TMC Engineering Hiring Exercise

Craig Wilding

# Task 1

One of our ETL scripts failed while bringing new data into the sample\_data.email\_activity table in

Redshift, producing an error message that references further diagnostic information available from the stl\_load\_errors system table.

When we query stl\_load\_errors for more details, this is what we see:



**What Redshift/SQL query would you run to resolve the issue before re-running the failed**

**portion of the data sync?**

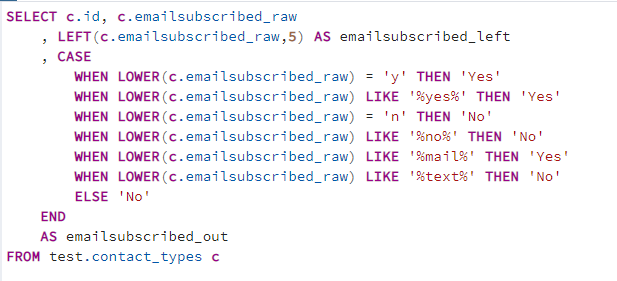
The problem is the raw value, ‘by-mail’ is larger than the column length (5) for the emailsubscribed column it is being loaded into.

The quick solution would be to simply truncate the raw field to a length of 5 so it fits. That is a simple LEFT() command in SQL. However, this still loads ambiguous on unusable data into the result table.

Along with loading the data, we also want to transform the data into meaningful and usable values so analysts do not have to repeatedly perform complicated transforms later. Because we are getting data from multiple sources, not all sources will have the values in the same format or options the analysts need. The ETL process needs to transform the different data sets before loading into the format the analysts need.

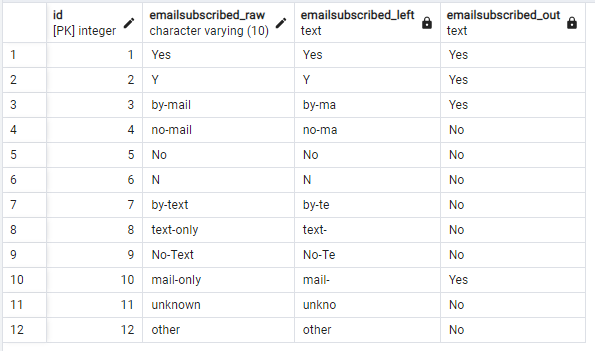
The better solution is to try and determine what end value for email subscribed the raw value is supposed to map to. Assuming the desired values for email subscribed are either Yes or No, then the way to do this in SQL is with a CASE statement that maps the incoming raw values to the allowed result values. In this case, I added a new case statement to map anything containing ‘mail’ to Yes after the rule that maps anything with ‘no’ to No. The default in this case is No since we have to assume they have not subscribed if we do not have a positive confirmation.

## SQL SCRIPT



## Results

I created a table with some sample raw values that I would expect for the field. The results below show the difference between the raw value, what a straight left truncation would give, and what the mapping would give for the raw values. The last column is the preferred answer for this question.



# Task 2

Processing text data can be challenging, and our syncs often take steps to normalize data.

**What Python code would you run to strip all formatting from a set of U.S. phone numbers and store them as a consistent 10-digit string?**

I wrote two python functions, one to normalize the data and one to format the phone number. The format method calls normalize to ensure that the data is normalized before formatting. This way programmers don’t have to remember to call normalize first.

## Normalize

In the normalize method, I mainly use the isdigit() method to filter out any non-digit characters. The exception is I converted any instances of the character ‘O’ to a zero, as this often gets typed incorrectly in the data. I tried to use the regex library for parsing out digits faster, but the resulting outcome from regex findall() for values that had invalid characters such as # or ‘-‘ was inconsistent so it seemed to take more time to reassemble the regex results into a valid phone number.

I did some data correction based on the size of the phone number coming in. If it was over 10 digits, I assume it was an international number and truncated it to 10 characters starting from the right. This removes leading country codes. If it was a 7-digit number, I added a default local area code. Otherwise if it did not have enough valid digits, I return a blank string.

## Format

I provided two possible formats, one with dashes and one with the area code in parenthesis. It will default to the dash format. I use format option codes which would be similar to how operating systems pass a value to indicate the localized phone number format. It would be better to have a method that pulls the format from the code then parse the phone number using sprint rather than breaking out each format as I did. I moved on to the next question instead.

## Code

The python code is attached in the file **TMC\_Q2\_phones.py**

It uses the input file **TMC\_Q2\_sample.csv** as sample data and **TMC\_Q2\_results.csv** as the output

import os

import shutil

import csv

import re # regex

from csv import DictReader

from csv import DictWriter

wrksp = r"G:\Safe Documents\Resume\PoliticJobs\DataEngineer\Tests\TMC"

sampleFileIn = os.path.join(wrksp, "TMC\_Q2\_sample.csv")

sampleFileOut = os.path.join(wrksp, "TMC\_Q2\_results.csv")

TAB = "\t"

DLM = "|"

EOL = '\n'

NULL = "NULL"

UP\_O = 'O'

Lower\_O = 'o'

ZERO = '0'

LOCAL\_AREA\_CODE = '888'

FORMAT\_DASH = 1

FORMAT\_AREA = 2

def normalizePhone(phoneIn) :

phoneOut = ""

try:

# parse for digits only

for char in phoneIn :

if (char.isdigit()) :

phoneOut += char

elif (UP\_O == char) or (Lower\_O == char) :

# convert letter O to zero

phoneOut += ZERO

# end if

# end for each char

# 7-digits assumes the area code was left off and will use a default local area code

if (len(phoneOut) == 7):

phoneOut = LOCAL\_AREA\_CODE + phoneOut

# truncate to 10 digits

# assumes international numbers have leading digits to truncate.

# so it will truncate from the left, pulling the right-most 10 digits

elif (len(phoneOut) > 10) :

phoneOut = phoneOut[-10:]

# check for min and max digits

# this will blank out the phone number as invalid

# over 16 assumes an invalid entry as international numbers are less than this.

elif (len(phoneOut) < 10) or (len(phoneOut) >= 16) :

phoneOut = ""

# end if

except:

print("ERROR converting phone value: " + phoneIn)

phoneOut = ""

# end try

return phoneOut

# end normalizePhone

def formatPhone(phone, formatCode) :

# for user convenience, this will assume the phone needs to be normalized first

# this way, it ensures the phone numbers are valid before formatting

phoneNorm = normalizePhone(phone)

phoneOut = phoneNorm

if (len(phoneOut) == 10) : # valid phone has 10 digits

if (FORMAT\_AREA == formatCode) :

phoneOut = "(" + phoneNorm[:3] + ") " + phoneNorm[3:6] + "-" + phoneNorm[6:]

# end if

else : # use FORMAT\_DASH

phoneOut = phoneNorm[:3] + "-" + phoneNorm[3:6] + "-" + phoneNorm[6:]

# end if

# end if valid phone

return phoneOut

# end formatPhone

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

# read each sample phone number and format it.

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

with open(sampleFileOut, 'w', newline='') as write\_csv:

# field names

fields = ['name', 'raw', 'normalized', 'dash-foramt', 'area-foramt']

# write column headers

csvwriter = csv.DictWriter(write\_csv, fieldnames = fields)

csvwriter.writeheader()

rowsOut = []

with open(sampleFileIn, 'r') as read\_obj:

csv\_dict\_reader = DictReader(read\_obj)

for row in csv\_dict\_reader:

name = row["name"]

phoneRaw = row["phone"]

# normalize phone number

phoneOut = normalizePhone(phoneRaw)

testOut = "Test: " + name + TAB+ "Raw[" + phoneRaw + "]" + TAB + "Norm[" + phoneOut + "]"

testOut += TAB + "Dash[" + formatPhone(phoneRaw, FORMAT\_DASH) + "]"

testOut += TAB + "Area[" + formatPhone(phoneRaw, FORMAT\_AREA) + "]"

print(testOut)

# Write results to csv

rowOut = {}

rowOut["name"] = name

rowOut["raw"] = phoneRaw

rowOut["normalized"] = phoneOut

rowOut["dash-foramt"] = formatPhone(phoneRaw, FORMAT\_DASH)

rowOut["area-foramt"] = formatPhone(phoneRaw, FORMAT\_AREA)

rowsOut.append(rowOut)

# end for each row

# end with csv read

#Write as CSV file

csvwriter.writerows(rowsOut)

print("WRITE file: " + sampleFileOut)

# end with write csv

del write\_csv

del rowsOut

# Task 3

One of our syncs is broken, and we need to communicate the outage to TMC’s member organizations

in our #bugs-and-outages Slack channel.

Here’s what we know:

- A vendor changed how they configure security on their side for a database mirror, and they

will now require an SSL certificate

- We run the data sync via a Civis Platform job template, and Civis support staff need to talk to

the vendor’s Engineering team about how to handle the certificate

- TLDR; we’re relying on outside entities to fix things that we don’t directly control, so we do

not have a clear timeline for resolution

**Can you write a member-facing message about this outage?**

## Response

**#bugs-and-outages**

We are experiencing an outage in our data syncs due to a connection problem with one of our vendors. There is no loss of data, but the data will not be updated and synced until the problem is resolved. The process is controlled by our Civis Platform and we are actively working with their support staff to resolve the problem as quickly as possible with the vendor. We do not yet have a timeline for a resolution, but we will keep you posted of any progress.

# Task 4

Build a pair of scripts that will:

1. pull voter file data from the Ohio Secretary of State website, and

2. match a provided input CSV file to that voter data, creating another CSV which looks like the input (including the row column) but has an extra column **matched\_voterid** that specifies the matches.

3. include a README explaining your approach.

The input CSV file: https://drive.google.com/open?id=1o3SWFV1oJ4Z3hr6nFPAQPO8y8wWtlfgL

For gathering the voter file: Ohio is one of the few states that makes it really easy for anyone to

download the voter file, which is the list of registered voters in the state.

● First check out https://www6.sos.state.oh.us/ords/f?p=111:1 to see where the data comes from, and what is the data format.

● Then you can download each county’s data by using URLs like

https://www6.sos.state.oh.us/ords/f?p=VOTERFTP:DOWNLOAD::FILE:NO:2:P2\_PRODUC

T\_NUMBER:1, where that last number is a county number from 1 to 88. For performance

reasons, you can limit your matching to the first 4 counties’ data.

Matching is a “fuzzy” process that often doesn’t have a clear right answer, so you’ll have to weigh tradeoffs and make decisions, eg. how much to normalize strings, how much to weigh certain columns, what to do when there isn’t a clear match, etc. Make decisions that seem reasonable to you, and document any interesting tradeoffs.

Spend no more than a few hours on this test, and feel free to wrap up sooner if you have a solid first pass. You will certainly have further improvements you would make with more time and with more input from users. As part of your README, explain those potential improvements and open questions. Those are as important as the code.

## README

TMC\_Q4\_README.txt

## Script:

TMC\_Q4\_MatchVoters.py

## Results:

TMC\_Q4\_Results.csv

## Code:

import os

import shutil

from datetime import datetime

import csv

from csv import DictReader

from csv import DictWriter

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

# TMC Q4:

# Match list of names, birth year, and address to Ohio Voter files

# Steps:

# Read voter info from voter files

# Build a data dictionary lookup based on birtyYear+zip5 as the key

# Read match file info

# lookup maches based on birtyYear+zip5 key

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

dirTMC = r"G:\Safe Documents\Resume\PoliticJobs\DataEngineer\Tests\TMC"

dirVoterFiles = os.path.join(dirTMC, "Ohio\_VoterFiles")

fileNameIn = os.path.join(dirTMC, "eng-matching-input-v3.csv")

fileNameOut = os.path.join(dirTMC, "TMC\_Q4\_Results.csv")

dirOut = dirTMC

wrksp = dirTMC

os.chdir(wrksp)

print(os.getcwd())

if not os.path.exists(dirOut):

os.makedirs(dirOut)

TAB = "\t"

DLM = "|"

EOL = '\n'

SPACE = ' '

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

# Voter info

# This class holds the voter's info from the voter file

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

class VoterInfoClass:

def \_\_init\_\_(self) :

self.voterID = ""

self.countyID = ""

self.lastName = ""

self.firstName = ""

self.dateOfBirth = ""

self.birthYear = ""

self.zip5 = ""

self.key = ""

# end init

def GetData(self, rowOut):

contacts = self.contactCount()

rowOut['contactCount'] = contacts

rowOut['phone'] = self.phone

rowOut['email'] = self.email

return rowOut

# end GetData()

def GetHeader(self):

header = ['contactCount', 'phone', 'email']

return header

# end GetHeader()

#end VoterInfoClass

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

# parseLastName

# Parse the last name from the input file which has it as one value

# Examples:

# Jeremy T Patterson -> Patterson

# Juanita J Dettwiller -> Dettwiller

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

def parseLastName(name) :

# Assumes last name is separated by a blank space from the rest of the name

lastName = ""

ixSpace = name.rfind(SPACE) # finds last instance of SPACE

if (ixSpace > -1) :

lastName = name[ixSpace+1:]

#endif

return lastName

# end parseLastName

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

# matchLastName

# Check if the last names match

# Do any cleaning on the names here to help ensure they match

# Examples:

# user lower to prevent case mismatch

# Remove any suffix such as JR. SR.

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

def matchLastName(matchLastNameIn, VoterLastNameIn) :

#

match = False

matchLastName = matchLastNameIn.lower().replace('.',"").replace("jr","").replace("sr","")

VoterLastName = VoterLastNameIn.lower().replace('.',"").replace("jr","").replace("sr","")

if (VoterLastName == matchLastName) :

match = True

#endif

return match

# end parseLastName

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

# data Dictionary

# This is a key-value dictionary

# key = birthYear + zip5

# returns List of VoterInfoClass objects with that key

# Example:

# key[1994+33344] = [Voter1, Voter2, Voter3]

# Voter# = an instance of VoterInfoClass

#

# dictVotersByBirth

# same as above, but the key is just the birth year.

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

dictVotersByBirthZip = {}

dictVotersByBirth = {}

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

# Get Voter info

# Recurse through voter files.

# This builds the data dictionary used for lookup

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

for countyFile in os.listdir(dirVoterFiles) :

fnameCounty = os.path.join(dirVoterFiles, countyFile)

with open(fnameCounty, 'r') as read\_obj:

csv\_dict\_reader = DictReader(read\_obj)

for row in csv\_dict\_reader:

Voter = VoterInfoClass()

Voter.voterID = row["SOS\_VOTERID"]

Voter.countyID = row["COUNTY\_ID"]

Voter.lastName = row["LAST\_NAME"]

Voter.firstName = row["FIRST\_NAME"]

Voter.dateOfBirth = row["DATE\_OF\_BIRTH"]

# Date Format: "1993-06-02"

Voter.birthYear = Voter.dateOfBirth[:4]

Voter.zip5 = row["RESIDENTIAL\_ZIP"]

Voter.key = Voter.birthYear + "+" + Voter.zip5

if (Voter.key not in dictVotersByBirthZip) :

# new key

voterList = []

voterList.append(Voter)

dictVotersByBirthZip[Voter.key] = voterList

else :

# add Voter to list of matches

voterList = dictVotersByBirthZip[Voter.key]

voterList.append(Voter)

dictVotersByBirthZip[Voter.key] = voterList

# end if

if (Voter.birthYear not in dictVotersByBirth) :

# new key

voterList = []

voterList.append(Voter)

dictVotersByBirth[Voter.birthYear] = voterList

else :

# add Voter to list of matches

voterList = dictVotersByBirth[Voter.birthYear]

voterList.append(Voter)

dictVotersByBirth[Voter.birthYear] = voterList

# end if

# end for each row

# end read csv

# end county voter files in dir

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

# Read input file

# Lookup BirthYear+zip5 for possible matches

# Check last name for matches

# Write matches to output file

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

with open(fileNameOut, 'w', newline='') as write\_csv:

# field names

fields = ['row', 'name', 'birth\_year']

fields += ['address', 'city', 'state', 'zip']

fields += ['matched\_voterid']

# write column headers

csvwriter = csv.DictWriter(write\_csv, fieldnames = fields)

csvwriter.writeheader()

rowsOut = []

with open(fileNameIn, 'r') as read\_obj:

csv\_dict\_reader = DictReader(read\_obj)

for row in csv\_dict\_reader:

# read the line in, save the values for output.

rowID = row["row"]

name = row["name"]

birth\_year = row["birth\_year"]

address = row["address"]

city = row["city"]

state = row["state"]

zip5 = row["zip"]

matched\_voterid = ""

key = birth\_year + "+" + zip5

lastName = parseLastName(name)

if (key in dictVotersByBirthZip) :

# MATCH 1: Birth Year + Zip5 + Last Name

voterList = dictVotersByBirthZip[key]

matchCount = 0

for Voter in voterList :

# compare name to voter file

if matchLastName(lastName, Voter.lastName) :

matchCount += 1

matched\_voterid = Voter.voterID

if (matchCount > 1) :

# multiple match on last name found

# TODO: do 2nd level matching here

# reset matched voter id

matched\_voterid = ""

# end matchCount

# end matchLastName

# end for voterList

elif (birth\_year in dictVotersByBirth) :

# Match 2: Birth Year + Last Name

voterList = dictVotersByBirth[birth\_year]

matchCount = 0

for Voter in voterList :

# compare name to voter file

if matchLastName(lastName, Voter.lastName) :

matchCount += 1

matched\_voterid = Voter.voterID

if (matchCount > 1) :

# multiple match on last name found

# TODO: do 2nd level matching here

# reset matched voter id

matched\_voterid = ""

# end matchCount

# end matchLastName

# end for voterList

# end if key found

rowOut = {}

rowOut["row"] = rowID

rowOut["name"] = name

rowOut["birth\_year"] = birth\_year

rowOut["address"] = address

rowOut["city"] = city

rowOut["zip"] = zip5

rowOut["matched\_voterid"] = matched\_voterid

rowsOut.append(rowOut)

# end for match file

#Write as CSV file

csvwriter.writerows(rowsOut)

print("WRITE file: " + fileNameOut)

# end with write csv

del write\_csv

del rowsOut

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("Finished")

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

# END - Cleanup

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

del dictVotersByBirthZip

del dictVotersByBirth