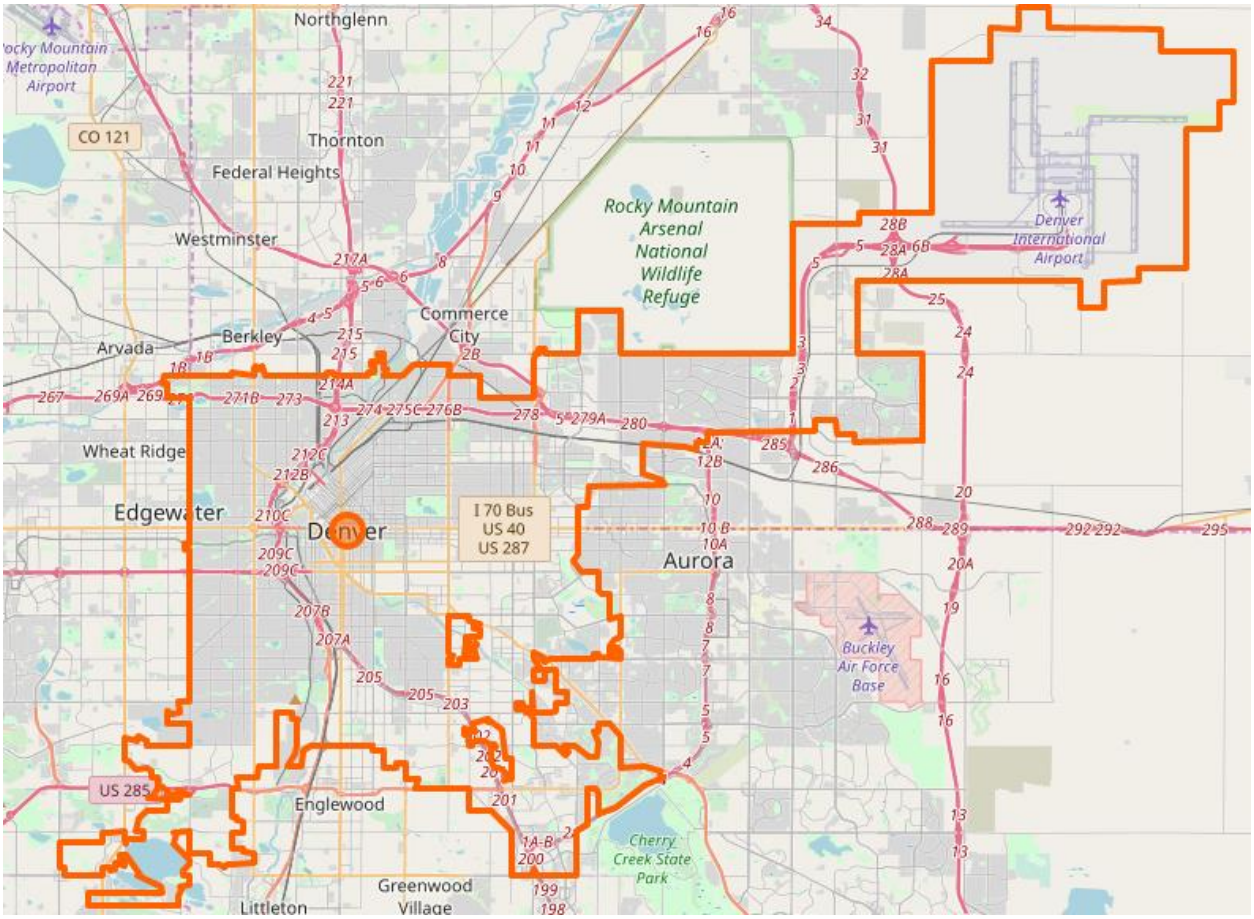
SQL Data Wrangling – Denver



Project Summary:

-For my project I decided to use the area surrounding Denver, CO. I've been to Denver before but the main reason I used this area is because I was unable to use my home location. Either way Denver proved to be very interesting and I would love to visit again.

Data Exploration

-Data was collected by OpenStreetMap, I was able to pull the data from within Denver's city limit. The file downloaded as a PBF file, which I was able to convert into OSM using osmconvert64. I then used the provided script to scale the file down to a much more manageable size. I have a few summary statistics below:

Note: *I used the sample for my calculations as well to save time running code, this can be changed if needed.*

Node Types & Quantities:

```
{'member': 4119,
 'nd': 294123,
 'node': 262658,
 'osm': 1,
 'relation': 170,
 'tag': 127280,
 'way': 32322}
```

Problematic Characters:

```
{'problemchars': 0, 'lower': 105140, 'other': 1550, 'lower_colon': 50368}
```

Cleaning Street Names

-The data overall looked pretty good, as you can see no problematic characters were found, however there were a couple things I wanted to change. It's just a personal preference for me but I like to see full street names without abbreviation, which is why I have implemented my fixer function to change the streets that do not conform.

```
'Ave': {'East Hampden Ave'},
'Broadway': {'Broadway', 'South Broadway'},
'Ct': {'Ash Ct',
 'South Knox Ct',
 'South Patton Ct',
 'South Winona Ct',
 'South Zurich Ct'},
'D': {'West Center Apt D'},
'East': {'Inverness Drive East',
 'Nassau Circle East',
 'Sanford Circle East',
 'Ulster Circle East'},
'Ford': {'West Ford'},
```

-My fixer function is very simple; it looks for the values I have decided are problematic and replaces that portion with the correct value. This works well if you have already identified the problems in your data, however it would not easily translate between data sets without some changes.

```
def fixer(name, mapping):

    for k, v in mapping.iteritems():
        if k in name:
            name = name.replace(k, v)
    return name

mapping =  {"Ct": "Court", "Ln": "Lane", "Hampden Ave": "Hampden Avenue"}
```

Data Overview and Thoughts

File Sizes:

DENVER.OSM	1.12GB
MINIDENVER.OSM	57.3MB
DENVER.DB	33.2MB
NODES.CSV	22.3MB
NODES_TAGS.CSV	1.02MB
WAYS.CSV	1.99MB
WAYS_NODES	7.03MB
WAYS_TAGS	2.71MB
DENVERCOMPRESS.ZIP	99.2MB

-The code then extracts the values from my osm file, makes corrections, and finally places them into csv files, one for each table in my SQL database. The Denver.db database contains the data, which I created using the code below:

```
import sqlite3

sqlite_file = 'Denver.db'

# Connect to the database
conn = sqlite3.connect(sqlite_file)

# Get a cursor object
cur = conn.cursor()

cur.execute(''' CREATE TABLE nodes(id INTEGER, lat INTEGER, lon INTEGER, user TEXT,
                uid INTEGER, version INTEGER, changeset TEXT, timestamp TEXT) ''')
cur.execute(''' CREATE TABLE nodes_tags(id INTEGER, key INTEGER, value TEXT, type TEXT) ''')
cur.execute(''' CREATE TABLE ways(id INTEGER, user TEXT, uid INTEGER, version INTEGER, changeset TEXT, timestamp TEXT) ''')
cur.execute(''' CREATE TABLE ways_nodes(id INTEGER, node_id INTEGER, position INTEGER) ''')
cur.execute(''' CREATE TABLE ways_tags(id INTEGER, key INTEGER, value TEXT, type TEXT) ''')

# commit the changes
conn.commit()
```

-After committing the changes to the newly created tables I inserted the data using the code below:

```
with open('nodes.csv','rb') as fin:
    dr = csv.DictReader(fin) # comma is default delimiter
    to_db = [(i['id'].decode("utf-8"), i['lat'].decode("utf-8"), i['lon'].decode("utf-8"), i['user'].decode("utf-8"),
              i['uid'].decode("utf-8"), i['version'].decode("utf-8"), i['changeset'].decode("utf-8"),
              i['timestamp'].decode("utf-8")) for i in dr]

cur.executemany("INSERT INTO nodes(id, lat, lon, user, uid, version, changeset, timestamp)
                 VALUES (?, ?, ?, ?, ?, ?, ?, ?);", to_db)

with open('nodes_tags.csv','rb') as fin:
    dr = csv.DictReader(fin) # comma is default delimiter
    to_db = [(i['id'].decode("utf-8"), i['key'].decode("utf-8"), i['value'].decode("utf-8"),
              i['type'].decode("utf-8")) for i in dr]

cur.executemany("INSERT INTO nodes_tags(id, key, value, type) VALUES (?, ?, ?, ?);", to_db)

with open('ways.csv','rb') as fin:
    dr = csv.DictReader(fin) # comma is default delimiter
    to_db = [(i['id'].decode("utf-8"), i['user'].decode("utf-8"), i['uid'].decode("utf-8"), i['version'].decode("utf-8"),
              i['changeset'].decode("utf-8"), i['timestamp'].decode("utf-8")) for i in dr]

cur.executemany("INSERT INTO ways(id, user, uid, version, changeset, timestamp) VALUES (?, ?, ?, ?, ?, ?);", to_db)

with open('ways_nodes.csv','rb') as fin:
    dr = csv.DictReader(fin) # comma is default delimiter
    to_db = [(i['id'].decode("utf-8"), i['node_id'].decode("utf-8"), i['position'].decode("utf-8")) for i in dr]

cur.executemany("INSERT INTO ways_nodes(id, node_id, position) VALUES (?, ?, ?);", to_db)

with open('ways_tags.csv','rb') as fin:
    dr = csv.DictReader(fin) # comma is default delimiter
    to_db = [(i['id'].decode("utf-8"), i['key'].decode("utf-8"), i['value'].decode("utf-8"),
              i['type'].decode("utf-8")) for i in dr]

cur.executemany("INSERT INTO ways_tags(id, key, value, type) VALUES (?, ?, ?, ?);", to_db)

conn.commit()
```

Findings

-After inserting the data into the database it's time to ask some questions and make some queries. I have attached below the code for each query, which I separated into their own functions:

```
def number_of_nodes():
    result = cur.execute('SELECT COUNT(*) FROM nodes')
    return result.fetchone()[0]

def number_of_ways():
    result = cur.execute('SELECT COUNT(*) FROM ways')
    return result.fetchone()[0]

def number_of_unique_users():
    result = cur.execute('SELECT COUNT(distinct(uid)) FROM (SELECT uid FROM nodes UNION ALL SELECT uid FROM ways)')
    return result.fetchone()[0]

def single_time_users():
    result = cur.execute('SELECT COUNT(*) FROM (SELECT k.user, COUNT(*) as num FROM (SELECT user
                          FROM nodes UNION ALL SELECT user FROM ways) as k GROUP BY k.user HAVING num=1);')
    return result.fetchone()[0]

def number_of_hydrants():
    result = cur.execute('SELECT COUNT(*) FROM nodes_tags WHERE value == ("fire_hydrant" or "hydrant")')
    return result.fetchone()[0]

def predominate_religion():
    result = cur.execute('SELECT value, COUNT(value) FROM nodes_tags WHERE key == "religion"
                          GROUP BY value ORDER BY 2 DESC LIMIT 1')
    return result.fetchone()[0]
```

-I then call each function which queries the database to find my answers:

```
print "Number of nodes: " ,number_of_nodes()
print "Number of ways: " ,number_of_ways()
print "Number of unique users: " ,number_of_unique_users()
print "Number of one-time contributors: " ,single_time_users()
print "Number of Tracked Denver Fire Hydrants: " ,number_of_hydrants()
print "Religion with the most locations: " ,predominate_religion()
```

```
Number of nodes:  262658
Number of ways:  32322
Number of unique users:  1432
Number of one-time contributors:  363
Number of Tracked Denver Fire Hydrants:  35
Religion with the most locations:  christian
```

Thoughts

-OpenStreetMap is a great resource for data collection and analyzing, I think what holds it back is the amount of user generated data, there isn't much. I'll admit I had never heard of it, and by asking around I find no one else has either. An attempt to increase visibility of this resource could go a long way in remedying this, whether it be an ad or social media campaign. This is a simple problem with a simple solution.

-Data validity and integrity are also big factors to consider when looking at data from OpenStreetMap as it is open source and relies on the public to populate their maps. Setting rules and standards for data would be a start, but I believe to standardize all entries means to lie down so much red tape that it could discourage participation.
I would begin somewhere simple, roadway standardized naming conventions. This is where I saw the most confusion within the data, for example (Road, Rd, Rd.). A simple cleaning function could be used as data is populated to convert values entered by users. This would not require users to do anything but enter data, as it could be cleaned during entry.

-The two improvements I suggest would increase the user base and set a single standard that could be expanded to include corrections for the rest of the data as well. It's not a complete solution by any means but could set the foundation for a good start.