Humboldt County OSM Project

Welcome to my project - an exploration and conversion of OSM XML data from the OpenStreetMap project to CSVs to be further analyzed in SQLite3.

This project demonstrates facility with the Python language with regard to data analysis as well as a few critical data analysis modules, primarily ElementTree. Further, a SQL database is constructed and queries run to demonstrate facility within a relational database environment.

Project Overview

The OSM Project incorporates many functions learned generally in the course of the Udacity Data Analysis NanoDegree. Much of the challenge of this project came from understanding the general functions and applying them correctly to a new and much larger dataset.

In brief, our goals are the following:

1. Audit OSM XML to better understand our data
2. Improve the data programmatically
3. Convert from OSM XML to CSVs
4. Construct schemata for structure within SQLite3 and import CSVs
5. Run a sampling of potential queries

I have chosen to use IPython Notebook to construct this document because it easily juxtaposes code with results and commentary.

Sampling a Large Dataset

The unzipped OSM XML file for Humboldt County is over five hundred megabytes in size. It is unwieldy and unable to be kept in memory buffer for easy manipulation (at least, on my home desktop).

Our very first task is to take a 1/100th sample of the information contained within our full OSM file.

In [9]:
```
# -*- coding: utf-8 -*-

import xml.etree.ElementTree as ET

OSM_FILE = r"C:\Users\Bash\Desktop\Udacity\2_Data_Analysis\P3\Project\fromscratch\humboldt_california.osm\hum
boldt_california.osm"
SAMPLE_FILE = r"C:\Users\Bash\Desktop\Udacity\2_Data_Analysis\P3\Project\fromscratch\humboldt_california.osm
\sample.osm"

k = 100

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


with open(SAMPLE_FILE, 'wb') as output:
    output.write('<?xml version="1.0" encoding="UTF-8"?>\n')
    output.write('<osm>\n  ')

    # Write every kth top level element
    for i, element in enumerate(get_element(OSM_FILE)):
        if i % k == 0:
            output.write(ET.tostring(element, encoding='utf-8'))

    output.write('</osm>')
```

This program, dubbed samplerfrag.py, runs a function to grab top-level elements with the tags 'node', 'way', and 'relation'. Rather than grabbing them all, it uses a conditional statement:

```
if i % k == 0,
```

a remainder division, allowing the user to adjust sample file size by altering k.

Our sample.osm file, as written, is 5,555KB. This is much more manageable than our full OSM file.

Auditing

Now that we have a sample of our dataset to perfect our Python code against before running it on the full dataset, we may begin the process of auditing the OSM.

The most important information contained in the XML are held in the tags and attributes within each element. We will first explore the tags that are present in our OSM file, then audit the values of certain tags for rigor.

Throughout the remainder of this document, programs may be run on the sample, the full file, or both. I have already crafted the Python programs to run well, but the comparison between the two can sometimes be interesting. When the program is run over the sample only, it is to conserve space - the full file's dictionary outputs can be quite expansive.

Tag Exploration

In [1]:
```
# -*- coding: utf-8 -*-
import xml.etree.cElementTree as ET
import pprint
import re

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


def key_type(element, keys):
    if element.tag == "tag":
        discountcaps = element.attrib['k'].lower()
        if problemchars.search(discountcaps):
            keys['problemchars'] += 1
        elif lower_colon.search(discountcaps):
            keys['lower_colon'] += 1
        elif lower.search(discountcaps):
            keys['lower'] += 1
        else:
            keys['other'] += 1

    return keys

def other_keys(element):                #prints 'other' keys, discounts all-caps 'others'
    if element.tag == "tag":
        discountcaps = element.attrib['k'].lower()
        if not problemchars.search(discountcaps):
            if not lower_colon.search(discountcaps):
                if not lower.search(discountcaps):
                    return discountcaps


def process_map(filename):
    keys = {"lower": 0, "lower_colon": 0, "problemchars": 0, "other": 0}
    otherlist = []
    otherdict = {}
    for _, element in ET.iterparse(filename):
        keys = key_type(element, keys)
        otherlist.append(other_keys(element))
    for other in otherlist:
        if other in otherdict:
            otherdict[other] += 1
        else:
            otherdict[other] = 1

    return keys, otherdict

pprint.pprint(process_map(r"C:\Users\Bash\Desktop\Udacity\2_Data_Analysis\P3\Project\fromscratch\humboldt_cal
ifornia.osm\sample.osm"))
```

```
({'lower': 3663, 'lower_colon': 7127, 'other': 15, 'problemchars': 0},
 {None: 68473,
  'is_in:iso_3166_2': 1,
  'name_1': 2,
  'source:hgv:national_network': 4,
  'tiger:name_base_1': 5,
  'tiger:zip_left_1': 3})
```

This program, dubbed tagexplore.py, functions by comparing tags against a few regular expressions in order to filter tags into dictionary keys. A further level of exploration is conducted into the "others" that fail to trigger the regular expressions filter. We find no problem characters, but quite a collection of "others".

Our problem children in this data seem to belong to four groups:

```
1. Tags with underscores
2. Tags with hyphens
3. Tags with more than one colon
4. The big one - Nones. Nearly seven million of them.
```

This first exploration of the OSM file's tags also shows us our first glimpse of Tiger GIS data from the federal government. We will be untangling Tiger tags in the next program.

One further observation - the tag proportion between our sample and our full OSM file are remarkably regular. Only "other" contains approximately half what we would expect with a 1/100th portion of the full file. This suggests that "others" may appear in dense clusters throughout the full OSM file.

Street Name Consistency Audit

In [2]:
```python
# -*- coding: utf-8 -*-
import re
import xml.etree.cElementTree as ET
from collections import defaultdict
import pprint


expected = ["Street", "Avenue", "Boulevard", "Drive", "Court", "Place", "Square", "Lane", "Road",
            "Trail", "Parkway", "Commons", "Access", "Acres", "Airport", "Alley", "Bridge", "Camp", "Center",
            "Circle", "Cove", "Creek", "Crossing", "Gulch", "Heights", "Highway", "Hill", "Hollow","Loop",
            "Mainline","Park", "Pass", "Point", "Railroad", "Ranch", "Route", "Row", "Run", "Spring", "Terrac
e",
            "View", "Vista", "Way"]

mapping = { "St": "Street",
            "St.": "Street",
            "street": "Street",
            "Dr": "Drive",
            "Ave": "Avenue",
            "Rd.": "Road",
            "Rd": "Road",
            "Int": "Intersection",
            "Blvd": "Boulevard",
            "Ln": "Lane",
            "Rnch": "Ranch",
            "Ctr": "Center",
            "lane": "Lane"
            }

street_type_re = re.compile(r'\b\S+\.?$', re.IGNORECASE) #used in audit_street_type()

def audit_street_type(street_types, street_name): #used in audit()
    m = street_type_re.search(street_name)
    if m:
        street_type = m.group()
        if street_type not in expected:
            street_types[street_type].add(street_name)

def is_street_name(elem):   #used in audit()
    if (elem.attrib['k'] == "addr:street"):
        return True
    else:
        return False


def audit(osmfile):
    osm_file = open(osmfile, "r")
    street_types = defaultdict(set)
    for event, elem in ET.iterparse(osm_file, events=("start",)):
        if elem.tag == "node" or elem.tag == "way":
            maybename = ""
            istiger = False
            for tag in elem.iter("tag"):
                if tag.attrib['k'] == "name":
                    maybename = tag.attrib['v']
                if tag.attrib['k'] == "tiger:name_base":
                    istiger = True
                if is_street_name(tag):
                    audit_street_type(street_types, tag.attrib['v'])
            if istiger:
                audit_street_type(street_types,maybename)
    osm_file.close()
    pprint.pprint(dict(street_types))
    return street_types


pprint.pprint(audit(r"C:\Users\Bash\Desktop\Udacity\2_Data_Analysis\P3\Project\fromscratch\humboldt_californi
a.osm\sample.osm"))
```

```
{'13N11': set(['Forest Route 13N11']),
 '1N18': set(['Forest Rt 1N18']),
 '2S05': set(['Forest Route 2S05']),
 '5N27': set(['Forest Route 5N27']),
 '6N12D': set(['Forest Route 6N12D']),
 'A': set(['Road A'])}
defaultdict(<type 'set'>, {'A': set(['Road A']), '13N11': set(['Forest Route 13N11']), '6N12D': set(['Forest
 Route 6N12D']), '2S05': set(['Forest Route 2S05']), '1N18': set(['Forest Rt 1N18']), '5N27': set(['Forest R
oute 5N27'])})
```

This program, dubbed audit_xml.py, identifies unique instances of street types (Avenue, Boulevard, Street, etc.). It is a program that may be used to identify error within our file - for example, lower case spellings (Jones street), contractions (Jones St), as well as a host of oddballs due primarily to highways and GIS data (North Highway 101, 13N11).

This audit allows us to iterate through the dictionaries, finding allowable street types to add to our list of "expected" street types. This causes them to be excluded from our output dictionary. After removing all allowable street types, we may begin to identify and map changes that will improve the data.

You may have noticed the handling of Tiger data in this program. Many street names in Humboldt Country's OSM are contained in Tiger GIS tags, which require special handling. The primary difficulty lies in distinguishing tags where k="name" and are Tiger GIS street names from tags where k="name" and might be a hairdresser or cemetary. I chose to handle it by implementing a sort of logic switch that will audit values where k="name" if it is followed later by a k="tiger:name_base".

The "mapping" dictionary is included in this program but isn't used. It is to demonstrate how the dictionary output of our audit program may be used to construct a mapping dictionary to be used in a tag-updating program later.

Improving the OSM programmatically

It's easy to forget that, up until now, we have not actually altered anything within our file. We have only surveyed the landscape of the millions of tags and values of the Humboldt Country OSM file. Our next program, tagmaul.py, will actually open the OSM and sample files and correct values to a standard format.

In [2]:
```
# -*- coding: utf-8 -*-

import xml.etree.cElementTree as ET


mapping = { "St": "Street",
            "St.": "Street",
            "street": "Street",
            "Dr": "Drive",
            "Ave": "Avenue",
            "Rd.": "Road",
            "Rd": "Road",
            "Int": "Intersection",
            "Blvd": "Boulevard",
            "Ln": "Lane",
            "Rnch": "Ranch",
            "Ctr": "Center"
            }


def update_name(name, mapping):     # used in updater
    newname = None
    namesplit = name.split()
    for key, value in mapping.iteritems():
        if namesplit[-1] == key:
                newname = name.replace(key,value)
    if newname == None:
        return name
    return newname


def updater(osm_file):
    osmfile = open(osm_file,"r+")
    treeparse = ET.iterparse(osm_file, events=("start",))
    for event, elem in treeparse:
        if elem.tag == "node" or elem.tag == "way":
            maybename = ""
            istiger = False
            for tag in elem.iter("tag"):
                if tag.attrib['k'] == "addr:street":
                    if tag.attrib['v'] != update_name(tag.attrib['v'], mapping):
                        print 'Before:', tag.attrib['v']
                        tag.attrib['v'] = update_name(tag.attrib['v'], mapping)
                        print 'After:', tag.attrib['v']
                if tag.attrib['k'] == 'name':
                    maybename = tag.attrib['v']
                if tag.attrib['k'] == "tiger:name_base":
                    istiger = True
            if istiger:
                for tag in elem.iter("tag"):
                    if tag.attrib['k'] == "name":
                        if tag.attrib['v'] != update_name(tag.attrib['v'], mapping):
                            print 'Before:', tag.attrib['v']
                            tag.attrib['v'] = update_name(tag.attrib['v'], mapping)
                            print 'After:', tag.attrib['v']
    osmfile.close()


updater(r"C:\Users\Bash\Desktop\Udacity\2_Data_Analysis\P3\Project\fromscratch\humboldt_california.osm\sampl
e.osm")
updater(r"C:\Users\Bash\Desktop\Udacity\2_Data_Analysis\P3\Project\fromscratch\humboldt_california.osm\humbol
dt_california.osm")
```

```
Before: 10th St
After: 10th Street
Before: 9th St
After: 9th Street
Before: E St
After: E Street
Before: F St
After: F Street
Before: 7th St
After: 7th Street
Before: I St
After: I Street
Before: 9th street
After: 9th Street
Before: L K Wood Blvd
After: L K Wood Boulevard
Before: Just N Rnch
After: Just N Ranch
Before: Just N Rnch
After: Just N Ranch
Before: Grnlf Rnch
After: Grnlf Ranch
Before: Se Cnr Kenmar Road Int
After: Se Cnr Kenmar Road Intersection
Before: Nw Cnr Elk River Int
After: Nw Cnr Elk River Intersection
Before: Ne Cnr Trinidad Int
After: Ne Cnr Trinidad Intersection
Before: Nw Cnr Trinidad Int
After: Nw Cnr Trinidad Intersection
```

We can observe a sampling of attributes that are updated via our method. In a more complete project, we would move on to writing a new OSM file with updated values, but this falls outside the scope of this project.

In fact, this update program is where most future improvements may be made. Note, for instance, that our highway intersections still have directional contractions (Se Cnr => Southeast Corner). These contractions also exist in a number of other street names.

You may also note that this program limits itself in its scope to the final item in a list made by string.split() for correction. This will not catch contractions in dirtier names like 'Bank Rd North' where the last member of the list is not the actual street type. These would require further work to handle.

Writing to CSV

Our most complex program by far, xml_to_csv.py must be able to discern different data and add them to their correct list or dictionary for export, in CSV, to SQLite3. What follows is the entire code followed by a breakdown of each segment of the code.

In [ ]:

```python
# -*- coding: utf-8 -*-
import xml.etree.cElementTree as ET
import re
import csv
import codecs
import cerberus
import schema

''' variables used in process_map and subfunctions shape_element and validate_element'''
NODES_PATH = r"C:\Users\Bash\Desktop\Udacity\2_Data Analysis\P3\Project\CSVs\nodes.csv"
NODE_TAGS_PATH = r"C:\Users\Bash\Desktop\Udacity\2_Data Analysis\P3\Project\CSVs\nodes_tags.csv"
WAYS_PATH = r"C:\Users\Bash\Desktop\Udacity\2_Data Analysis\P3\Project\CSVs\ways.csv"
WAY_NODES_PATH = r"C:\Users\Bash\Desktop\Udacity\2_Data Analysis\P3\Project\CSVs\ways_nodes.csv"
WAY_TAGS_PATH = r"C:\Users\Bash\Desktop\Udacity\2_Data Analysis\P3\Project\CSVs\ways_tags.csv"

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']

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

'''used in process_map()'''
def shape_element(element, node_attr_fields=NODE_FIELDS, way_attr_fields=WAY_FIELDS,
                  problem_chars=PROBLEMCHARS, default_tag_type='regular'):  #used in process_map()
    """Clean and shape node or way XML element to Python dict"""

    way_nodes = []
    tags = []  # Handle secondary tags the same way for both node and way elements
    attr_list = [0000, 0.00, 0.00, "Missing Value", 0000, "Missing Value", 0000, "Missing Value"]

    for field in NODE_FIELDS:
        fieldindex = NODE_FIELDS.index(field)
        if field in element.attrib:
            attr_list[fieldindex] = element.attrib[field]

        node_attribs = dict(zip(NODE_FIELDS,attr_list))
        node_attribs['id'] = int(node_attribs['id'])
        node_attribs['lat'] = float(node_attribs['lat'])
        node_attribs['lon'] = float(node_attribs['lon'])
        node_attribs['uid'] = int(node_attribs['uid'])
        node_attribs['changeset'] = int(node_attribs['changeset'])
        node_attribs['user'] = unicode(node_attribs['user'])

    ndcounter = -1

    for child in element:
        if child.tag == 'tag':
            kval = ""
            tagtype = ""
            if re.search(PROBLEMCHARS,child.attrib['k']) != None:
                pass
            elif re.search(LOWER_COLON,child.attrib['k']) != None:
                colon_index = child.attrib['k'].find(':')
                tagtype = child.attrib['k'][:colon_index]
                kval = child.attrib['k'][colon_index+1:]
            else:
                kval = child.attrib['k']

            if tagtype == "":
                tagtype = default_tag_type

            tags.append({
                'id': element.attrib['id'],
                'key': kval,
                'value': child.attrib['v'],
                'type': tagtype
            })
        if child.tag == 'nd':
            ndcounter += 1
            way_nodes.append({
                'id': element.attrib['id'],
                'node_id': child.attrib['ref'],
                'position': ndcounter
            })

    if element.tag == 'node':
        return {'node': node_attribs, 'node_tags': tags}
    elif element.tag == 'way':
        way_attribs = dict(node_attribs)
        del way_attribs['lat'], way_attribs['lon']
        return {'way': way_attribs, 'way_nodes': way_nodes, 'way_tags': tags}

def get_element(osm_file, tags=('node', 'way', 'relation')):    #used in process_map()
    """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):    #used in process_map()
    """Raise ValidationError if element does not match schema"""
    if validator.validate(element, schema) is not True:
        field, errors = next(validator.errors.iteritems())
        message_string = "\nElement of type '{0}' has the following errors:\n{1}"
        error_strings = (
            "{0}: {1}".format(k, v if isinstance(v, str) else ", ".join(v))
            for k, v in errors.iteritems()
        )
        raise cerberus.ValidationError(
            message_string.format(field, "\n".join(error_strings))
        )

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

    def writerow(self, row):
        super(UnicodeDictWriter, self).writerow({
            k: (v.encode('utf-8') if isinstance(v, unicode) else v) for k, v in row.iteritems()
        })

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

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

    with codecs.open(NODES_PATH, 'w') as nodes_file, \
         codecs.open(NODE_TAGS_PATH, 'w') as nodes_tags_file, \
         codecs.open(WAYS_PATH, 'w') as ways_file, \
         codecs.open(WAY_NODES_PATH, 'w') as way_nodes_file, \
         codecs.open(WAY_TAGS_PATH, 'w') 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'])

process_map(r'C:\Users\Bash\Desktop\Udacity\2_Data Analysis\P3\Project\humboldt_california.osm\humboldt_calif
ornia.osm')
```

process_map()

The function process_map() is the primary function in the program that calls the rest of our functions. It contains the writers for each of our CSVs, our cerberus validator to check for integrity in our data structure before writing, and a for-loop over our previously defined get_element() function. The get_element() function uses ET.iterparse() to yield elements of the correct tag (in this case, "node", "way", and "relation"). Each element yielded is sifted by shape_element (explained below), then validated by validate_element(), another function that checks the integrity of the data against a pre-defined schema.

Finally, if the element passes validation, its data is written to the appropriate CSV.

shape_element()

The workhorse of this program is shape_element(), which accepts the iterated elements from get_element() and returns dictionaries - 'node', 'node_tags', 'way', 'way_tags', and 'way_nodes' - according to the schemata defined.

'node' and 'way'

The function begins by setting up a default missing value list with data types that will validate correctly. Then, the function iterates over each field in NODE_FIELDS, each time recording its own index so that when it finds a value in the element to replace the default, it replaces the correct value. For each found attribute within the element, the default attribute list is updated with the element's actual attributes.

We use the NODE_FIELDS for 'way' as well because 'way' is the same as 'node' other than missing 'lat' and 'lot' attributes. Before the data are returned, 'lat' and 'lon' are deleted from 'way' dictionaries identified as such.

'node_tags' and 'way_tags'

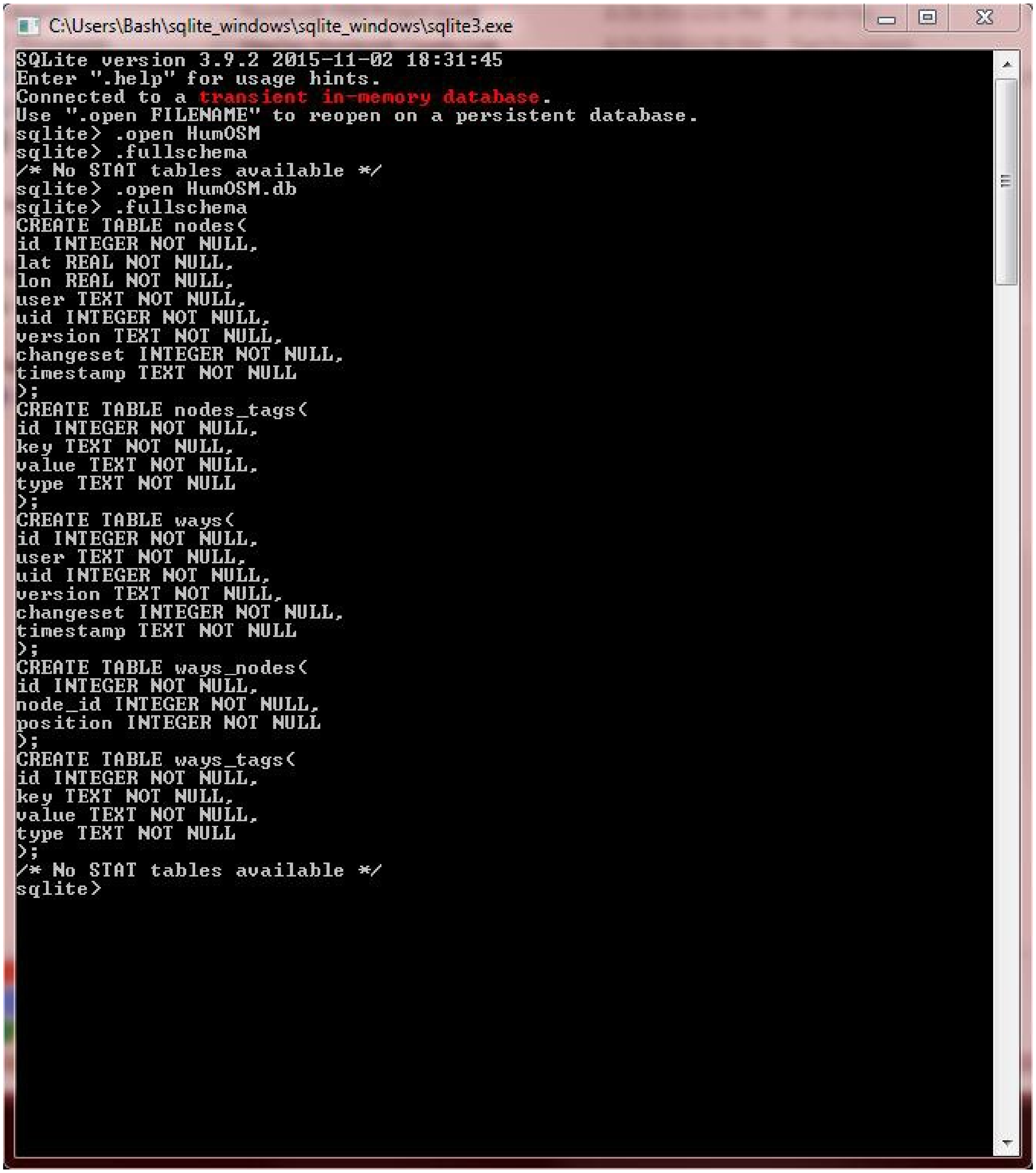
These two groups of data are handled identically, possessing identical schemata. A single tags list is created, then populated by looping through the children of the target element where the tag = "tag". Regular expressions are used to separate different tag types, and a dictionary is returned containing the parent element's id, the tag key and value attributes, as well as the type of tag (default is regular, but often tiger, in the case of our dataset).

'way_nodes'

Within the same loop over the children of the target element, the program checks for tag = "nd". These special children only exist within 'way' elements. A special position counter ('ndcounter') tells which nd tag the data belong to, and the dictionaries returned contain the parent element's id and the nd 'ref' attribute.

lConstructing the HumOSM SQL Database

Our CSVs complete and in order, the next task is to pivot to SQLite3 to construct our database. The first step in creating most SQL databases is to write the schemata:

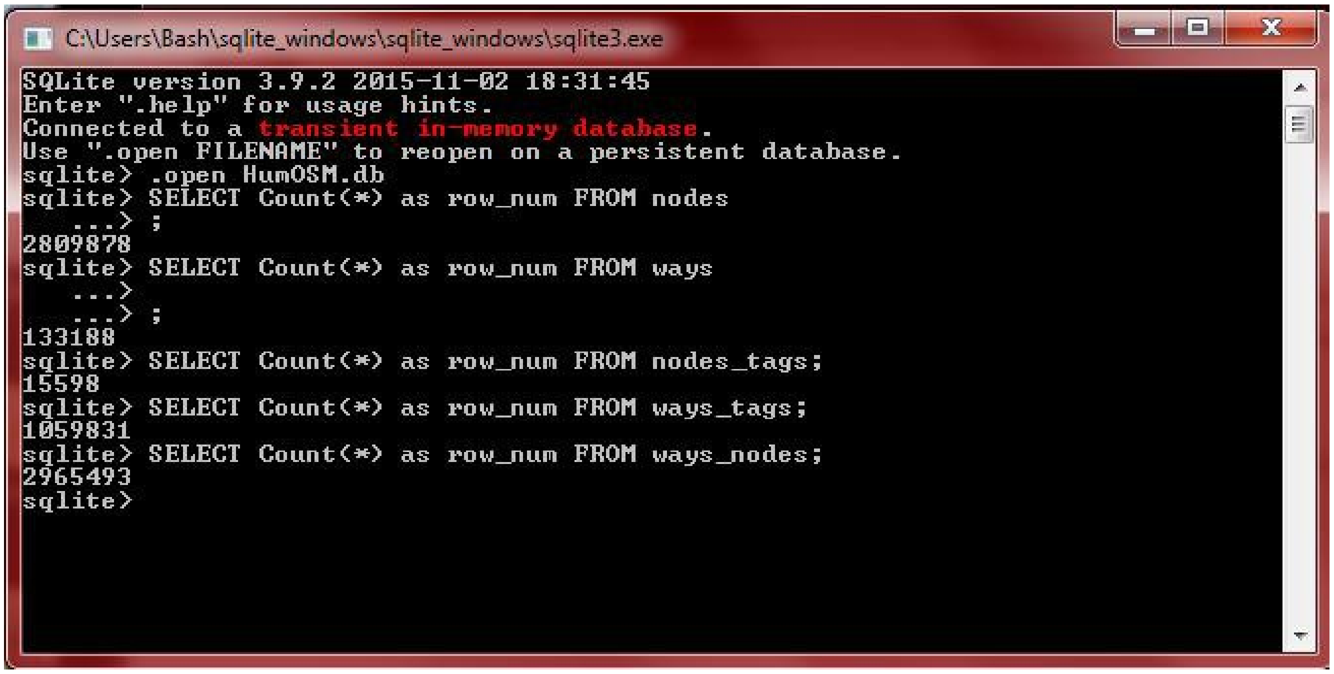


C:\Users\Bash\sqlite_windows\sqlite_windows\sqlite3.exe

```
SQLite version 3.9.2 2015-11-02 18:31:45
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .open HumOSM
sqlite> .fullschema
/* No STAT tables available */
sqlite> .open HumOSM.db
sqlite> .fullschema
CREATE TABLE nodes(
id INTEGER NOT NULL,
lat REAL NOT NULL,
lon REAL NOT NULL,
user TEXT NOT NULL,
uid INTEGER NOT NULL,
version TEXT NOT NULL,
changeset INTEGER NOT NULL,
timestamp TEXT NOT NULL
);
CREATE TABLE nodes_tags(
id INTEGER NOT NULL,
key TEXT NOT NULL,
value TEXT NOT NULL,
type TEXT NOT NULL
);
CREATE TABLE ways(
id INTEGER NOT NULL,
user TEXT NOT NULL,
uid INTEGER NOT NULL,
version TEXT NOT NULL,
changeset INTEGER NOT NULL,
timestamp TEXT NOT NULL
);
CREATE TABLE ways_nodes(
id INTEGER NOT NULL,
node_id INTEGER NOT NULL,
position INTEGER NOT NULL
);
CREATE TABLE ways_tags(
id INTEGER NOT NULL,
key TEXT NOT NULL,
value TEXT NOT NULL,
type TEXT NOT NULL
);
/* No STAT tables available */
sqlite>
```

Once the data are imported into their respective tables, we may begin to call queries on our database!

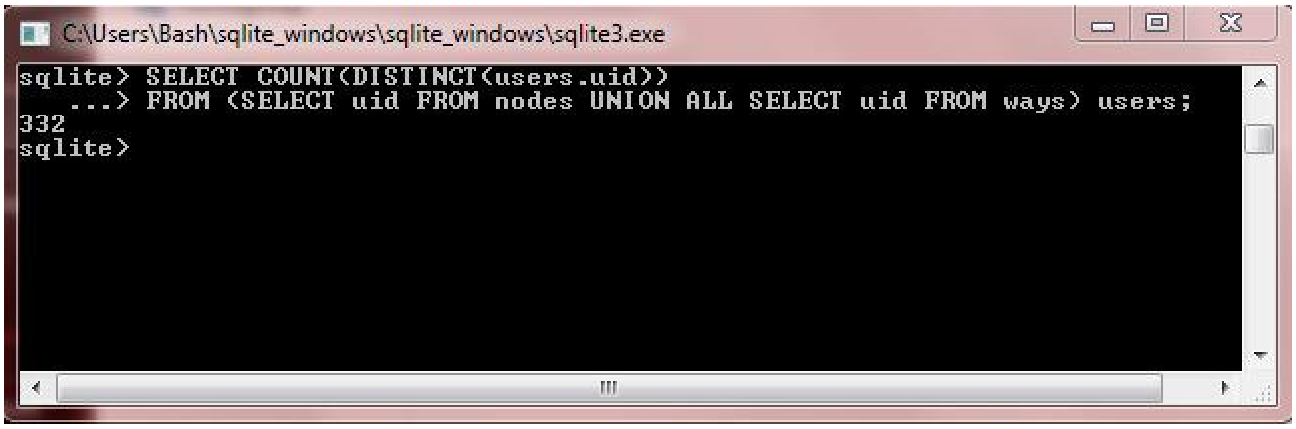
How many rows of information do we have in each of our tables?



C:\Users\Bash\sqlite_windows\sqlite_windows\sqlite3.exe

```
SQLite version 3.9.2 2015-11-02 18:31:45
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .open HumOSM.db
sqlite> SELECT Count(*) as row_num FROM nodes
   ...> ;
2809878
sqlite> SELECT Count(*) as row_num FROM ways
   ...>
   ...> ;
133188
sqlite> SELECT Count(*) as row_num FROM nodes_tags;
15598
sqlite> SELECT Count(*) as row_num FROM ways_tags;
1059831
sqlite> SELECT Count(*) as row_num FROM ways_nodes;
2965493
sqlite>
```

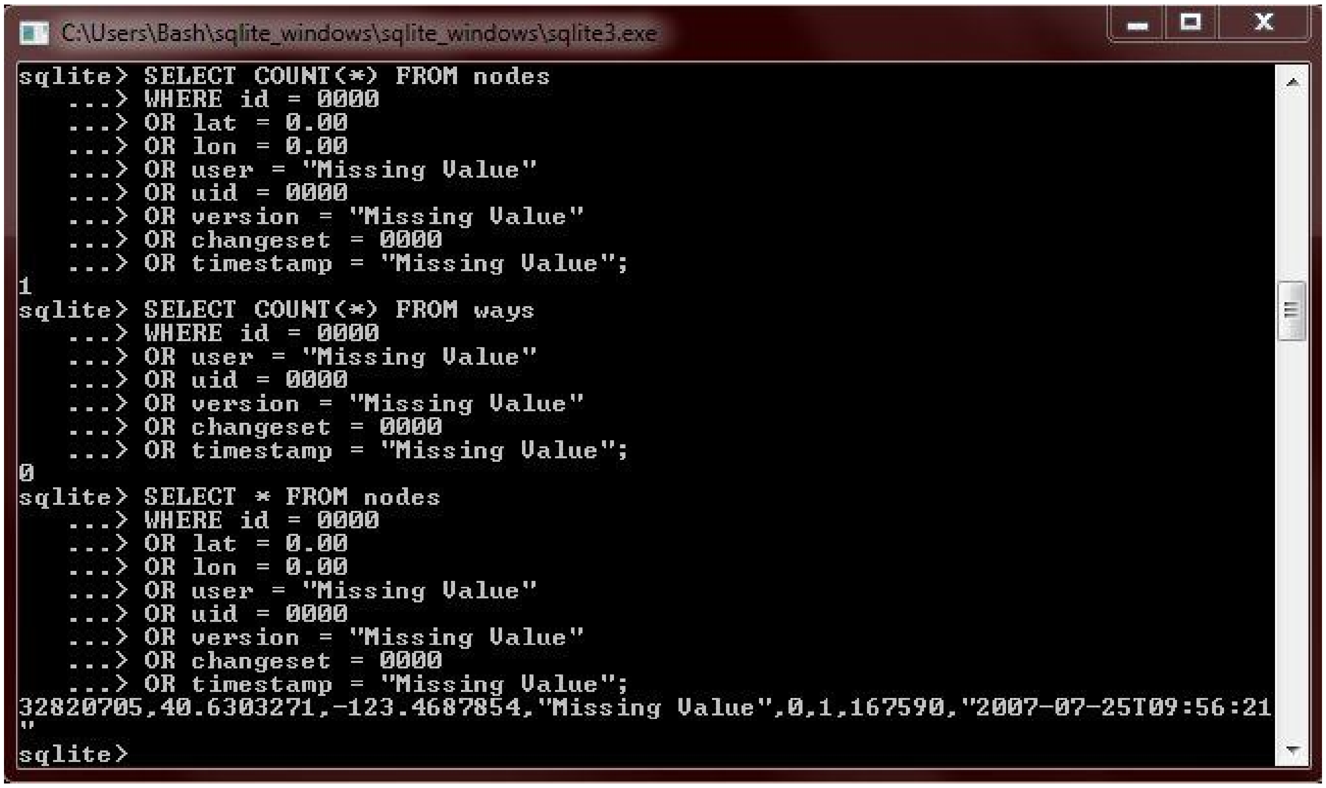
How many unique users contributed to the dataset? (Special thanks to the SQL Sample Project for help with syntax)



C:\Users\Bash\sqlite_windows\sqlite_windows\sqlite3.exe

```
sqlite> SELECT COUNT(DISTINCT(users.uid))
   ...> FROM (SELECT uid FROM nodes UNION ALL SELECT uid FROM ways) users;
332
sqlite>
```

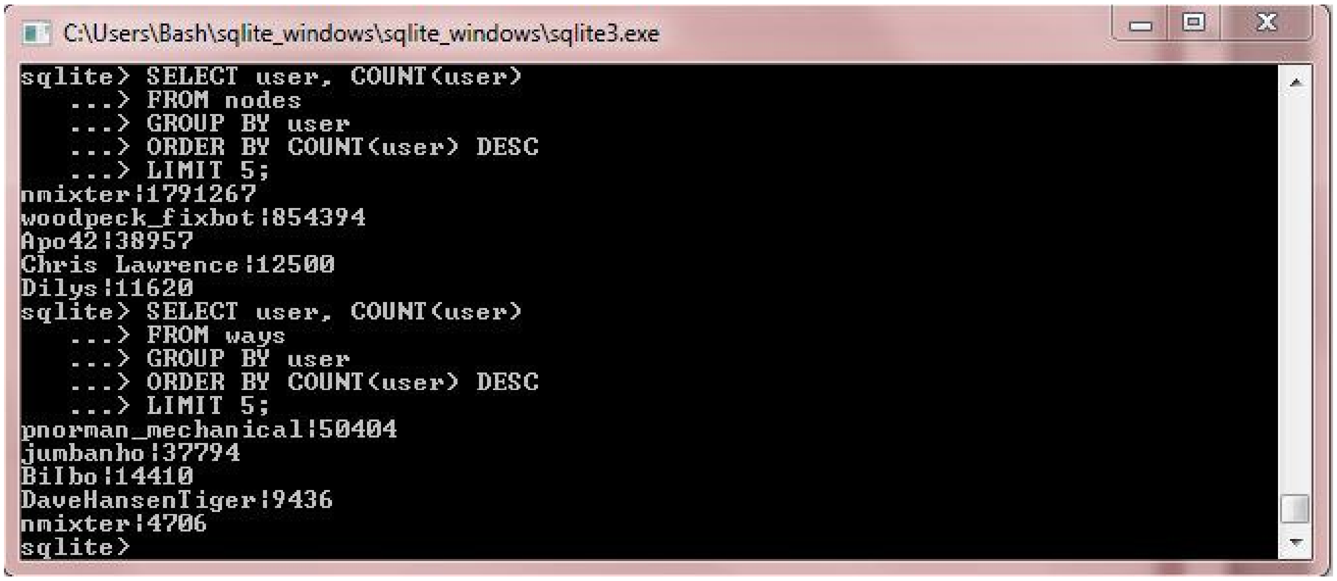
Given the special attention paid to dealing with missing values in the dataset, I wanted to know just how many rows from our 'nodes' and 'ways' tables were filled with the default values.



```
sqlite> SELECT COUNT(*) FROM nodes
   ...> WHERE id = 0000
   ...> OR lat = 0.00
   ...> OR lon = 0.00
   ...> OR user = "Missing Value"
   ...> OR uid = 0000
   ...> OR version = "Missing Value"
   ...> OR changeset = 0000
   ...> OR timestamp = "Missing Value";
1
sqlite> SELECT COUNT(*) FROM ways
   ...> WHERE id = 0000
   ...> OR user = "Missing Value"
   ...> OR uid = 0000
   ...> OR version = "Missing Value"
   ...> OR changeset = 0000
   ...> OR timestamp = "Missing Value";
0
sqlite> SELECT * FROM nodes
   ...> WHERE id = 0000
   ...> OR lat = 0.00
   ...> OR lon = 0.00
   ...> OR user = "Missing Value"
   ...> OR uid = 0000
   ...> OR version = "Missing Value"
   ...> OR changeset = 0000
   ...> OR timestamp = "Missing Value";
32820705,40.6303271,-123.4687854,"Missing Value",0,1,167590,"2007-07-25T09:56:21
"
sqlite>
```

Naturally, all of the extra trouble I went through to handle missing values was apparently caused by a single 'node' element that was missing its 'user' value. C'est la vie!

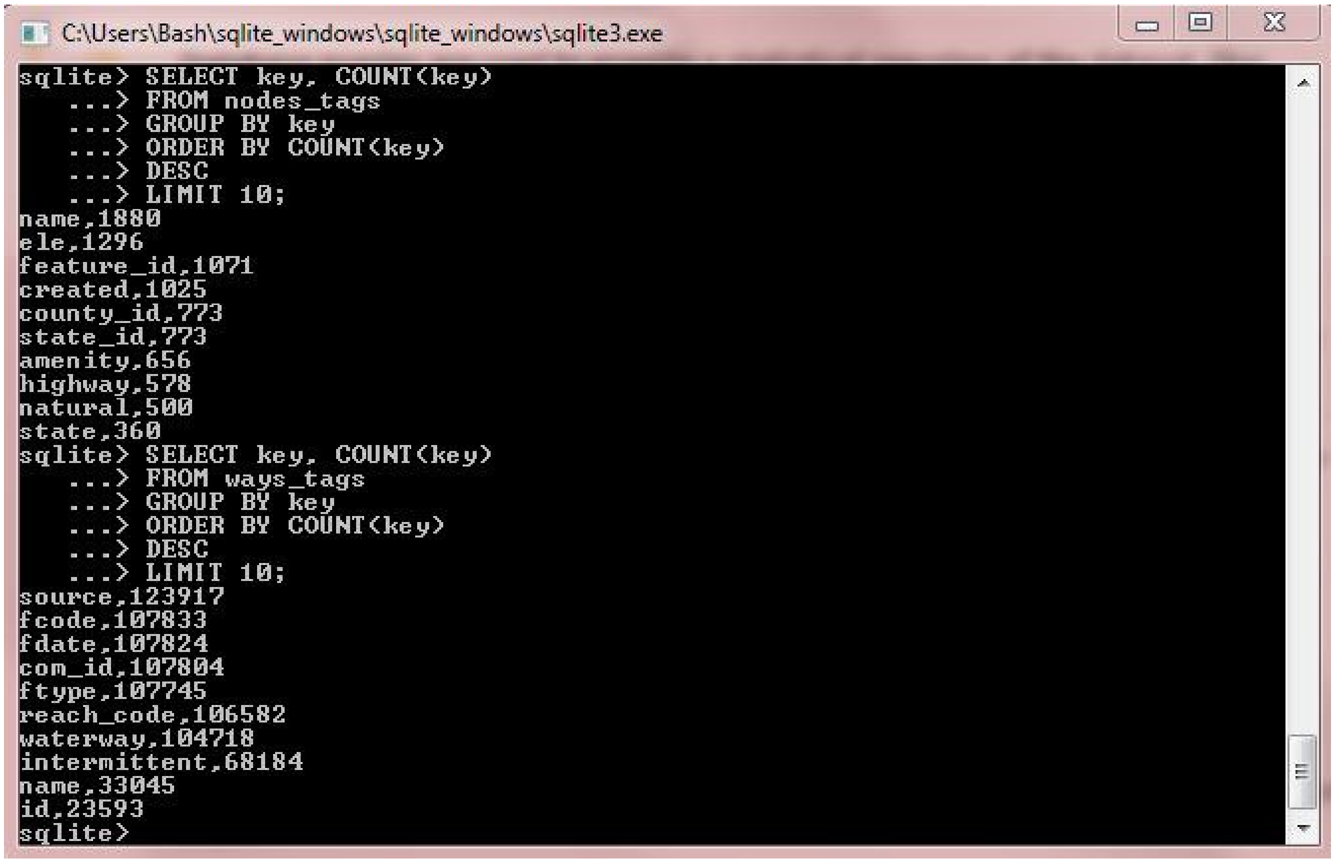
Next, I was curious about the proportion of contributions by different users as well as whether ways and nodes had significantly different user contributors.



```
sqlite> SELECT user, COUNT(user)
   ...> FROM nodes
   ...> GROUP BY user
   ...> ORDER BY COUNT(user) DESC
   ...> LIMIT 5;
nmixter|1791267
woodpeck_fixbot|854394
Apo42|38957
Chris Lawrence|12500
Dilys|11620
sqlite> SELECT user, COUNT(user)
   ...> FROM ways
   ...> GROUP BY user
   ...> ORDER BY COUNT(user) DESC
   ...> LIMIT 5;
pnorman_mechanical|50404
jumbanho|37794
Bilbo|14410
DaveHansenTiger|9436
nmixter|4706
sqlite>
```

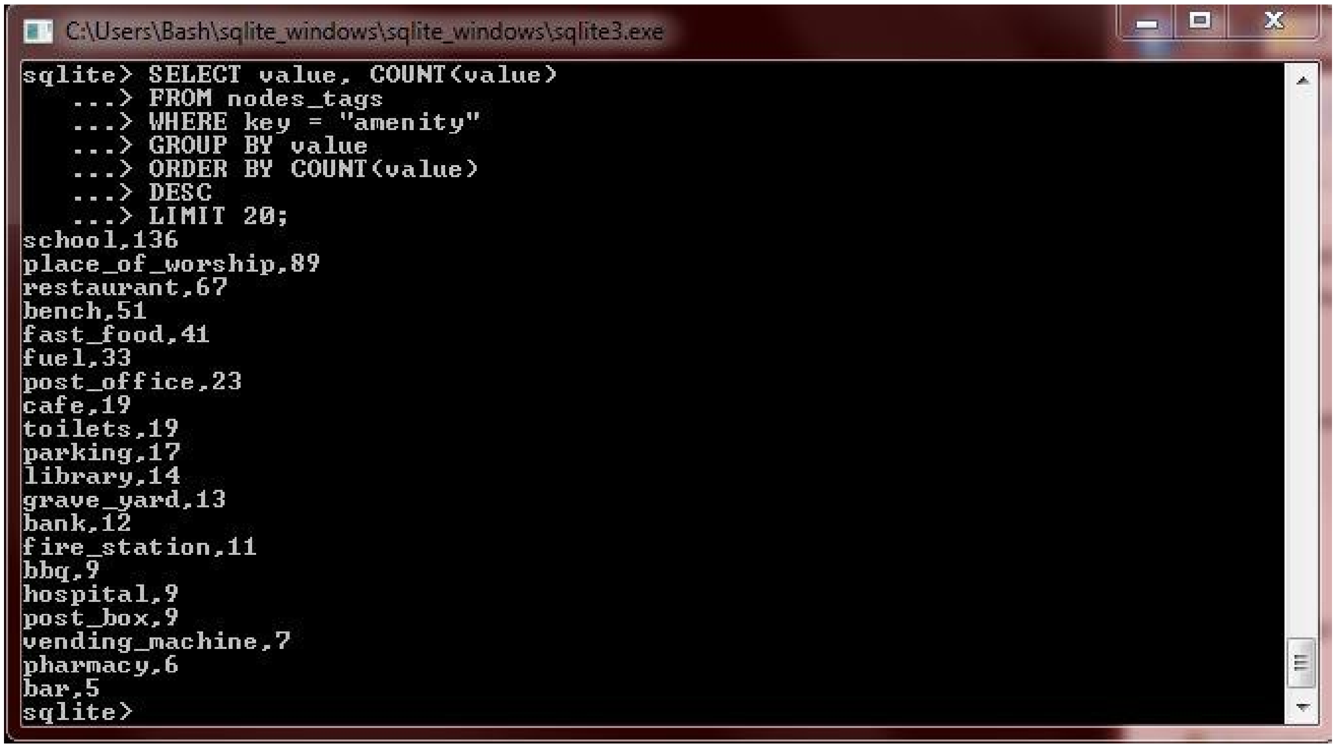
Interesting! It appears that the top five contributors to ways and nodes share only a singer user (nmixter), despite the proportion of contribution by the top five users in each case being very high.

I wanted to dig a little deeper into our nodes_tags and ways_tags tables to see if we can learn more about the dataset.


C:\Users\Bash\sqlite_windows\sqlite_windows\sqlite3.exe

```
sqlite> SELECT key, COUNT(key)
   ...> FROM nodes_tags
   ...> GROUP BY key
   ...> ORDER BY COUNT(key)
   ...> DESC
   ...> LIMIT 10;
name,1880
ele,1296
feature_id,1071
created,1025
county_id,773
state_id,773
amenity,656
highway,578
natural,500
state,360
sqlite> SELECT key, COUNT(key)
   ...> FROM ways_tags
   ...> GROUP BY key
   ...> ORDER BY COUNT(key)
   ...> DESC
   ...> LIMIT 10;
source,123917
fcode,107833
fdate,107824
com_id,107804
ftype,107745
reach_code,106582
waterway,104718
intermittent,68184
name,33045
id,23593
sqlite>
```

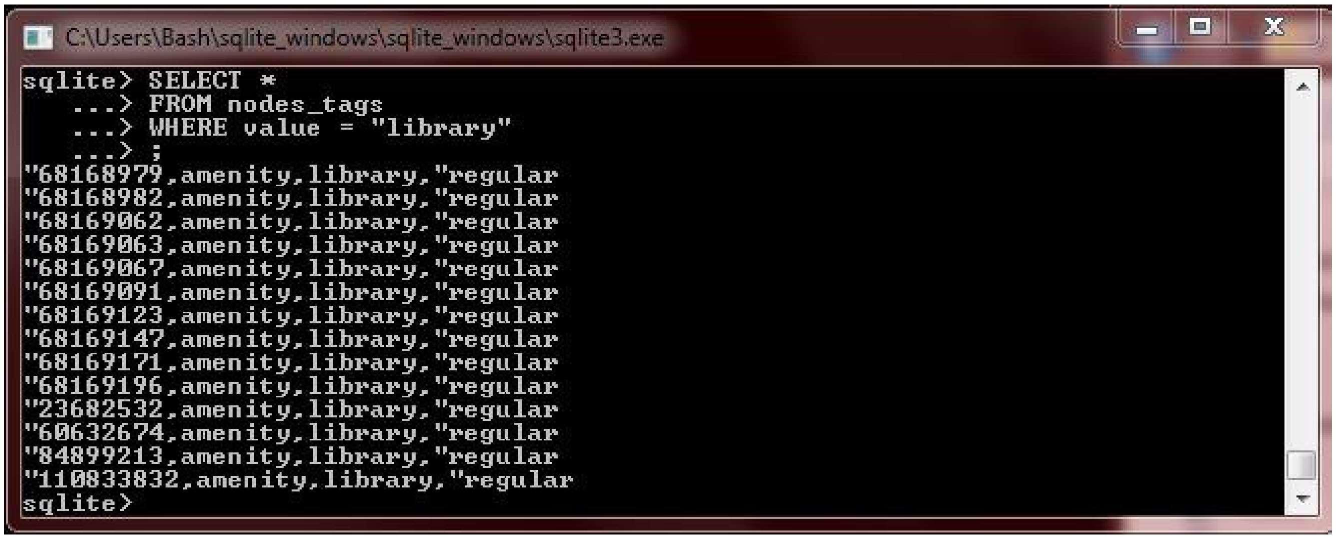
What kind of amenities are available in Humboldt?



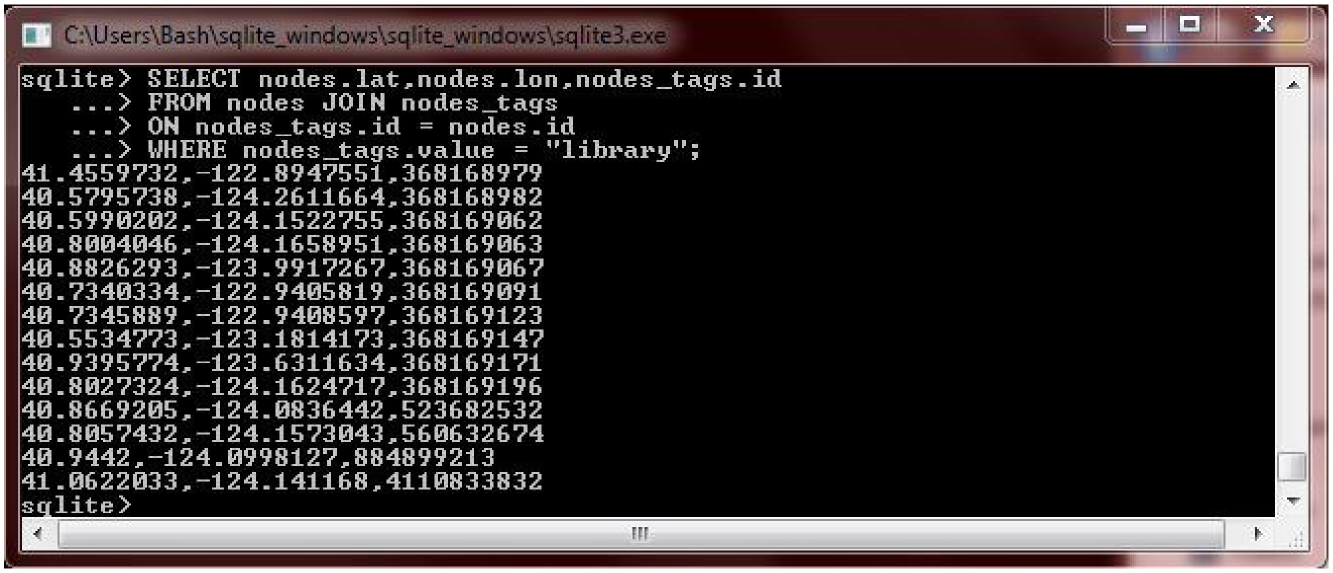
```
sqlite> SELECT value, COUNT(value)
   ...> FROM nodes_tags
   ...> WHERE key = "amenity"
   ...> GROUP BY value
   ...> ORDER BY COUNT(value)
   ...> DESC
   ...> LIMIT 20;
school,136
place_of_worship,89
restaurant,67
bench,51
fast_food,41
fuel,33
post_office,23
cafe,19
toilets,19
parking,17
library,14
grave_yard,13
bank,12
fire_station,11
bbq,9
hospital,9
post_box,9
vending_machine,7
pharmacy,6
bar,5
sqlite>
```

Finally, I noticed that the 'node_id' field in ways_nodes references the 'id' field in nodes. I'm curious which of the 'id' numbers are referenced the most.

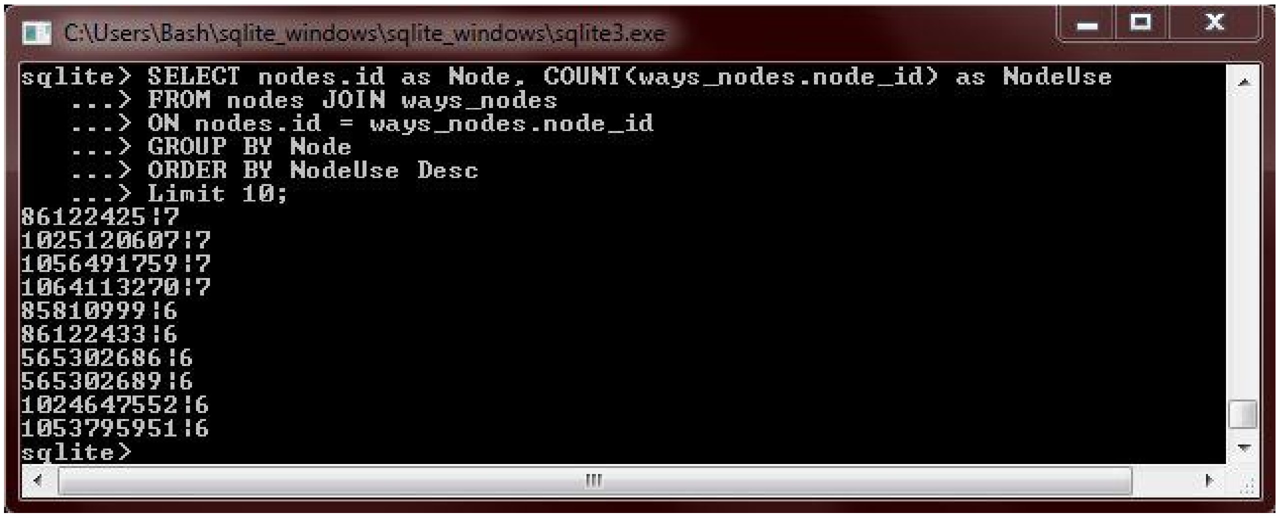
Let's get a list of Humboldt's libraries.


```
C:\Users\Bash\sqlite_windows\sqlite_windows\sqlite3.exe
sqlite> SELECT *
   ...> FROM nodes_tags
   ...> WHERE value = "library"
   ...> ;
"68168979,amenity,library,"regular
"68168982,amenity,library,"regular
"68169062,amenity,library,"regular
"68169063,amenity,library,"regular
"68169067,amenity,library,"regular
"68169091,amenity,library,"regular
"68169123,amenity,library,"regular
"68169147,amenity,library,"regular
"68169171,amenity,library,"regular
"68169196,amenity,library,"regular
"23682532,amenity,library,"regular
"60632674,amenity,library,"regular
"84899213,amenity,library,"regular
"110833832,amenity,library,"regular
sqlite>
```

Now for a little joinery. What are the coordinates for the node associated with each library?


```
C:\Users\Bash\sqlite_windows\sqlite_windows\sqlite3.exe
sqlite> SELECT nodes.lat,nodes.lon,nodes_tags.id
   ...> FROM nodes JOIN nodes_tags
   ...> ON nodes_tags.id = nodes.id
   ...> WHERE nodes_tags.value = "library";
41.4559732,-122.8947551,368168979
40.5795738,-124.2611664,368168982
40.5990202,-124.1522755,368169062
40.8004046,-124.1658951,368169063
40.8826293,-123.9917267,368169067
40.7340334,-122.9405819,368169091
40.7345889,-122.9408597,368169123
40.5534773,-123.1814173,368169147
40.9395774,-123.6311634,368169171
40.8027324,-124.1624717,368169196
40.8669205,-124.0836442,523682532
40.8057432,-124.1573043,560632674
40.9442,-124.0998127,884899213
41.0622033,-124.141168,4110833832
sqlite>
```

Finally, I've notices that the node_id field in the ways_nodes table seems to refer to the id field of nodes. I'm interested to know which nodes are referenced most often.


C:\Users\Bash\sqlite_windows\sqlite_windows\sqlite3.exe

```
sqlite> SELECT nodes.id as Node, COUNT(ways_nodes.node_id) as NodeUse
   ...> FROM nodes JOIN ways_nodes
   ...> ON nodes.id = ways_nodes.node_id
   ...> GROUP BY Node
   ...> ORDER BY NodeUse Desc
   ...> Limit 10;
86122425|7
1025120607|7
1056491759|7
1064113270|7
85810999|6
86122433|6
565302686|6
565302689|6
1024647552|6
1053795951|6
sqlite>
```

Suggestions for Improving the Dataset

My first recommendation is a fairly obvious one – Tiger GIS keys and values need to be integrated into the OSM in a way that doesn't cause name overlap with other types of keys and values. It may be a special challenge for an open map system, as everybody contributing to the map may have different ideas about structure, but the Tiger tags in particular took quite a bit of time and effort to untangle.

Tiger is not the only offender, it seems. There are very many tag.attrib['k'] = 'name' values that are not useful, including a few different coordinate notations.

Open projects like these make me think of Wikipedia. Naturally, Wikipedia's user base is very large and fiercely competitive when it comes to the rigor of the information contained, a standard perhaps unlikely for OSM. However, the process of providing existing structure and alteration auditing by a group of trusted volunteers tends to give Wikipedia the advantage of flexibility and integrity.

In short, if a similar system for auditing data contributions to OSM could be managed, there may be less potential for overlap of tag keys and values from different user notation systems.