# Exploring Denver Colorado

In this project I will be wrangling a dataset from Open Street Map for Denver Colorado.

I will:

- Audit the data.
- Clean the data.
- Finally import and analyze the data in a SQL database.

## Data Auditing

To Audit the data I will follow these steps:

1. Establish and count the tags.
2. Determine the structure.
3. Convert tree structure to schema
4. Discover which data may present a problem in upload.

### First I will import all of my tools.

In [10]:
```
import xml.etree.cElementTree as ET
import pprint
import csv
import codecs
import re
import cerberus
from itertools import chain
from collections import defaultdict
import schema
```

### Next I will count the individual tags

I will do this by defining a function to iteratively parse through the xml file and count the occurance of each unique tag.

In [11]:
```
Denver = './denver_colorado_osm.xml'

def count_tags(xml):
    tags = {}
    for event, elem in ET.iterparse(xml):
        if elem.tag in tags:
            tags[elem.tag] += 1
        else:
            tags[elem.tag] = 1
    return tags
```

In [3]:
```
Tags = count_tags(Denver)
print(Tags)
```

```
{'note': 1, 'meta': 1, 'bounds': 1, 'node': 3132214, 'tag': 1842489, 'nd': 3526619, 'way': 396041, 'member': 44664, 'rela
tion': 1725, 'osm': 1}
```

## Tag Count

There are 10 unique tag types that exist in the following amounts:

1. osm : 1
2. note : 1
3. meta : 1
4. bounds : 1
5. relation : 1,725
6. member : 44,664
7. way : 396,041
8. tag : 1,842,489
9. node : 3,132,214
10. nd : 3,526,619

## Define Structure

I will first identify the root using the .getroot function. Knowing this I can explore the structure further by iterating through the children of each tag.

In [12]:
```
dtree = ET.parse(Denver)
```

```
droot = dtree.getroot()
print(droot)
```

```
<Element 'osm' at 0x000001F8AEB06220>
```

In [13]:

```
def child_count(rname):
    children = {}
    for child in rname:
        if child.tag in children:
            children[child.tag] +=1
        else:
            children[child.tag] = 1
    return children

print(child_count(droot))
```

```
{'note': 1, 'meta': 1, 'bounds': 1, 'node': 3132214, 'way': 396041, 'relation': 1725}
```

## OSM Structure

Below the root are 6 unique tag types in the following quantities:

1. note : 1
2. meta : 1
3. bounds : 1
4. node : 3,132,214
5. way : 396,041
6. relation : 1,725

This accounts for all tags, of those types, found in the initial count.

Because note has only one tag and was discovered first I will assume it is located at the 0 position of the root. I expect to find meta and bounds at 1 and 2 or as children of note. I will now explore these three.

In [6]:

```
#Examining note
print(droot[0])
print(child_count(droot[0]))

#Examining meta
print(droot[1])
print(child_count(droot[1]))
```

```
#Examining bounds
print(droot[2])
print(child_count(droot[2]))
```

```
<Element 'note' at 0x000001A3B58855E0>
{}
<Element 'meta' at 0x000001A3B5885540>
{}
<Element 'bounds' at 0x000001A3B5885770>
{}
```

Note, meta, and bounds were found where they were expected and, unsurprisingly, contained no children.

Because the node, way, and relation tags have significantly more entries I will need a new function to examine their children.

In [14]:
```
def count_grandchildren(tname):
    grandchildren = []
    count = 0
    for i in droot:
        if i.tag == tname:
            if child_count(droot[count]) not in grandchildren:
                grandchildren.append(child_count(droot[count]))
        count += 1
    return grandchildren

print(count_grandchildren('node'))
```

```
[{}, {'tag': 1}, {'tag': 3}, {'tag': 2}, {'tag': 11}, {'tag': 14}, {'tag': 12}, {'tag': 13}, {'tag': 21}, {'tag': 15},
{'tag': 4}, {'tag': 9}, {'tag': 8}, {'tag': 5}, {'tag': 7}, {'tag': 6}, {'tag': 17}, {'tag': 10}, {'tag': 16}, {'tag': 1
9}, {'tag': 24}, {'tag': 26}, {'tag': 18}, {'tag': 25}, {'tag': 30}]
```

## Node Structure

Node has exaclty one type of tag, the "tag" tag. Each node has between 0 and 30 tags of this type.

I will now explore the ways tag in similar fashion.

In [15]:
```
ways = count_grandchildren('way')
print(ways)
```

```
[{'nd': 11, 'tag': 11}, {'nd': 29, 'tag': 2}, {'nd': 45, 'tag': 2}, {'nd': 10, 'tag': 3}, {'nd': 8, 'tag': 3}, {'nd': 71,
'tag': 10}, {'nd': 13, 'tag': 10}, {'nd': 6, 'tag': 10}, {'nd': 2, 'tag': 5}, {'nd': 2, 'tag': 8}, {'nd': 2, 'tag': 9},
{'nd': 21, 'tag': 7}, {'nd': 17, 'tag': 1}, {'nd': 17, 'tag': 12}, {'nd': 2, 'tag': 11}, {'nd': 7, 'tag': 12}, {'nd': 7,
```

## Ways Structure

From the above list it appears we have two types of unique tags as children to way: nd and tag. But the number of tag combinations makes it difficult to read exactly what is going on. I'll make a couple of new functions to help make this more readable.

In [16]:
```
def uniquetags(dlist):
    # uses chain from itertools to find unique keys in a list of dictionaries
    tlist = list(set(chain.from_iterable(sub.keys() for sub in dlist)))
    return str(tlist)

print(uniquetags(ways))
```

```
['tag', 'nd']
```

In [17]:
```
def tag_range(dlist,tname):
    # iterates through a list of dictionaries to find the highest and lowest value of a given key returns (0,0) if key do
    mtag = 0
    for dict in dlist:
        if dict.get(tname) is not None:
            if dict[tname] > mtag:
                mtag = dict[tname]
    mintag = mtag
    for dict in dlist:
        if dict.get(tname) is not None:
            if dict[tname] < mintag:
                mintag = dict[tname]
        else:
            # returns (0,mtag) if key is missing from at least one dictionary in list
            mintag = 0
    return(mintag, mtag)



print('way>nd:')
print(tag_range(ways,'nd'))
print()
print('way>tag:')
print(tag_range(ways,'tag'))
```

```
way>nd:
(2, 1529)
```

```
way>tag:
(0, 29)
```

I found that "way" does have exactly two child tags "nd" and "tag". There are between 2 and 1,529 nd tags for each way tag. There are between 0 and 29 tag tags.

### Relation Structure

Using the same process and functions I will explore the structure of the relation tab.

In [11]:
```
relations = count_grandchildren('relation')

print(uniquetags(relations))
```

```
['member', 'tag']
```

In [12]:
```
print('relation>member:')
print(tag_range(relations,'member'))
print()
print('relation>tag:')
print(tag_range(relations,'tag'))
```

```
relation>member:
(1, 2640)

relation>tag:
(0, 280)
```

I found that "relation" also has exactly two child tags "member" and "tag". There are between 1 and 2,640 member tags for each relation tag. There are between 0 and 280 tag tags.

## Tree Structure

From my probing I was able to find that the xml datatree is structured something as follows:

I. osm

1. note
2. meta
3. bounds

4. node

    a. tag (0-30)

5. way

    a. nd (2-1,529)

    b. tag (0-29)

6. relation

    a. member (1-2,640)

    b. tag (0-280)

Next I will explore the attributes of each tag.

In [13]:

```
print('osm')
print(droot.attrib)
print(droot.text)
print()
print('note')
print(droot[0].attrib)
print(droot[0].text)
print()
print('meta')
print(droot[1].attrib)
print(droot[1].text)
print()
print('bounds')
print(droot[2].attrib)
print(droot[2].text)
```

```
osm
{'version': '0.6', 'generator': 'Overpass API 0.7.54.13 ff15392f'}


note
```

```
{}
The data included in this document is from www.openstreetmap.org. The data is made available under ODbL.

meta
{'osm_base': '2018-04-24T23:50:02Z'}
None

bounds
{'minlat': '39.5956000', 'minlon': '-105.2298000', 'maxlat': '39.9329000', 'maxlon': '-104.4800000'}
None
```

To determine the attributes/text of the other tags I will need a new function.

In [14]:

```
def attrib_compile(file, tname, gname):
    #iterates through a given file (file), to find attribute keys for a given tag (tname) and it's child (gname)
    attributes = []
    for event, elem in ET.iterparse(file, events=("start",)):
        if elem.tag == tname:
            if gname == 0:
                # if the third argument is set to 0 function instead returns attribute keys for a given tag.
                if elem.attrib.keys() not in attributes:
                    attributes.append(elem.attrib.keys())
            else:
                for t in elem.iter(gname):
                    if t.attrib.keys() not in attributes:
                        attributes.append(t.attrib.keys())
    return attributes
attribnode = attrib_compile(Denver, 'node', 0)
print(attribnode)
```

```
[dict_keys(['id', 'lat', 'lon', 'version', 'timestamp', 'changeset', 'uid', 'user'])]
```

The node tag has the following attributes: 'id', 'lat', 'lon', 'version', 'timestamp', 'changeset', 'uid', 'user'

In [15]:

```
attribnodetag = attrib_compile(Denver, 'node', 'tag')
print(attribnodetag)
```

```
[dict_keys(['k', 'v'])]
```

The tag tag under the node tag has the following attributes: 'k', 'v'

In [134…

```
attribway = attrib_compile(Denver, 'way', 0)
print(attribway)
```

```
[dict_keys(['id', 'version', 'timestamp', 'changeset', 'uid', 'user'])]
```

The way tag has the following attributes: 'id', 'version', 'timestamp', 'changeset', 'uid', 'user'

In [135…
```
attribwaytag = attrib_compile(Denver, 'way', 'tag')
print(attribwaytag)
```

```
[dict_keys(['k', 'v'])]
```

The tag tag under the way tag has the following attributes: 'k', 'v'

In [136…
```
attribwaynd = attrib_compile(Denver, 'way', 'nd')
print(attribwaynd)
```

```
[dict_keys(['ref'])]
```

The nd tag under the way tag has the following attribute: 'ref'

In [137…
```
attribrelation = attrib_compile(Denver, 'relation', 0)
print(attribrelation)
```

```
[dict_keys(['id', 'version', 'timestamp', 'changeset', 'uid', 'user'])]
```

The relation tag has the following attributes: 'id', 'version', 'timestamp', 'changeset', 'uid', 'user'

In [138…
```
attribrelationtag = attrib_compile(Denver, 'relation', 'tag')
print(attribrelationtag)
```

```
[dict_keys(['k', 'v'])]
```

The tag tag under the relation tag has the following attributes: 'k', 'v'

In [139…
```
attribrelationmember = attrib_compile(Denver, 'relation', 'member')
print(attribrelationmember)
```

```
[dict_keys(['type', 'ref', 'role'])]
```

The member tag under the relation tag has the following attributes: 'type', 'ref', 'role'

## Table Structure

Now that I know what attributes each tag has I can begin structuring my schema.

### Tablename: nodes

```
id INTEGER PRIMARY KEY NOT NULL
lat REAL
lon REAL
user TEXT
uid INTEGER
version INTEGER
changeset INTEGER
timestamp TEXT
```

### Tablename: nodes_tags

```
nid INTEGER
key TEXT
value TEXT
type TEXT
```

FOREIGN KEY (nid) REFERENCES nodes(id)

### Tablename: ways

```
id INTEGER PRIMARY KEY NOT NULL
user TEXT
uid INTEGER
version TEXT
```

```
changeset INTEGER

timestamp TEXT
```

### Tablename: ways_tags

```
wid INTEGER NOT NULL

key TEXT NOT NULL

value TEXT NOT NULL

type TEXT
```

FOREIGN KEY (wid) REFERENCES ways(id)

### Tablename: ways_nodes

```
wid INTEGER NOT NULL

nid INTEGER NOT NULL

position INTEGER NOT NULL
```

FOREIGN KEY (wid) REFERENCES ways(id)

FOREIGN KEY (nid) REFERENCES nodes(id)

## Looking Closer

Whether a child of node or way the tag tag has the same attributes 'k' and 'v.'

This appears to be a key and value pair. I will write a function to see what keys ('k' value) exist in the tag tags.

In [166...

```python
def keytypes(file, tname):
    # creates a list of values assigned to attribute k for a tag tag that is a child of a given tag (tname)
    keylist = []
    for event, elem in ET.iterparse(file, events=("start",)):
        if elem.tag == tname:
            for t in elem.iter('tag'):
                if t.attrib['k'] not in keylist:
                    keylist.append(t.attrib['k'])
    return keylist

waykey = keytypes(Denver, 'way')
print(waykey)
```

```
['NHS', 'bicycle', 'hgv', 'hgv:national_network', 'highway', 'lanes', 'maxspeed', 'name', 'oneway', 'ref', 'source:hgv:na
tional_network', 'service', 'surface', 'electrified', 'gauge', 'old_railway_operator', 'operator', 'railway', 'tiger:cfc
c', 'tiger:county', 'tiger:name_base', 'usage', 'bridge', 'layer', 'toll', 'foot', 'horse', 'motor_vehicle', 'access', 'm
inspeed', 'tiger:name_direction_prefix', 'tiger:name_type', 'tiger:reviewed', 'source', 'tracktype', 'lit', 'maxspeed:typ
e', 'alt_name', 'tiger:separated', 'tiger:source', 'tiger:tlid', 'parking:lane:both', 'tiger:upload_uuid', 'destination',
'destination:ref', 'CDOT_route', 'maxspeed:advisory', 'note', 'destination:ref:to', 'destination:street', 'cycleway', 'na
me_1', 'tiger:name_base_1', 'tiger:name_type_1', 'bus', 'turn:lanes', 'sidewalk', 'junction', 'old_name', 'lanes:backwar
d', 'lanes:both_ways', 'lanes:forward', 'placement:both_ways', 'turn:lanes:both_ways', 'tiger:name_direction_prefix_1',
'tiger:name_base_2', 'tiger:name_base_3', 'tiger:name_direction_suffix', 'name_2', 'tiger:name_direction_prefix_2', 'tige
r:name_type_2', 'source:maxspeed', 'turn:lanes:backward', 'level', 'lcn', 'nat_ref', 'hov', 'note:lanes', 'tiger:zip_righ
t', 'old_ref', 'FIXME', 'unsigned_ref', 'tiger:name_type_3', 'name:es', 'turn:lanes:forward', 'cycleway:right', 'noname',
'parking:lane:right', 'source:name', 'construction', 'tiger:name_direction_suffix_1', 'name:en', 'cycleway:left', 'segreg
ated', 'footway', 'bicycle:conditional', 'name:etymology:wikidata', 'created_by', 'cables', 'power', 'voltage', 'man_mad
e', 'unsigned', 'cycleway:both', 'cycleway:both:segregated', 'tiger:zip_right_2', 'tiger:zip_right_3', 'proposed', 'descr
iption', 'planned:highway', 'name_3', 'tiger:name_direction_prefix_3', 'note:name', 'tiger:name_base_4', 'name_4', 'tige
r:name_direction_prefix_4', 'tiger:name_type_4', 'abandoned:highway', 'CODT_route', 'sidewalk:right', 'amenity', 'waterwa
y', 'tracks', 'historic', 'note:cycleway:lane', 'history', 'hiu', 'note:bus', 'aeroway', 'length', 'width', 'addr:city',
'addr:housenumber', 'addr:postcode', 'addr:state', 'addr:street', 'building', 'ele', 'gnis:feature_id', 'height', 'leisur
e', 'name:uk', 'sport', 'start_date', 'wheelchair', 'wikidata', 'wikipedia', 'building:colour', 'building:material', 'roo
f:colour', 'tourism', 'year_of_construction', 'building:height', 'building:levels', 'boat', 'natural', 'shop', 'phone',
'parking', 'gnis:county_id', 'gnis:created', 'gnis:state_id', 'water', 'landuse', 'website', 'seperate', 'fee', 'note:seg
regated:no', 'note:access', 'frequency', 'line', 'sac_scale', 'dogs', 'hikers', 'physical', 'singletrack', 'terrain', 'ke
y', 'is_in:city', 'is_in:state', 'admin_level', 'border_type', 'boundary', 'is_in', 'is_in:country', 'is_in:country_cod
e', 'is_in:iso_3166_2', 'is_in:state_code', 'tiger:CLASSFP', 'tiger:CPI', 'tiger:FUNCSTAT', 'tiger:LSAD', 'tiger:MTFCC',
'tiger:NAME', 'tiger:NAMELSAD', 'tiger:PCICBSA', 'tiger:PCINECTA', 'tiger:PLACEFP', 'tiger:PLACENS', 'tiger:PLCIDFP', 'ti
ger:STATEFP', 'designation', 'place', 'name:ru', 'place_name', 'barrier', 'lcn_ref', 'separated', 'FIXME:bicycle', 'gnis:
county_name', 'gnis:import_uuid', 'building:part', 'address', 'tunnel', 'building:roof:colour', 'maxstay', 'religion', 's
ervices', 'Use', 'cost', 'capacity', 'levels', 'Cost', 'roof:shape', 'roof:orientation', 'cutting', 'embankment', 'buildi
ng:roof:shape', 'mtb:scale', 'mtb:scale:imba', 'trail_visibility', 'crossing', 'smoothness', 'park_ride', 'atm', 'maxspee
d:backward', 'maxspeed:forward', 'parking:lane:left', 'addr:housename', 'undefined', 'gwv', 'destination:lanes:backward',
'destination:ref:lanes:backward', 'crosswalk', 'dangerous', 'area', 'addr:country', 'opening_hours', 'ref:walmart', 'rest
riction', 'cycleway:right:buffer', 'cuisine', 'wifi', 'automated', 'self_service', 'placement', 'automatic', 'internet_ac
cess', 'internet_access:fee', 'intermittent', 'note:building', 'fax', 'rooms', 'smoking', 'stars', 'office', 'note:destin
ation', 'capacity:disabled', 'hoops', 'oneway:bus', 'supervised', 'drive_through', 'emergency', 'healthcare', 'takeaway',
'contact:email', 'contact:fax', 'contact:phone', 'contact:website', 'outdoor_seating', 'contact:facebook', 'owner', 'ref:
```

Examining the generated list of 'k' values from the tag tags under the way tag a couple of values stand out to me.

1. addr:street In my experiance street addresses in databases often lack consistency indicating one area that may be in need of standardization.

2. phone There are several tags with a variation on 'phone' if I were to write a function that cleaned that data, I could clean a large and varied amount of data with that singular effort

In [167…

```
nodekey = keytypes(Denver, 'node')
print(nodekey)
```

```
['highway', 'created_by', 'old_ref', 'ref', 'amenity', 'note:ref', 'ele', 'gnis:Class', 'gnis:County', 'gnis:County_num',
'gnis:ST_alpha', 'gnis:ST_num', 'gnis:id', 'import_uuid', 'is_in', 'name', 'place', 'census:population', 'population', 'w
ikidata', 'wikipedia', 'capital', 'is_in:continent', 'is_in:country', 'is_in:country_code', 'name:en', 'name:eo', 'name:j
a', 'name:oc', 'name:pl', 'name:ru', 'name:ta', 'name:uk', 'source:name:oc', 'state_capital', 'railway', 'source', 'traff
ic_signals', 'noexit', 'direction', 'access', 'barrier', 'man_made', 'surveillance:type', 'noref', 'ref:left', 'ref:righ
t', 'note', 'bicycle', 'crossing', 'traffic_calming', 'surface', 'stop', 'junction', 'addr:city', 'addr:housenumber', 'ad
dr:postcode', 'addr:state', 'addr:street', 'leisure', 'operator', 'website', 'foot', 'horse', 'motor_vehicle', 'tactile_p
aving', 'crossing_ref', 'supervised', 'traffic_signals:direction', 'gnis:county_id', 'gnis:created', 'gnis:feature_id',
'gnis:state_id', 'power', 'access:icy', 'access:windy', 'capacity:disabled', 'parking', 'crossing:barrier', 'crossing:bel
l', 'crossing:light', 'lanes', 'atm', 'cuisine', 'description', 'food', 'microbrewery', 'opening_hours', 'outdoor_seatin
g', 'phone', 'wheelchair', 'tourism', 'addr:housename', 'shop', 'natural', 'landuse', 'religion', 'denomination', 'gnis:e
dited', 'emergency', 'fax', 'phone:attendance', 'old_name', 'disused:amenity', 'historic', 'note:name', 'alt_name', 'buil
ding', 'waterway', 'tower:type', 'service_times', 'email', 'old_website', 'lutheran', 'addr:unit', 'gnis:county_name', 'g
nis:import_uuid', 'office', 'private', 'phone:emergency', 'sport', 'gnis:feature_type', 'aeroway', 'source_ref', 'close
d', 'aerodrome', 'closest_town', 'faa', 'iata', 'icao', 'name_1', 'destination', 'admin_level', 'border_type', 'boundar
y', 'route_ref', 'shelter', 'local_ref', 'park_ride', 'cuisine_1', 'destination:ref', 'line', 'internet_access', 'capacit
y', 'addr:country', 'addr:full', 'ref:walmart', 'dispensing', 'designation', 'level', 'takeaway', 'material', 'bench', 't
elephone', 'phone:guest_support', 'catering', 'wifi', 'brand', 'contact:website', 'monitoring_station', 'radar', 'source:
pkey', 'entrance', 'fuel:electricity', 'fuel:octane_100', 'fuel:octane_91', 'fuel:octane_95', 'fuel:octane_98', 'theatre:
genre', 'drive_in', 'smoking', 'contact:email', 'services', 'type', 'odbl', 'layer', 'activity', 'covered', 'fee', 'star
s', 'bicycle_parking', 'location:transition', 'motorcar', 'motorcycle', 'delivery', 'drive_through', 'opening_hours:url',
'undefined', 'internet_access:fee', 'opening_hours:dinner', 'opening_hours:lunch', 'phone:catering', 'wpt_symbol', 'attra
ction', 'animal', 'dog', 'beauty', 'recycling_type', 'diet:vegetarian', 'amenity:restaurant', 'FIXME:ref', 'fuel:diesel',
'fuel:octane_85', 'fuel:octane_87', 'automated', 'self_service', 'toilets:wheelchair', 'highway_1', 'generator:output:ele
ctricity', 'generator:source', 'historic:power', 'public_transport', 'stop_id', 'geological', 'artwork_type', 'addr:suit
e', 'healthcare', 'craft', 'tower', 'access:opening_hours', 'day_off', 'day_on', 'hour_off', 'hour_on', 'contact:phone',
'happy_hours', 'recycling:batteries', 'recycling:books', 'recycling:cans', 'recycling:cardboard', 'recycling:cartons', 'r
ecycling:clothes', 'recycling:electrical_appliances', 'recycling:glass', 'recycling:glass_bottles', 'recycling:green_wast
e', 'recycling:magazines', 'recycling:newspaper', 'recycling:paper', 'recycling:paper_packaging', 'recycling:plastic', 'r
ecycling:plastic_bottles', 'recycling:plastic_packaging', 'recycling:scrap_metal', 'recycling:small_appliances', 'recycli
ng:waste', 'recycling:wood', 'fuel:HGV_diesel', 'shelter_type', 'fuel:GTL_diesel', 'fuel:biodiesel', 'fuel:biogas', 'fue
l:cng', 'fuel:e10', 'fuel:e85', 'fuel:lpg', 'website:orders', 'wheelchair:description', 'payment:bitcoin', 'organic', 'ad
dr:source', 'contact:fax', 'cost:coffee', 'collection_times', 'url', 'fuel:1_25', 'fuel:1_50', 'backrest', 'addr:door',
'phone:mobile', 'phone:tollfree', 'female', 'male', 'car_wash', 'information', 'artist:wikidata', 'artist_name', 'subjec
```

```
t:wikidata', 'vending', 'naptan:Bearing', 'diet:vegan', 'payment:coins', 'payment:credit_cards', 'payment:debit_cards',
'payment:notes', 'naptan:AtcoCode', 'crossing:lights', 'ref:store_number', 'station', 'int_name', 'club', 'subway', 'lock
ed', 'clothes', 'brewery', 'fire_hydrant:type', 'barrier_2', 'drive_thru', 'toilets:disposal', 'payment:litecoin', 'socia
l_facility', 'social_facility:for', 'traffic_sign', 'maxspeed', 'opening_hours:summer', 'opening_hours:winter', 'design',
'payment:american_express', 'payment:cash', 'payment:discover_card', 'payment:mastercard', 'payment:paypal', 'payment:squ
are_wallet', 'payment:visa', 'support', 'opening_hours:drive_thru', 'note:cuisine', 'alt_phone', 'alt_website', 'note:sho
p', 'artist', 'building:levels', 'phone:alternate', 'bus', 'network', 'healthcare:speciality', 'opening_hours:drive-up',
'opening_hours:lobby', 'structure', 'maxstay', 'culture', 'pastry_shop', 'second_hand', 'note:line', 'survey:date', 'shor
t_name', 'shop_1', 'lines', 'platforms', 'board_type', 'voltage-high', 'voltage-low', 'studio', 'service:bicycle:chain_to
ol', 'country', 'diplomatic', 'exit', 'construction', 'phone:english', 'phone:espanol', 'waste', 'voltage', 'opening_hour
s:gate', 'note:shop:doityourself', 'opening_hours:gym', 'opening_hours:kids_club', 'opening_hours:school_year', 'opening_
hours:carryout', 'opening_hours:delivery', 'opening_hours:access', 'opening_hours:office', 'height', 'note:local_ref', 'n
ote:shelter', 'service', 'lamp_type', 'socket:type1', 'socket:nema_5_20', 'socket:chademo', 'socket:type1_combo', 'name:z
h', 'socket:tesla_supercharger', 'leaf_type', 'transformer', 'crossing:activation', 'contact:name', 'leaf_cycle', 'recycl
ing:shoes', 'opening_hours:pharmacy', 'opening_hours:store', 'service:bicycle:pump', 'camera:mount', 'denotation', 'ameni
ty_1', 'recycling:computers', 'amenity_2', 'indoor', 'post_box:type', 'fuel:dyed_diesel', 'payment:checks', 'seasonal',
'street_lamp:type', 'traffic_signals:sound', 'latitude', 'longitude', 'contact:google_plus', 'contact:yelp', 'contact:you
tube', 'charge', 'train', 'military', 'currency:USD', 'light_rail', 'bin', 'rooms', 'distillery', 'reuse:books', 'chargin
g_station:plug', 'telescope:type', 'advertising', 'addr:postbox', 'service:bicycle:repair', 'suite', 'levels', 'binocular
s', 'payment', 'service:vehicle:Transmission_Repair', 'payments', 'fuel:gasoline', 'website:reservations', 'instagram',
'side', 'traffic_sign:forward', 'service:vehicle:Sale', 'service:vehicle:car_parts', 'service:vehicle:car_repair', 'unise
x', 'fire_hydrant:position', 'lit', 'generator:method', 'opening_hours:breakfast', 'area', 'bollard', 'industrial', 'colo
ur', 'min_height', 'opening_hours:brunch', 'payment:cheque', 'lamps', 'addr:flats', 'opening_hours:KatieDid', 'phone:Kati
eDid', 'website:KatieDid', 'financial_advice:investment_advisor', 'phone:appointments', 'opening_hours:fall', 'contact',
'phone:booking', 'website:contact', 'opening_hours:happy_hour', 'opening_hours:sunday_brunch', 'phone:customer_service',
'opening_hours:hotpot', 'door', 'kerb', 'website:alternate', 'manager', 'start_date', 'facebook', 'email:operator', 'phon
e:operator:mobile', 'fountain', 'calming', 'service:bicycle:tools', 'dine_in', 'opening_hours:pickup', 'restriction', 'op
ening_hours:grooming', 'comments', 'web', 'phone:toll-free', 'water', 'artwork_type_1', 'artwork_type_2', 'amenity_3', 'b
uilding_1', 'status', 'government', 'playground', 'opening_hours:car_repair', 'year', 'date', 'toilet', 'apartments', 'FI
XME']
```

# Data Cleaning

I have identified two areas where data may need to be cleaned: addr:street, and phone. I will proceed with the following steps:

1. Audit street types to discover any outliers
2. Create a function to clean and standardize street types
3. Audit phone numbers to discover any issues
4. Create a function to cleand and standardize phone numbers.

In [4]:

```
street_type_re = re.compile(r'\b\S+\.?$', re.IGNORECASE)
street_types = defaultdict(set)

expected_street = ['Street', 'Broadway', 'Crescent', 'Row', 'Center', 'Loop', 'Terrace', 'Avenue', 'Boulevard', 'Drive',
```

```
def audit_street_type(street_types, street_name):
    m = street_type_re.search(street_name)
    if m:
        street_type = m.group()
        if street_type not in expected_street:
            street_types[street_type].add(street_name)
def is_street_name(elem):
    return (elem.attrib['k'] == 'addr:street')

def streetaudit():
    for event, elem in ET.iterparse(Denver, events=('start',)):
        if elem.tag == 'way':
            for tag in elem.iter('tag'):
                if is_street_name(tag):
                    audit_street_type(street_types, tag.attrib['v'])
    print(street_types)

streetaudit()
```

```
defaultdict(<class 'set'>, {'Mall': {'Lawrence Street Mall'}, '73': {'County Highway 73'}, 'Ave': {'West Quincy Ave', 'W
96th Ave', 'w 67th Ave', 'East Hampden Ave', 'W 92nd Ave', 'W 86th Ave'}, 'Pl': {'W 88th Pl'}, 'West': {'Park Avenue Wes
t', 'Ulster Circle West'}, 'North': {'Central Court North', 'East Speer Boulevard North'}, '287': {'US 287', 'US Highway
287'}, '93': {'State Highway 93'}, '380': {'South Colorado Boulevard #380'}, '8': {'CO 8'}, '2': {'State Highway 2'}, 'MU
P': {'128th Ave MUP'}, 'Blvd': {'Federal Blvd', 'Sheridan Blvd'}, 'St': {'S Delaware St'}, '128': {'CO 128'}, '36': {'US
Highway 36'}, 'East': {'Ulster Circle East'}, '72': {'West Highway 72'}, '80237': {'80237'}})
```

## Street errors

Street types that need correcting: Ave, Pl, Blvd, St

In [5]:

```
street_mapping = { "St": "Street",
            "St.": "Street",
            "Ave": "Avenue",
            "Pl": "Place",
            "Blvd": "Boulevard",
            "Rd" : "Road",
            "Rd.": "Road"
            }

def update_name(name, mapping):
    m = street_type_re.search(name)
    if m:
        street_type = m.group()
        if street_type not in expected_street:
```

```
            for key,value in mapping.items():
                if street_type == key:
                    name = name.replace(key,value)

    return name
```

In [6]:

```
phone_re = re.compile(r'^([0-9]{3}-)[0-9]{3}-[0-9]{4}$')
pleasecheckthenumber = defaultdict(set)
phonelist = ['phone', 'phone:attendance', 'phone:emergency', 'telephone', 'phone:guest_support', 'phone:catering', 'conta
             'phone:mobile', 'phone:tollfree', 'alt_phone', 'phone:alternate', 'phone:english', 'phone:espanol', 'phone:K
             'phone:appointments', 'phone:customer_service', 'phone:operator:mobile', 'phone:toll-free', 'phone:reservati
             'phone:parts', 'phone:sales', 'phone:service', 'source:contact:phone', 'phone:pharmacy', 'note:phone', 'phor
             'phone_UCHTC', 'phone:leasing', 'phone:maintenance', 'phone:howell', 'phone:nurse', 'old_phone', 'phone:pro-
             'phone:toll_and_truck_rental', 'phone:ProServices_Desk']


def audit_phone(pleasecheckthenumber, number):
    m = phone_re.search(number)
    if not m:
        pleasecheckthenumber[number].add(number)
def is_number(elem):
    return (elem.attrib['k'] in phonelist)

def phoneaudit():
    for event, elem in ET.iterparse(Denver, events=('start',)):
        if elem.tag == 'node':
            for tag in elem.iter('tag'):
                if is_number(tag):
                    audit_phone(pleasecheckthenumber, tag.attrib['v'])
    print(pleasecheckthenumber)

phoneaudit()
```

```
defaultdict(<class 'set'>, {'3032972700': {'3032972700'}, '+1 303 424 4445': {'+1 303 424 4445'}, '(303) 744-1069': {'(30
3) 744-1069'}, '(720) 424-5380': {'(720) 424-5380'}, '(303) 333-8275': {'(303) 333-8275'}, '(303) 900-2027': {'(303) 900-
2027'}, '+1 303 320 5871': {'+1 303 320 5871'}, '+1 303-364-8715': {'+1 303-364-8715'}, '(720) 865-0955': {'(720) 865-095
5'}, '(303) 239-4500': {'(303) 239-4500'}, '911': {'911'}, '(303) 239-4300': {'(303) 239-4300'}, '3036320805': {'30363208
05'}, '(303) 455-9991': {'(303) 455-9991'}, '3037337448': {'3037337448'}, '(720) 612-7698': {'(720) 612-7698'}, '30329257
67': {'3032925767'}, '3036239600': {'3036239600'}, '+1-303-979-2064': {'+1-303-979-2064'}, '3037770707': {'3037770707'},
'+1 303 534 9505': {'+1 303 534 9505'}, '(303) 297-1229': {'(303) 297-1229'}, '303 428 6500': {'303 428 6500'}, '(303) 57
4-1200': {'(303) 574-1200'}, '(303) 371-4114': {'(303) 371-4114'}, '(888) 592-7753': {'(888) 592-7753'}, '(720) 374-722
0': {'(720) 374-7220'}, '(303) 307-1234': {'(303) 307-1234'}, '(303) 307-1119': {'(303) 307-1119'}, '(303) 373-9000':
{'(303) 373-9000'}, '(303) 373-1133': {'(303) 373-1133'}, '(303) 576-6633': {'(303) 576-6633'}, '(303) 371-1555': {'(303)
371-1555'}, '(720) 974-7315': {'(720) 974-7315'}, '(303) 302-8500': {'(303) 302-8500'}, '(303) 220-1404': {'(303) 220-140
```

{'(303) 856-5533'}, '(303) 337-7163': {'(303) 337-7163'}, '(866) 916-4648': {'(866) 916-4648'}, '(303) 337-0776': {'(303) 337-0776'}, '(303) 750-0660': {'(303) 750-0660'}, '(303) 720-7259': {'(303) 720-7259'}, '(303) 756-3560': {'(303) 756-3560'}, '(303) 744-1961': {'(303) 744-1961'}, '(720) 897-7160': {'(720) 897-7160'}, '(303) 778-8922': {'(303) 778-8922'}, '(214) 532-3761': {'(214) 532-3761'}, '(303) 722-2226': {'(303) 722-2226'}, '(720) 250-8626': {'(720) 250-8626'}, '(303) 777-2770': {'(303) 777-2770'}, '(720) 379-3816': {'(720) 379-3816'}, '(720) 216-0573': {'(720) 216-0573'}, '(720) 475-1302': {'(720) 475-1302'}, '+1-303-377-8483': {'+1-303-377-8483'}, '(303) 812-2000': {'(303) 812-2000'}, '(303) 825-3818': {'(303) 825-3818'}, '(303) 366-1353': {'(303) 366-1353'}, '(800) 275-8777': {'(800) 275-8777'}, '(303) 344-1897': {'(303) 344-1897'}, '+1 303-466-4209': {'+1 303-466-4209'}, '303364544': {'303364544'}, '3033078888': {'3033078888'}, '(303) 368-1520': {'(303) 368-1520'}, '(844) 847-9517': {'(844) 847-9517'}, '(303) 337-3400': {'(303) 337-3400'}, '(844) 453-5058': {'(844) 453-5058'}, '(303) 731-1115': {'(303) 731-1115'}, '(844) 792-5355': {'(844) 792-5355'}, '(720) 213-0012': {'(720) 213-0012'}, '(303) 745-0291': {'(303) 745-0291'}, '(303) 750-2102': {'(303) 750-2102'}, '(303) 755-6205': {'(303) 755-6205'}, '+1-303-813-1000': {'+1-303-813-1000'}, '(702) 324-2858': {'(702) 324-2858'}, '(303) 525-0200': {'(303) 525-0200'}, '(303) 246-1589': {'(303) 246-1589'}})

## Phone Errors

Phone errors that need correcting: Missing hyphens, superfluos country code, spaces in phone number, parenthesis around area code

In [18]:

```
def phone_format(phone_number):
    if '+1' in phone_number:
            phone_number = re.sub('\+1','', phone_number)
    clean_phone_number = re.sub('[^0-9]+', '', phone_number)
    if len(clean_phone_number) == 10:
        formatted_phone_number = re.sub("(\d)(?=(\d{3})+(?!\d))", r"\1-", "%d" % int(clean_phone_number[:-1])) + clean_ph
    else:
        formatted_phone_number = phone_ext(clean_phone_number)
    return formatted_phone_number

def phone_ext(tel):
    phonex = f"{tel[:3]}-{tel[3:6]}-{tel[6:10]}x{tel[10:]}"
    return phonex

def fixphone(phone_num):
    m = phone_re.search(phone_num)
    if not m:
        #format phone number
        if len(phone_num)>= 10:
            phone_num = phone_format(phone_num)
    return phone_num

print(fixphone('+1(303) 867-5309.x1981'))
```

```
303-867-5309x1981
```

# Data Extraction

I will now extract the data to csv files so that I may easily import it into a sql database. I will do so by following the steps as follows:

- Step through each top level element in the XML using iterparse.
- Use a custom function to shape each elemebnt inot data structures
- Ensure the data is in the correct format with a schema and validation library
- Finally write the data to the appropriate .csv file

# Shape Element Function

The function should take as input an iterparse Element object and return a dictionary.

## If the element top level tag is "node":

The dictionary returned should have the format {"node": .., "node_tags": ...}

The "node" field should hold a dictionary of the following top level node attributes:

- id
- user
- uid
- version
- lat
- lon
- timestamp
- changeset All other attributes can be ignored

The "node_tags" field should hold a list of dictionaries, one per secondary tag. Secondary tags are child tags of node which have the tag name/type: "tag". Each dictionary should have the following fields from the secondary tag attributes:

- id: the top level node id attribute value
- key: the full tag "k" attribute value if no colon is present or the characters after the colon if one is.
- value: the tag "v" attribute value
- type: either the characters before the colon in the tag "k" value or "regular" if a colon

```
    is not present.
```

Additionally,

- if the tag "k" value contains problematic characters, the tag should be ignored
- if the tag "k" value contains a ":" the characters before the ":" should be set as the tag type and characters after the ":" should be set as the tag key
- if there are additional ":" in the "k" value they and they should be ignored and kept as part of the tag key. For example:

  should be turned into {'id': 12345, 'key': 'street:name', 'value': 'Lincoln', 'type': 'addr'}

- If a node has no secondary tags then the "node_tags" field should just contain an empty list.

The final return value for a "node" element should look something like:

{'node': {'id': 757860928, 'user': 'uboot', 'uid': 26299, 'version': '2', 'lat': 41.9747374, 'lon': -87.6920102, 'timestamp': '2010-07-22T16:16:51Z', 'changeset': 5288876}, 'node_tags': [{'id': 757860928, 'key': 'amenity', 'value': 'fast_food', 'type': 'regular'}, {'id': 757860928, 'key': 'cuisine', 'value': 'sausage', 'type': 'regular'}, {'id': 757860928, 'key': 'name', 'value': "Shelly's Tasty Freeze", 'type': 'regular'}]}

## If the element top level tag is "way":

The dictionary should have the format {"way": ..., "way_tags": ..., "way_nodes": ...}

The "way" field should hold a dictionary of the following top level way attributes:

- id
- user
- uid
- version
- timestamp
- changeset

All other attributes can be ignored

The "way_tags" field should again hold a list of dictionaries, following the exact same rules as for "node_tags".

Additionally, the dictionary should have a field "way_nodes". "way_nodes" should hold a list of dictionaries, one for each nd child tag. Each dictionary should have the fields:

- id: the top level element (way) id
- node_id: the ref attribute value of the nd tag
- position: the index starting at 0 of the nd tag i.e. what order the nd tag appears within the way element

The final return value for a "way" element should look something like:

{'way': {'id': 209809850, 'user': 'chicago-buildings', 'uid': 674454, 'version': '1', 'timestamp': '2013-03-13T15:58:04Z', 'changeset': 15353317}, 'way_nodes': [{'id': 209809850, 'node_id': 2199822281, 'position': 0}, {'id': 209809850, 'node_id': 2199822390, 'position': 1}, {'id': 209809850, 'node_id': 2199822392, 'position': 2}, {'id': 209809850, 'node_id': 2199822369, 'position': 3}, {'id': 209809850, 'node_id': 2199822370, 'position': 4}, {'id': 209809850, 'node_id': 2199822284, 'position': 5}, {'id': 209809850, 'node_id': 2199822281, 'position': 6}], 'way_tags': [{'id': 209809850, 'key': 'housenumber', 'type': 'addr', 'value': '1412'}, {'id': 209809850, 'key': 'street', 'type': 'addr', 'value': 'West Lexington St.'}, {'id': 209809850, 'key': 'street:name', 'type': 'addr', 'value': 'Lexington'}, {'id': '209809850', 'key': 'street:prefix', 'type': 'addr', 'value': 'West'}, {'id': 209809850, 'key': 'street:type', 'type': 'addr', 'value': 'Street'}, {'id': 209809850, 'key': 'building', 'type': 'regular', 'value': 'yes'}, {'id': 209809850, 'key': 'levels', 'type': 'building', 'value': '1'}, {'id': 209809850, 'key': 'building_id', 'type': 'chicago', 'value': '366409'}]}

In [8]:

```
OSM_PATH = "denver_colorado_osm.xml"
NODES_PATH = "nodes.csv"
NODE_TAGS_PATH = "nodes_tags.csv"
WAYS_PATH = "ways.csv"
WAY_NODES_PATH = "ways_nodes.csv"
WAY_TAGS_PATH = "ways_tags.csv"

LOWER_COLON = re.compile(r'^([a-z]|_)+:([a-z]|_)+')
PROBLEMCHARS = re.compile(r'[=\+/&<>;\'"\?%#$@\,\. \t\r\n]')

SCHEMA = schema.schema

# Make sure the fields order in the csvs matches the column order in the sql table schema
NODE_FIELDS = ['id', 'lat', 'lon', 'user', 'uid', 'version', 'changeset', 'timestamp']
NODE_TAGS_FIELDS = ['id', 'key', 'value', 'type']
WAY_FIELDS = ['id', 'user', 'uid', 'version', 'changeset', 'timestamp']
WAY_TAGS_FIELDS = ['id', 'key', 'value', 'type']
WAY_NODES_FIELDS = ['id', 'node_id', 'position']


def shape_element(element, node_attr_fields=NODE_FIELDS, way_attr_fields=WAY_FIELDS,
                  problem_chars=PROBLEMCHARS, default_tag_type='regular'):
```

```
    """Clean and shape node or way XML element to Python dict"""

    node_attribs = {}
    way_attribs = {}
    way_nodes = []
    tags = []  # Handle secondary tags the same way for both node and way elements

    if element.tag == 'node':
        # create dictionary for node
        for i in node_attr_fields:
            node_attribs[i] = element.attrib[i]
        # create list of dictionaries for node_tags
        for t in element.findall('tag'):
            nt = {}
            nt['id'] = node_attribs['id']
            if (t.attrib['k'] in phonelist):
                result = fixphone(t.attrib['v'])
                if result:
                    nt['value'] = result
                else:
                    continue
            else:
                nt['value'] = t.attrib['v']
            m = problem_chars.search(t.attrib['k'])
            l = LOWER_COLON.search(t.attrib['k'])
            if m:
                return
            elif t:
                nt['key'] = t.attrib['k'].split(':',1)[1]
                nt['type'] = t.attrib['k'].split(':',1)[0]
            else:
                nt['key'] = t.attrib['k']
                nt['type'] = default_tag_type
            tags.append(nt)
        return {'node': node_attribs, 'node_tags': tags}
    elif element.tag == 'way':
        # create dictionary for way
        for i in way_attr_fields:
            way_attribs[i] = element.attrib[i]
        # create list of dictionaries for way_nodes
            count = 0
            for n in element.findall('nd'):
                nd = {}
                nd['id'] = way_attribs['id']
                nd['node_id'] = n.attrib['ref']
```

```
                nd['position'] = count
                count += 1

                way_nodes.append(nd)
        # create list of dictionaries for way_tags
        for t in element.findall('tag'):
            wt = {}
            wt['id'] = way_attribs['id']
            if t.attrib['k'] in phonelist:
                result = fixphone(t.attrib['v'])
                if result:
                    wt['value'] = result
                else:
                    continue
            elif t.attrib['k'] == 'addr:street':
                bettername = update_name(t.attrib['v'], street_mapping)
                if bettername:
                    wt['value'] = bettername
                else:
                    continue
            else:
                wt['value'] = t.attrib['v']
            m = problem_chars.search(t.attrib['k'])
            l = LOWER_COLON.search(t.attrib['k'])
            if m:
                return
            elif l:
                wt['key'] = t.attrib['k'].split(':',1)[1]
                wt['type'] = t.attrib['k'].split(':',1)[0]
            else:
                wt['key'] = t.attrib['k']
                wt['type'] = default_tag_type
            tags.append(wt)

        return {'way': way_attribs, 'way_nodes': way_nodes, 'way_tags': tags}



# ================================================== #
#               Helper Functions                     #
# ================================================== #

def get_element(osm_file, tags=('node', 'way', 'relation')):
    """Yield element if it is the right type of tag"""
```

```
    context = ET.iterparse(osm_file, events=('start', 'end'))
    _, root = next(context)
    for event, elem in context:
        if event == 'end' and elem.tag in tags:
            yield elem
            root.clear()


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))


class UnicodeDictWriter(csv.DictWriter, object):
    """Extend csv.DictWriter to handle Unicode input"""

    def writerow(self, row):
        super(UnicodeDictWriter, self).writerow({
            k: v for k, v in row.items()
            # k: (repr(v).encode('utf-8') if not isinstance(v, bytes) else v) for k, v in row.items()
        })

    def writerows(self, rows):
        for row in rows:
            self.writerow(row)


# ================================================== #
#               Main Function                        #
# ================================================== #
def process_map(file_in, validate):
    """Iteratively process each XML element and write to csv(s)"""

    with codecs.open(NODES_PATH, 'w', "utf-8") as nodes_file, \
         codecs.open(NODE_TAGS_PATH, 'w', "utf-8") as nodes_tags_file, \
         codecs.open(WAYS_PATH, 'w', "utf-8") as ways_file, \
         codecs.open(WAY_NODES_PATH, 'w', "utf-8") as way_nodes_file, \
         codecs.open(WAY_TAGS_PATH, 'w', "utf-8") as way_tags_file:

        nodes_writer = UnicodeDictWriter(nodes_file, NODE_FIELDS)
```

```
        node_tags_writer = UnicodeDictWriter(nodes_tags_file, NODE_TAGS_FIELDS)
        ways_writer = UnicodeDictWriter(ways_file, WAY_FIELDS)
        way_nodes_writer = UnicodeDictWriter(way_nodes_file, WAY_NODES_FIELDS)
        way_tags_writer = UnicodeDictWriter(way_tags_file, WAY_TAGS_FIELDS)

        nodes_writer.writeheader()
        node_tags_writer.writeheader()
        ways_writer.writeheader()
        way_nodes_writer.writeheader()
        way_tags_writer.writeheader()

        validator = cerberus.Validator()

        for element in get_element(file_in, tags=('node', 'way')):
            el = shape_element(element)
            if el:
                if validate is True:
                    validate_element(el, validator)

                if element.tag == 'node':
                    nodes_writer.writerow(el['node'])
                    node_tags_writer.writerows(el['node_tags'])
                elif element.tag == 'way':
                    ways_writer.writerow(el['way'])
                    way_nodes_writer.writerows(el['way_nodes'])
                    way_tags_writer.writerows(el['way_tags'])


if __name__ == '__main__':
    # Note: Validation is ~ 10X slower. For the project consider using a small
    # sample of the map when validating.
    process_map(OSM_PATH, validate=True)
```

# Data Import and Analysis

With the data now converted to tabular form in CSVs I will begin to analyze it with the following steps:

1. Create appropriate tables in my database and upload the data from my CSVs
2. Querry the data to answer questions.

I will use the following to create my tables:

```
CREATE TABLE nodes (

    id INTEGER PRIMARY KEY NOT NULL,
    lat REAL,
    lon REAL,
    user TEXT,
    uid INTEGER,
    version INTEGER,
    changeset INTEGER,
    timestamp TEXT

);

CREATE TABLE nodes_tags (

    id INTEGER,
    key TEXT,
    value TEXT,
    type TEXT,
    FOREIGN KEY (id) REFERENCES nodes(id)

);

CREATE TABLE ways (

    id INTEGER PRIMARY KEY NOT NULL,
    user TEXT,
    uid INTEGER,
    version TEXT,
    changeset INTEGER,
    timestamp TEXT

);

CREATE TABLE ways_tags (

    id INTEGER NOT NULL,
    key TEXT NOT NULL,
```

```
    value TEXT NOT NULL,
    type TEXT,
    FOREIGN KEY (id) REFERENCES ways(id)

);

CREATE TABLE ways_nodes (

    id INTEGER NOT NULL,
    node_id INTEGER NOT NULL,
    position INTEGER NOT NULL,
    FOREIGN KEY (id) REFERENCES ways(id),
    FOREIGN KEY (node_id) REFERENCES nodes(id)

);
```

## Analyze Data

Now that my schema is created and my data is imported I will answer the following questions:

1. What is the number of unique users?
2. How many nodes?
3. How many ways?
4. What is the number of doctors?

### What is the number of unique users?

Using the following querry I was able to identify the number of unique users:

Select count(DISTINCT users.uid) as 'usercount' From (Select uid from nodes UNION ALL Select uid from ways) users;

- There are 1,294 unique users.

### What is the number of nodes?

Using the following querry I was able to identify the number of nodes:

Select count(*) from nodes;

- There are 3,132,214 nodes.

### What is the number of ways?

Using the following querry I was able to identify the number of ways:

Select count(*) from ways;

- There are 396,041 ways.

### How many doctors are there?

Using the following querry I was able to identify the number of doctors:

Select value, count(id) as 'number' from nodes_tags where key in ('amenity', 'amenity_1', 'amenity_2', 'amenity_3') and value = 'doctors' group by value

- There are 80 doctors

## Ideas for Improvement

It is difficult to maintain consitency in a database when it is rigidly structured and includes input from only approved users. These problems compound when the database is open source and fluid. If the database were too rigid we would no doubt lose a great deal of the information that has been included.

However, when we look at the vast range of fields that contain phone numbers:

'phone', 'phone:attendance', 'phone:emergency', 'telephone', 'phone:guest_support', 'phone:catering', 'contact:phone', 'phone:mobile', 'phone:tollfree', 'alt_phone', 'phone:alternate', 'phone:english', 'phone:espanol', 'phone:KatieDid', 'phone:appointments', 'phone:customer_service', 'phone:operator:mobile', 'phone:toll-free', 'phone:reservations', 'phone:parts', 'phone:sales', 'phone:service', 'source:contact:phone', 'phone:pharmacy', 'note:phone', 'phone:school', 'phone_UCHTC', 'phone:leasing', 'phone:maintenance', 'phone:howell', 'phone:nurse', 'old_phone', 'phone:pro-desk', 'phone:toll_and_truck_rental', 'phone:ProServices_Desk'

We can immediately see the benefit of some standardization. For example:

What is the differance between 'phone', 'telephone', 'contact:phone', or 'source:contact:phone'?

Isn't 'alt_phone' astonishingly similar to 'phone:alternate'?

Do we expect to see many entries in the 'phone:howell', 'phone:ProServices_Desk', or 'phone:toll_and_truck_rental'?

As a matter of fact if we run the following querry:

Select value from ways_tags where type = 'phone' and key = 'toll_and_truck_rental';

We return only one value:

- 303-698-7082

It would be much simpler if phone were a special sort of tag that held phone numbers associated with a node or way. If this were the case we could apply data validation to make certain the number was valid and formatted correctly. These two changes would save a great deal of trouble to anyone trying to make use of the data.

In [ ]: