OpenStreetMap Data Case Study

By Eric Fraser

Map Area

Originally I planned on analyzing the OSM data at the link below for San Diego, CA because it is the city in which I currently live:

https://www.openstreetmap.org/relation/253832

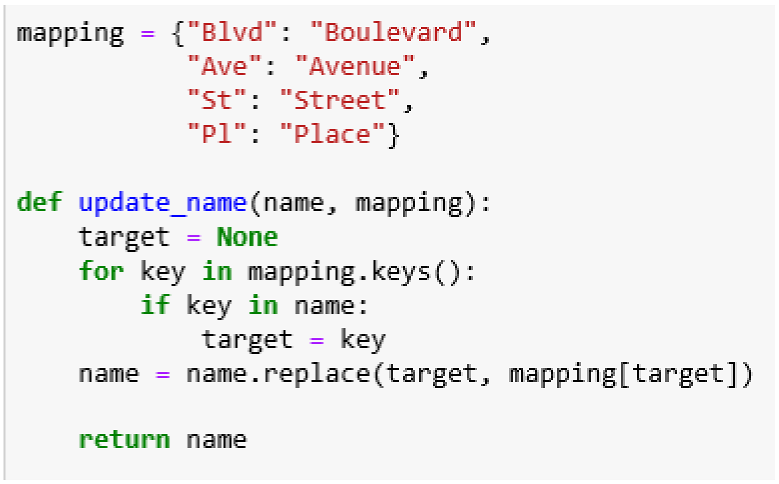
However, the map size contained more data than would be able to be compressed for the project, so I had to find an alternative.

I chose the Denver Colorado OSM because I spent 9 years living there. Because of my relative familiarity with the city, I'm interested and curious to see what kinds of conclusions we can draw from the data we wrangle and query.

Problems Encountered in the Project

1. Street names inconsistently abbreviated

I was surprised how consistent street names in this data set were in general. However, there were still several examples of over-abbreviation that had to be cleaned and corrected. After auditing the file, there were only four legitimate opportunities for adjustments as listed in the mapping dictionary below. Several streets were abbreviated using Blvd, Ave, St, or Pl. Using the update_name function and the mapping dictionary I was able to update the OSM file with the preferred unabbreviated street name types.



```python
mapping = {"Blvd": "Boulevard",
           "Ave": "Avenue",
           "St": "Street",
           "Pl": "Place"}

def update_name(name, mapping):
    target = None
    for key in mapping.keys():
        if key in name:
            target = key
    name = name.replace(target, mapping[target])

    return name
```

2. Udacity OSM module is out of date

It was a struggle to get the correct XML file type and size initially. The module in Udacity walks through instructions for this step-by-step, but OSM has changed drastically since the Udacity module has been updated. I eventually navigated my way to the correct page, downloaded a PBF type file, and had to scour the internet for a conversion tool to convert the file I downloaded from OSM to actual workable XML. I would suggest Udacity update the module so as not to confuse and frustrate future students.

3. Python Version syntax issues

Much of the sample code given in the data.py file was python code designed for implementation in Python 2.7. I actually tried to create a new kernel within Jupyter Notebook using the command line to downgrade my current version (3.4), but after some research this proved to be not worth the trouble, and I decided to run with Python 3. After completing the actual programming work, I started to get errors that I had not seen before. It took a few hours of trial and error, research, and looking at Python upgrade documentation to switch the tiny details in the sample code so that they would run properly in Python 3.

For example:

In Python 2.7, the correct syntax for the highlighted function below is ".iteritems()". I scoured the programs and changed all the areas that were ".iteritems()" to the correct Python 3 syntax: .items().

```
def validate_element(element, validator, schema=SCHEMA):
    """Raise ValidationError if element does not match schema"""
    if validator.validate(element, schema) is not True:
        field, errors = next(validator.errors.items())
        message_string = "\nElement of type '{0}' has the following errors:\n{1}"
        error_string = pprint.pformat(errors)

        raise Exception(message_string.format(field, error_string))
```

4. Data in database was imported on only ODD row_ids

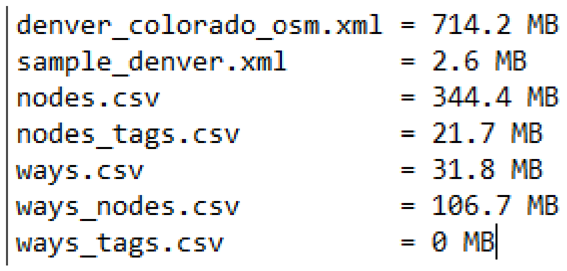
When the CSV's containing our data were entered into the database, the data only populated every other row. By this I mean in every table, ROW_ID 1 contained data, ROW_ID 2 was all NULLs, ROW_ID 3 contained data, etcetera. This data format looked very messy and basically doubled the number of rows per table – it would have to be fixed. To remedy this problem I went through each table and deleted the null rows using an SQL script. To fix the issue in other tables, just replace the Table and Column of the fixed table with the details of the new column you are trying to fix.

```
DELETE
FROM Nodes
WHERE User = '';
```

Overview of the Data:

Size of the Files:

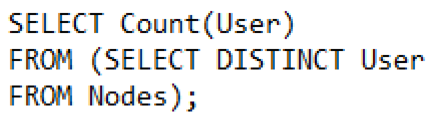
Here are the sizes of the files used for the project. The sample_denver.xml file is the stripped down Denver_colorado_osm.xml file used for iterating through the python code. I tried using the large file first and it took ages to compile – the sample file was much more manageable.



```
denver_colorado_osm.xml = 714.2 MB
sample_denver.xml       = 2.6 MB
nodes.csv               = 344.4 MB
nodes_tags.csv          = 21.7 MB
ways.csv                = 31.8 MB
ways_nodes.csv          = 106.7 MB
ways_tags.csv           = 0 MB
```

Number of Unique Users:

To find the number of unique users, I used a simple subquery against the Nodes table. Executing this query I found that there were 1298 unique contributors to this OSM dataset.

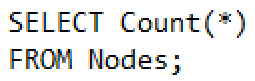


```
SELECT Count(User)
FROM (SELECT DISTINCT User
FROM Nodes);
```

Number of Nodes and Ways:

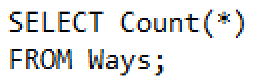
These values were found by executing simple count queries against the Nodes and Ways table respectively.

# of Nodes: 3,132,214



```
SELECT Count(*)
FROM Nodes;
```

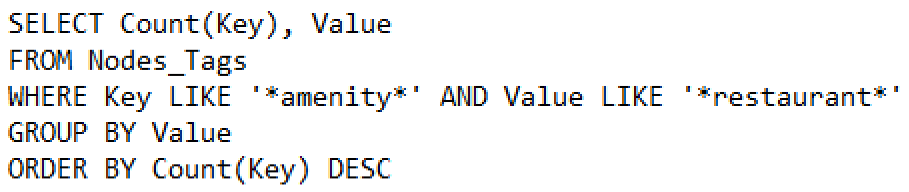
# or Ways: 396,041



```
SELECT Count(*)
FROM Ways;
```

Number of Restaurants:

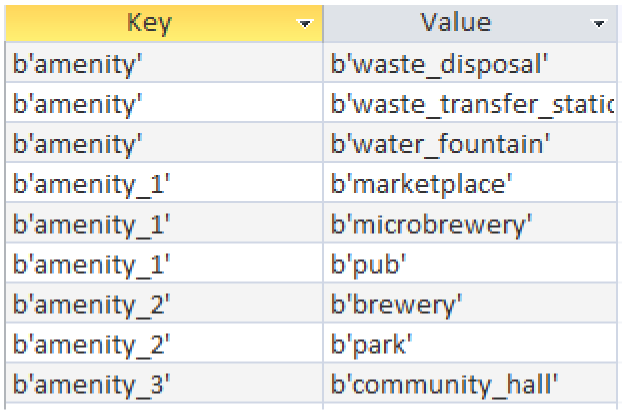
I was curious how many individual restaurants were counted in the data, so I ran the below query and found that there are 1498 in Denver. Interestingly, fast food was considered a separate amenity, and its count was 578.



```
SELECT Count(Key), Value
FROM Nodes_Tags
WHERE Key LIKE '*amenity*' AND Value LIKE '*restaurant*'
GROUP BY Value
ORDER BY Count(Key) DESC
```

A major drawback of utilizing open sourced data is that each person adding content and context to the OSM data has a different method and style of doing so. There is little to no consistency in how nodes are labeled, how the tags are used, how street names and house numbers are abbreviated, etcetera. One interesting fix to this problem could be to implement a universal standard for how new data is added to the currently existing OSM data. The data is already structured uniformly utilizing XML, so there must be some program or algorithm that would look for common inconsistencies, change them on the spot, and add them to the now-uniform OSM file. This would allow manipulation and analysis of the data at a much higher level, leaving most of the cleaning and organizing to the programs instead of people.

You can see an example below from this data set – there are several different types of amenity keys used and there is no explanation for why they are labeled differently. Examples like this are littered throughout the data, and it would be much easier to clean and analyze if there was a set standard for what was expected for certain key categories.



Key	Value
b'amenity'	b'waste_disposal'
b'amenity'	b'waste_transfer_statio
b'amenity'	b'water_fountain'
b'amenity_1'	b'marketplace'
b'amenity_1'	b'microbrewery'
b'amenity_1'	b'pub'
b'amenity_2'	b'brewery'
b'amenity_2'	b'park'
b'amenity_3'	b'community_hall'

This is a nice solution, but there are a couple problems with it. First of all, it sounds simple in theory to be able to create an algorithm like that from scratch, but I would imagine there is much more that goes into it. Each person who enters data on OSM has a personalized way of doing things, so the code solving this problem would have to be nimble and extremely detailed to cover a variety of unique scenarios. Furthermore, implementing restrictions on how users enter data on OSM would cease to make the program truly open source. Though there are different disadvantages to open source, one of the advantages is that users are allowed tremendous leeway, meaning they can be creative and flex their coding muscles to solve interesting problems and add valuable and interesting data to the data set.

Though the data provided by OSM could be much more easy to clean, wrangle, and analyze if users were forced to conform to a certain standard, it may come at the cost of the freedom and autonomy that makes OSM so unique.