

TwitterFollow Cloud Project

CAB432: Cloud Computing

Michael Leontieff, n9455396
Anneke Kotze, n9451013

Introduction	3
Use Cases	4
Architecture & Services	4
Development Phases	5
Architecture & Implementation	5
High Level Overview	5
Architecture Components	6
Twitter Stream	6
Single running stream per Application	6
Filters cannot be altered whilst stream is open	6
Pre-Cache Data Alterations	6
In Memory Cache	6
Stream Caching	6
Application State Management	7
Language Classification	7
APIs & Services	8
Twitter Stream	8
Natural Library for Node	8
Redis Cache/DataStore	9
Deployment process	9
VM Images	9
Redis Cache	9
Application Node	9
Azure Template Configuration	10
Template	10
Microsoft.Network/publicIPAddresses	10
Microsoft.Network/loadBalancers	11
Microsoft.Network/networkSecurityGroups	11
Microsoft.Resources/deployments	11
Microsoft.Compute/virtualMachineScaleSets	11
Microsoft.Insights/autoscaleSettings	11
Parameters	11
Cloud Init	12
Scaling & Performance	12
A Typical Worker Node	13
Most Taxing Operation	14
Scaling Methodology Chosen	14

User Count	14
Filter Count	14
Filter Popularity	14
Scaling Situations	14
Minimal Load - Single Node Operation	15
Medium to High Load - >1 Node Operation	15
Workload-Request Load Correlation	15
Scaling Graphs	16
Testing & Limitations	16
Future Extensions	18
Appendices	19
Appendix A: User Guide	19
Appendix B: Adding a Search Filter	20
Appendix C: Removing a Search Filter	21
Appendix D: Adding Multiple Search Filters	22
Appendix E: Removing Multiple Search Filters	23
Appendix F: Detail View	24
Appendix G: Stopping Data Update	25

Introduction

The intention of this application is to provide a user with the ability to search for and track a selection of manually entered hashtags or search terms on Twitter. The resulting data will then be displayed in two forms. The primary form of display is a summary bubble graph (a screenshot is below) that will show the general sentiment and popularity of a given query in comparison to other queries.

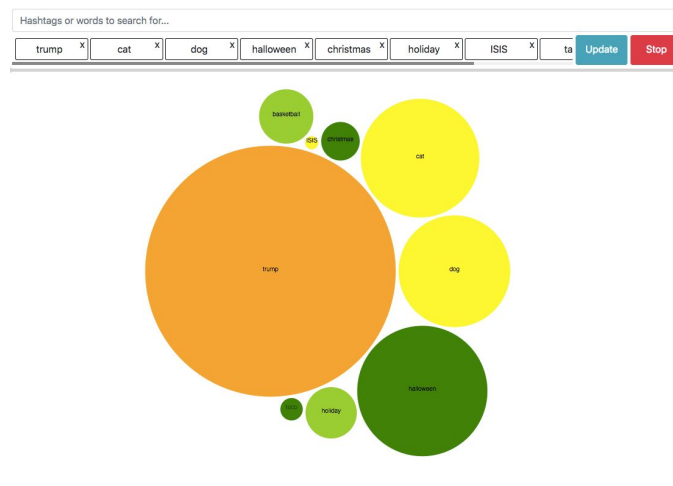


Figure 1 - Screenshot of Bubble Graph

The second form of display is on an individual search term level. This will allow the user to select one of the search terms and then see further information for that query term. This extra data will include a stream of tweets that is added to as more tweets are processed, along with the results of the sentiment analysis done on them. An example of this is shown below:

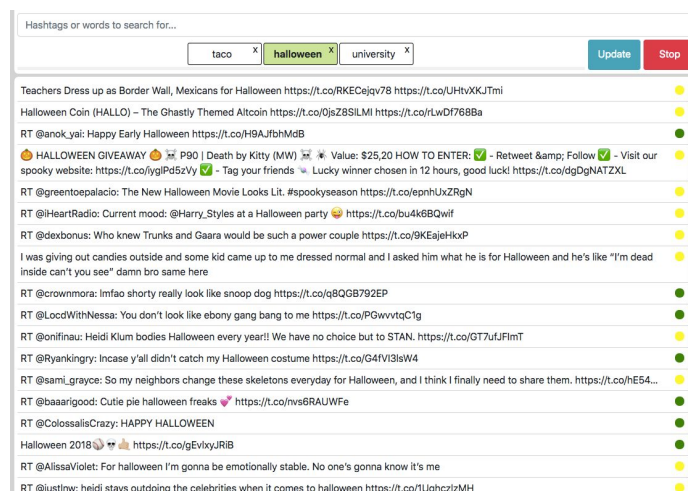


Figure 2 - Screenshot of Detail View

Use Cases

The primary use case is when a user wishes to determine the general sentiment and popularity of a number of terms in a specific category. An example is supermarket brands. The user could enter the terms Coles, Woolworths, Aldi, Foodworks, and IGA, and then they will see the overall sentiments and number of tweets about each brand. This would allow the user to see which brands are more commonly tweeted about and whether or not the tweets are generally positive or negative. As a secondary use case, a user could also enter a number of terms that are less closely linked in order to compare general popularity of different things, such as football or rugby compared to e-sports, or comparing different holidays (as seen below).

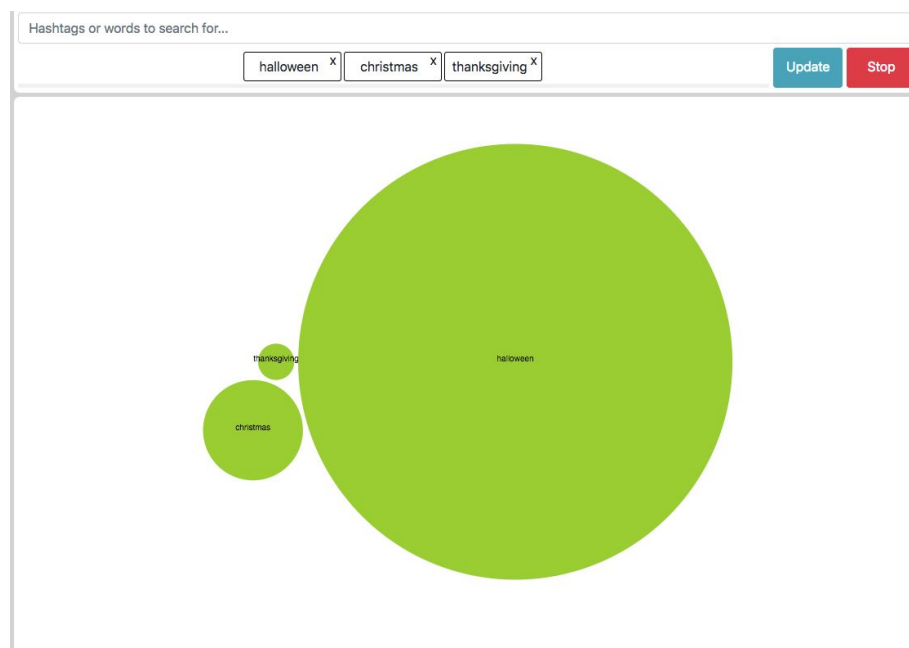


Figure 3 - Example Use Case

Architecture & Services

The general architecture of the application is shown in the diagram below.

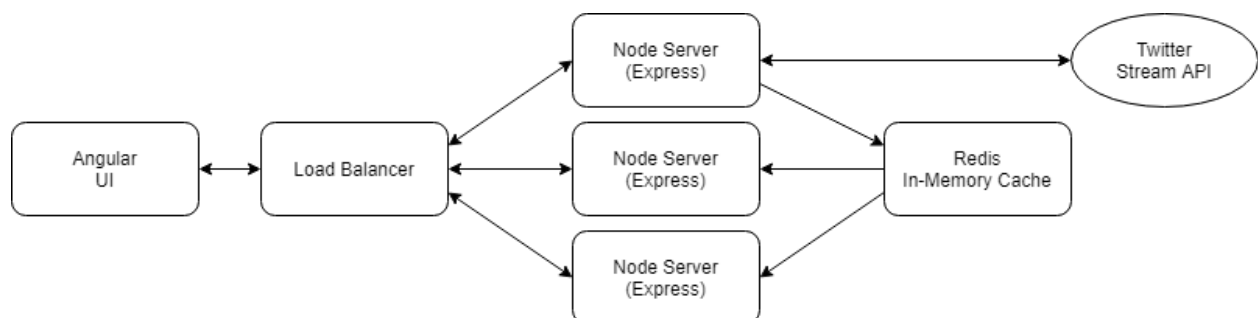


Figure 4 - Architecture of Application

The application consists of an Angular UI served through a browser, statically served by a Node server using the Express framework, which uses a Redis in-memory cache for data storage. The Node server communicates with the Twitter API in order to find tweets matching given search terms, which are then saved for later access in the Redis cache. When requested by the UI, each server that receives a request via the load balancer will pull a number of records from the cache for processing. Each tweet is processed using the Natural library for Node in combination with a pre-trained model to ascertain the sentiment associated with that tweet. The UI makes use of d3.js, a data display javascript library, to display a bubble graph representing the summarised sentiment and total number of processed tweets returned by the Node server for each search term.

Development Phases

The development approach taken was to divide the application into two sections, the UI and the server, and develop these independently of each other. Even though the development was done independently, it was a priority to ensure that the entire application was able to be dockerized at all times. This division of development was chosen as the only interaction between the two components is via an API on the server. This also made adding a load balancer a simpler operation. Once the UI and server were completed and able to communicate in a locally running docker container, the scaling could be implemented. This is a required step, as each instance in the scale set will pull and run the image so increase the level of automation in the dev pipeline. Testing was undertaken throughout the project in order to maintain a higher quality application and to reduce potential issues towards the completion of the project.

Architecture & Implementation

High Level Overview

In summary, the application consists of an Angular front end, which is statically served by a node server that uses the Express Framework. This UI interfaces with a collection of /api/ REST endpoints exposed by our application to allow the client to initiate and terminate stream processing. This is in addition to supplying the resulting, processed data for display by the graphing library.

Once a stream is initiated, the client repeatedly polls the server for updates, whereby each update request will process a chunk of tweets stored in the cache. This results in the load being directly tied to a request, which itself is stateless and may be directed to any node in a scaling group to be resolved without issue.

Architecture Components

This application is a composition of services that deliver the defined use case, these components are described below.

Twitter Stream

The Twitter Stream is the data source of the application, of which all of the processing hinges on. A connection to the stream exhibits many characteristics that need to be catered for in the application architecture, of which are described with mitigation below:

Single running stream per Application

An application may only have a single running stream, which in turn requires the application to maintain a state of active stream connections so that nodes don't initiate connections if one is already running. Without these measures, access to the Twitter stream may be interrupted.

Filters cannot be altered whilst stream is open

The filter supplied to a stream connection cannot be changed whilst running, meaning that if a user changes their query, existing streams need to be terminated before be re-initiated with the updated parameters. This extends the requirement for a shared stream state, with the requirement for a node to be able to request the termination of a stream that is currently running (either on that node or another within the scaling group).

Pre-Cache Data Alterations

Before being stored in the cache, each tweet is stripped of meaningless data and serialised as a JSON string for storage. This also primes the data for the language classification phase.

In Memory Cache

The In-Memory cache takes the form of the persistence pillar in the application, and provides two main functionalities to support the experience.

Stream Caching

To support the torrent of data outputted by the Twitter stream, the application employs an in-memory cache to facilitate flow control for the downstream processing that is performed on the data. This allows all filter-matching tweets to be processed on-demand, allowing the language processing to be performed on tweets after they have been returned by the filter. In-memory was chosen as opposed to alternatives as it allows for higher read/write thresholds to better support the consuming nodes and the incoming stream.

A second key reason is to facilitate horizontal scaling, as the in-memory cache provides a common point of call for data to be used by all nodes in a scaling group.

Application State Management

The requirements for horizontal scaling introduce complexities that are in part tackled by the in-memory pillar. Application State Management refers to the need to maintain a shared state across the nodes so that they may perform their operations without stepping on each other. An example of this is the storing of the twitter stream state, whereby any node may determine if a stream is running, and if so it's current state, that being either *RUNNING*, *TERMINATING* or *TERMINATED*.

With the ability for each node to poll the shared state, this allows any node to initiate and terminate a connection to the Stream, without knowing what node the stream resides on. A diagram of this is supplied below.

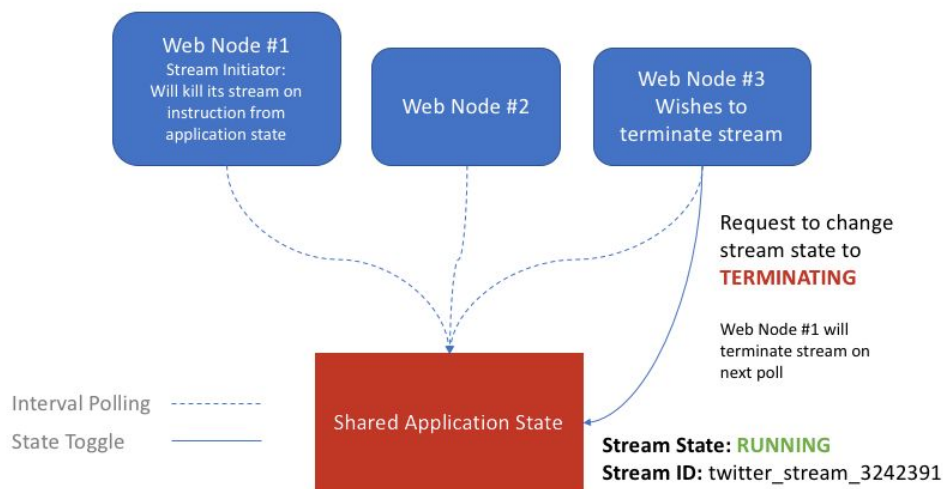


Figure 5 - Shared State Operation Diagram

Language Classification

To deliver the use case described, the incoming data from the Twitter Stream needs to undergo classification to dictate the polarity of the tweet. This is performed against a pre-trained model - a model which has been constructed from a pre-classified data set of tweets sourced online.

The pre-trained data exhibits three potential labels: Positive, Negative and Neutral with the incoming tweets from the stream being classified into these groups using the Naive Bayes classifier as provided by the Natural library for NPM.

APIs & Services

Each component of the Application, as described above is underpinned by a Service that provides the core functionality of the component. Details of these Services are described below:

Twitter Stream

Link: <https://www.npmjs.com/package/twitter>

The Twitter stream is consumed through the use of the Twitter NPM library, which in turn is wrapped in a singleton and instantiated upon request. This instance creates a connection to the *statuses/filter* stream with an input configuration supplied by the initiating request.

```
public createStream(filters: string[], cacheInstance: CacheService) {
  if (!TwitterServiceAPI.twitterStream) {
    TwitterServiceAPI.twitterStream
      = TwitterServiceAPI.twitterClient.stream('statuses/filter', {
        track: filters.join(","),
        language: "en"
      });

    // not actually coupled to the stream, signifies presence rather than link of/to a stream
    TwitterServiceAPI.twitterStreamId = this.createTwitterStreamId();
    // it's a promise but we don't care - as long as it's set
    cacheInstance.setStreamStatus(StreamStates.RUNNING);

    // Interval poll for termination flag
    let interval = setInterval(() => {
      cacheInstance.getStreamStatus().then(streamStatus => {
        if (streamStatus === StreamStates.TERMINATED) {
          console.info("Twitter stream terminated");
          if (TwitterServiceAPI.twitterStream) {
            TwitterServiceAPI.twitterStream.destroy();
            TwitterServiceAPI.twitterStream = null;
          }
          clearInterval(interval);
        }
      });
    }, 100);
  }
}
```

Figure 6 - Twitter Stream Sample Code

Natural Library for Node

Link: <https://www.npmjs.com/package/natural>

To process the incoming stream data, text classification will be applied against the data using the Natural library for Node. This module allows for the parsing and training of pre-classified data which may be used as a model to label unclassified text.

Redis Cache/DataStore

Link: <https://redis.io/>

Redis was chosen as the underlying technology to supply the In-Memory Cache. This is due to the data structures it supports and its cache-like behaviour. The latter of which complemented our requirements for operations like popping data from the cache, and associating an expiry to each record to prevent staleness.

Deployment process

In keeping with the DevOps methodology, the deployment process has been streamlined to be largely automated, allowing for quick rollouts of functionality and bug fixes. This works in tandem with the requirements for deploying scaling groups, such as generalised images and Cloud Init configuration.

VM Images

To deploy this application in a functioning state, two VM images are required. The process of creating each image involves provisioning a new VM, installing the software pertaining to the role of the VM in the architecture and then generalising and imaging the VM for later use. The Images featured in our deployment are as follows:

Redis Cache

This image contains a pre-compiled and installed redis service, which was installed within the `/etc/` directory so as to persist after the user deprovisioning process performed during image creation.

Application Node

This image contains a docker installation, which is used to pull a particular containerised version of the application from docker hub and then run the container with arguments sourced from the Cloud Init scripts.

Azure Template Configuration

To manage the abundance of configuration that needs to be performed, all configuration pertaining to scale rules, health probes and security are expressed through both a template and parameters JSON file.

Template

The template is the schema for all of the resources required by the application. The template is a high level view of the deployment and is devoid of fine-grained parameters such as instance counts and scaling CPU boundaries. To alleviate errors, the template.json file contains a variables object, which allows for a single source for application-specific identifiers that in turn will reduce errors.

```
{
  "type": "Microsoft.Network/networkSecurityGroups",
  "apiVersion": "[variables('networkApiVersion')]",
  "name": "[variables('networkSecurityGroupName')]",
  "location": "[parameters('location')]",
  "properties": {
    "securityRules": [
      {
        "name": "allowSSH",
        "properties": {
          "description": "Allow SSH traffic",
          "protocol": "Tcp",
          "sourcePortRange": "*",
          "destinationPortRange": "22",
          "sourceAddressPrefix": "*",
          "destinationAddressPrefix": "*",
          "access": "Allow",
          "priority": 1000,
          "direction": "Inbound"
        }
      }
    ]
  }
},
```

Figure 7 - Template snippet

The template contains three primary fields of interest: *parameters*; which state what configuration settings can be substituted in and their data type, *variables*; the common points of reference mentioned earlier and finally *resources*; which summarises each resource that is required for deployment. These resources types, roles and configuration are listed below:

Microsoft.Network/publicIPAddresses

Manages public IP's for the load balancer and for worker nodes during development testing. In a production environment, worker nodes will not be directly exposed to the web.

Microsoft.Network/loadBalancers

Describes the backend pools and health probes required for automated scaling.

Microsoft.Network/networkSecurityGroups

States the ports exposed to the web, including SSH, HTTP and DNS to resolve docker pull issues in the latest Ubuntu LTS.

Microsoft.Resources/deployments

Describes the virtual network and subnets required for the application

Microsoft.Compute/virtualMachineScaleSets

Describes the scale set, including the network profile and coordination with supporting resources.

Microsoft.Insights/autoscaleSettings

Describes the metrics to be monitored and the min/max scaling.

Parameters

With the template describing the dependencies of the service, the parameters contain the deployment-specific configuration. This abstraction allows the deployments variables to be manipulated without dealing with the clutter of the resource schema.

This is represented through a collection of key, object-value pairs that associate a value with a variable used in the template.json file.

```
"autoscaleMax": {  
  "value": "5"  
},  
"autoscaleMin": {  
  "value": "1"  
},  
"scaleInCPUPercentageThreshold": {  
  "value": "20"  
},  
"scaleInInterval": {  
  "value": "1"  
},
```

Figure 8 - Parameters snippet

Cloud Init

Having created VM Images with the required software for the application, Cloud Init is used to start the relevant services on the instances after provisioning has succeeded. Examples of these scripts are included in the appendix.

```
#!/usr/bin/env bash

# link systemd dns config to resolv.conf
sudo ln -s /run/systemd/resolve/resolv.conf /etc/resolv.conf

# Fetch a particular docker image and run with config
# run as daemon to prevent cloud init exec timeout
sudo docker run -p 80:8000 \
  --env redis_host=10.1.0.9 \
  --env redis_port=6379 \
  -d mleontieff/cloud_computing_two:2.1
```

Figure 9 - Parameters snippet

Scaling & Performance

The cloud-oriented architecture of the application is designed to address multiple scaling situations, described below is the high-level infrastructure diagram, which exhibits the worker/web nodes with respect to the memory cache and the requesting user.

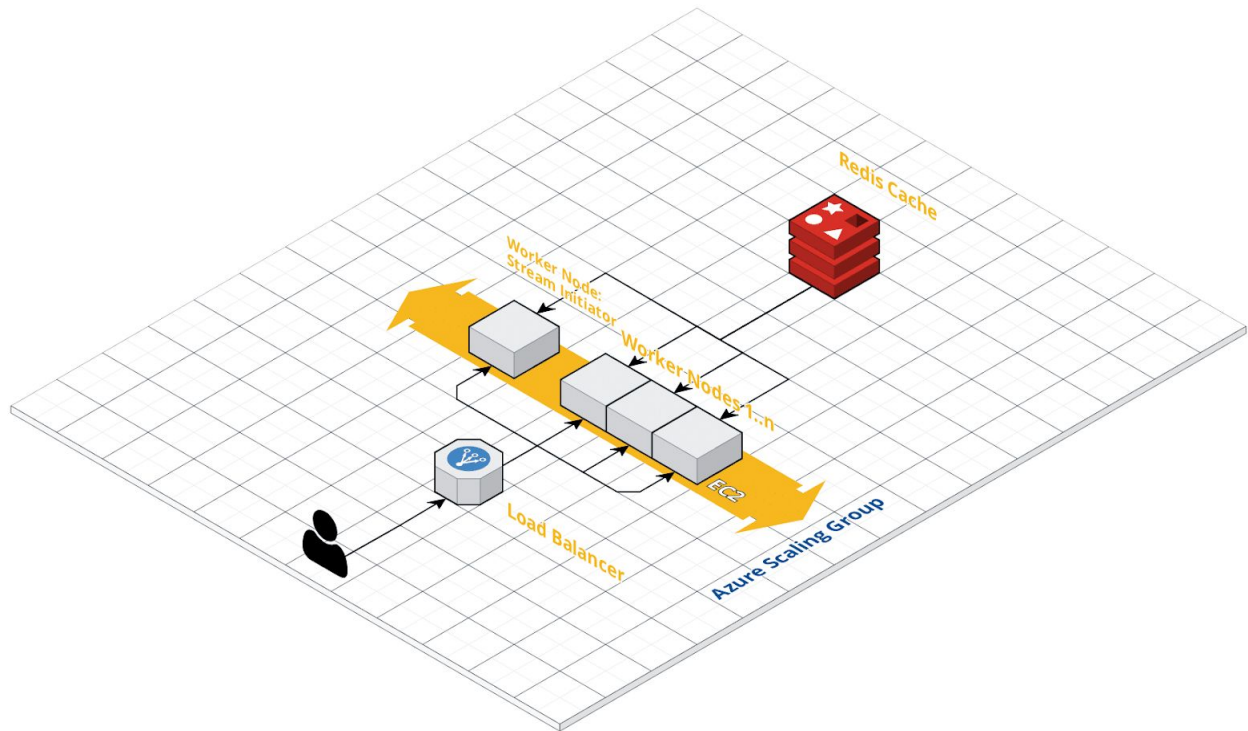


Figure 10 - Infrastructure Diagram

A Typical Worker Node

Load balancing is delivered through the horizontal scaling of the nodes within the Azure auto-scaling group. Each node represents a complete replication of the application, both front-end and /api/ endpoints. When these nodes are combined with a connection to the memory cache, they allow for a functioning application.

The replication is intentional, as it allows for any request, whether that be regarding client-side page load or api endpoints to be resolved by any node in a stateless manor. No vertical or horizontal scaling is applied to the memory cache currently, as the load imparted upon the cache is too insignificant to warrant it (<5% CPU load on default VM with app at highest cache read/writes).

Most Taxing Operation

The most demanding operation within the application is the classification of the cached data. Therefore, the goal of the application architecture is to revolve the scaling component around the need to distribute this work to any number of nodes without issue.

With an active stream running against a single trending topic, the CPU load imparted upon a single B1MS instance is roughly 50%. With the option to add up to 250 filters the potential for scaling is significant.

Scaling Methodology Chosen

It was decided that automated scaling offerings provided by Azure would best fit this application as they perfectly align with the request-based load architecture that is utilised.

This is because the load is driven by multiple factors, factors those combined scenarios make for a very unpredictable load situation that cannot be solved through predictive, non-reactive scaling. These factors are as follows:

User Count

Each user of the application will be initiating a constant stream of polls for updates, and therefore the load will scale with respect to the user count.

Filter Count

The load generated by an individual will also vary, based upon the number of filters being queried for. Queries with more filters aren't necessarily more intensive however, as popularity may vary.

Filter Popularity

Each filter being processed may not impart the same processing cost, in the case of less popular topics the interval polling will process the cache faster than it can be populated. In addition, the results yielded by a filter will fluctuate based upon current events and other factors.

For example, it was observed in testing that the trending filter *Halloween* was netting thousands of tweets per minute on the 31st, however, a couple of days later the popularity had slumped to only several hundred in the same time period.

Scaling Situations

To address these characteristics, several scaling situations need to be accounted for by the application and underlying architecture in order to deliver a consistent user experience.

Minimal Load - Single Node Operation

In single node operation, the scaling group contains a single node which addresses all requests by incoming clients. Given there's only a single node, it will both supply the static client-side content, host the stream connection, push results to cache and finally perform text classification upon the data for return to clients.

Medium to High Load - >1 Node Operation

A node exhibits the same characteristics in >1 node operation as if it were a singular instance, being able to deliver the entire experience. The application architecture has been tailored to this situation, and as such the architecture allows for even distribution of work loads across all nodes in the group. This is achieved in the following ways.

Workload-Request Load Correlation

The core of the applications scalability is the idea that any /api/ endpoint request may be resolved by any node in the application. This affords the application the ability to leverage the load balancer to distribute requests to these endpoints and have any node respond to it without issue through this process:

1. Client will repeatedly poll endpoint
2. Each request will pop an arbitrary number of tweets from the cache, classify them, and update the summary object stored on the cache with the results.
3. When this is complete, the summary object is returned.

Therefore, processing of the tweets is solely driven by client requests, allowing for severe cost-savings regarding infrastructure as it is the clients that initiate and maintain the behind-the-scenes classification of data.

Scaling Graphs

The following graph represents the load imparted upon a scale set and its subsequent nodes throughout a 45 min period. This confirms that the methodology selected regarding scaling is functioning correctly for our application.

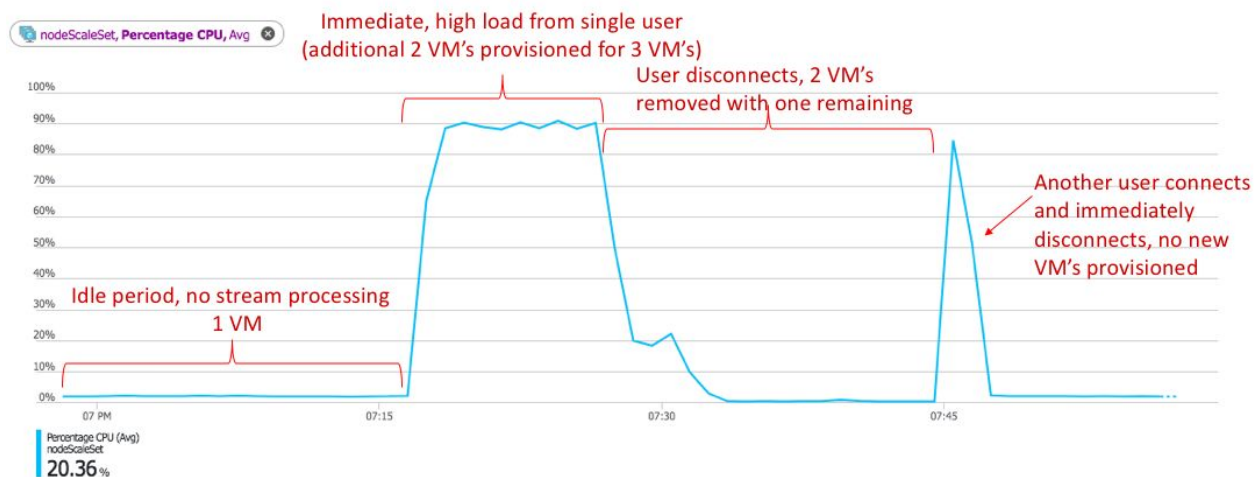


Figure 11 - Scaling Graph

Testing & Limitations

As part of ensuring a high-quality application, a large amount of testing was required. This confirmed the functionality was correct and provided through a usable and intuitive user interface. A summary table of the testing undertaken is below:

Test Case	Expected Result	Actual Result
Adding a search filter	The new filter should show up in the result graph	The graph shows the new filter once the user presses 'Update' and after the next server poll returns See Appendix B
Removing a search filter	The old filter should be removed from the result graph	The graph no longer shows the removed filter once the user presses 'Update' and the next server poll occurs See Appendix C
Adding multiple search filters	Each new filter should be added to the graph all at once	The graph shows all new filters added once the user

	after the user presses 'Update'	presses 'Update' and the next server poll occurs See Appendix D
Removing multiple search filters	Each removed filter should be removed from the graph all at once after the user presses 'Update'	The graph shows the remaining filters once the user presses 'Update' and the next server poll occurs See Appendix E
Selecting a filter to see more detail	The graph should be hidden and a new view should open that shows a selection of current tweets and their sentiment	The graph view is hidden and the new view with processed tweets & sentiments opens after the next server poll See Appendix F
Stopping the update of data	The data should stop being updated and no more requests should be sent	The graph view is cleared after the user presses 'Stop' See Appendix G

Overall, the results of the test matched the expected outcome well. There were some limitations in regards to server-polling. Most notably, the frequency with which this can be done is less than ideal. However, if this frequency is increased it is far more likely for the user's browser tab or window to become unresponsive due to the sheer frequency of requests sent. This is because each time the server is polled for updated data, an individual request is sent for each search term. This is required as returning all possible data for each API request would result in an unsustainable load on a single server as it would need to process all data currently stored in the in-memory cache. Splitting each request also allows for load-balancing to be done more effectively.

The main compromise made in the architecture of the application is the reliance on a single stream that needs to be killed and recreated as soon as the filters are changed. This is less efficient and increases the risk of multiple users dramatically impacting performance. Unfortunately, the alternative approach of using Twitter REST endpoints would have resulted in an excessive number of duplicate requests being made, with a higher financial cost per request, making it infeasible for the purposes of this application.

The main impact of this compromise on the user experience is the requirement for the user to press the 'Update' button (as can be seen in Figure 8 below) before the stream will be updated with the new search terms. This results in a slightly less responsive application, however it ensure more accurate results are displayed and assists in keeping the behaviour of the application consistent, even when the user performs many actions quickly.



The screenshot shows a search interface. At the top is a text input field with the placeholder text "Hashtags or words to search for...". Below this input field is a row of seven buttons, each containing a search term and a small 'x' icon to its right. The search terms are: "taco", "halloween", "university", "basketball", "rugby", "football", and "holiday". To the right of these buttons are two more buttons: a blue "Update" button and a red "Stop" button.

Figure 11 - Screenshot of Search Section

Future Extensions

The functionality provided by the application is sufficient for the primary use case of comparing the popularity of hashtags and search terms to each other. However, there is ample room for improvement and extension of features. Some examples of these extensions are: storing the gathered data in a database rather than a temporary data structure to allow the user to see changes in popularity and overall sentiment over time rather than only for the duration of the cache; the ability to group search terms under a common heading and compare to other groups of search terms, for example comparing 'sports' to 'e-sports', with sub-filters of 'football' and 'rugby', and 'DOTA' and 'League of Legends' respectively; or, the ability to do historical analysis of a specific search term to see changes over the history of Twitter, not just the lifetime of the application's datastore.

Appendices

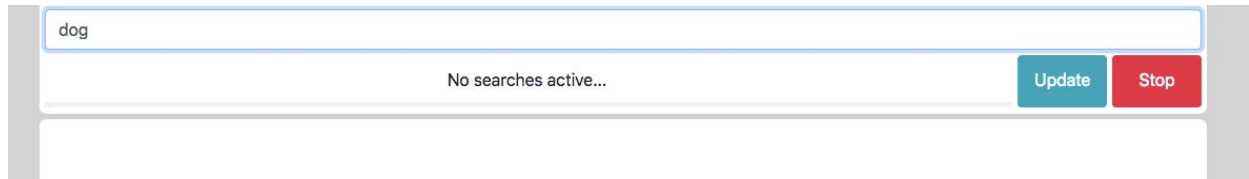
Appendix A: User Guide

In order to deploy the application, follow these steps:

1. Log into azure-cli and navigate to the “deployment” directory of the repository
2. Execute the “deploy-memory-cache.sh” script to deploy the redis VM.
3. Adjust the parameters.json file to contain the redis connection information then run the “deploy-scale-set.sh” script.
4. Once deployment completes, access the application through the IP address of the Load Balancer.

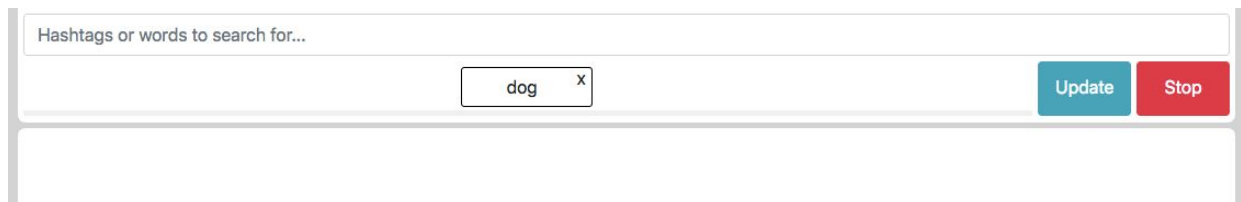
Appendix B: Adding a Search Filter

The user entered 'dog' into the search field



A search interface with a text input field containing 'dog'. Below the input field, the text 'No searches active...' is displayed. To the right of the text are two buttons: 'Update' (teal) and 'Stop' (red).

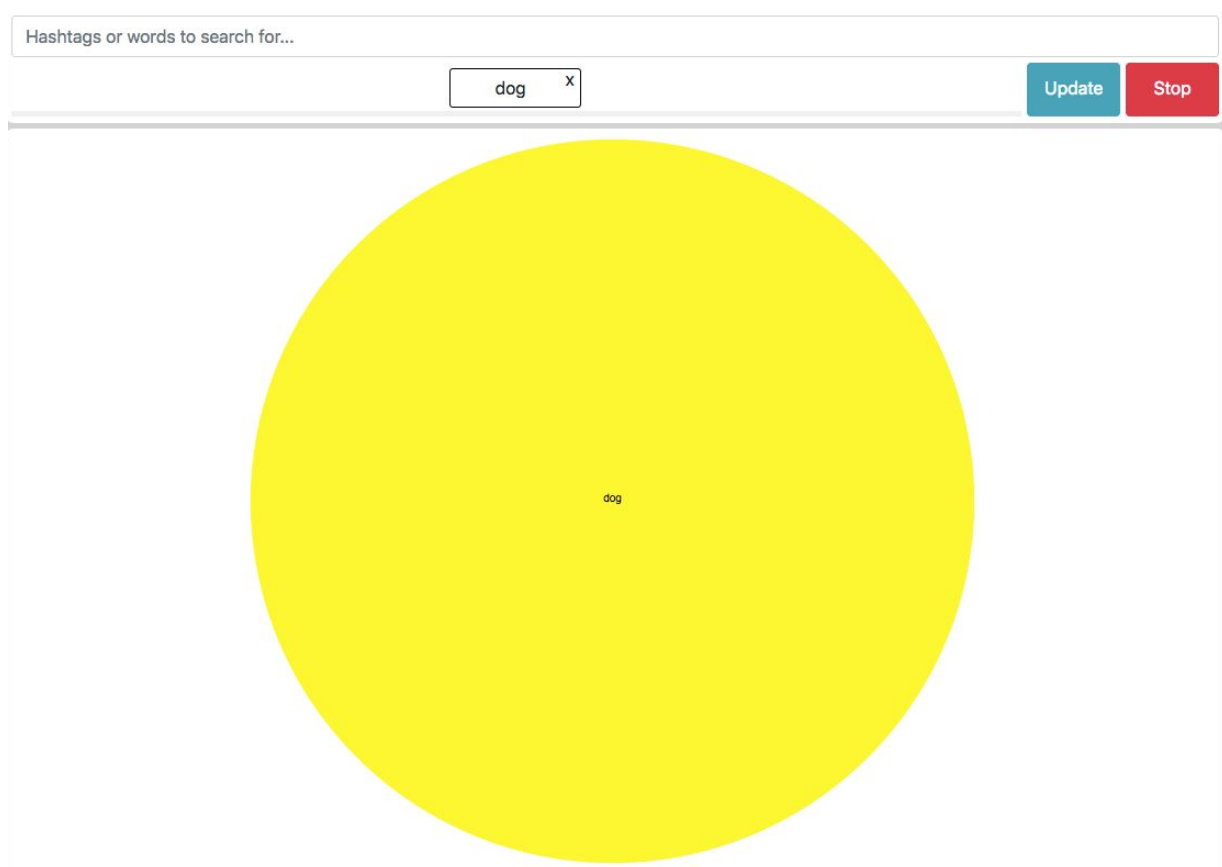
The user pressed Enter



A search interface with a text input field containing 'dog'. Below the input field, the text 'No searches active...' is displayed. To the right of the text are two buttons: 'Update' (teal) and 'Stop' (red). The 'Update' button is highlighted with a blue border.

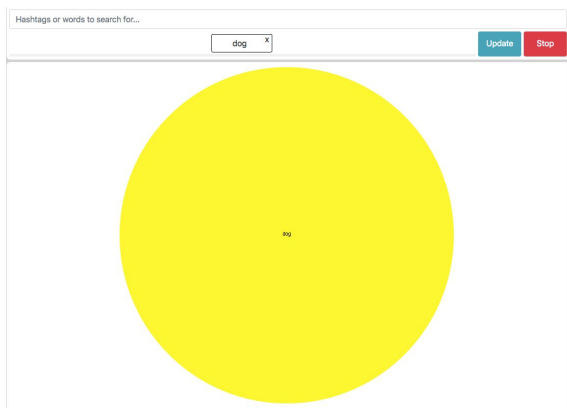
The user pressed the 'Update' button

There was a 1 second wait



The graph was displayed

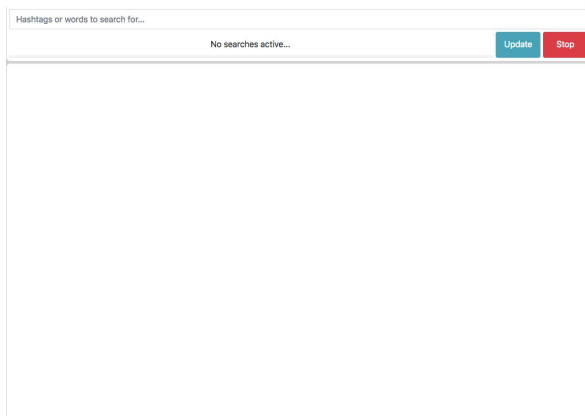
Appendix C: Removing a Search Filter



The user pressed the 'x' on the search term



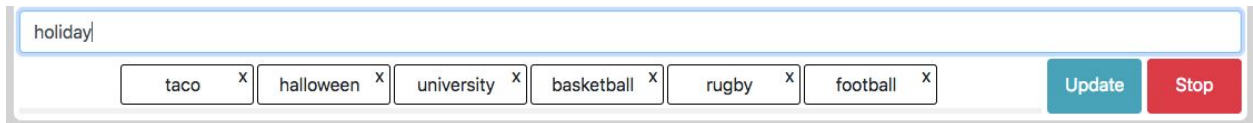
The user pressed the 'Update' button
There was a less than 1 second wait



The graph was cleared

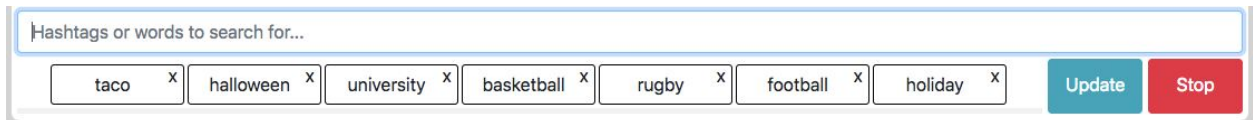
Appendix D: Adding Multiple Search Filters

The user entered the search terms into the field



Search interface showing the initial state. The search field contains "holiday". Below the field, there are filter buttons for "taco", "halloween", "university", "basketball", "rugby", and "football", each with an "x" to remove it. To the right are "Update" and "Stop" buttons.

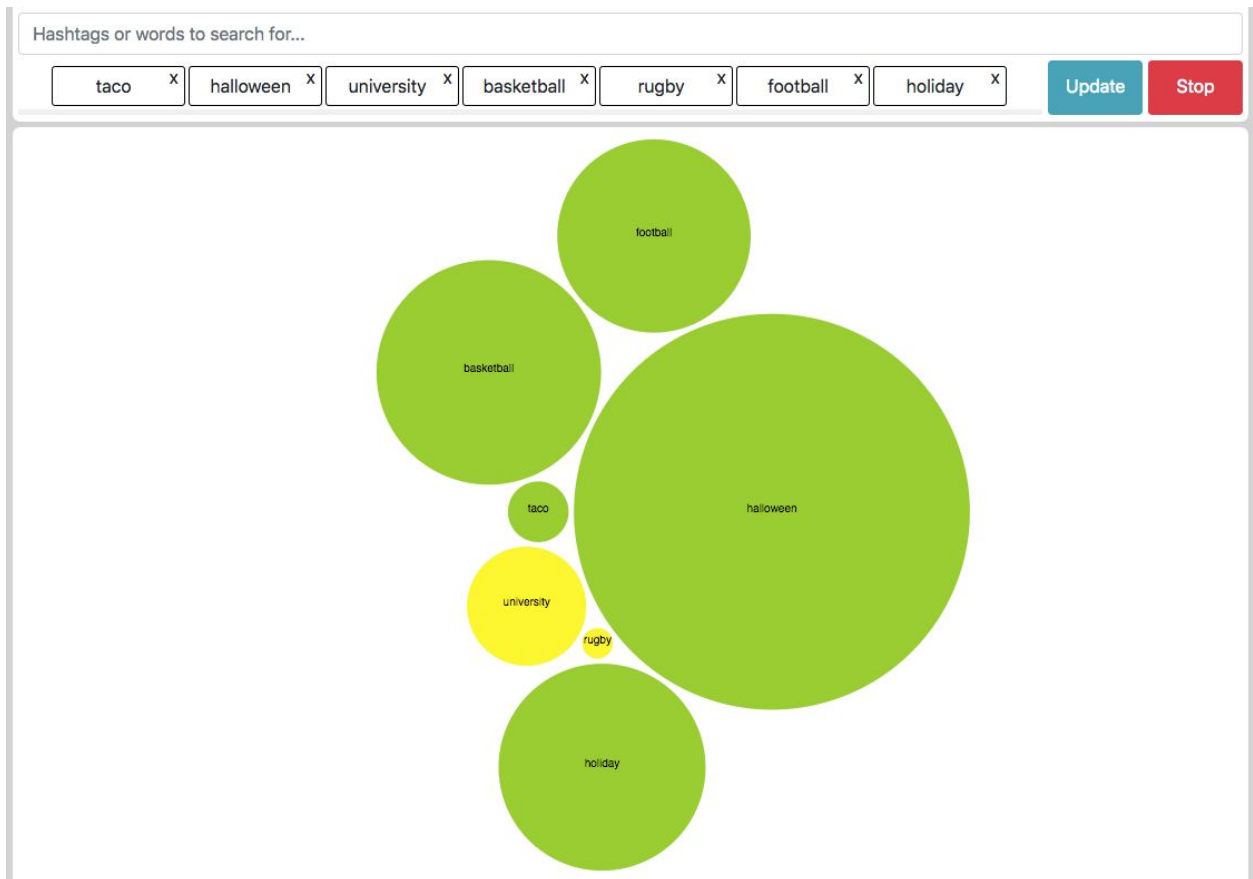
The user pressed Enter



Search interface after pressing Enter. The search field now contains "Hashtags or words to search for...". The filter buttons now include "taco", "halloween", "university", "basketball", "rugby", "football", and "holiday", each with an "x" to remove it. The "Update" and "Stop" buttons remain.

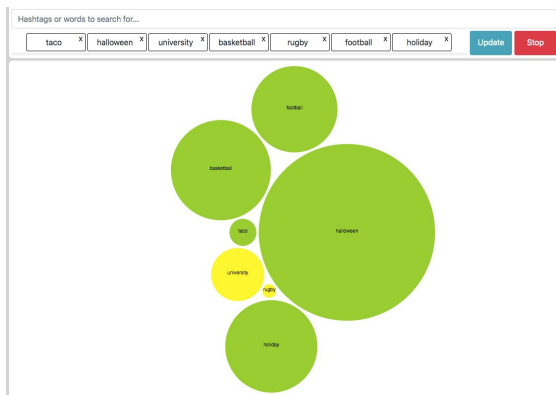
The user pressed the 'Update' button

There was a 1 second wait

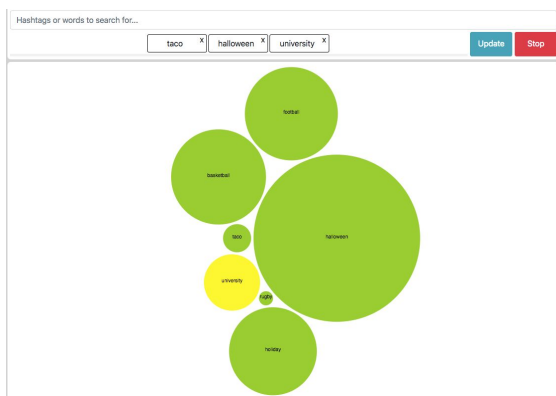


The graph was displayed

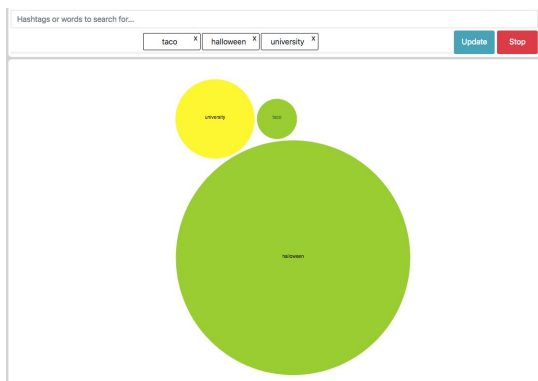
Appendix E: Removing Multiple Search Filters



The user pressed the 'x' on the search terms to remove

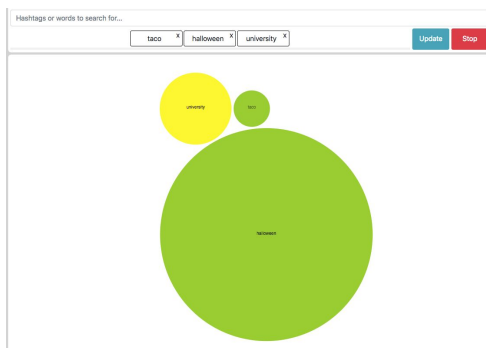


The user pressed the 'Update' button
There was a 1.5 second wait

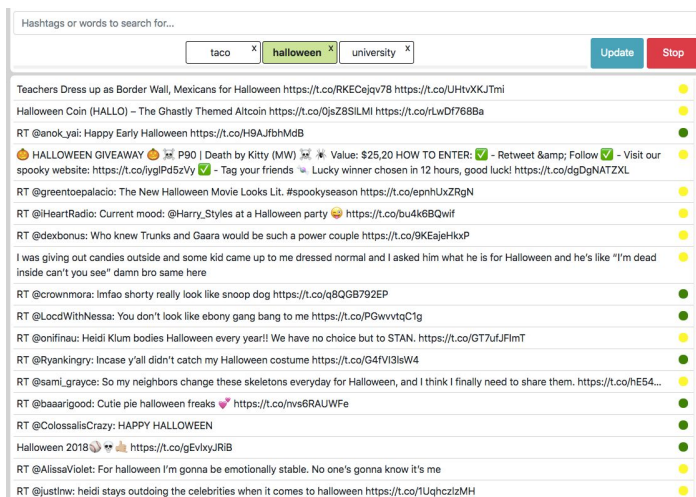


The graph was updated

Appendix F: Detail View



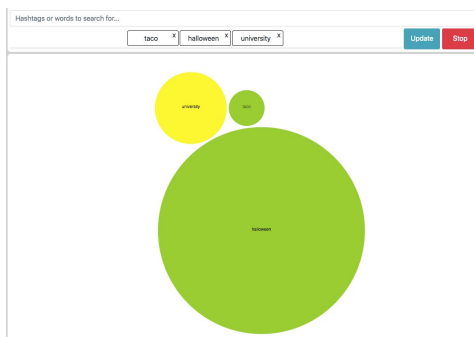
The user selected 'halloween'



There was basically no delay

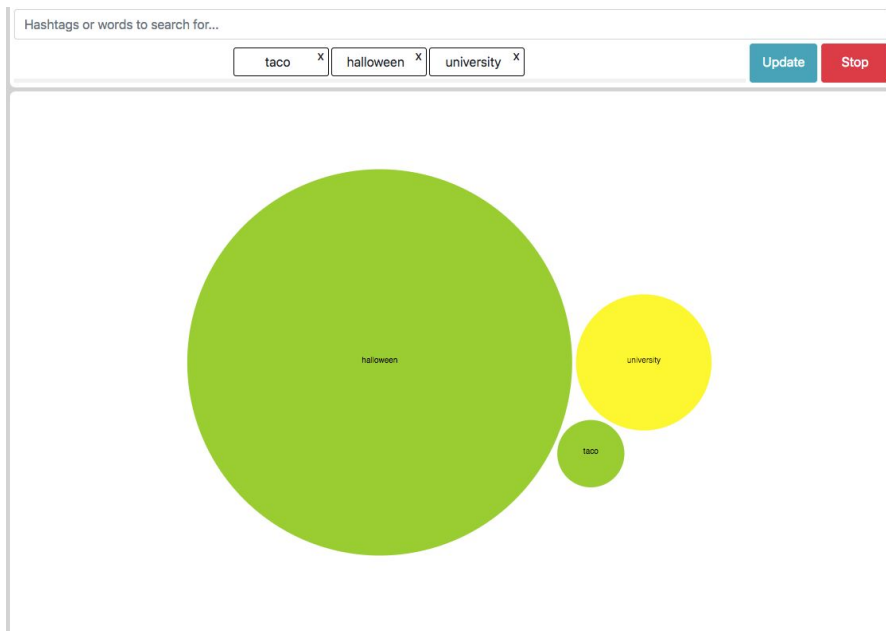
The list continued to update with tweets processed

The user clicked on 'halloween' again

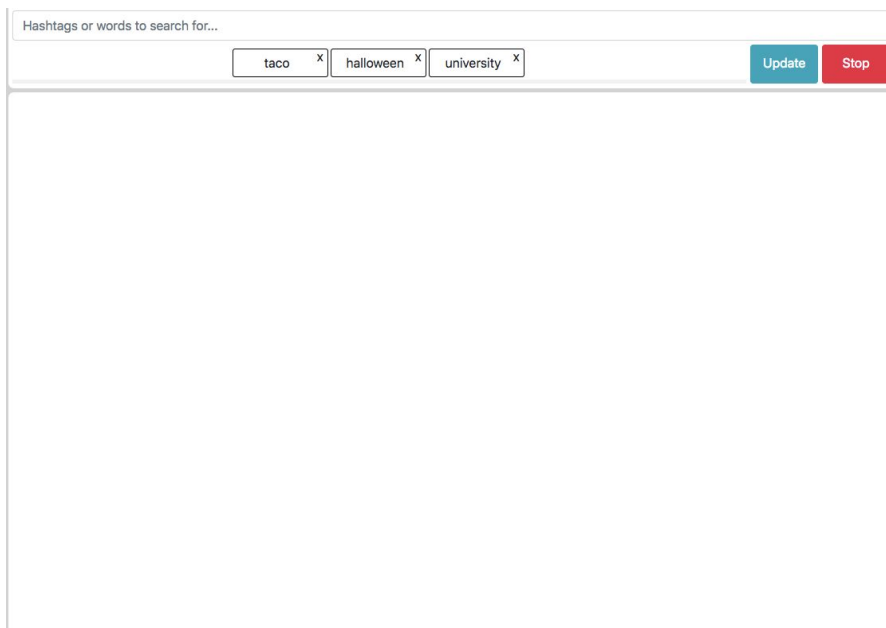


The graph was redisplayed

Appendix G: Stopping Data Update



The user pressed the 'Stop' button
There was a 2 second delay



The graph was removed