Assignment 4 - Group Project Final Report

Group Members: Mingwei Gao, Yuwei Chen, Hui Lyu

# 1 Overview and Initial Assessment of Dataset

## 1.1 Overview

Farmer markets is an important part of local economies. It not only serves as a way for people to purchase locally grown produce, but also as an opportunity for them to connect with others within their communities. Such experience helps to make individuals feel tied to their communities and promoted a sense of belonging.

In our project, we pick US Farmers Markets as our dataset. (<https://www.ams.usda.gov/local-food-directories/farmersmarkets>) It contains 8665 rows (instances) and 59 columns. For each row, it represents information of a single farmer market with a unique FMID. The information includes market names, websites, social medias, market locations, season dates and times, directions, operating times, product offerings, accepted forms of payment, and more.

## 1.2 Quality Issues

From an initial inspection, we find the following quality issues.

|  |  |  |
| --- | --- | --- |
| **Problem Type** | **Dirty Data** | **Reasons** |
| Missing values | County field is null | Unavailable values during data entry |
| Misspelling words | City = “Amhersts” | Typos and phonetic errors |
| Mislocated values | County = “Missouri” | Human errors |
| Varying representations for null value | Including “no”,”none”,”n/a”,”-”, etc | Different representations |
| Varying representations for same value | Different URLs for the same website of the same market | Typos |
| Inconsistency of data  (in different columns) | Zipcode does not match latitude and longitude | The same real world entity is described by different values |
| Inconsistency of format  (in the same column) | Different descriptions in updateTime, Season1Date, etc | Different representations in different circumstances |

## 1.3 Use Case of Dataset

Strengthening local food system is important to every government. Particularly, food distribution by local agriculture producer, namely farmer markets, is an important aspect of food distribution.

In particular, for users who need exact location information (especially zip code) of farmer markets, this dataset may not provide precise enough information. While, a reference postal code is offered accompanied with its origin given zip code for validation.

In addition, for users who need precise dates and times, not all the farmers markets have corresponding exact information.

## 1.4 Cleaning Goals

There are several cleaning goals we attempt to realize:

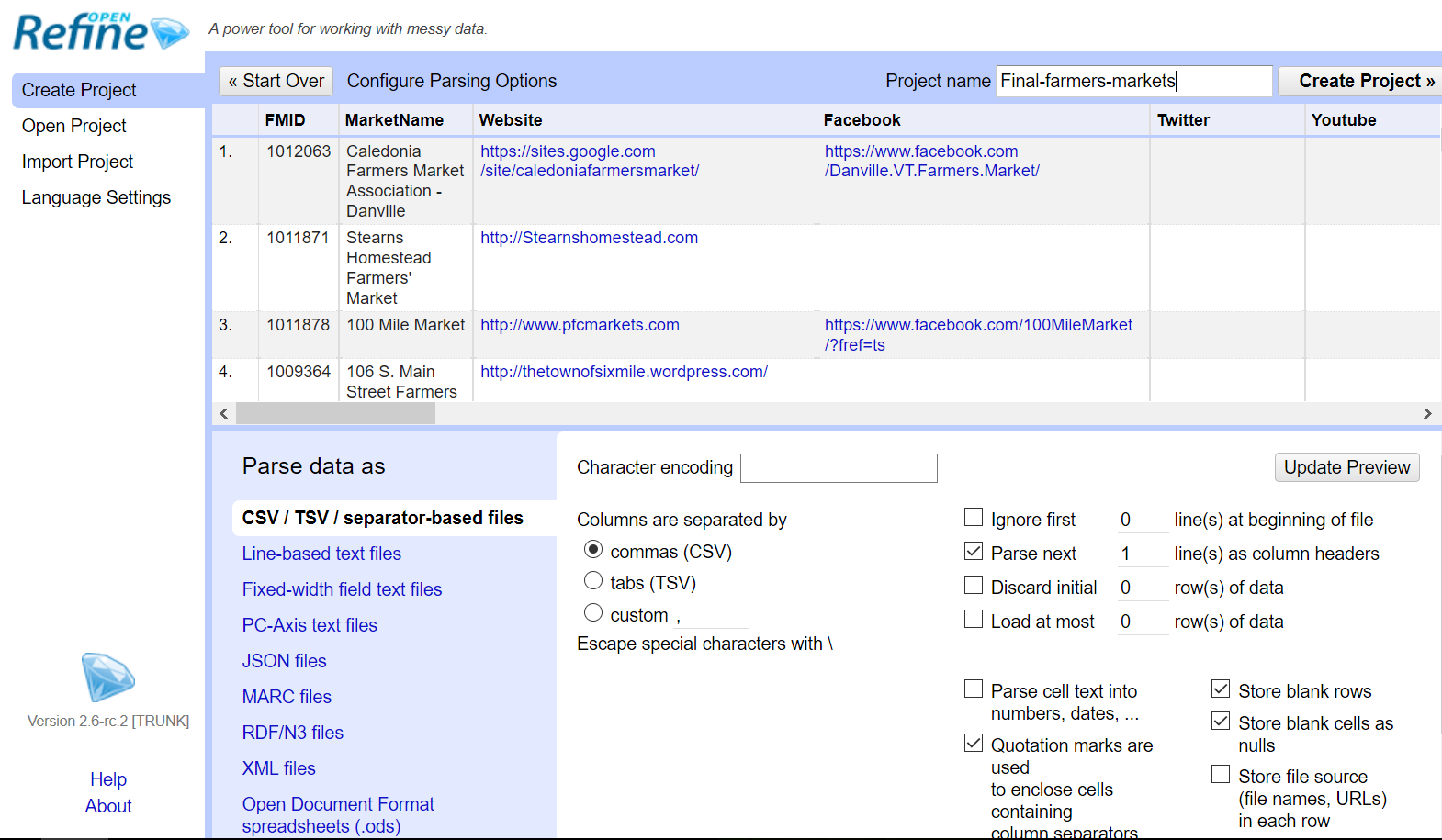
1. Eliminate errors
2. Eliminate redundancy
3. Increase data reliability
4. Deliver accuracy
5. Ensure consistency and completeness
6. Provide feedback for improvement

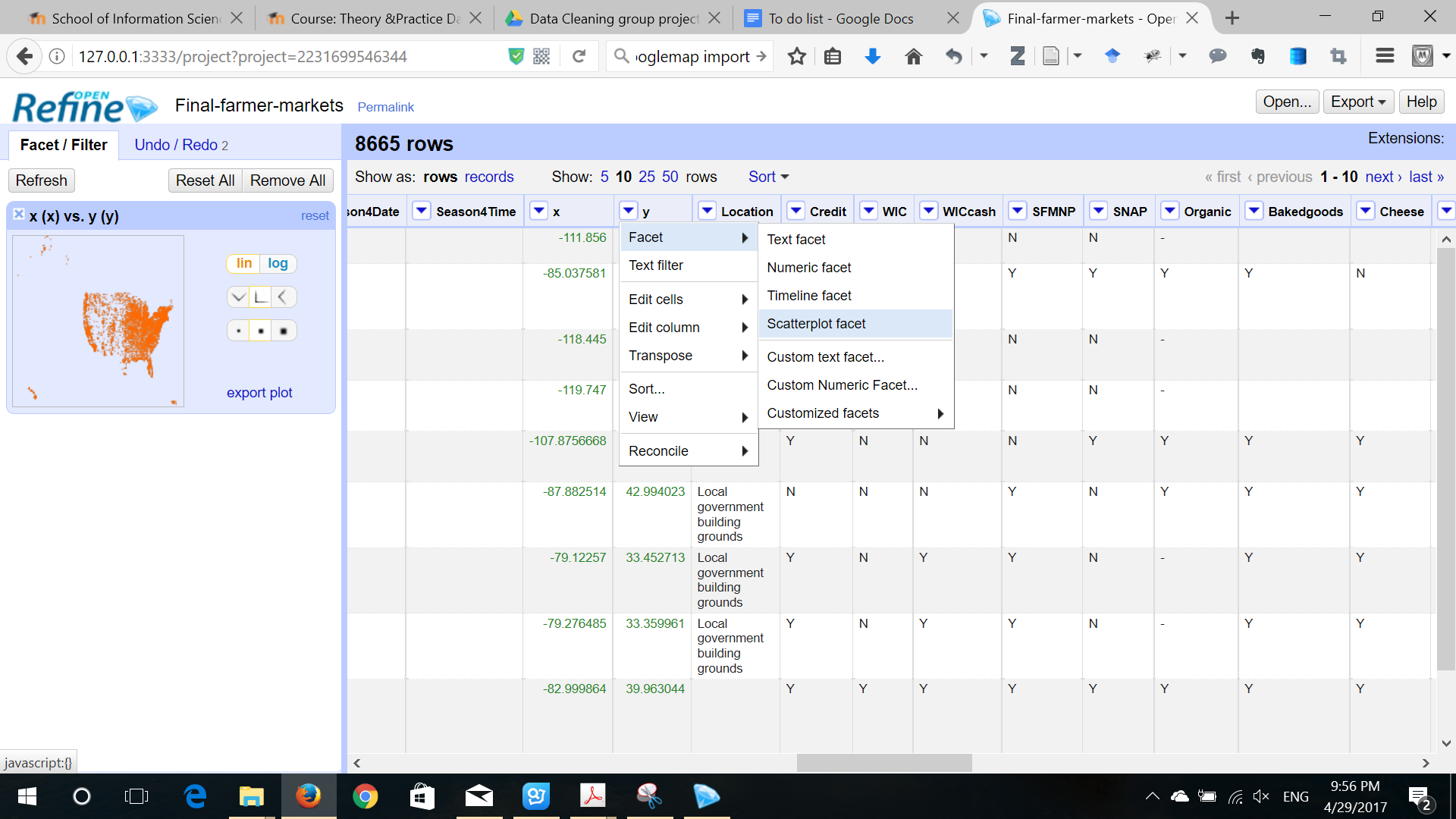
In essence, the goal of data cleaning is to minimize errors, prevent dirty data and keep data up to date. It helps to make following workflows smooth and accurate and increase efficiency of transaction.

# 2 Data Cleaning with OpenRefine

## 2.1 Complete Cleaning Steps

1. Create a new project and load the csv file into it (first row as header)

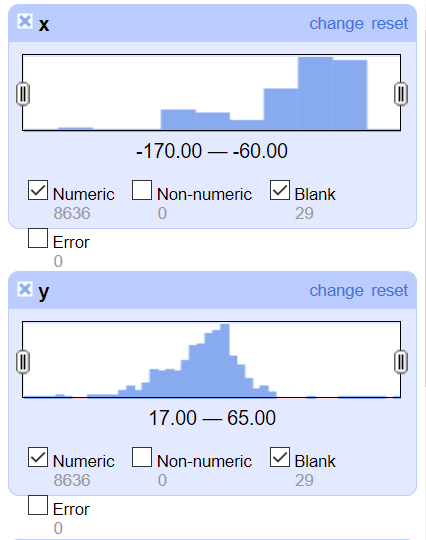


1. Transform text values of “x” and “y” columns into number values 
2. Make a scatter plot of values of “x” and “y” columns 
3. Develop numeric facet of “x” and “y” to check number of blank (null) values

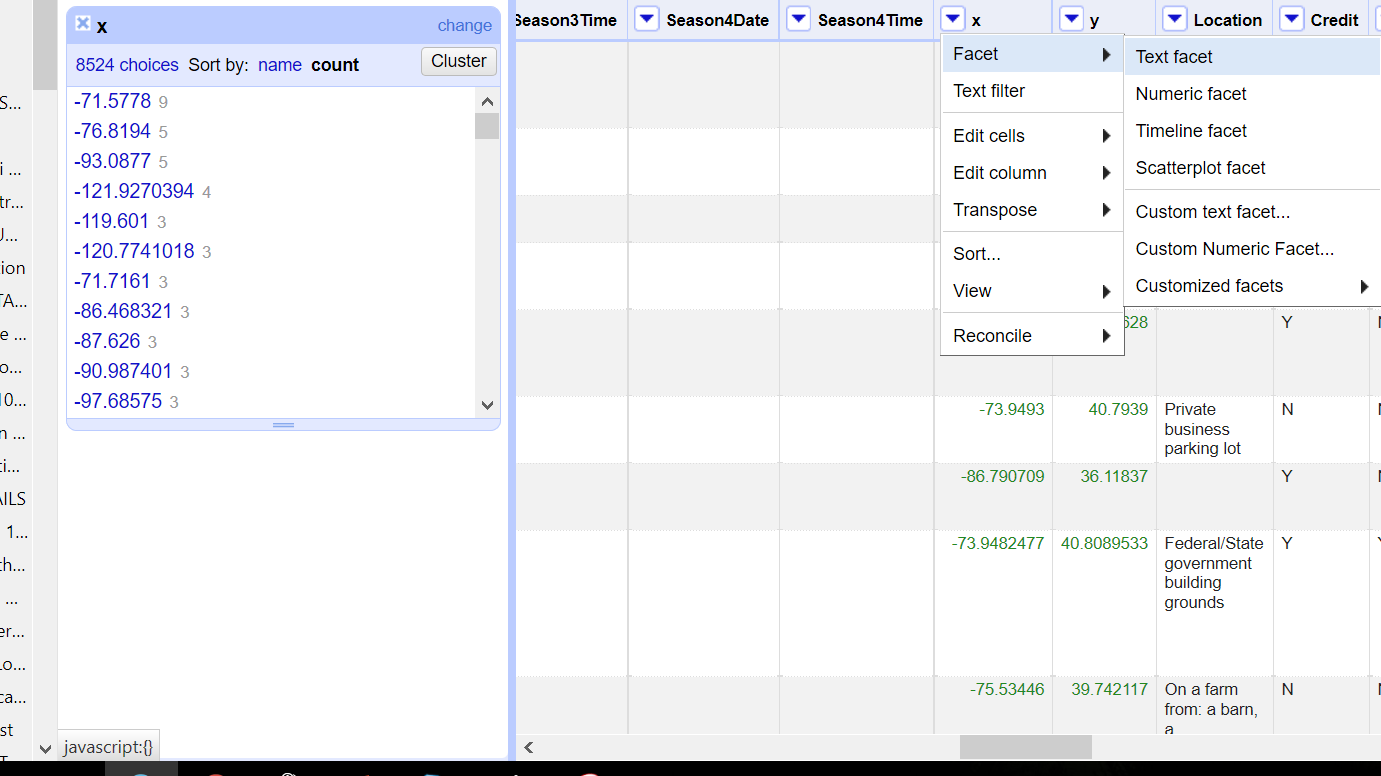


Result:

* 29 null values for each
* The ranges of latitude and longitude are reasonable



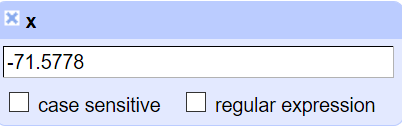
1. Develop text facet of “x” and “y” to check duplicates



Result: There are duplicate values in both “x” and “y”.

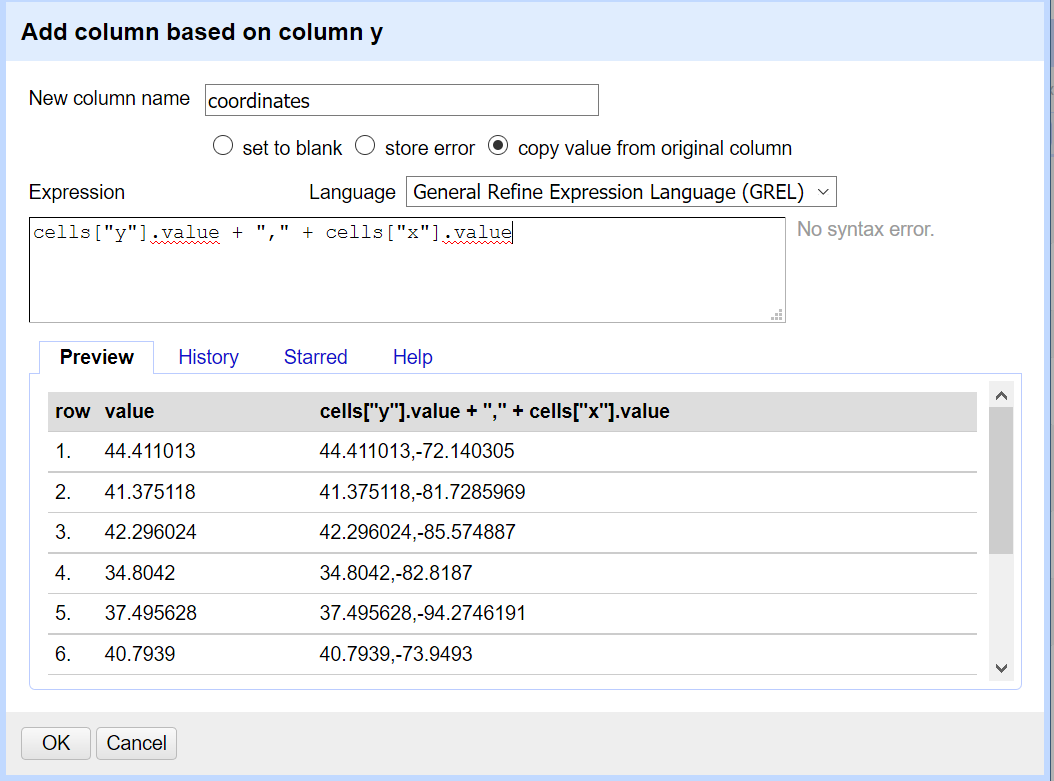


1. Filter by “x=-71.5778”. We find 9 rows with the same value of both “x” and “y”, but of different addresses.

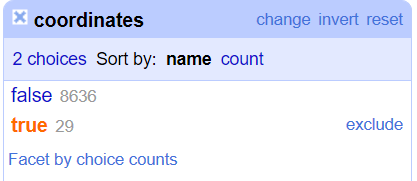




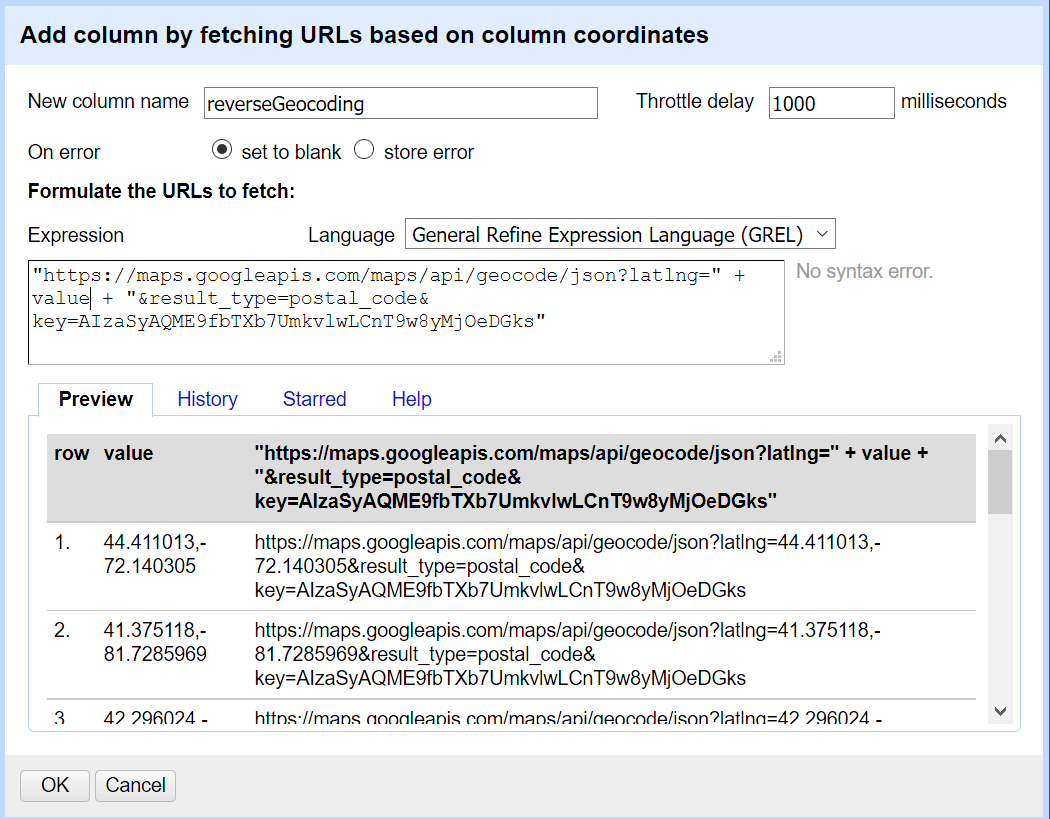
1. Implement value.trim() to each column to delete the leading and trailing whitespace of the values
2. Add a new column called “coordinates” to concatenate latitude and longitude in order to use Google Map API reverse geocoding
   1. We assume the not blank coordinates values are basically correct since they have been validated on Google Map visualization (no unreasonable points).
   2. Then we use the assumed correct coordinates to get the corresponding postal code through reverse geocoding.



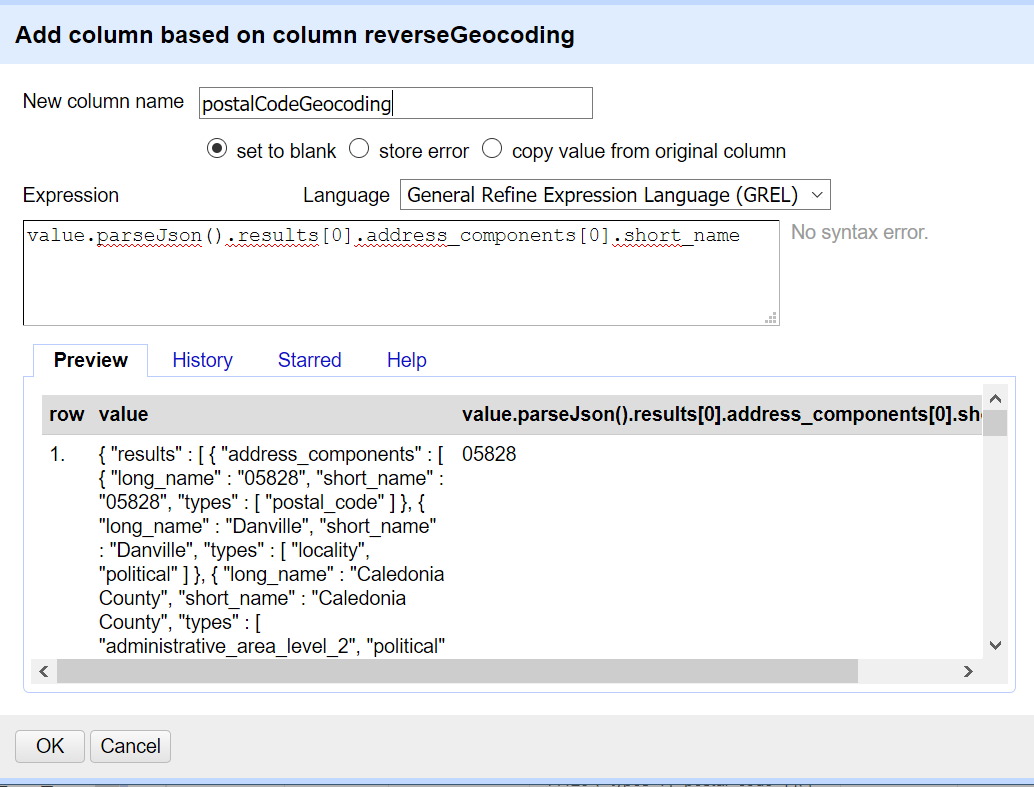
Facet by blank: 29 blank rows



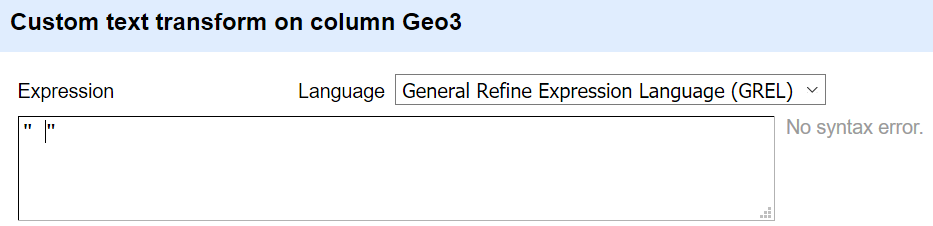
1. Add a new column by fetching URLs with json responses of Google Map API reverse geocoding



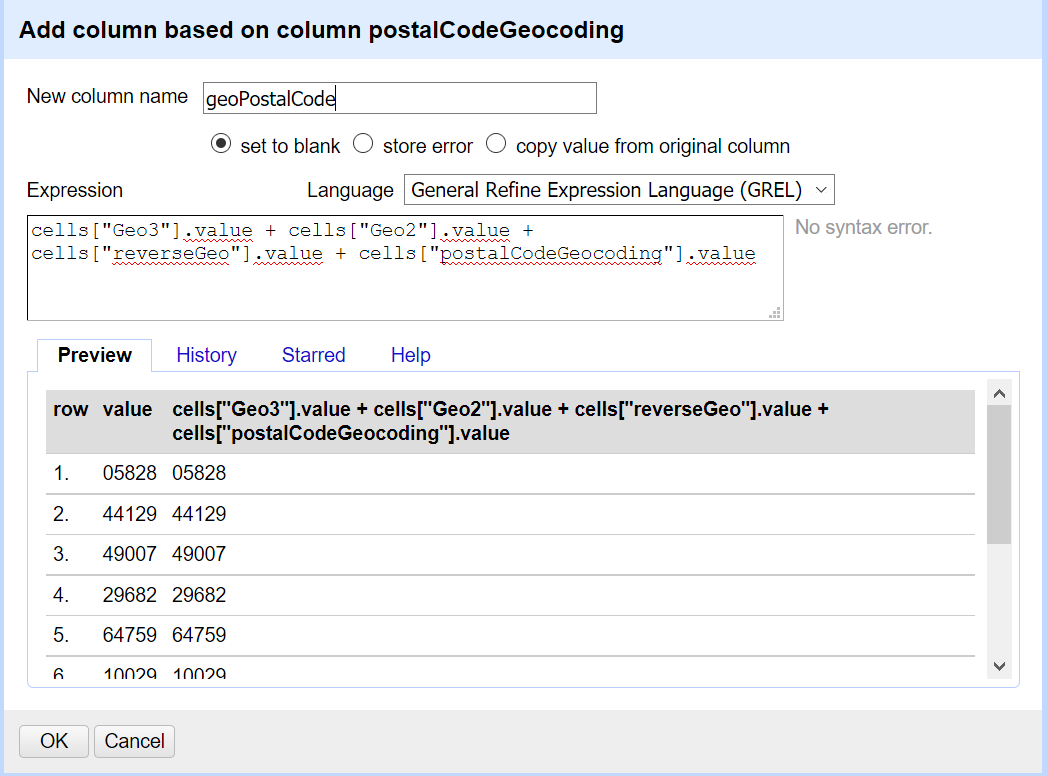
1. Parse JSON and extract the postal code using “value.parseJson().results[0].address\_components[0].short\_name”



1. Repeat step 8 & 9 for several times using different API Keys, and then generate multiple columns
2. Transform blank cells in these columns to a whitespace for future use

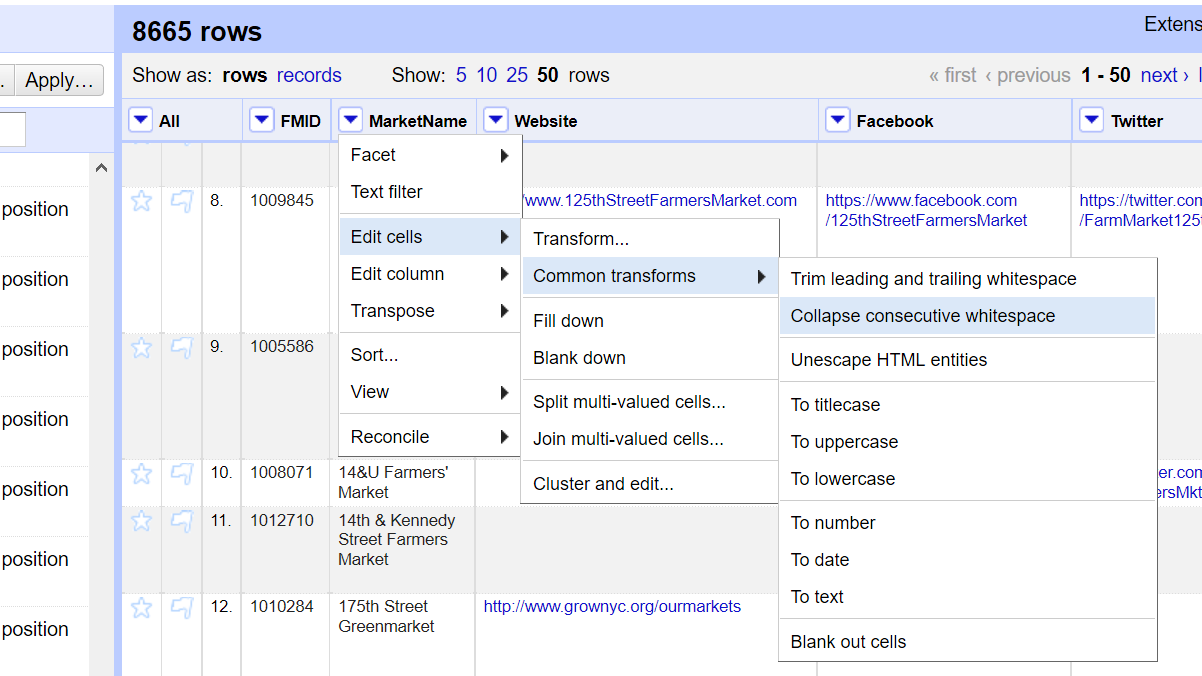


1. Merge all the columns of postal codes to a newly added column called “geoPostalCode”, and then transform value.trim() to all the cells

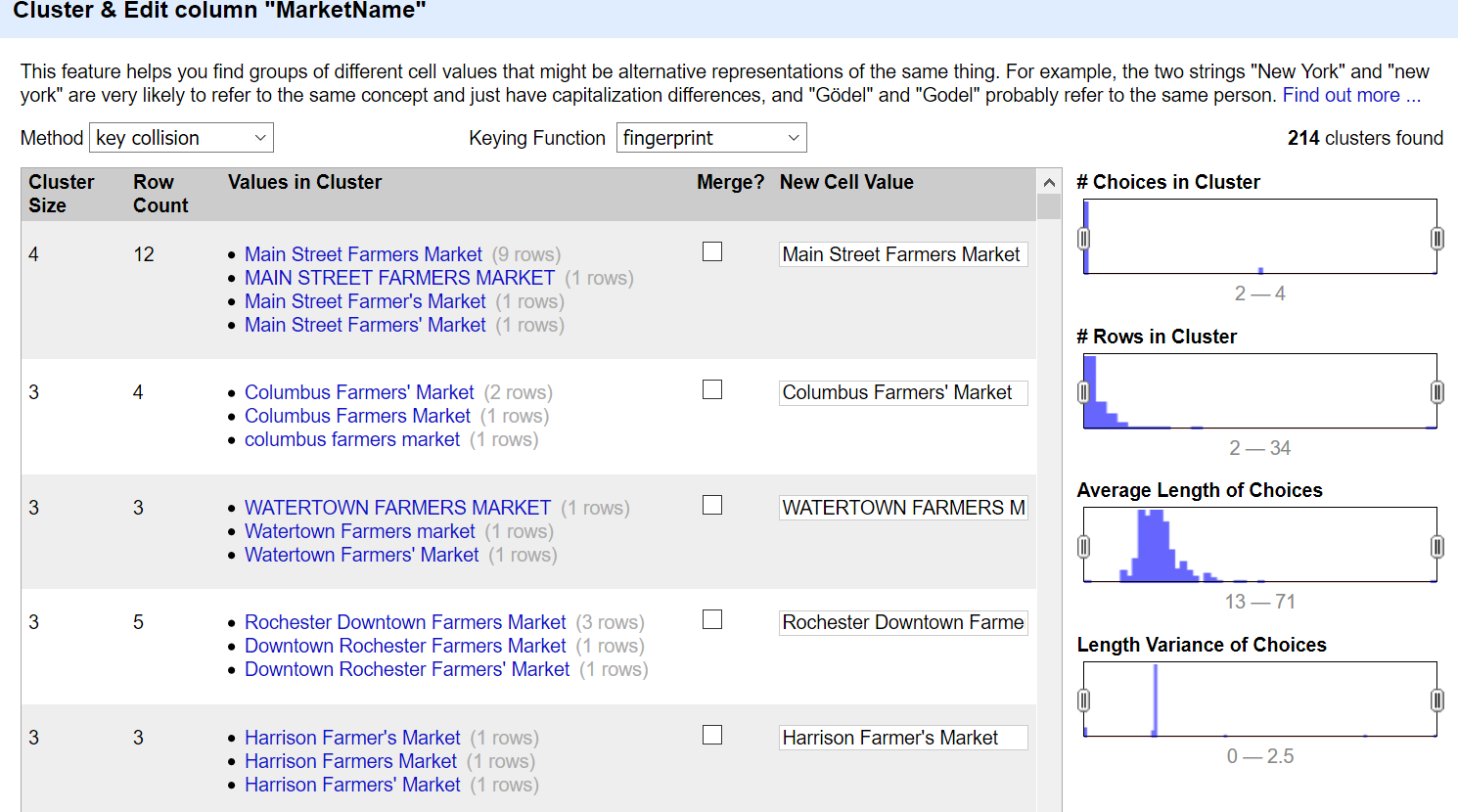


Consequently, “geoPostalCode” column includes the data coming from Google Map reverse geocoding result based on latitude and longitude.

1. Move column “geoPostalCode” to position 12, right after “zip” column for comparison
2. Collapse consecutive whitespaces for “MarketName” column

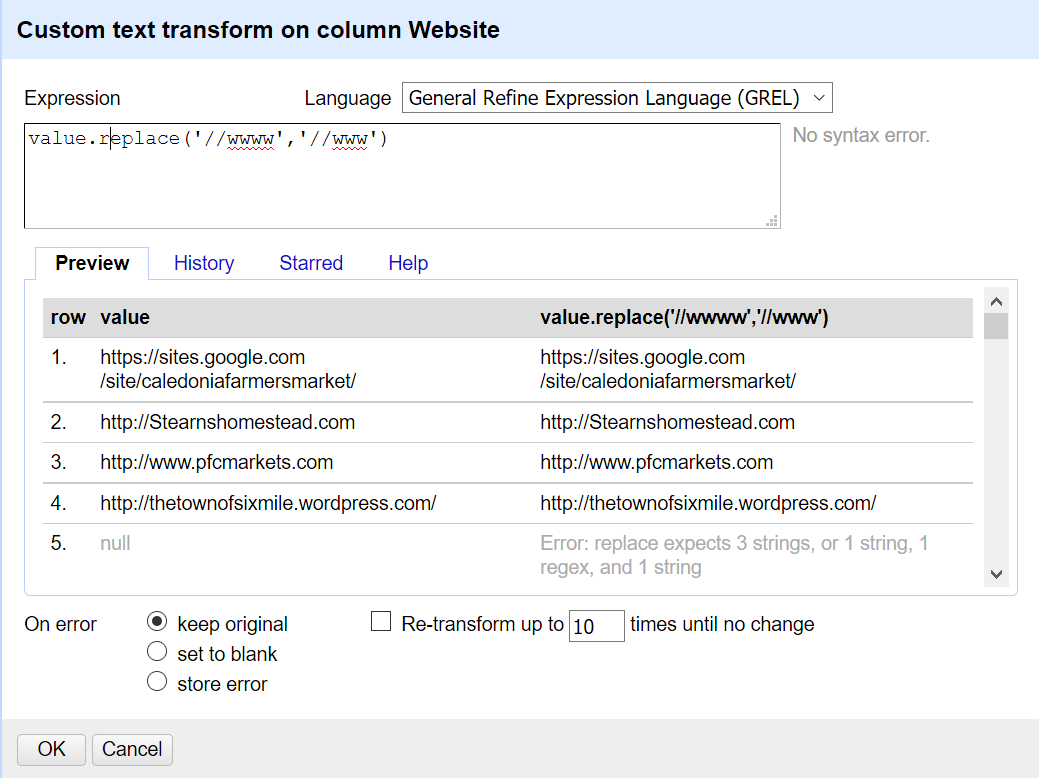


1. Consider about clustering and merging cells in column “MarketName”

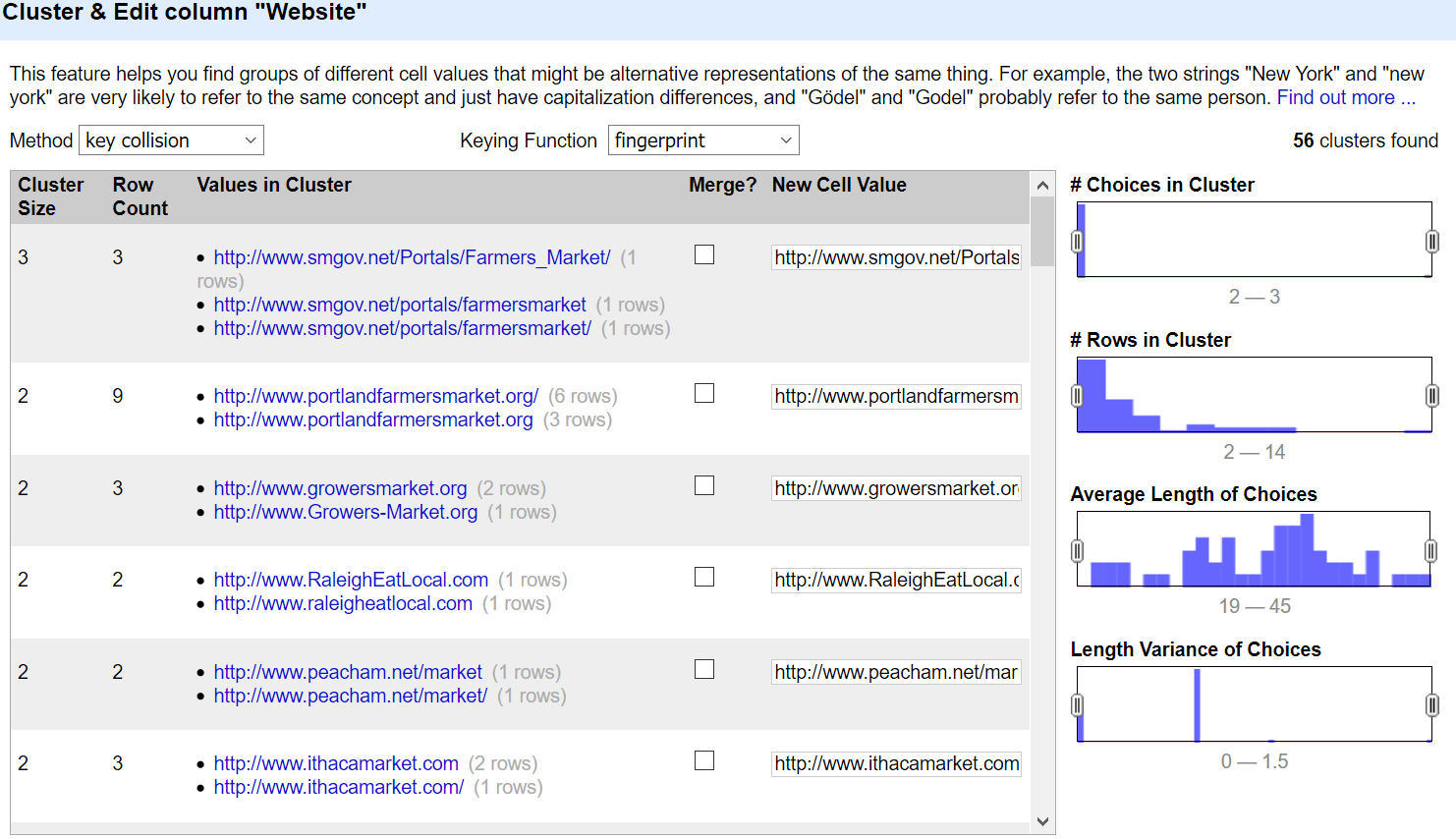


After browsing these clusters, we find that they are different markets located in different places. Therefore, we decide not to merge anything.

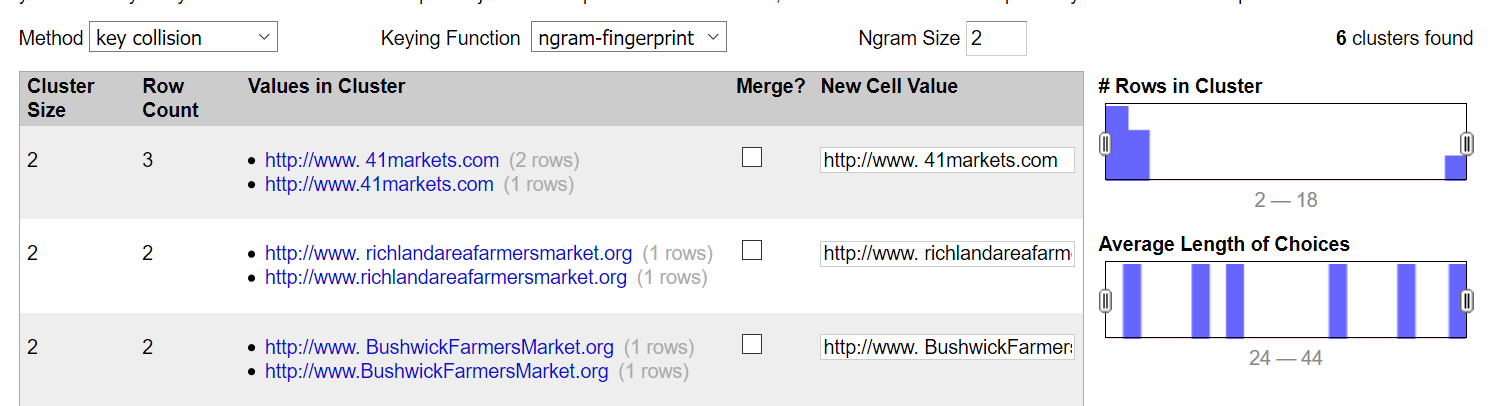
1. Modify potential typo by replacing “//wwww” with “//www”



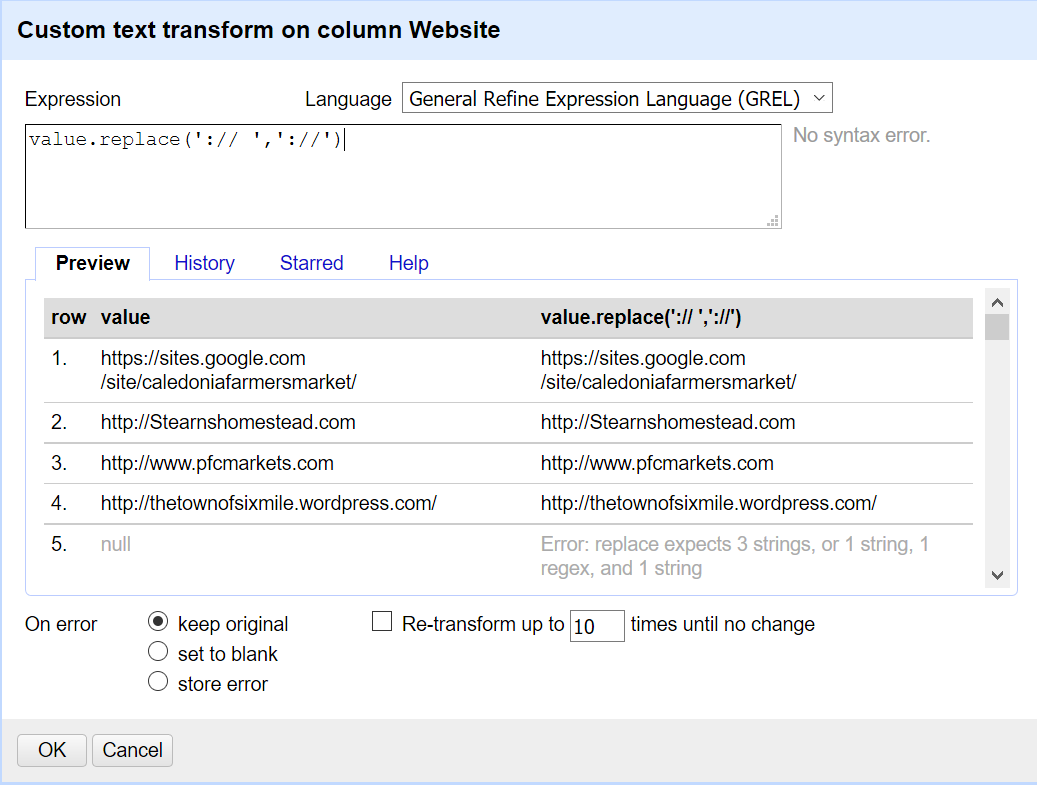
1. Cluster and merge cells in “Website” after manual browsing and checking



1. Modify cluster keying function to “ngram-fingerprint” and set a value for ngram size



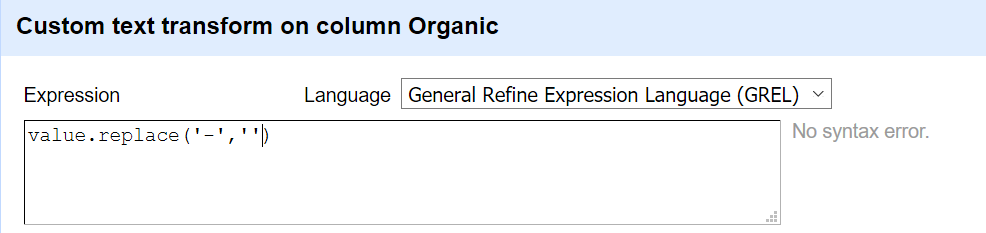
1. Modify typo by replacing “:// ” with “://”



1. Repeat step 16 through step 19 for “Facebook”, “Twitter”, “Youtube”, and “OtherMedia” columns
2. Dealing with NULL values:
3. Replace **“none”, “no” and “n/a”** with blank values in “Facebook”, “Twitter”, “Youtube”, and “OtherMedia” columns



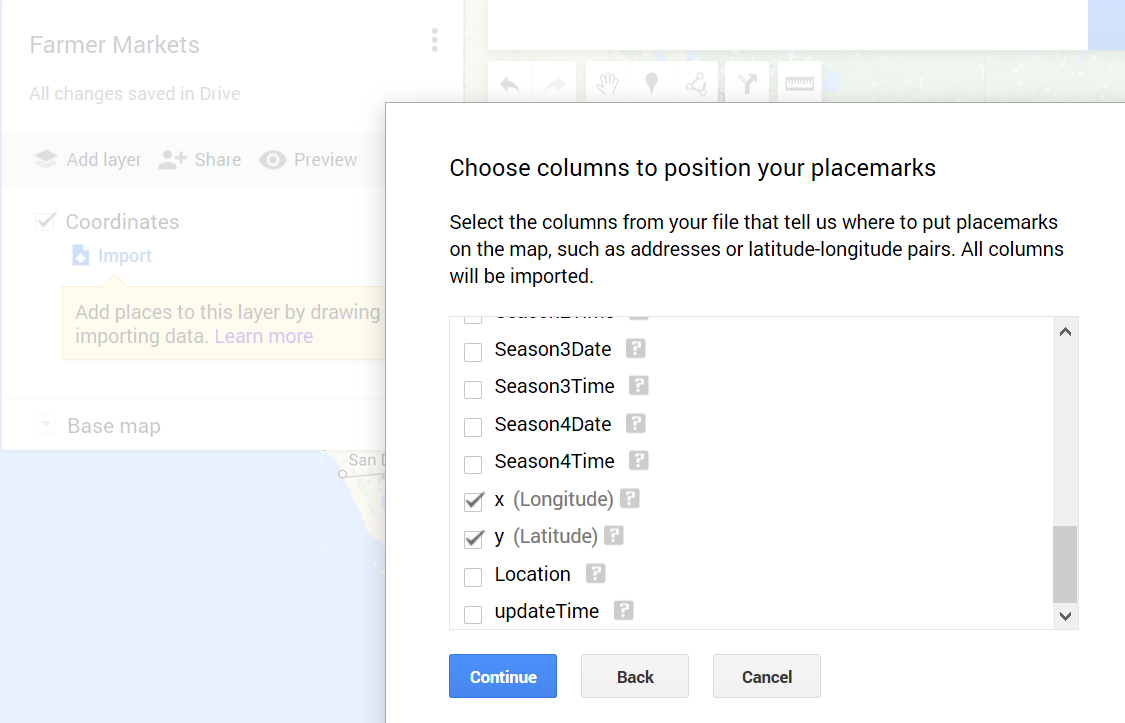
1. Collapse consecutive whitespaces, cluster and merge values for “city”, “street”, “County”, “State” columns
2. Replace **“none”, “n/a”, and “no”** values with blank values in “city”, “street”, “County”, “State” and “zip” columns
3. Replace **‘-’** with blank values in “Organic” column



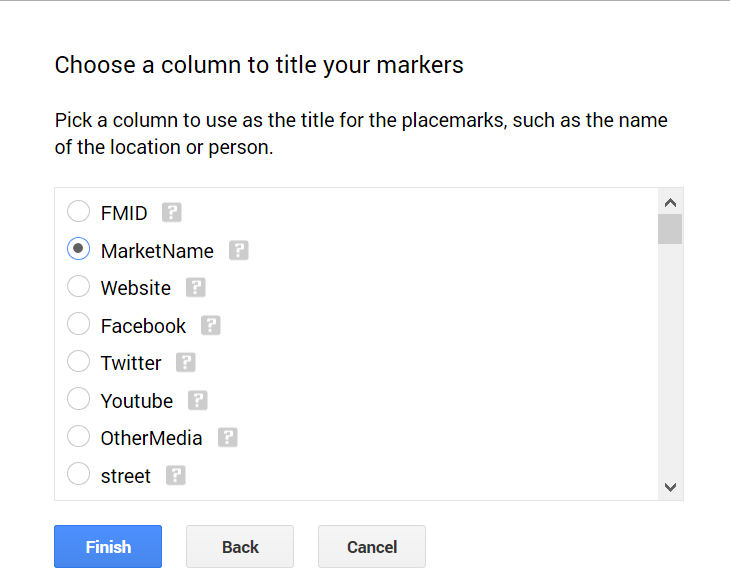
1. We decide not to transform date or datetime in OpenRefine since it will automatically add unexpected information. For instance, “2007” will be transformed to a format similar to “2007-01-01T00:00:00”. The detailed datetime information which may not be correct will be added. Therefore, formatting of date and time will be done in SQLite.

## 2.2 Google Map Visualization of Coordinates

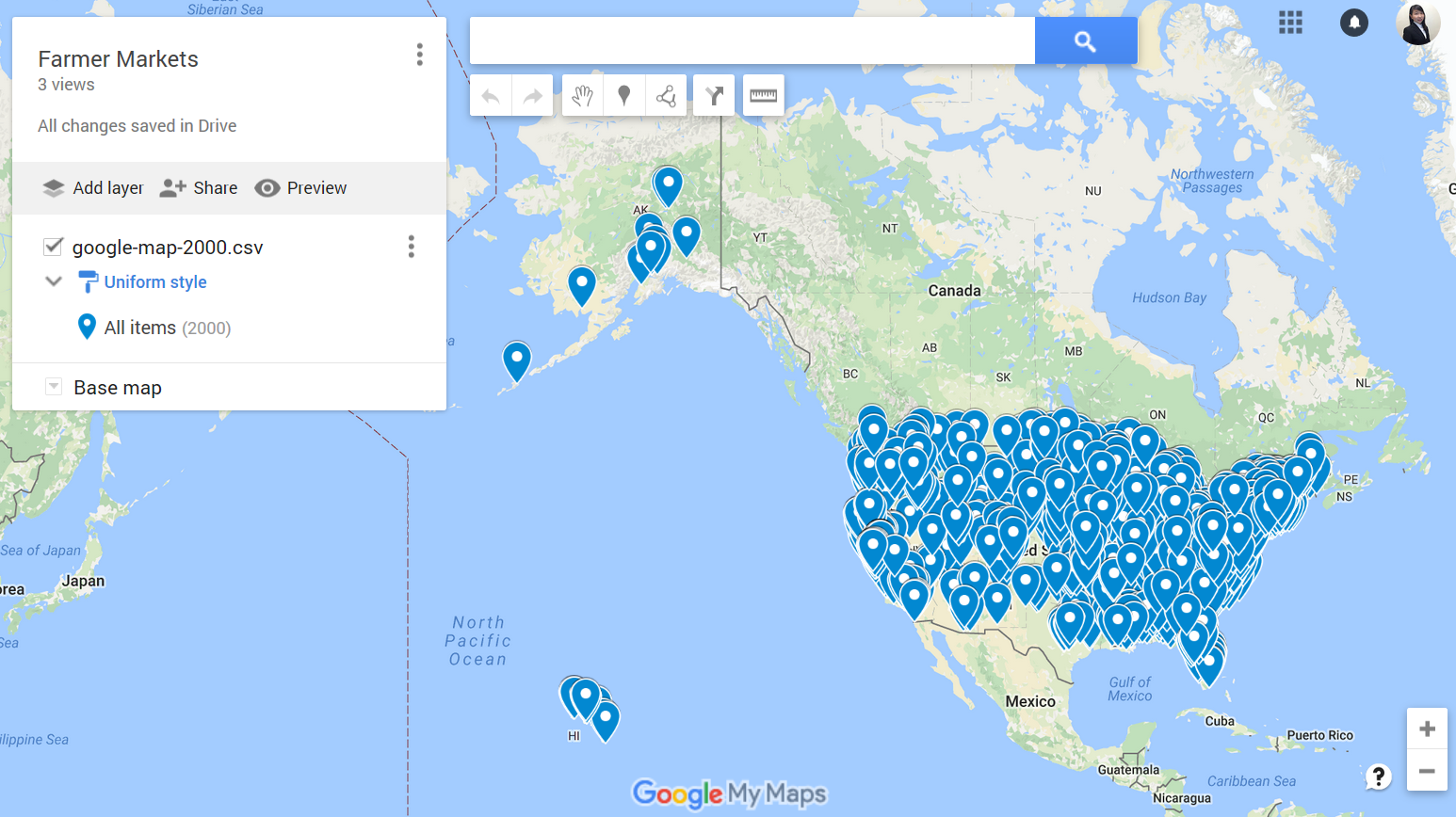
1. Sort by FMID in Excel
2. Separate the whole sorted csv file into 5 files in Excel
   1. 1-2000 rows
   2. 2001-4000 rows
   3. 4001-6000 rows
   4. 6001-8000 rows
   5. 8001-8665 rows
3. Create new map called “Farmer Markets” in Google Map
4. Add new layer for each individual csv files (because Google Map can display at most 2000 items per layer)
5. Import data “google-map-2000.csv”
   1. Remove columns from “Credit” to “WildHarvested” (because Google Map allow to import csv with no more than 50 columns)
   2. Choose columns to position the placemarks
      1. X as longitude
      2. Y as latitude



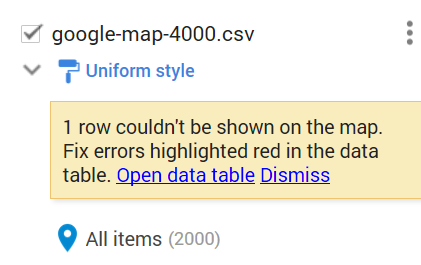
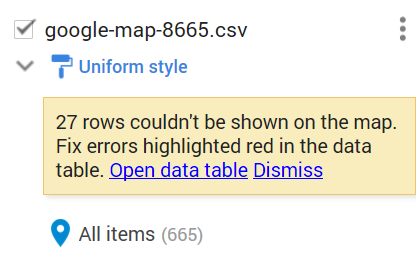
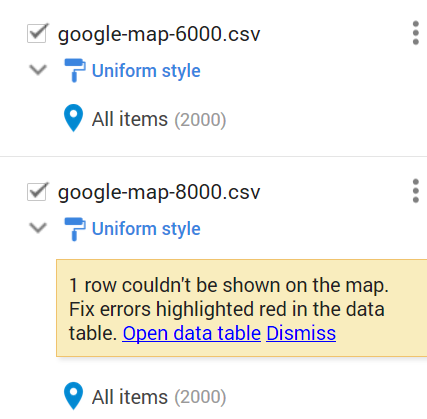
* 1. Choose columns to title the markers



1. Result for the first layer

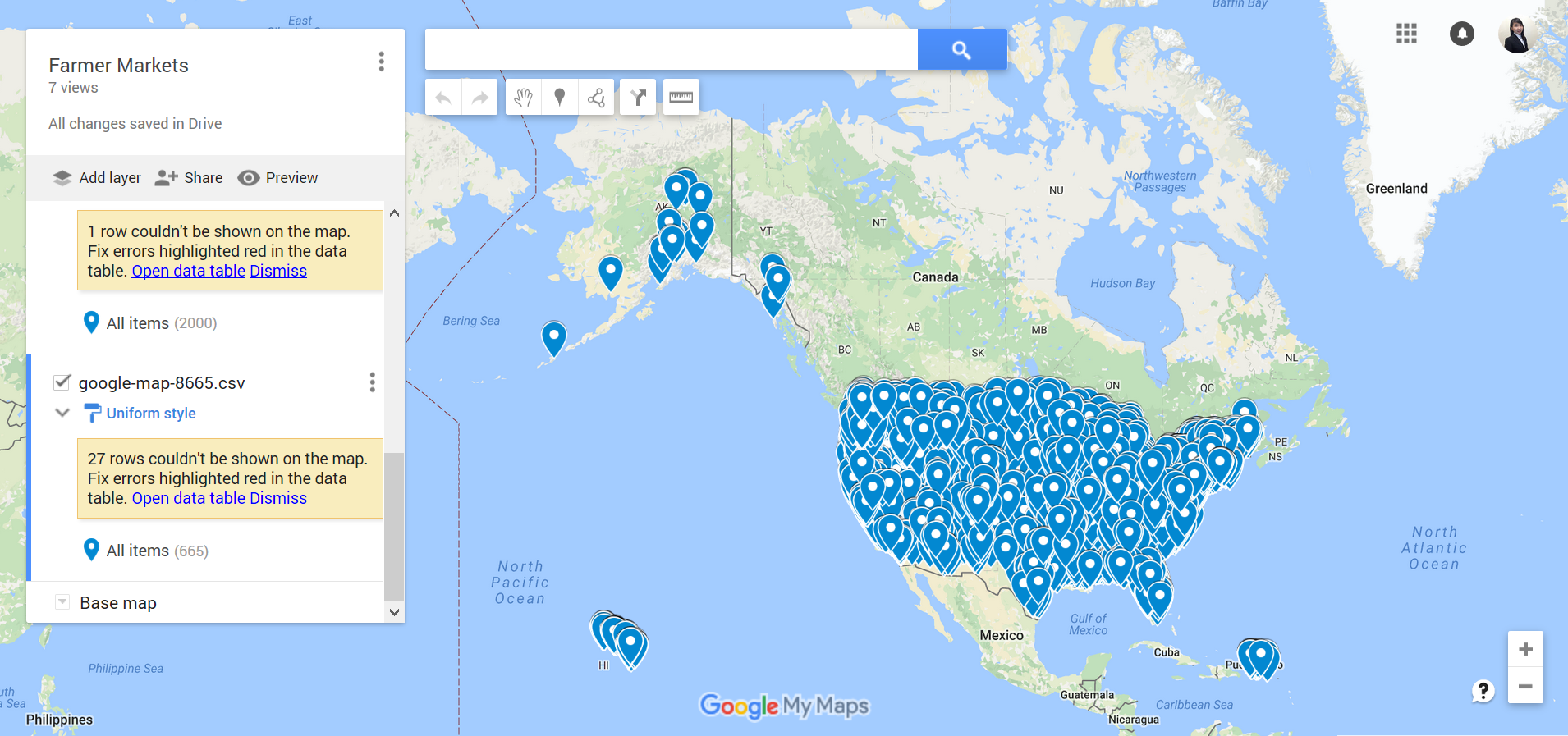


1. Repeat step 5 for the other csv files. There are 5 layers eventually.
2. Find 29 rows in total which do not have value in (x,y). **The null values of “x” and “y” are in pairs. In other words, if one of “x” or “y” is null, the other is also null.**

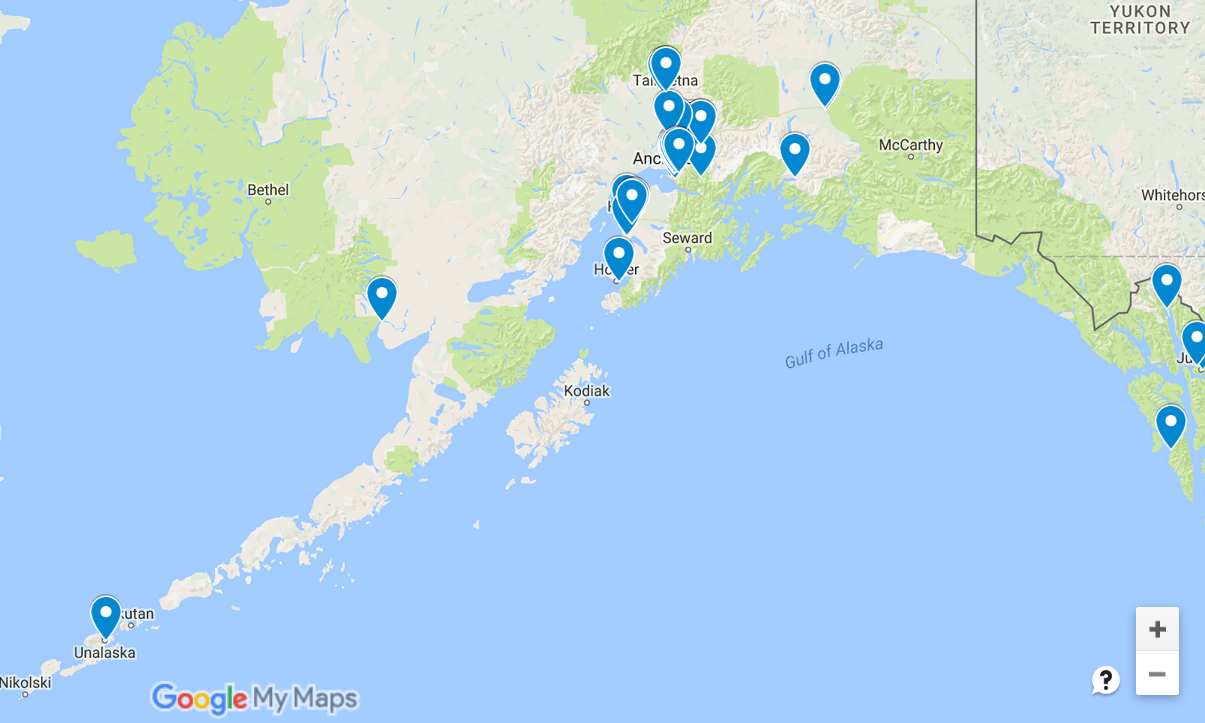
 

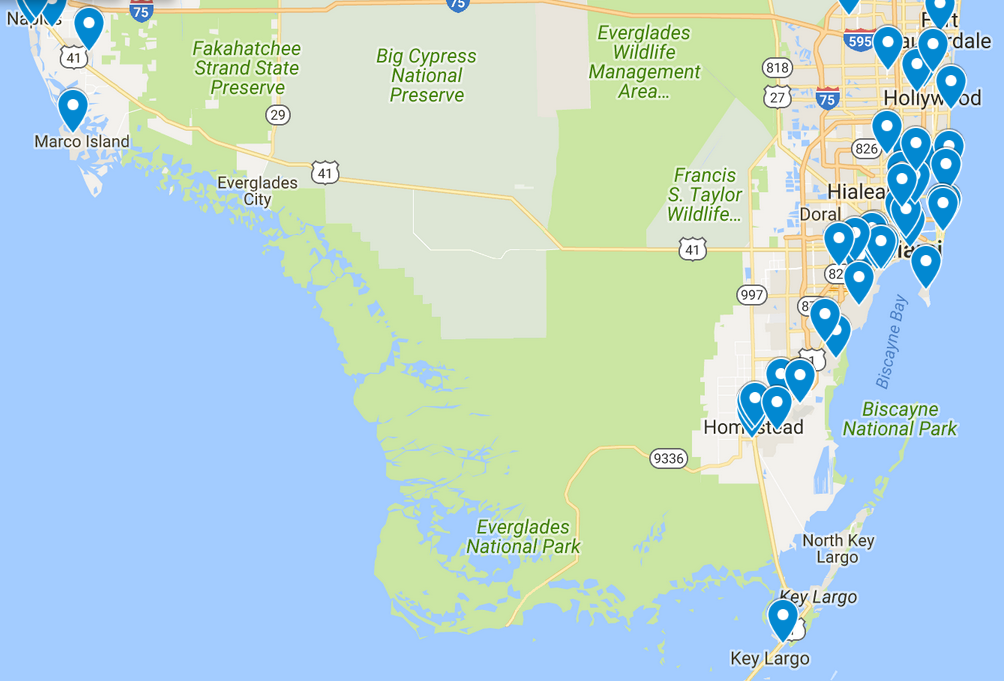


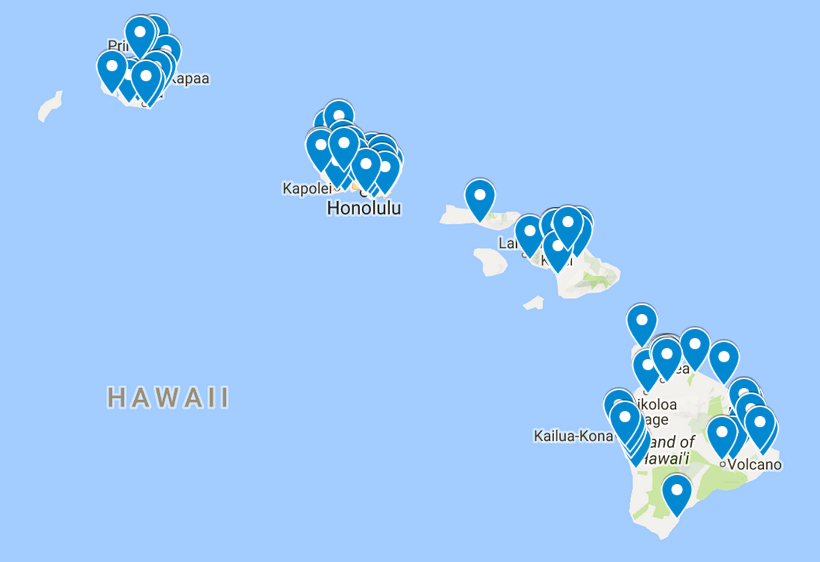
1. Final result



Zoom in:

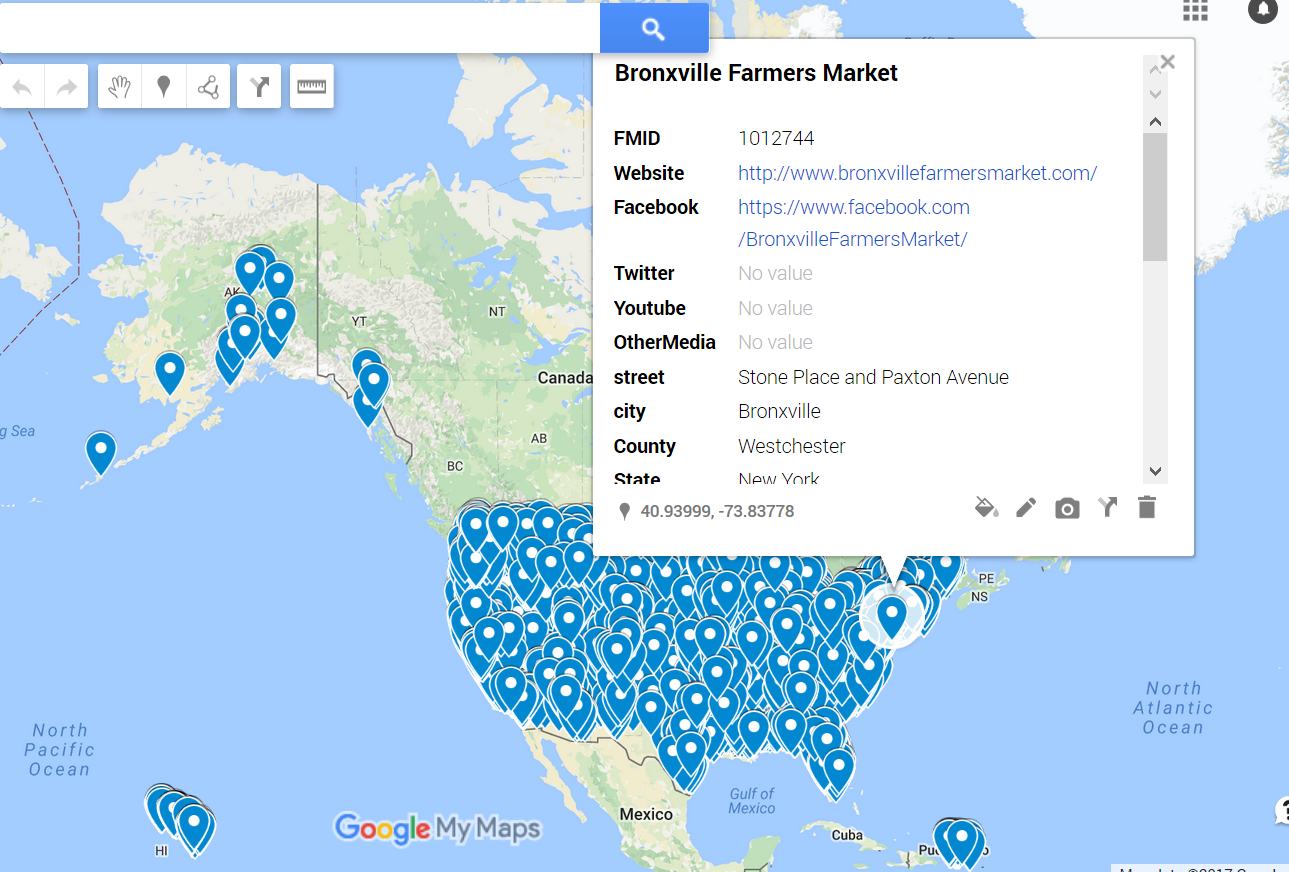








1. Validate the coordinates on map
   1. no unreasonable (x,y) values in the given dataset
   2. no point is located in the ocean
2. Labels can also be shown on the map



## 2.3 Google Map API Reverse Geocoding

API: <https://developers.google.com/maps/documentation/geocoding/intro>

API Key:

* AIzaSyAQME9fbTXb7UmkvlwLCnT9w8yMjOeDGks
* AIzaSyCO3shtN23zrr6LjENrt1lQ1611myysis4
* AIzaSyDUB0N3Xlr2LvyuwlsX3DZa49Gw0hdcJY0
* AIzaSyAs\_QgNHBLIjykmR-rNPnKcmHMOwh2vHgE

Use latitude and longitude to get its postal code:

* result\_type=postal\_code

For example:

[https://maps.googleapis.com/maps/api/geocode/json?latlng=44.41101,-72.1403&result\_type=postal\_code&key=AIzaSyAQME9fbTXb7UmkvlwLCnT9w8yMjOeDGks](https://maps.googleapis.com/maps/api/geocode/json?latlng=44.41101,-72.1403&key=AIzaSyAQME9fbTXb7UmkvlwLCnT9w8yMjOeDGks)

# 3 Data Cleaning with SQLite

## 3.1 Relational Database Schema

Table: fmv2

(FMID,MarketName, Website, Facebook, Twitter, Youtube, OtherMedia, street, city, County, State, zip, geoPostalCode, Season1Date, Season1Time, Season2Date, Season2Time, Season3Date, Season3Time, Season4Date, Season4Time, x, y, Location, Credit, WIC, WICcash, SFMNP, SNAP, Organic, Bakedgoods, Cheese, Crafts, Flowers, Eggs, Seafood, Herbs, Vegetables, Honey, Jams, Maple, Meat, Nursery, Nuts, Plants, Poultry, Soap, Trees, Wine, Coffee, Beans, Fruits, Grains, Juices, Mushrooms, PetFood, Tofu, WildHarvested, updateTime)

## 3.2 Integrity Constraints Checking and Fixing

### Primary key checking

Check whether FMID could act as the primary key in the table (whether there is duplicate FMID):

SELECT T1.FMID FROM fmv2 T1, fmv2 T2

WHERE T1.FMID=T2.FMID

AND T1.x != T2.x

Since we got 0 result after running the query above, FMID could be assigned as the primary key for the table.

### Further processing for zip code validation

We noticed that there are records whose geoPostalCode value is NULL. That means zip code cannot be found through Reverse Geocoding API according to x and y values, which indicates that these zip codes may be wrong.

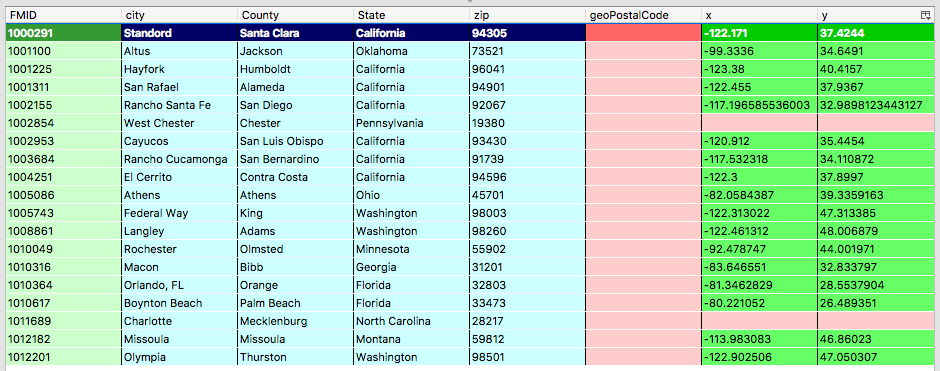
SELECT FMID, zip, geoPostalCode

FROM fmv2

WHERE zip is not null

AND geoPostalCode is null;

In total, there are 19 records whose zip code may be wrong. Here is the screenshot of the results:



Among these 19 records, two of them are without latitude and longitude values. For the rest of them, since it is not a huge amount of work, we decide to manually validate the zip code.

The website we use: <http://www.melissadata.com/lookups/latlngzip4.asp?lat=-122.171&lng=37.4244>

Through manual check, we found that the zip code of whose FMID are 1000291, 1001100, 1002953, 1003684, 1005743, 1008861, 1010049, 1010316, 1010364, 1010617, 1012201 are exactly the same with the zip code we got according to latitude and longitude from the website. The zip code of whose FMID are 1001225, 1001311,1002155, 1004251, 1012182, are different from what we got from the website.

Therefore, we update those rows whose zip codes are exactly the same with what we got from the website.

UPDATE fmv2 SET geoPostalCode=zip

WHERE FMID=1000291

OR FMID=1001100

OR FMID=1002953

OR FMID=1003684

OR FMID=1005743

OR FMID=1008861

OR FMID=1010049

OR FMID=1010316

OR FMID=1010364

OR FMID=1010617

OR FMID=1012201;

Besides, there are some records whose zip code are not the same with what we got based on latitude and longitude. Similarly, there may be problems for those zip codes.

SELECT zip, geoPostalCode

FROM fmv2

WHERE zip <> geoPostalCode;

We got 1164 results after running the script above. That is too much. Do we really need to clean all of them? Are those zip codes all wrong? After the discussion, our conclusion is: small differences between zip in dataset and zip we got from the third party should be tolerated. (Think about zip code 61820 (in Champaign) and 61801 (in Urbana). There is a difference of 19 between the zip code but in fact the distance between geographic locations of these two zip codes is not that far. ) Our group feel that the difference of 50 could be a value to determine whether there is significant problem with the zip code. Thus, we ran the following script.

SELECT zip, geoPostalCode, cast(zip as numeric)-cast(geoPostalCode as numeric)

FROM fmv2

WHERE cast(zip as numeric)-cast(geoPostalCode as numeric)>50

OR cast(zip as numeric)-cast(geoPostalCode as numeric)<-50;

This time we got 335 results. However, we found that some of them, the values in zip column are not 5 digits, but 9 digits, while what we got from Reverse Geocoding would always be 5 digits. That would be one of the reasons leading to big differences. Another possible reason would be that there are non-numeric values in zip column, such as ‘IL’, ‘FL’. Due to those facts, we found that we need to clean those who cells whose contents are not 5-digit numbers.

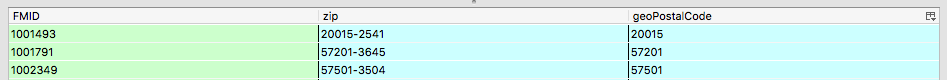
Moreover, we found that there are some zip codes who are not 5 digits.

SELECT FMID, zip, geoPostalCode

FROM fmv2

WHERE length(zip)<>5;

After running the script above, we got 29 results in total. Some of them, as the screenshot (1) shows, are 9 digits, and the first 5 digits are the them with what we got from the Reverse Geocoding. We believe those zip codes are highly possibly with no problem.



UPDATE fmv2 SET geoPostalCode=zip

WHERE FMID=1001493

OR FMID=1001791

OR FMID=1002349

OR FMID=1005109

OR FMID=1006828

OR FMID=1007541

OR FMID=1007758;

The reason why we put value in zip column to geoPostalCode, rather than copying from geoPostalCode to zip is that the information contained in 9-digit zip code would definitely more than in 5-digit zip code. We do not want to lose them.

Then we ran script below after updating:

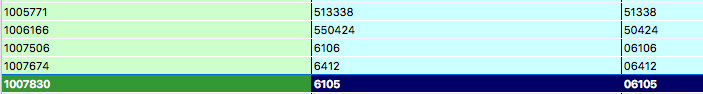
SELECT FMID, zip, geoPostalCode

FROM fmv2

WHERE length(zip)<>5

AND zip<>geoPostalCode;

We got 22 results this time. Some of them, as shown below, obviously has typo problem, or they lose the leading zero. We believe that we could update those to values we got from Reverse Geocoding.



UPDATE fmv2 SET zip=geoPostalCode

WHERE FMID=1002457

OR FMID=1005771

OR FMID=1006166

OR FMID=1007506

OR FMID=1007674

OR FMID=1007830;

For those using short names of states, such as IL, MA, in zip column, we decide to put the value as NULL, if the short name in zip column matches the State name in State column, since it is duplicative information. The reason why we do not copy the zip code from geoPostalCode directly is that we feel that may be too arbitrary because the value in geoPostalCode column are from latitude and longitude, but we did not check whether those latitudes and longitudes match the addresses.

The code we use to update here:

UPDATE fmv2 SET zip=''

WHERE length(zip)=2;

Then we ran the code below again:

SELECT zip, geoPostalCode, cast(zip as numeric)-cast(geoPostalCode as numeric) AS DIFFERENCE

FROM fmv2

WHERE cast(zip as numeric)-cast(geoPostalCode as numeric)>50

OR cast(zip as numeric)-cast(geoPostalCode as numeric)<-50

ORDER BY DIFFERENCE DESC;

This time, we got a result of 331. Besides, accidentally, we found an obvious typo problem:

Screen Shot 2017-05-06 at 9.06.24 PM.png

We choose to update this error using the following query:

UPDATE fmv2 SET zip=geoPostalCode

WHERE zip='726o1';

For the rest of them, we choose to leave them there. But for the sake of potential dataset users, we recommend a warning here to alert users that these zip codes are not that reliable.

### Uniforming format for updateTime column

For updateTime column, we found that some are using words like ‘May’, ‘Apr’ to represent months, rather than using numbers. The reason for we choose to uniform the format of date is that we found that there is comma when using letters to represent months and it caused problems when we try to import data into MySQL. The original dataset is a csv file (Comma Separated Values). Thus, it is hard for the MySQL to really distinguish which comma is for separating columns and which are within the format of dates.

We runs codes below:

SELECT UpdateTime FROM fmv2

WHERE UpdateTime like 'Jan%’;

SELECT UpdateTime FROM fmv2

WHERE UpdateTime like 'Jan%'

OR UpdateTime like 'Feb%’;

…

Every time we adding a new ‘OR’ statement and write down the total number of results we get. And we found that there are only May, Apr and Jun used for months in updateTime columns. Therefore we ran code below:

SELECT UpdateTime FROM fmv2

WHERE UpdateTime like 'Jun%'

OR UpdateTime like 'May%'

OR UpdateTime like 'Apr%';

And we got 312 results in total. Then we ran codes below to update those rows.

UPDATE fmv2 SET UpdateTime=replace(UpdateTime,'Apr ','4/')

WHERE UpdateTime like 'Apr %';

UPDATE fmv2 SET UpdateTime=replace(UpdateTime,'May ','5/')

WHERE UpdateTime like 'May %';

UPDATE fmv2 SET UpdateTime=replace(UpdateTime,'Jun ','6/')

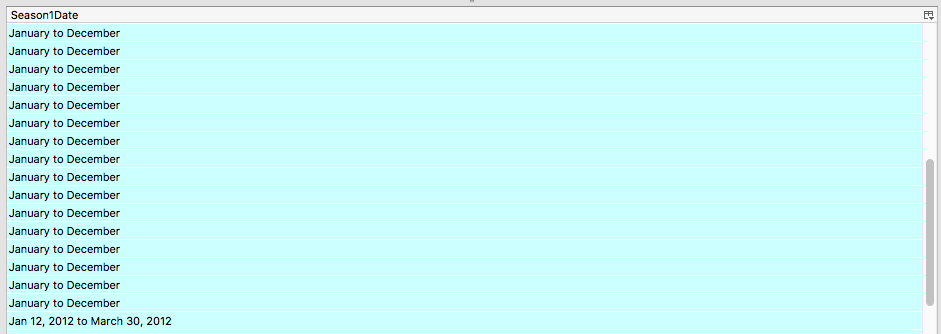
WHERE UpdateTime like 'Jun %';

UPDATE fmv2 SET UpdateTime=replace(UpdateTime,' 20','/20')

WHERE UpdateTime like '% 20%';

### Uniforming format for Season1Date, Season2Date and Season3Date

For Season1Date, similarly, there are also some cells in which months are represnted using words, rather than numbers. See the last row in the following screenshot.



For cells like what is shown in the last row of the screenshot above, we decide to use the similar SQL scripts to update it to the same format as the rest of cells. For those who only have month values, we choose to keep them unchanged since if we change them into numbers, it will be confused for users to understand whether it means days or months.

SELECT Season1Date

FROM fmv2

WHERE Season1Date like 'Jan %';

Screen Shot 2017-05-06 at 9.39.43 PM.png

UPDATE fmv2 SET Season1Date=replace(Season1Date,'Jan ', '01/')

WHERE Season1Date like 'Jan %';

SELECT Season1Date

FROM fmv2

WHERE Season1Date like '%to Oct %';



UPDATE fmv2 SET Season1Date=replace(Season1Date, 'Oct ','10/')

WHERE Season1Date like "% Oct %";

After finishing steps above, we found a problem. Since we cannot use Regular Expression in SQLite Manager, how could we distinguish, for example, “May” in “May 5, 2012 to Oct 6, 2012” from “May” in “May to October” ? We want to change the format of the previous one but do not want to change the latter one. Therefore, we went back to our initial concern: removing the comma, and we ran codes below to fulfil the task:

UPDATE fmv2 SET Season1Date=replace(Season1Date, 'May ','05/')

WHERE Season1Date like "%, 2012";

UPDATE fmv2 SET Season1Date=replace(Season1Date, 'June ','06/')

WHERE Season1Date like "%, 2012";

UPDATE fmv2 SET Season1Date=replace(Season1Date, 'April ','04/')

WHERE Season1Date like "%, 2012";

UPDATE fmv2 SET Season1Date=replace(Season1Date, 'July ','07/')

WHERE Season1Date like "%, 2012";

UPDATE fmv2 SET Season1Date=replace(Season1Date, 'September ','09/')

WHERE Season1Date like "%, 2012";

UPDATE fmv2 SET Season1Date=replace(Season1Date, 'October ','10/')

WHERE Season1Date like "%, 2012";

UPDATE fmv2 SET Season1Date=replace(Season1Date, 'November ','11/')

WHERE Season1Date like "%, 2012";

UPDATE fmv2 SET Season1Date=replace(Season1Date, ', 2012','/2012')

WHERE Season1Date like "%, 2012";

SELECT FMID, Season1Date

FROM fmv2

WHERE Season1Date like "%, 2011";



UPDATE fmv2 SET Season1Date=replace(Season1Date, 'June ','06/')

WHERE Season1Date like "%, 2011";

UPDATE fmv2 SET Season1Date=replace(Season1Date, 'November ','11/')

WHERE Season1Date like "%, 2011";

UPDATE fmv2 SET Season1Date=replace(Season1Date, 'May ','05/')

WHERE Season1Date like "%, 2011";

UPDATE fmv2 SET Season1Date=replace(Season1Date, 'September ','09/')

WHERE Season1Date like "%, 2011";

UPDATE fmv2 SET Season1Date=replace(Season1Date, 'Sept ','09/')

WHERE Season1Date like "%, 2011";

UPDATE fmv2 SET Season1Date=replace(Season1Date, 'Oct. ','10/')

WHERE Season1Date like "%, 2011";

UPDATE fmv2 SET Season1Date=replace(Season1Date, 'October ','10/')

WHERE Season1Date like "%, 2011";

UPDATE fmv2 SET Season1Date=replace(Season1Date, ', 2011','/2011')

WHERE Season1Date like "%, 2011";

Due to codes we run above, in fact we missed some rows in which only the part before “to” contains comma. So we ran the code below:

SELECT FMID, Season1Date

FROM fmv2

WHERE Season1Date like "%,%";



And then we used codes below to fix them:

UPDATE fmv2 SET Season1Date=replace(Season1Date, 'June ','06/')

WHERE Season1Date like "%,%";

UPDATE fmv2 SET Season1Date=replace(Season1Date, 'May ','05/')

WHERE Season1Date like "%,%";

UPDATE fmv2 SET Season1Date=replace(Season1Date, 'July ','07/')

WHERE Season1Date like "%,%";

UPDATE fmv2 SET Season1Date=replace(Season1Date, ', 2011','/2011')

WHERE Season1Date like "%,%";

UPDATE fmv2 SET Season1Date=replace(Season1Date, ', 2012','/2012')

WHERE Season1Date like "%,%";

We ran the similar queries for Season2Date and Season3Date:

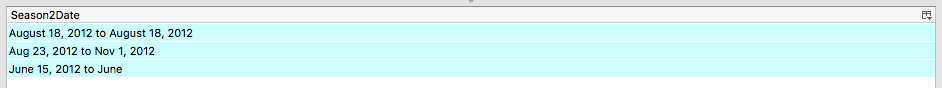
Season2Date:

SELECT Season2Date

FROM fmv2

WHERE Season2Date like "%,%";

Result:



Queries to fix:

UPDATE fmv2 SET Season2Date=replace(Season2Date, 'Nov ','11/')

WHERE Season2Date like "%,%";

UPDATE fmv2 SET Season2Date=replace(Season2Date, 'Aug ','08/')

WHERE Season2Date like "%,%";

UPDATE fmv2 SET Season2Date=replace(Season2Date, 'August ','08/')

WHERE Season2Date like "%,%";

UPDATE fmv2 SET Season2Date=replace(Season2Date, 'June ','06/')

WHERE Season2Date like "%,%";

UPDATE fmv2 SET Season2Date=replace(Season2Date, ', 2012','/2012')

WHERE Season2Date like "%,%";

Season3Date:

SELECT Season3Date

FROM fmv2

WHERE Season3Date like "%,%";

Result:

Screen Shot 2017-05-10 at 2.54.28 PM.png

Queries to fix:

UPDATE fmv2 SET Season3Date=replace(Season3Date, 'July ','07/')

WHERE Season3Date like "%,%";

UPDATE fmv2 SET Season3Date=replace(Season3Date, 'September ','09/')

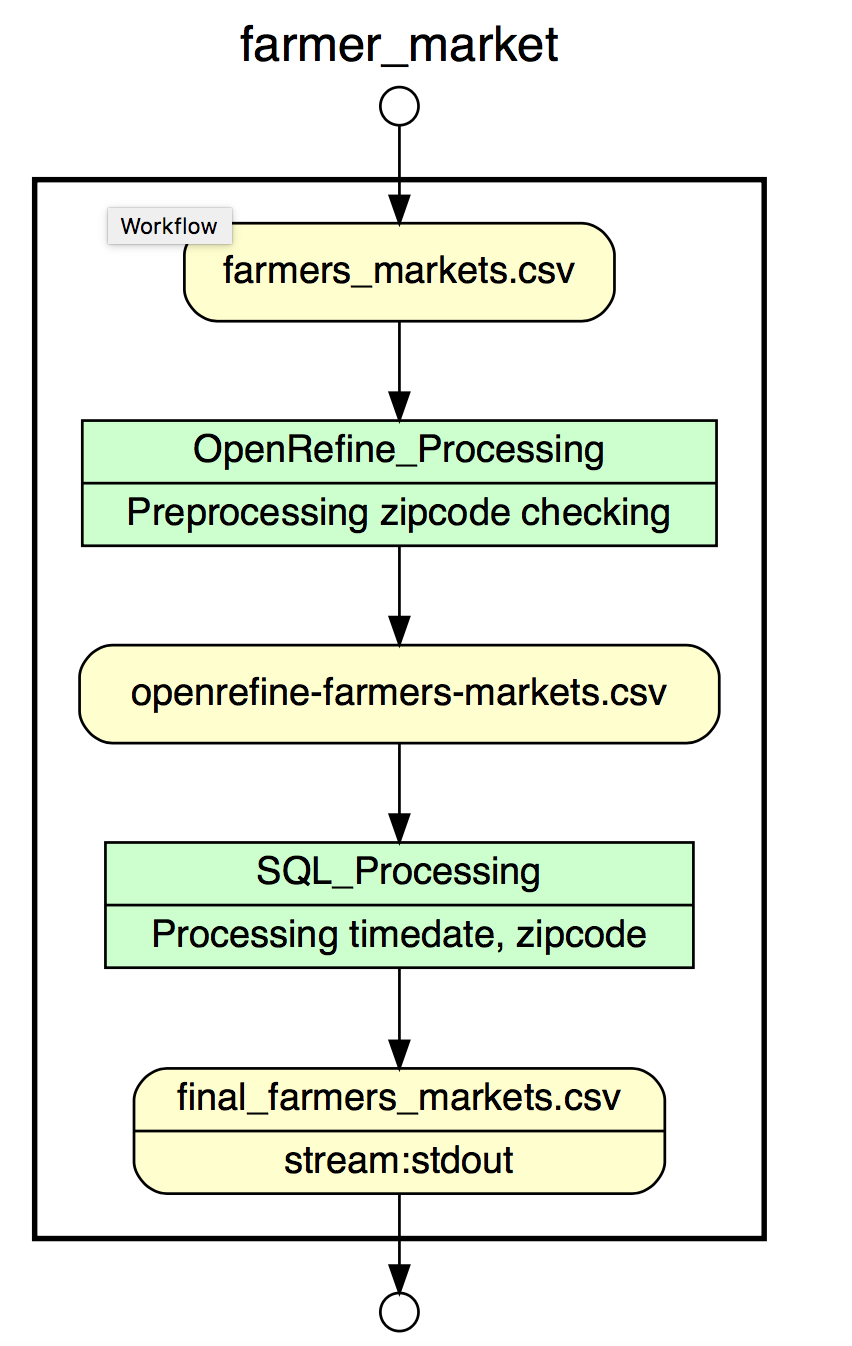
WHERE Season3Date like "%,%";

UPDATE fmv2 SET Season3Date=replace(Season3Date, ', 2012','/2012')

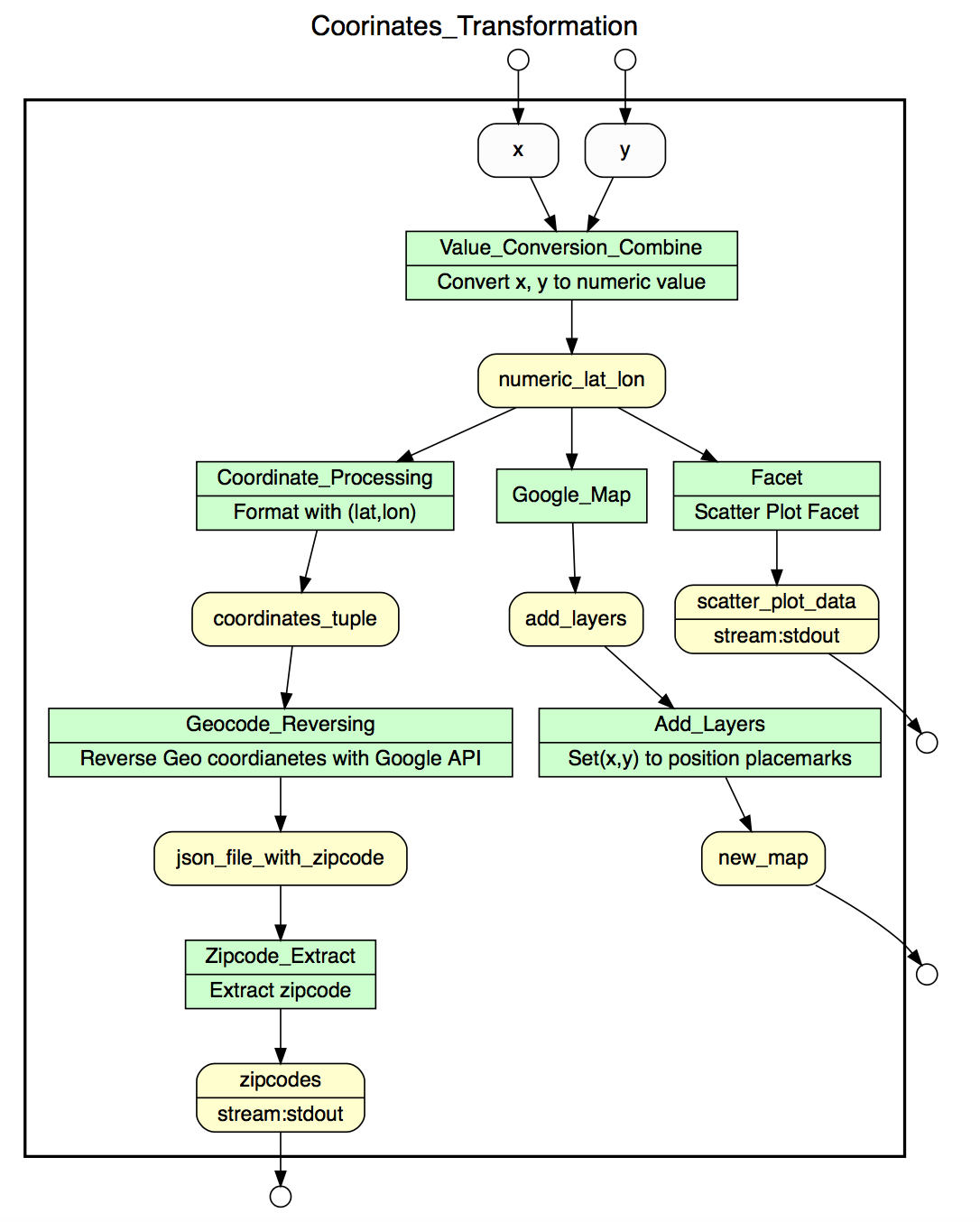
WHERE Season3Date like "%,%";

# 4 Data Cleaning Workflow

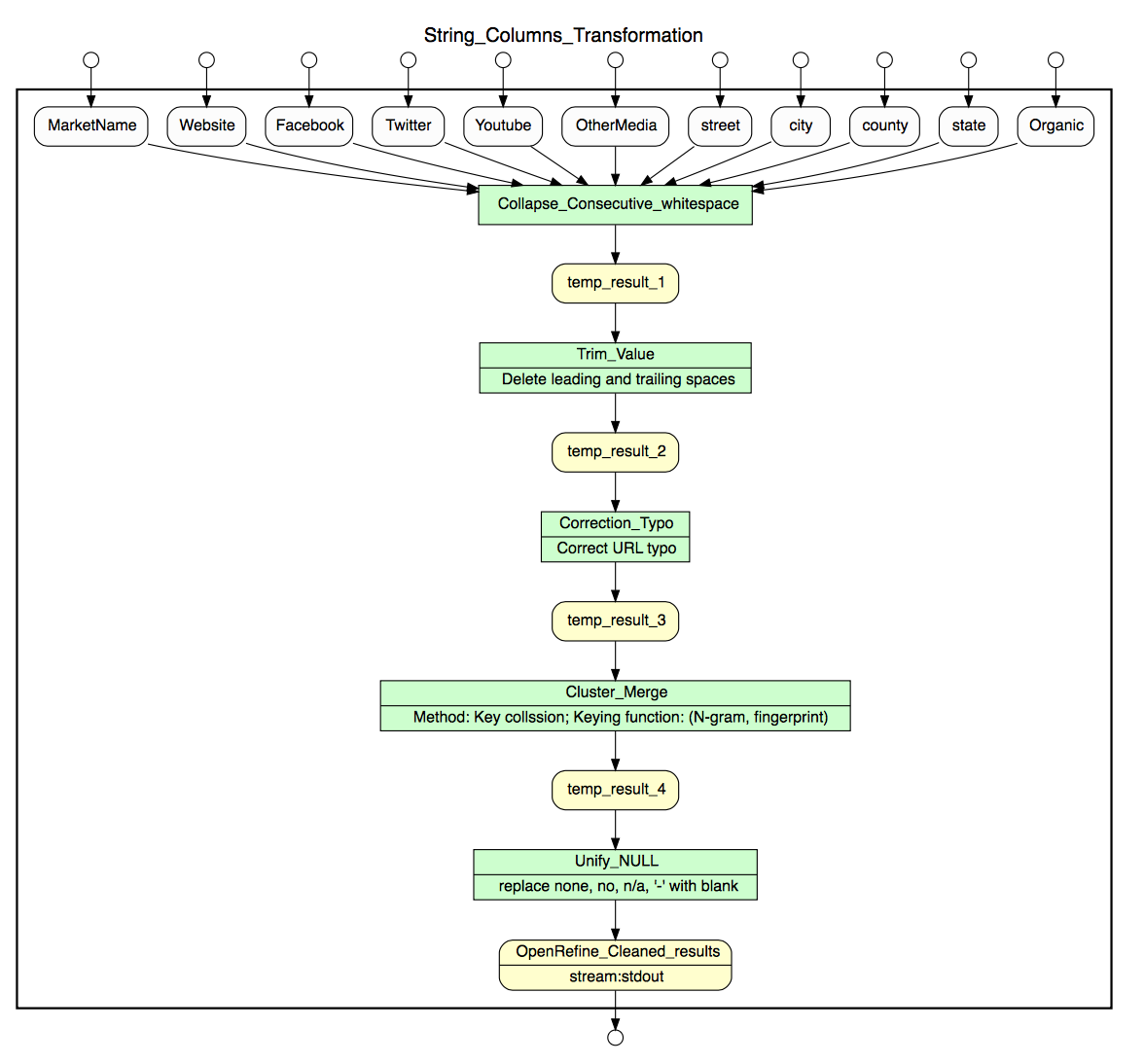
## 4.1 Overall Cleaning Processing



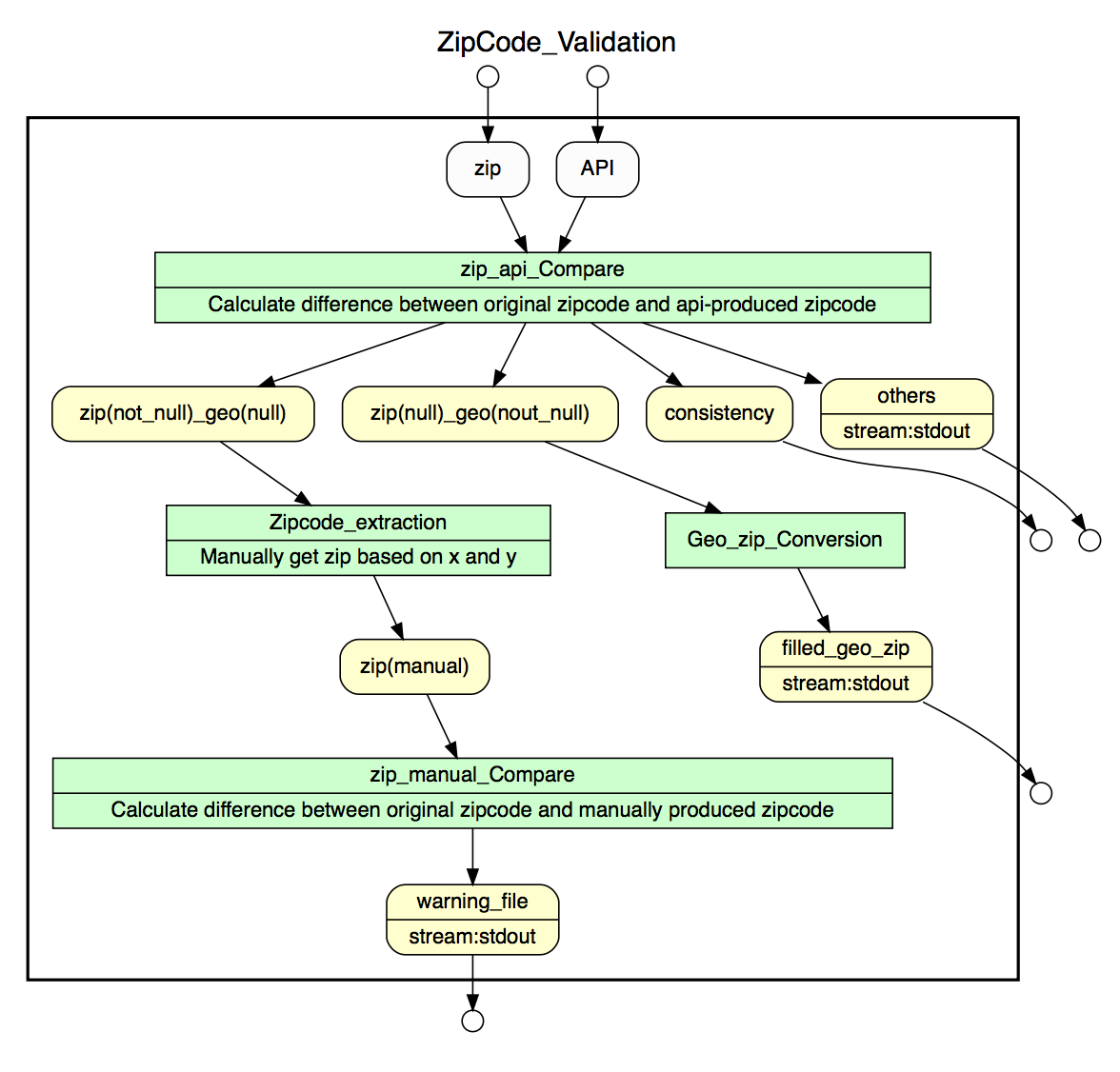
## 4.2 Coordinates Transformation in OpenRefine and Google Map



## 4.3 String Columns Transformation in OpenRefine



## 4.4 Zip Code Validation in SQLite



## 4.5 Date Format Transformation in SQLite

