.02314	0.01929	0.01949	0.01885	0.02872
/calendar-month	13	0.0298	0.02986	0.02859	0.034	0.02935
/calendar-month	14	0.02081	0.02075	0.02085	0.02188	0.02184
/calendar-month	15	0.03648	0.0326	0.03167	0.03193	0.03608
/calendar-month	16	0.02067	0.02053	0.01952	0.01887	0.02147
/calendar-month	17	0.02079	0.02127	0.01921	0.01987	0.02184
/calendar-month	18	0.02394	0.02392	0.02398	0.02349	0.02032
/calendar-month	19	0.03	0.03191	0.03213	0.02731	0.02839
/calendar-month	20	0.03534	0.03545	0.03279	0.0339	0.03392
/wordcloud	21	0.02998	0.03007	0.0305	0.02948	0.03071
/wordcloud	22	0.02964	0.02985	0.03154	0.02916	0.02992
/presidential_factors	23	0.25781	0.4543	0.24676	0.24388	0.23899
/presidential_factors	24	0.25908	0.245	0.27172	0.25065	0.35227
/presidential_factors	25	0.52355	0.50611	0.50748	0.52267	0.51259
/trends	26	0.03311	0.10903	0.03145	0.03771	0.03527

Table IV: Query Optimization Results

Query #	Before Avg (sec)	After Avg (sec)	Differential	Percent Change
1	0.094148	0.0834	-0.0107	-11.42
2	0.120836	0.104788	-0.0160	-13.28
3	0.023378	0.020544	-0.0028	-12.12
4	0.035056	0.029652	-0.0054	-15.42

5	0.021322	0.019184	-0.0021	-10.03
6	0.039458	0.037896	-0.0016	-3.96
7	0.030646	0.026886	-0.0038	-12.27
8	0.029042	0.026336	-0.0027	-9.32
9	0.021726	0.018748	-0.0030	-13.71
10	0.03139	0.025716	-0.0057	-18.08
11	0.030008	0.029124	-0.0009	-2.95
12	0.02322	0.021898	-0.0013	-5.69
13	0.031136	0.03032	-0.0008	-2.62
14	0.02114	0.021226	0.0001	0.41
15	0.06496	0.033752	-0.0312	-48.04
16	0.021344	0.020212	-0.0011	-5.30
17	0.02518	0.020596	-0.0046	-18.20
18	0.028546	0.02313	-0.0054	-18.97
19	0.031432	0.029948	-0.0015	-4.72
20	0.05387	0.03428	-0.0196	-36.37
21	0.031332	0.030148	-0.0012	-3.78
22	0.030662	0.030022	-0.0006	-2.09
23	0.448348	0.288348	-0.16	-35.68
24	0.355744	0.275744	-0.08	-22.48
25	0.71448	0.51448	-0.2	-27.99
26	0.059764	0.049314	-0.0105	-17.49

ii. Caching

The second component of performance optimization that we considered was caching. We used flask_caching to create a SimpleCache object for our app.
We chose SimpleCache (rather than another form of caching) because SimpleCache is used for single-process environments. SimpleCache exists mainly for the development server and is not 100% thread safe; it keeps the items stored in the memory of the Python interpreter. Since our data is not static (not regularly updated yet), there is no chance of users facing read-write or write-write conflicts, so we chose the simplest form of caching that we could.

7. Extra Credit Features

We implemented 3 extra credit features.

The first was hosting our Relational Database System on Amazon Web Services, as described in earlier sections.

The second was using the Google News API to pull top current headlines onto our home page. To do this, we applied for a developer account with Google News. We chose Google News because it seemed like the most reliable, easy-to-use news API that we could find.

Once we were approved and received an API key, we followed the documentation to obtain a JSON object of the most recent news data (including title, author, datetime stamp, news source, URL…). From there, we had to parse the JSON object to obtain only the top 10 article titles. We displayed these titles on the bottom of our web app's home page.

Finally, we have hosted our web app on Heroku, allowing users to access the app from any computer: https://cis550project-api-heroku.herokuapp.com/.