

Supporting Visualizations on Large Twitter Datasets Using Cloudberry

Shengjie Xu

June 2017

Advisor: Chen Li

Department of Computer Science

Abstract

Cloudberry is a powerful middleware layer that supports fast analytics on big data, and TwitterMap is a web application that utilizes Cloudberry to offer interactive visualization on huge datasets from Twitter.

This paper will focus on the implementation details of TwitterMap's new features such as query result normalizations, content sentiment analysis, and live tweets count. These details should guide developers to support visualization on very large datasets using Cloudberry, and hopefully, inspire them to take full advantage of Cloudberry to create distinct and innovative analytical applications on big data.

Table of contents

Introduction.....	3
Development Guide.....	5
1. Normalization.....	5
1.1 Data Crawling.....	9
1.2 Data Cleaning and Integration.....	11
1.3 Data Ingestion.....	13
1.4 Data Schema Registration.....	16
2. Sentiment Analysis.....	18
3. Live Tweets Count.....	21
References.....	24
Acknowledgement.....	25

Introduction

Cloudberry is a powerful middleware layer which excels in supporting real-time complex analytical operations on huge datasets (Figure 1). Furthermore, it is a general-purpose system so it can be configured to run on different databases and front-end applications.

TwitterMap is a frontend application developed by the Cloudberry team to demonstrate the capability of Cloudberry (Figure 2). It offers map-based interactive visualization on social media data from Twitter. Specifically, when the user asks for the popularity of certain keywords, it responds with a heat map which shows the count of all the tweets that mention these keywords. Most importantly, thanks to the optimizations offered by Cloudberry, it can finish analysis on hundreds of millions of records and send responses back to the viewers within a few seconds. Additionally, the search results are always up-to-date as new tweets are ingested into the back-end database every day.

As Cloudberry receives new updates, the TwitterMap evolves accordingly to test and show the new changes. New features of the TwitterMap include normalization, sentiment analysis, and live count. They account for Cloudberry's "lookup," "append," and "estimable" APIs respectively. To fully utilize these APIs on TwitterMap, it requires changes on the frontend written in JavaScript, the configurations of Cloudberry, and datasets stored in the Apache AsterixDB. So, it is worthwhile to talk in depth about these details since other developers may experience the same situations when using Cloudberry to build different applications.

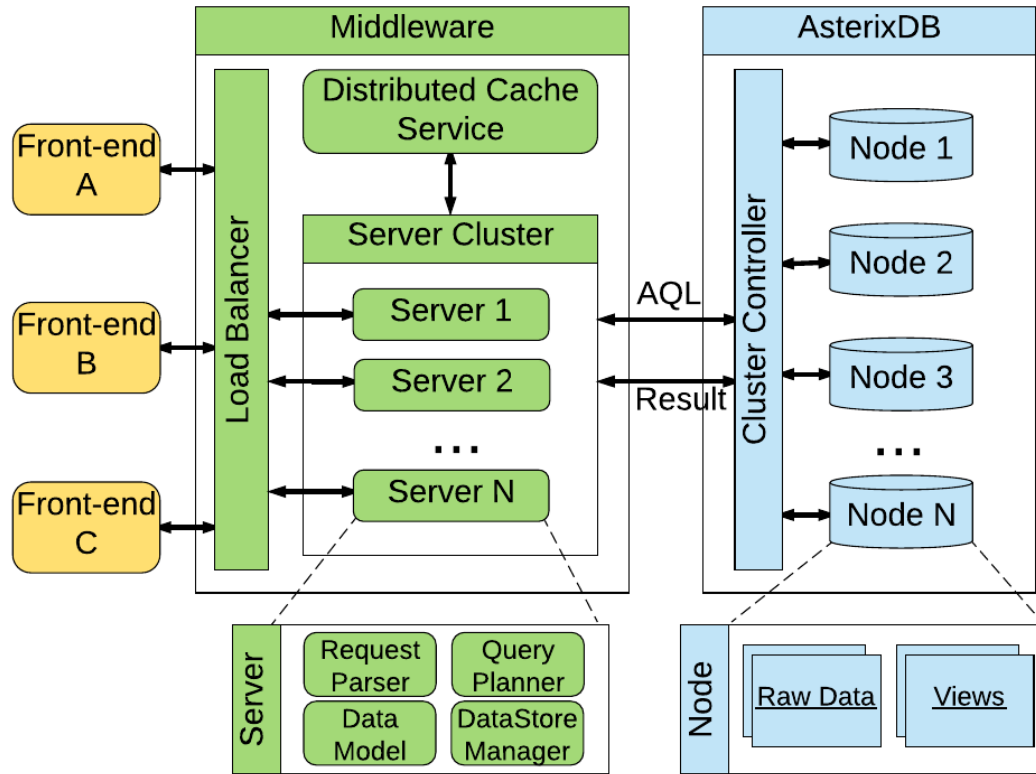


Figure 1: A Generic Architecture of Cloudberry

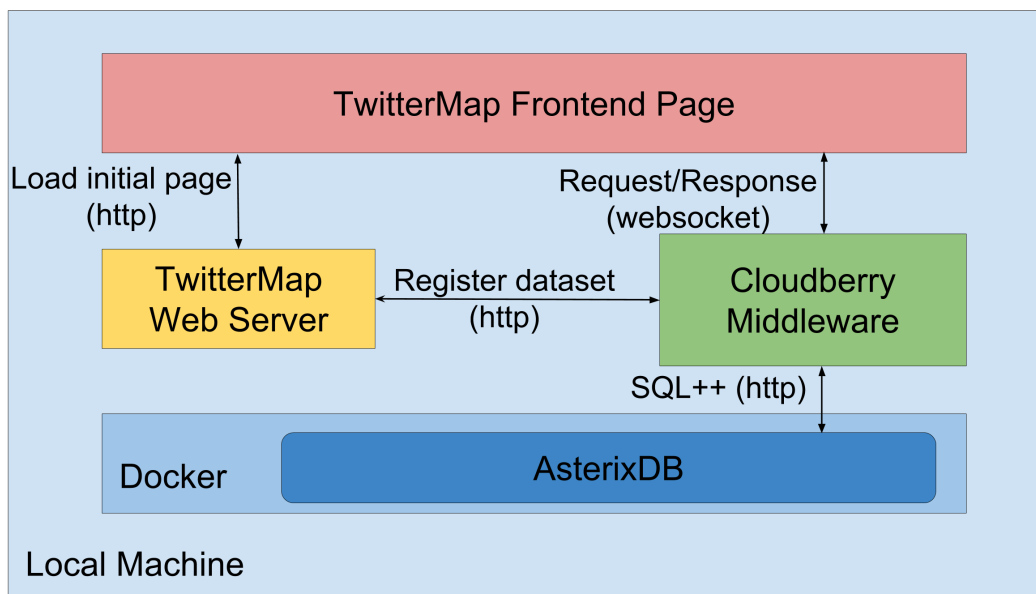


Figure 2: The Architecture of TwitterMap Which Uses Cloudberry and Apache AsterixDB

Development Guide

1 – Normalization

This feature offers more insights of the data by incorporating the demographic context into the analytics. Originally, TwitterMap can only show the count of all the tweets that mention the keywords. Now, the count is normalized by the population of the associated geographical location. For example, the population in California is approximately 39.1 million. If the count of all the tweets that mention the keyword “trump” is 391,000 in California, the normalized count will be $391,000 / 39.1 = 10,000$. It means that there are 10,000 tweets per one million people. See Figure 3 and 4 for demonstrations.

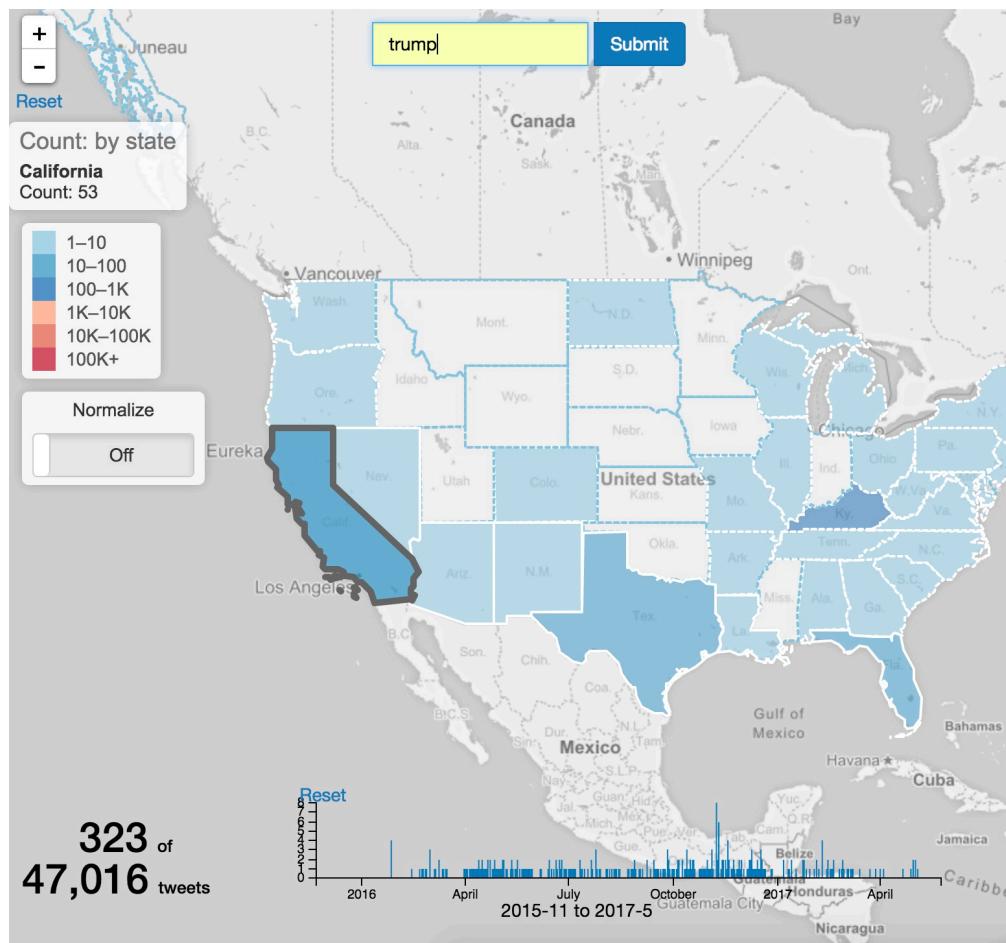


Figure 3: The Raw Count of All the Tweets that Mention “Trump”

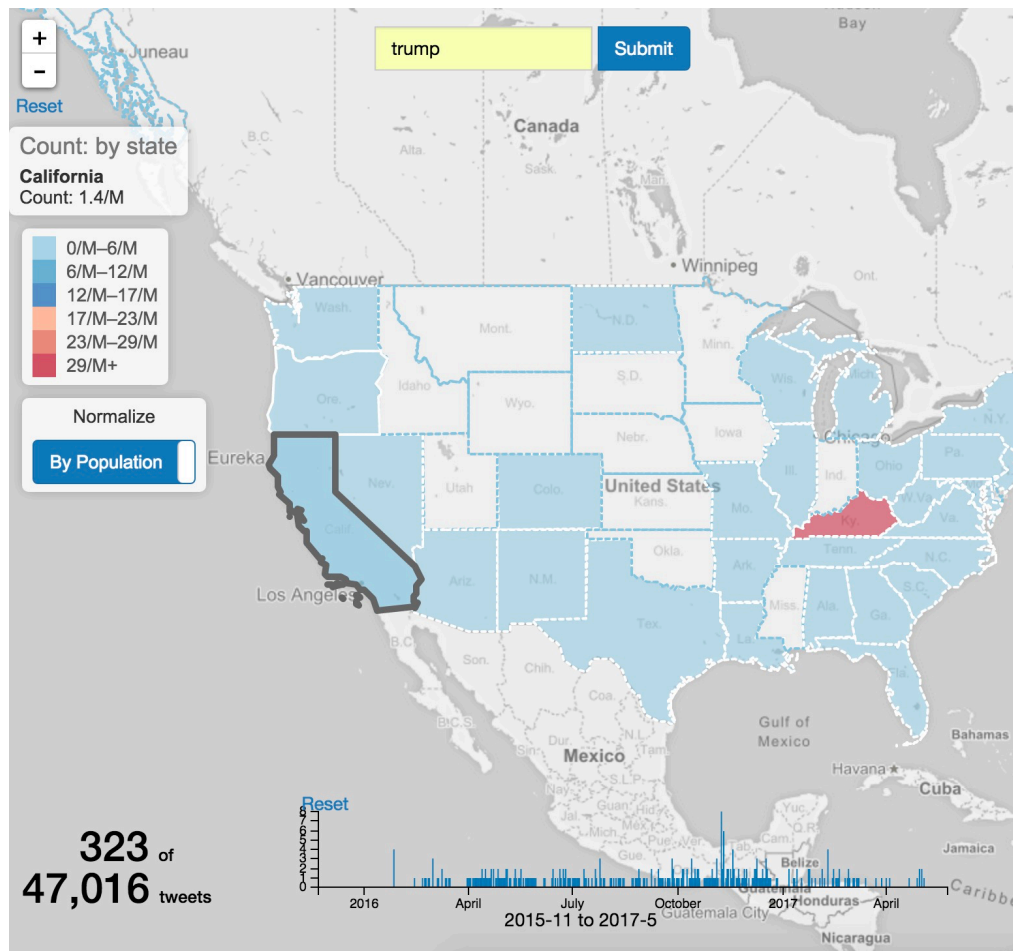


Figure 4: Normalized by Population Showing the Count of Tweets per 1 Million People

This feature uses Cloudberry’s “lookup” API, which enables the middleware to retrieve information from multiple datasets in a single query. Specifically, it offers a quick look up for the grouped results using some lookup datasets. Normally, these datasets store factual information such as population and their cardinalities tend to be way smaller than the main dataset.

Firstly, the frontend needs to add the “lookup” field in the query JSON and specify how the lookup dataset should join the main dataset (Figure 5 & 6).

```
{
  dataset: parameters.dataset,
  filter: getFilter(parameters, defaultNonSamplingDayRange),
  group: {
    by: [{
```

```

    field: "geo",
    apply: {
      name: "level",
      args: {
        level: parameters.geoLevel
      }
    },
    as: parameters.geoLevel
  }],
  aggregate: [{
    field: "*",
    apply: {
      name: "count"
    },
    as: "count"
  }],
  lookup: [
    cloudberryConfig.getPopulationTarget(parameters)
  ]
}
}

```

Figure 5: Sample TwitterMap Query with Lookup

```

getPopulationTarget: function(parameters){
  switch (parameters.geoLevel) {
    case "state":
      return {
        joinKey: ["state"],
        dataset: "twitter.dsStatePopulation",
        lookupKey: ["stateID"],
        select: ["population"],
        as: ["population"]
      };
    case "county":
      return {
        joinKey: ["county"],
        dataset: "twitter.dsCountyPopulation",
        lookupKey: ["countyID"],
        select: ["population"],
        as: ["population"]
      };
    case "city":
      return {
        joinKey: ["city"],
        dataset: "twitter.dsCityPopulation",
        lookupKey: ["cityID"],
        select: ["population"],
        as: ["population"]
      };
  }
}

```



```
}
}
```

Figure 6: Helper Function to Specify the Population Dataset

For example, in Figure 5, when the “geoLevel” is “state,” the lookup dataset “dsStatePopulation” will perform a left outer join with the main dataset “ds_tweet” using the “state” field as the join key. After that, the result dataset will use the “stateID” field as the lookup key. Finally, it will output the “population” field. The details about setting up lookup datasets will be discussed in detail in Sections 1.1, 1.2, 1.3, and 1.4.

Notice that the “lookup” is inside the “group by” field (Figure 5). It means that Cloudberry will group all the tweets by its “geo” first, and then for each group, it will look up the population dataset using the “geo” of the group as a key to retrieve the population. For example, the “geo” for a state is “stateID,” which has been set up as a lookup key when joining the lookup dataset with the main dataset. A sample response is shown in Figure 7. Without calling this API, the “population” field will not appear in these results.

```
[
  {
    "state": 37,
    "count": 1,
    "population": 10146788
  },
  {
    "state": 13,
    "count": 3,
    "population": 10310371
  },
  {
    "state": 21,
    "count": 1,
    "population": 4436974
  }
]
```

Figure 7: Sample Result with Lookup

The “lookup” API is very versatile. Its future applications on the TwitterMap can be looking up on user statistics table to normalize the count by the total number of users or check a dictionary table to filter out curse words before displaying some of the most popular tweets.

1.1 – Data Crawling

The lookup dataset is about the US population. It is crawled from <https://www.citypopulation.de/>. The crawler is implemented using Scrapy, a fast yet simple Python web crawling framework (<https://scrapy.org/>). The source code of the crawler can be found in the Cloudberry repository under [noah/src/main/resources/population/crawler](#). The key to effectively crawling desired data is to analyze and determine the target information. For instance, the existing county geo data has the county name, county ID, and the same information of the state it belongs to (Figure 8). Since the ID is self-generated by the database, it cannot be crawled from the data source (Figure 9). Therefore, the target information must include both the county name and state name in order to provide more context when matching the population data to the geo data.

```
{  
  "area": 594.436,  
  "name": "Autauga",  
  "stateName": "Alabama",  
  "stateID": 1,  
  "geoID": "0500000US01001",  
  "countyID": 1001  
}
```

Figure 8: One Piece of Existing County Geo Data

Home → America → USA				
USA: Alabama				
Counties				
The population of all counties in the State of Alabama by census years.				
Name	Status	Population Census 1990-04-01	Population Census 2000-04-01	Population Census 2010-04-01
Autauga	County	34,222	43,751	54,571
Baldwin	County	98,280	140,416	182,265

Figure 9: The Website Only Contains County Name and State Name

The next step is to configure and deploy the crawlers. In the case of Scrapy, developers need to analyze the URL pattern for the “start_requests()” function and identify the CSS selectors of the target HTML elements for the “parse()” function (Figure 10).

```
import scrapy

class CountyPopulationSpider(scrapy.Spider):
    """
    A customized crawler inherited from the scrapy.Spider base class.
    To run this crawler, install scrapy https://scrapy.org/, and run:
    scrapy crawl county_population -o county_population.json
    """
    name = "county_population"
    def start_requests(self):
        state_list = ['alabama', 'alaska', 'arizona', 'arkansas',
'california',
                    'colorado', 'connecticut', 'delaware',
'districtofcolumbia',
                    'florida', 'georgia', 'hawaii', 'idaho', 'illinois',
'indiana',
                    'iowa', 'kansas', 'kentucky', 'louisiana', 'maine',
'maryland',
                    'massachusetts', 'michigan', 'minnesota',
'mississippi', 'missouri',
                    'montana', 'nebraska', 'nevada', 'newhampshire',
'newjersey', 'newmexico',
                    'newyork', 'northcarolina', 'northdakota', 'ohio',
'oklahoma', 'oregon',
                    'pennsylvania', 'rhodeisland', 'southcarolina',
'southdakota',
                    'tennessee', 'texas', 'utah', 'vermont', 'virginia',
```

```

'washington',
    'westvirginia', 'wisconsin', 'wyoming']
base_url = 'https://www.citypopulation.de/php/usa-census-'
url_list = []
for state in state_list:
    url_list.append(base_url + state + '.php')
for url in url_list:
    yield scrapy.Request(url, self.parse)

def parse(self, response):
    state_name = response.css('header.citypage h1
span.smalltext::text').extract_first()
    for county in response.css('article#table section#adminareas
table#tl tbody tr'):
        yield {
            'county_name': county.css('td.rname span
a::text').extract_first(),
            'county_population':
county.css('td.rpop.priol::text').extract_first(),
            'state_name': state_name
        }

```

Figure 10: Crawler for County Data

1.2 – Data Cleaning and Integration

Real world data is messy. After crawling, it is almost always necessary to clean the data. For example, the population is in the string format with commas separating digits, which is not good for computations (Figure 11). Another common problem is the encoding of the data as some strings contain non-ASCII letters. For instance, Doña Ana is a county in New Mexico. Its letter “ñ” is interpreted as a Unicode character “\u00f1” in the crawled results but the default representation for all the Unicode characters in the existing the geo data is “?”. Therefore, rigorous data cleaning methods need to be applied to help match crawled data to the existing geo data.

```

{
    "county_name": "Autauga",
    "county_population": "54,571",
    "state_name": "Alabama"
}

```

Figure 11: Crawled Data

After data cleaning, the next step is to integrate the population data (Figure 11) with the existing geo data (Figure 8) by tagging the population data with IDs. For example, state population will need to add a “stateID” and county population will need to add both a “countyID” and a “stateID”. The final result is shown in Figure 12.

```
{
  "countyID": 1001,
  "name": "Autauga",
  "population": 54571,
  "stateID": 1,
  "stateName": "Alabama"
}
```

Figure 12: Final Cleaned and Integrated Population Data

Taking the county as an example, the starting point for the integration is to construct a Python dictionary from the crawled data (Figure 13) and iterate through each existing county geo record (Figure 8) and look up its population using its “name” and “stateName”. Most of the time the combination of the two will produce a unique result, but there exist a few counties having the same state names as well. So, the dictionary should contain some mechanism to detect duplicates, which in this case is the “duplicate” field in Figure 13. In the case of the population data, the duplicates are so few that they can be handled manually.

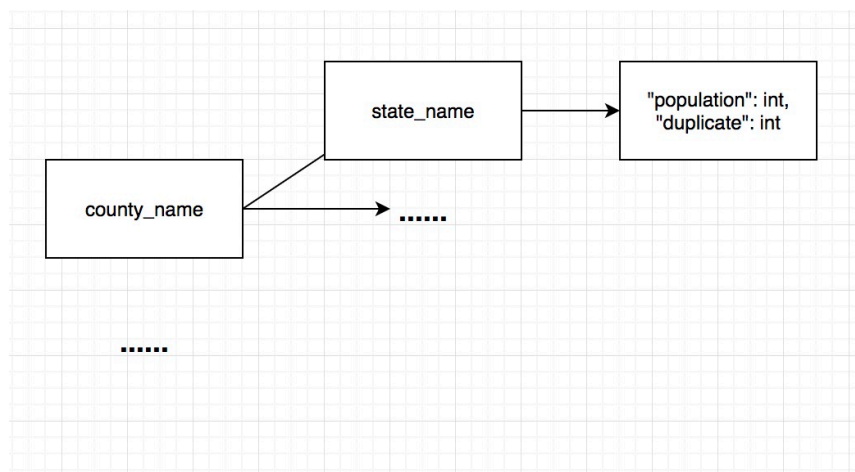


Figure 13: The Dictionary for the County Population

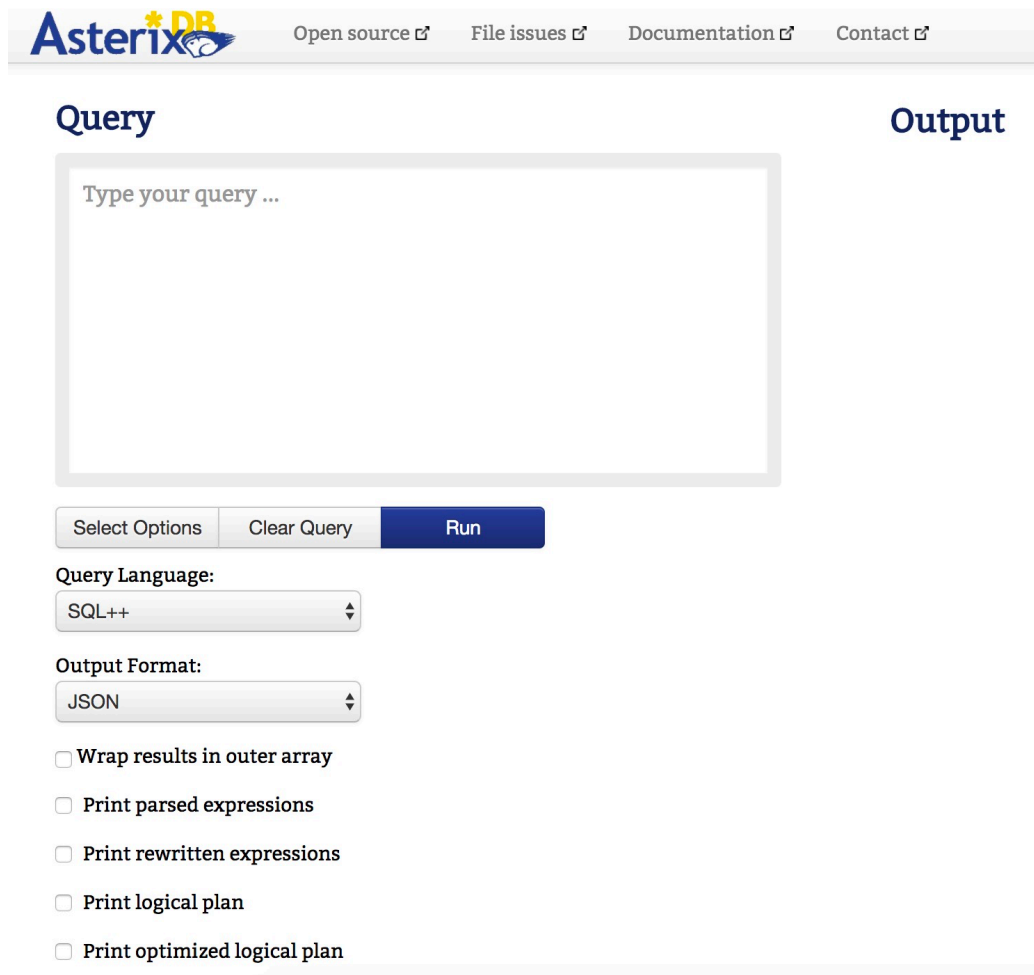
The real challenge is to handle special cases. For example, the existing geo data includes Puerto Rico and all of its counties and cities but they do not exist in the data source. So, developers need to find other data sources to fill the gap. Another common issue is that quite a few city-level population data contain multiple county names because they may belong to more than one county. To make it worse, some cities or counties have changed their official name at some point in history, which makes it impossible to automate the integration without manually looking up their old names on the Internet. Approximately, there are 2,000 special cases in total. However, compared to the rest 31,000 records which have been correctly integrated, these special cases only account for about 6%. Therefore, one feasible solution to handle them is to estimate their populations from their upper-level geo region. For instance, a city's population can be approximated using the average population of its county, and a county's population can be estimated using the average population of its state. Therefore, developers can move on to focus on other tasks that are more important.

The data cleaning scripts can be found in the Cloudberry repository under [noah/src/main/resources/population/data_cleaning](https://github.com/Cloudberry/noah/src/main/resources/population/data_cleaning).

1.3 – Data Ingestion

After pre-processing the raw data, developers need to ingest the data into the database, which in the case of TwitterMap is the Apache AsterixDB. The query language it uses is SQL++ and it is documented in <https://ci.apache.org/projects/asterixdb/index.html>. Developers can go to the AsterixDB console (Figure 14) at localhost:19001 to test their queries. After gaining confidence with the ingestion queries, it is more efficient to integrate them into the existing bash scripts for data ingestion, which is located in the Cloudberry repository under

[/script/ingestAllTwitterToLocalCluster.sh](#). A sample query for ingesting county data is shown in Figure 15.



The screenshot shows the AsterixDB web interface. At the top is the AsterixDB logo and navigation links: Open source, File issues, Documentation, and Contact. Below this is a header with 'Query' and 'Output' tabs. The 'Query' tab is active, showing a large text input area with the placeholder 'Type your query ...'. Below the input area are three buttons: 'Select Options', 'Clear Query', and 'Run'. Under these buttons are two dropdown menus: 'Query Language:' set to 'SQL++' and 'Output Format:' set to 'JSON'. At the bottom, there are five unchecked checkboxes: 'Wrap results in outer array', 'Print parsed expressions', 'Print rewritten expressions', 'Print logical plan', and 'Print optimized logical plan'.

Figure 14: AsterixDB Console Page

```
use twitter;
create type typeCountyPopulation if not exists as open{
  name:string,
  population:int64,
  countyID:int64,
  stateName:string,
  stateID:int64
}
create dataset dsCountyPopulation(typeCountyPopulation) if not exists
primary key countyID;
create index countyIDIndex if not exists on dsCountyPopulation(countyID)
type btree;
-- you need to change '172.17.0.3' to the ip address of the host
-- and change '/home/county_population_cleaned_final.adm' to the path of
the data file
```

```
LOAD DATASET dsCountyPopulation USING localfs
(("path"="172.17.0.3:///home/allCountyPopulation.adm"),("format"="adm"));
-- or use insert if there is little data
-- insert into dsCountyPopulation();
```

Figure 15: SQL++ Queries for Ingesting County Data

Note these ingestion commands work only for the AsterixDB that is setup on a Docker image, whose installation guide can be found on the quick start page in the Cloudberry documentation. Using the Docker image, developers need to copy the population data files from the local file system into the Docker image called “NC1” and change the IP address in the last line of this ingestion command to be the IP address of the “NC1.”

Advanced developers can use the file feed feature of the AsterixDB to make the data ingestion process even simpler (Figure 16).

```
set -o nounset # Treat unset variables as an error
host=${1:-'http://localhost:19002/aql'}
nc=${2:-"nc1"}
# ddl to register the twitter dataset
cat <<EOF | curl -XPOST --data-binary @- $host
use dataverse twitter;
create type typeCountyPopulation if not exists as open{
    name:string,
    population:int64,
    countyID:int64,
    stateName:string,
    stateID:int64
}
create dataset dsCountyPopulation(typeCountyPopulation) if not exists
primary key countyID;
create feed CountyPopulationFeed using socket_adapter
(
    ("sockets"="$nc:10003"),
    ("address-type"="nc"),
    ("type-name"="typeCountyPopulation"),
    ("format"="adm")
);
connect feed CountyPopulationFeed to dataset dsCountyPopulation;
start feed CountyPopulationFeed;
EOF
echo 'Created population datasets in AsterixDB.'
#Serve socket feed using local file
```



```

cat ./noah/src/main/resources/population/adm/allCountyPopulation.adm
| ./script/fileFeed.sh $host 10003
echo 'Ingested county population dataset.'
cat <<'EOF' | curl -XPOST --data-binary @- $host
use dataverse twitter;
stop feed CountyPopulationFeed;
drop feed CountyPopulationFeed;
EOF

```

Figure 16: Using File Feed to Simplify Count Data Ingestion

1.4 – Data Schema Registration

After ingesting datasets into the backend, the last step is to register the data schemas into the middleware to enable Cloudberry's optimization. Developers need to register data schemas such as the one shown in Figure 17 for state population dataset via a POST request to the Cloudberry's registration URL <http://localhost:9000/admin/register>. A more sophisticated schema for the main dataset of TwitterMap is shown in Figure 18.

```

{
  "dataset": "twitter.dsStatePopulation",
  "schema": {
    "typeName": "twitter.typeStatePopulation",
    "dimension": [
      { "name": "name", "isOptional": false, "datatype": "String" },
      { "name": "stateID", "isOptional": false, "datatype":
"Number" },
      { "name": "create_at", "isOptional": false, "datatype":
"Time" }
    ],
    "measurement": [
      { "name": "population", "isOptional": false, "datatype":
"Number" }
    ],
    "primaryKey": ["stateID"]
  }
}

```

Figure 17: Data Schema of the Lookup dataset "dsStatePopulation"

It is a good practice to write only one script to include all the necessary registration requests to simplify the setup process (Figure 19). Alternatively, developers can send registration

requests via the Cloudberry's admin console (Figure 20), <http://localhost:9000/>, but the result is the same as using a bash script.

```
{
  "dataset": "twitter.ds_tweet",
  "schema": {
    "typeName": "twitter.typeTweet",
    "dimension": [
      {"name": "create_at", "isOptional": false, "datatype": "Time"},
      {"name": "id", "isOptional": false, "datatype": "Number"},
      {"name": "coordinate", "isOptional": false, "datatype": "Point"},
      {"name": "lang", "isOptional": false, "datatype": "String"},
      {"name": "is_retweet", "isOptional": false, "datatype": "Boolean"},
      {"name": "hashtags", "isOptional": true, "datatype": "Bag", "innerType": "String"},
      {"name": "user_mentions", "isOptional": true, "datatype": "Bag", "innerType": "Number"},
      {"name": "user.id", "isOptional": false, "datatype": "Number"},
      {"name": "geo_tag.stateID", "isOptional": false, "datatype": "Number"},
      {"name": "geo_tag.countyID", "isOptional": false, "datatype": "Number"},
      {"name": "geo_tag.cityID", "isOptional": false, "datatype": "Number"},
      {"name": "geo", "isOptional": false, "datatype": "Hierarchy", "innerType": "Number"},
      {"levels": [
        {"level": "state", "field": "geo_tag.stateID"},
        {"level": "county", "field": "geo_tag.countyID"},
        {"level": "city", "field": "geo_tag.cityID"}
      ]},
      {"name": "in_reply_to_status", "isOptional": false, "datatype": "Number"},
      {"name": "in_reply_to_user", "isOptional": false, "datatype": "Number"},
      {"name": "favorite_count", "isOptional": false, "datatype": "Number"},
      {"name": "retweet_count", "isOptional": false, "datatype": "Number"},
      {"name": "user.status_count", "isOptional": false, "datatype": "Number"}
    ],
    "measurement": [
      {"name": "text", "isOptional": false, "datatype": "Text"}
    ],
    "primaryKey": ["id"],
    "timeField": "create_at"
  }
}
```

Figure 18: Data Schema of the Main Dataset "ds_tweet"

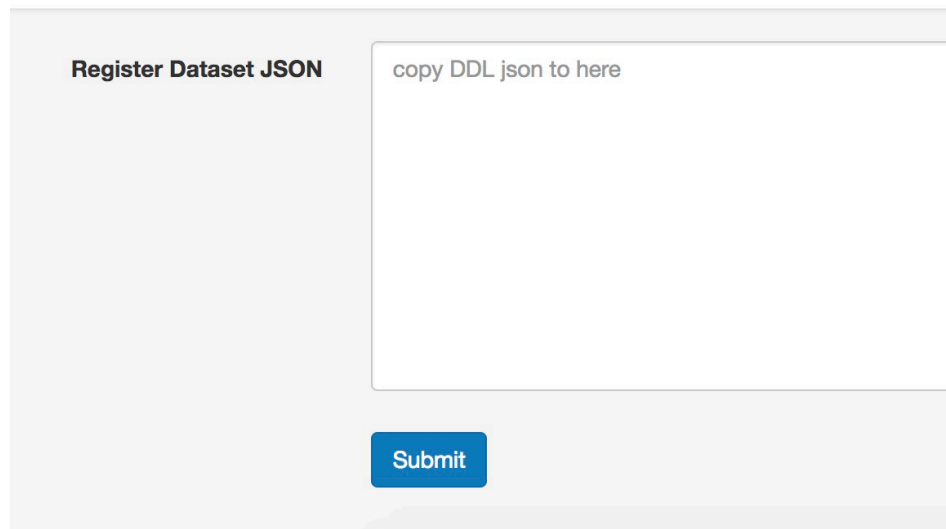
```
curl -d "@./script/dataModel/registerTweets.json" -H "Content-Type: application/json" -X POST http://localhost:9000/admin/register
curl -d "@./script/dataModel/registerStatePopulation.json" -H "Content-
```

```

Type: application/json" -X POST http://localhost:9000/admin/register
curl -d "@./script/dataModel/registerCountyPopulation.json" -H "Content-
Type: application/json" -X POST http://localhost:9000/admin/register
curl -d "@./script/dataModel/registerCityPopulation.json" -H "Content-
Type: application/json" -X POST http://localhost:9000/admin/register

```

Figure 19: Automated Data Schema Registration



The screenshot shows a web interface for registering a dataset. It features a light gray background with a white form area. The form is titled 'Register Dataset JSON' in bold black text. Below the title is a large text input field with the placeholder text 'copy DDL json to here'. At the bottom right of the form is a blue button with the text 'Submit' in white.

Figure 20: Sending Registration Requests via the Cloudberry Admin Console GUI

2 – Sentiment Analysis

This feature enables TwitterMap to show users' attitudes towards the keywords. The attitudes are estimated using third-party natural language processing libraries such as the Stanford NLP and Open NLP. These libraries are stored as UDFs in the AsterixDB. Cloudberry can take advantage of these UDFs and append the results to the response on the fly (Figure 21).

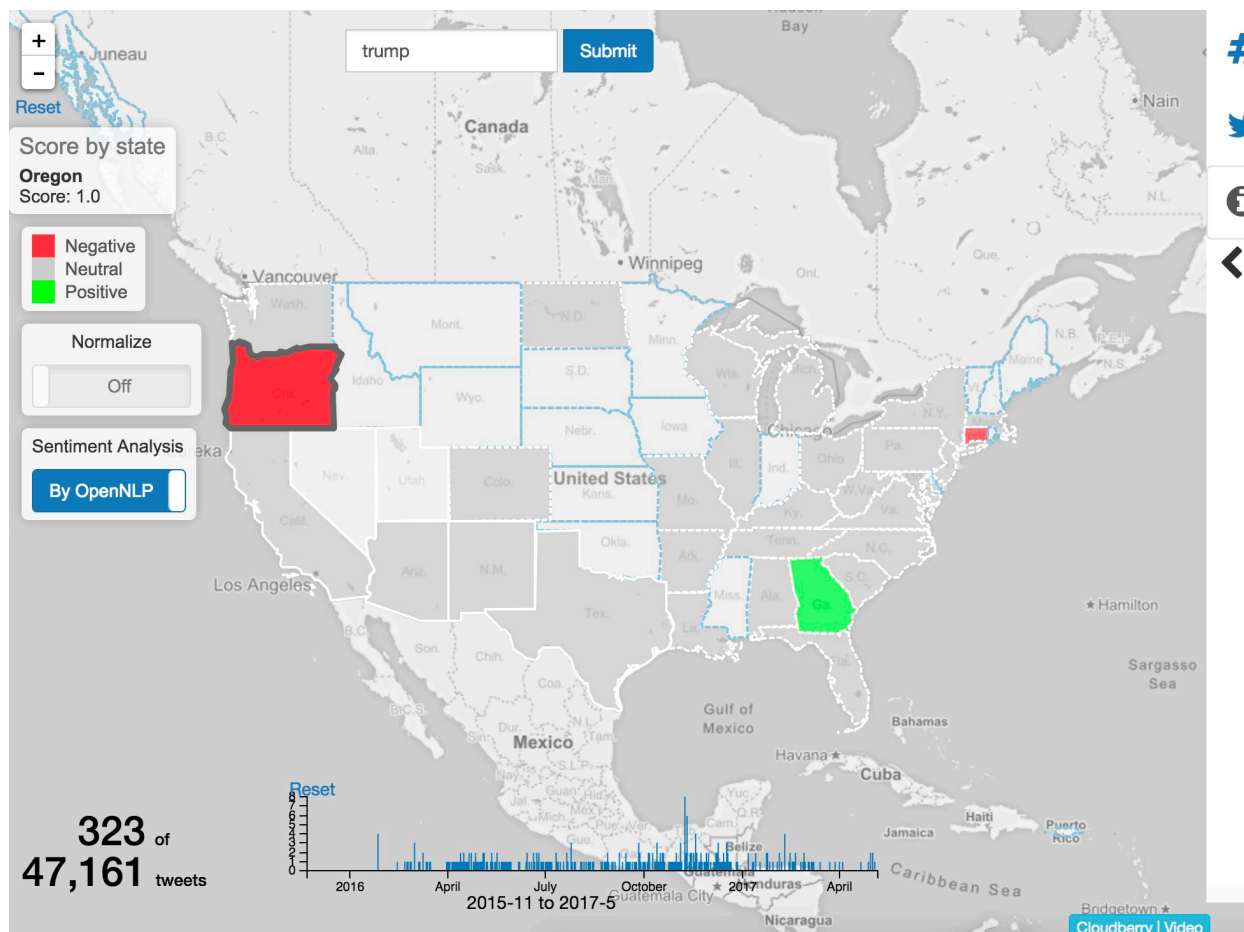


Figure 21: Sentiment Analysis with Open NLP

This feature relies on Cloudberry’s “append” API, which enables the middleware to add derived fields using UDFs to the response. As shown in Figure 22, developers need to add the “append” field in the query JSON on the frontend. The “definition” field under the “append” field stands for the invocation form of the UDF, which in this case refers to “twitter.`snlp#getSentimentScore`(text).” This is the Open NLP UDF, which analyzes the text of each tweet to calculate a “sentimentScore.” Notice that in Figure 22, this query asks for the aggregation of the “sum” and the “count” of the “sentimentScore.” These values are used to calculate the average sentiment score in each geographical location in the map/controller.js. Therefore, the sentiment score in each region shown in Figure 21 is the average sentiment score of all the tweets in that region.

Since Cloudberry is a general-purpose system, it has no restrictions on the UDFs. So, as long as the UDF is valid, Cloudberry can leverage its value by appending the derived value to the response in real-time. That being said, future applications of the “append” API can be flexible as developers can use it with whatever valid UDFs they can provide.

```
{
  dataset: parameters.dataset,
  append: [{
    field: "text",
    definition: cloudberryConfig.sentimentUDF,
    type: "Number",
    as: "sentimentScore"
  }],
  filter: getFilter(parameters, defaultNonSamplingDayRange),
  group: {
    by: [{
      field: "geo",
      apply: {
        name: "level",
        args: {
          level: parameters.geoLevel
        }
      },
      as: parameters.geoLevel
    }],
    aggregate: [{
      field: "*",
      apply: {
        name: "count"
      },
      as: "count"
    }, {
      field: "sentimentScore",
      apply: {
        name: "sum"
      },
      as: "sentimentScoreSum"
    }, {
      field: "sentimentScore",
      apply: {
        name: "count"
      },
      as: "sentimentScoreCount"
    }],
    lookup: [
      cloudberryConfig.getPopulationTarget(parameters)
    ]
  }
}
```

```
}
}
```

Figure 22: JSON Query to Append the Sentiment Analysis UDF

3. Live Tweet Count

This feature enables the TwitterMap to display the live count of all the tweets. The first number means the count of all the tweets in the current view and the second number stands for the count of all the tweets collected in the database. Notice that the second number is always increasing in the production system (<http://cloudberry.ics.uci.edu/demos/twittermap/>) as the backend is constantly ingesting new tweets from Twitter’s API at a rate of approximately 30 tweets per second (Figure 23). It provides the most direct visual aids to the viewers to understand Cloudberry’s capability of handling a massive amount of data in real-time.



Figure 23: Live Tweet Count in the Production System

The implementation of this feature relies on Cloudberry’s “estimable” API. As shown in Figure 24, developers need to specify “estimable” to be “true” in the request JSON. This request is asking Cloudberry to give an estimated count of all the tweets in the database.

```
{
  dataset: "twitter.ds_tweet",
  global: {
    globalAggregate: {
      field: "*",
      apply: {
```

```

        name: "count"
      },
      as: "count"
    }},
    estimable : true,
    transform: {
      wrap: {
        key: "totalCount"
      }
    }
  }
}

```

Figure 24: Count JSON Request Using the Estimable API

The purpose of using “estimable” is that calculating statistics such as the count of all the data usually requires expensive operations such as scanning the whole database, which brings unnecessary computational overhead to the system. The middleware will be better off if it can spend most of its resources on useful operations such as query slicing and caching. So, the “estimable” API provides a workaround for this problem by scanning the database only once a day, calculating the difference between the new count and the old count, and dividing the result by $24 * 3600$ seconds to get the increasing rate. After that, the real-time total count can be estimated via the new count and the increasing rate, which only requires very little computational overhead on Cloudberry.

The frontend needs to periodically query the middleware for the statistics using a web socket (Figure 25). In the case of TwitterMap, the frontend will receive the updated count every one second. This new information is reflected in the view using AngularJS’ “\$watch” service and the two-way data binding. As shown in Figure 26, the statistics, “totalCount”, is watched by the “\$watch” service. Whenever its value is updated, AngularJS will assign the new count to the scope variable “\$scope.totalCount”, which is bound to the view via double curly braces. Meanwhile, the view will also be updated without the need to refresh the page (Figure 27).

```

function requestLiveCounts() {
  if(ws.readyState === ws.OPEN){

```

```

        ws.send(countRequest);
    }
}
var myVar = setInterval(requestLiveCounts, 1000);

```

Figure 25: Sending Count Requests Using a Web Socket Periodically

```
$scope.$watch(  
    function () {  
        return cloudberry.totalCount;  
    },  
  
    function (newCount) {  
        if(newCount) {  
            $scope.totalCount = newCount;  
        }  
    }  
);
```

Figure 26: Watch the Live Count

[illegible]

Figure 27: Update the Count on the TwitterMap in Real Time Using the Angular Way

References

Cloudberry. "Cloudberry." *Cloudberry*. UCI ISG, n.d. Web. 02 June 2017.

<<http://cloudberry.ics.uci.edu/>>.

Cloudberry. "Quick-start." *Cloudberry*. UCI ISG, n.d. Web. 06 June 2017.

<<http://cloudberry.ics.uci.edu/quick-start/>>.

Acknowledgement

Foremost, I would like to thank all members of the Cloudberry Team of the Information System Group at University of California, Irvine for your dedication, enthusiasm, and knowledge.

I would also like to give special thanks to my faculty advisor of the ICS Honors Program Prof. Chen Li and to my mentor PhD. candidate Jianfeng Jia for your motivation and guidance. I could not imagine having a better advisor and a mentor.