

P3: Wrangle OpenStreetMap Data

Alex Bugrimenko

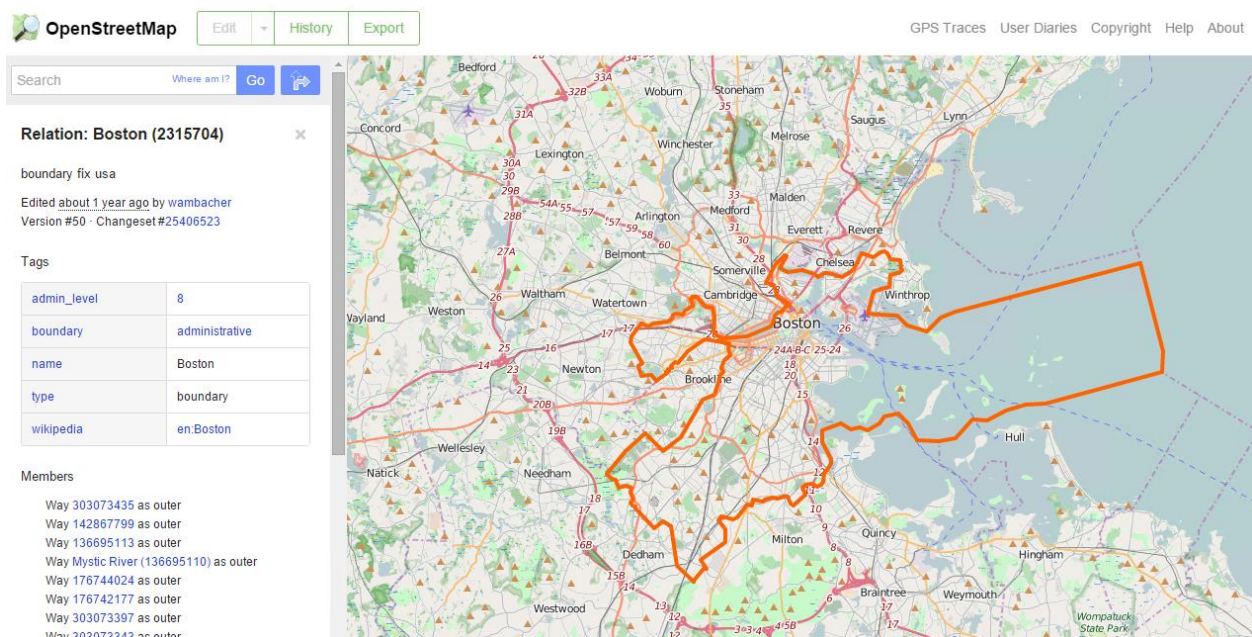
Section 1. Overview of the Data

Dataset description

I live in south New Hampshire and work in Boston area for more than 15 years. I've been a witness of big changes in the city landscape, including The Big Digg reconstruction of the major high way and was very excited to work with the Open Street map of the Boston City.

The dataset I've been working with, describes Boston area - City of Boston, Suffolk County, Massachusetts, United States of America. On the OpenStreetMap.org this map can be viewed using the following url:

<https://www.openstreetmap.org/relation/2315704#map=11/42.3124/-70.9979>



OpenStreetMap Edit History Export GPS Traces User Diaries Copyright Help About

Search Where am I? Go

Relation: Boston (2315704)

boundary fix usa

Edited about 1 year ago by wambacher
Version #50 - Changeset #25406523

Tags

admin_level	8
boundary	administrative
name	Boston
type	boundary
wikipedia	en:Boston

Members

- Way 303073435 as outer
- Way 142867799 as outer
- Way 136695113 as outer
- Way Mystic River (136695110) as outer
- Way 176744024 as outer
- Way 176742177 as outer
- Way 303073397 as outer
- Way 303073343 as outer

The osm file describing the area has been downloaded from the mapzen.com:

https://s3.amazonaws.com/metro-extracts.mapzen.com/boston_massachusetts.osm.bz2

Unpacked file is over 421M in size (421,590,851). A sample of the file is attached to the project. It is produced by selecting every 50th element from the original file (to generate the sample file, I used slightly modified version of the code provided in the project description – function get_samplefile in the attached P3_BostonMA.py file).

Open Street Map file (osm) is an xml file. The goal of the project is to process the dataset and load it into MongoDB for further analysis and use. During this process data must be transformed from xml format to JSON and loaded into MongoDB in a form of JSON documents.

Section 2. Data Cleaning and Transformation

Dataset Structure

In order to effectively work with data we need to know its structure. There is a Wiki documentation available for Open Street Map, which describes dataset:

https://wiki.openstreetmap.org/wiki/OSM_XML

I could not find an official xsd schema and even when we have a comprehensive data schema description (usually, an xsd schema for xml docs), it is always a good idea to validate data structure that we have to work with. In my case, I want to analyze osm file for Boston Area. In particular, I'm interested to know what xml elements presented there, what attributes they have and what is the hierarchical structure of those elements.

For a relatively large files it is always the best to use xml readers that work with a stream and do not have to load complete document into memory. As a result of this nature, it is somewhat challenging to extract document schema from an xml reader that reads only one element at a time. Therefore I developed a simple function, which can discover xml structure and report it in a simple yet helpful form.

All high level elements have total number of such elements shown in brackets next to them. Child nodes represent direct children for each element, in other words show relationship only one level deep. All attributes are listed in alphabetical order.

For the analyzed data set I got the following:

```

===== bounds [1] =====
--- attributes:
maxlat, maxlon, minlat, minlon

===== member [9590] =====
--- attributes:
ref, role, type

===== nd [2244151] =====
--- attributes:
ref

===== node [1888262] =====
--- children nodes:
tag
--- attributes:
changeset, id, lat, lon, timestamp, uid, user, version

===== osm [1] =====
--- children nodes:
bounds, node, relation, way
--- attributes:
generator, timestamp, version

===== relation [1188] =====
--- children nodes:
member, tag
--- attributes:
changeset, id, timestamp, uid, user, version

```

```

===== tag [846720] =====
--- attributes:
k, v

===== way [294396] =====
--- children nodes:
nd, tag
--- attributes:
changeset, id, timestamp, uid, user, version

```

By viewing the file and checking the values of different elements, it is easy to notice that element *tag* used to provide extra flexibility to the file structure: each tag element is a key-value pair defined by attributes *k* and *v* respectively. Based on Open Street documentation, tags describe physical features on the ground and there is quite detailed guidance on tags here:

https://wiki.openstreetmap.org/wiki/Map_Features

It is interesting to see what tags presented in the Boston area. I expanded my python function to get distinct values of a “k” attribute of a “tag” element. Here is a simple list of discovered values (complete list with the generated structure is attached - boston_ma_struc.txt):

```

--- tag attrib:k values:
    _Shape_Area_ [3]
    _Shape_Leng_ [3]
    abandoned [1]
    abandoned:railway [1]
    abutters [166]
    access [2758]
    access:employee [51]
    addr:city [1992]
    addr:city_parent [7]
    addr:country [122]
    addr:county [8]
    addr:floor [1]
    addr:full [4]
    addr:housename [144]
    addr:housenumber [4448]
    addr:inclusion [35]
    addr:interpolation [48]
    addr:neighbourhood [6]
    addr:place [1]
    addr:postcode [1737]
    addr:state [1276]
    addr:street [3122]
    addr:street_1 [1]
    addr:unit [4]
    address [651]
    admin_level [113]
    advertising [1]
    ...

```

Python code of the function used to discover xml structure:

```
def get_structure(source):
    ''' Gets simple structure of the file:
    - all elements with count
    - a distinct list of attributes
    - a distinct list of child nodes
    - distinct list of <tag k values
    '''
    nodes = defaultdict(dict)
    for _, element in ET.iterparse(source):
        k = element.tag
        if element.tag in nodes:
            nodes[k]["Count"] += 1
        else:
            nodes[k]["Count"] = 1
            nodes[k]["attrib"] = set()
            nodes[k]["children"] = set()
            nodes[k]["tag"] = defaultdict(int)
        for attr_name in element.attrib:
            nodes[k]["attrib"].add(attr_name)
            if k == "tag" and attr_name == "k":
                nodes[k]["tag"][element.attrib[attr_name]] += 1
        children = list(element) #element.getchildren()
        if children != None:
            for child in children:
                nodes[k]["children"].add(child.tag)
    return nodes
```

Elements of Interest

There are three major data elements (data primitives) in the dataset: nodes, ways and relations. We are interested to work with data elements that describe physical objects, therefore I will ignore “relations” in this project and process only two types of top level tags: “node” and “way”.

For each node or way we are interesting in knowing, when possible, its name, address, geospatial coordinates and who and when created that node. We also want to preserve information about references between nodes.

The final goal is to load the dataset into MongoDB and in order to make database structure more useful the following data transformations must be performed:

- all attributes of "node" and "way" should be turned into regular key/value pairs, except:
 - attributes in the CREATED array should be added under a key "created"
 - attributes for latitude and longitude should be added to a "pos" array, for use in geospatial indexing. Make sure the values inside "pos" array are floats and not strings.
- if second level tag "k" value contains problematic characters, it should be ignored
- if second level tag "k" value starts with "addr:", it should be added to a dictionary "address"
- if second level tag "k" value does not start with "addr:", but contains ":", it should be processed same as any other tag.
- if there is a second ":" that separates the type/direction of a street, the tag should be ignored

Note, that in the “tag” element I can see values “type”. It creates duplicate keys and therefore in the JSON structure I have attribute “node_type” to represent actual “type” of the node.

So overall JSON structure may look like this:

```
{
  "id": "2406124091",
  "node_type": "node",
  "visible": "true",
  "created": {
    "version": "2",
    "changeset": "17206049",
    "timestamp": "2013-08-03T16:43:42Z",
    "user": "linuxUser16",
    "uid": "1219059"
  },
  "pos": [41.9757030, -87.6921867],
  "address": {
    "housenumber": "5157",
    "postcode": "60625",
    "street": "North Lincoln Ave"
  },
  "node_refs": ['2199822281', '2199822284', '2199822281'],
  "amenity": "restaurant",
  "cuisine": "mexican",
  "name": "La Cabana De Don Luis",
  "phone": "1 (773)-271-5176"
}
```

Data Cleanup & Validation

There are few basic clean up procedures and validation we can perform.

Geospatial Coordinates

Coordinates of each element should be placed into a separate property “pos”, which is an array of coordinates (latitude, longitude). Data must be validated to make sure that coordinates are float numbers.

I can also check data consistency and use min and max coordinates of the dataset provided in the “bounds” element to validate that every coordinate is within described range. The boundaries for the downloaded Boston area are

```
<bounds minlat="42.228" minlon="-71.191" maxlat="42.42" maxlon="-70.923"/>
```

The function I used to audit position coordinates returns two dictionaries:

- wrongType - contains all latitude and longitude values which cannot be recognized as a float data type
- outOfRange – contains all coordinates that are recognized as float numbers, but located outside of the defined boundaries.

When I ran this check for the Boston dataset, I found no inconsistency of any kind. The python code of the function I used is:

```
def audit_pos(source):
    ''' Audits geospatial coordinates and returns all wrong values '''
    minPos = [42.228, -71.191] # [minlat, minlon] from <bounds />
    maxPos = [42.42, -70.923] # [maxlat, maxlon] from <bounds />

    wrongType = { "lat": [], "lon": [] }
    outOfRange = { "lat": [], "lon": [] }
    for event, elem in ET.iterparse(source):
        if (elem.tag == "node" or elem.tag == "way"):
            lat = ""
```

```

lon = ""
for attr in elem.attrib:
    if attr == "lat":
        lat = elem.attrib[attr]
    elif attr == "lon":
        lon = elem.attrib[attr]
# check the data
if len(lat) > 0 and len(lon) > 0:
    if not isFloat(lat):
        wrongType["lat"].append(lat)
    elif float(lat) < minPos[0] or float(lat) > maxPos[0]:
        outOfRange["lat"].append(lat)
    if not isFloat(lon):
        wrongType["lon"].append(lon)
    elif float(lon) < minPos[1] or float(lon) > maxPos[1]:
        outOfRange["lon"].append(lon)
return wrongType, outOfRange

```

Address – Street Name

Address is located in “tag” nodes where key (attribute “k”) starts with “address:”. During data extract, some cleanup could be done to make sure that all street types are normalized and presented in a unified form. For instance, “Ave” and “Ave.” values could be replaced with “Avenue”.

Usually street type is a last word in the tag with k=“addr:street”, which represents street name. However, it is not always the case. To audit street type I used the following function, which returns two dictionaries – first one with the list of all presented street types and the second one with the list of unexpected or not unified street types (street type mapping dictionary is incomplete here for the sake of example):

```

street_type_re = re.compile(r'\b\S+\.?$', re.IGNORECASE) # last word
street_type_mapping = { "St": "Street", "ST": "Street", "st": "Street" }
def audit_street_type(source):
    ''' Audits all found street types '''
    street_type_expected = ["Street", "Avenue", "Boulevard", "Drive", "Court",
                           "Place", "Square", "Lane", "Road", "Trail",
                           "Parkway", "Commons", "Terrace", "Artery", "Circle",
                           "Corner", "Park", "Row", "Turnpike", "Way", "Wharf"]

    street_types = defaultdict(int)
    unexpected_street_types = defaultdict(int)
    for event, elem in ET.iterparse(source):
        if (elem.tag == "tag") and (elem.attrib['k'] == "addr:street"):
            street_name = elem.attrib['v']
            m = street_type_re.search(street_name)
            if m:
                street_type = m.group()
                street_types[street_type] += 1
                if street_type in street_type_mapping.keys():
                    street_type = street_type_mapping[street_type]
                if street_type not in street_type_expected:
                    unexpected_street_types[street_type] += 1

    return street_types, unexpected_street_types

```

Some values, such as numeric values, may not be used as a street type. But others, such as “ave.”, “Ave”, “Ave.”, represent different ways to name street type and could be used as a mapping to transform data to JSON format, using the following function:

```
def update_streetname(name):
    ''' Cleans up street name '''
    new_name = name
    m = street_type_re.search(new_name)
    if m:
        stype = m.group()
        if stype in street_type_mapping.keys():
            new_name = re.sub(street_type_re, street_type_mapping[stype], new_name)
    return new_name
```

Based on this list I also added several values to the list of expected (valid) street types:

"Corner", "Park", "Row", "Turnpike", "Way", "Wharf"

Address – Postal code

Postal codes presented in the data set in several different formats. Most of them are regular 5 digit codes, but there are also codes with extensions, for example “02138-1901” and a few entries with codes started with MA, such as “MA 02186”. All postal codes are transformed into 5 digit codes and validated that they contain numeric values only, which is true for Boston area:

```
def update_postcode(postcode):
    ''' Cleans up postal code '''
    new_code = postcode
    if postcode.find(" ", 0) > 0:
        new_code = postcode[postcode.find(" ", 0)+1:]
    elif postcode.find("-", 0) > 0:
        new_code = postcode[:postcode.find("-", 0)]

    if (not isInt(new_code)):
        new_code = ""
    return new_code
```

Address – Inconsistent Placement

As discovered in the osm file structure analysis, element “tag” may have value “address” and also have more structured address defined in multiple tags where attribute “k” starts with “addr:”. For example:

```
<node id="257489494" lat="42.2510718" lon="-71.0138915" version="3" timestamp="2015-08-26T15:10:01Z" changeset="33598844" uid="3185877" user="KBREILLY">
  <tag k="name" v="Steward Satellite Emergency Facility - Quincy"/>
  <tag k="address" v="114 Whitwell Street, Quincy, MA"/>
  <tag k="amenity" v="hospital"/>
  <tag k="addr:city" v="Quincy"/>
  <tag k="addr:state" v="MA"/>
  <tag k="massgis:id" v="36"/>
  <tag k="short_name" v="Quincy Med Ctr"/>
  <tag k="source_url" v="http://mass.gov/mgis/hospitals.htm"/>
  <tag k="addr:street" v="Whitwell Street"/>
  <tag k="attribution" v="Office of Geographic and Environmental"/>
  <tag k="emergency_room" v="yes"/>
  <tag k="addr:housenumber" v="114"/>
</node>
```

It creates a conflict with defined JSON structure, where “address” is supposed to be a dictionary of structured information about address. To overcome it, all “tag” elements with k=“address” will be placed into “address” dictionary as “fulladdress”. For this example:


```
"address": {"city": "Quincy", "fulladdress": "114 Whitwell Street, Quincy, MA",
"street": "Whitwell Street", "housenumber": "114", "state": "MA"}, "node_type":
"node", "id": "257489494"}
```

Exceptions

Certain data elements simply cannot be imported to MongoDB in the format we want. For example, element attributes must be transformed into key-value pairs where key must not contain special characters. Therefore for attributes with special characters we have to make a choice – either reject it or transform into MongoDB compatible format, i.e. strip unsupported values or perform some other type of transformation. In this project not supported values will be rejected. Not supported characters defined as:

```
problemchars = re.compile(r'[\+/\&<>;\\"\?%#$@\.\. \t\r\n]')
```

Generating JSON File

All described above validation and transformation used in the **shape_element** function, which transforms a single xml element of osm file into JSON element with desired structure. A JSON version of the sample osm file is attached to the project.

```
def shape_element(element):
    ''' Shapes the xml element into JSON format '''
    node = {}
    node["created"] = {}
    node["address"] = {}
    if (element.tag == "node" or element.tag == "way"):
        lat = 0.0
        lon = 0.0
        node["node_type"] = element.tag
        for attr in element.attrib:
            if audit_CanNOTBeUsedAsKey(attr):
                # ignore
                continue
            elif (attr in CREATED):
                node["created"][attr] = element.attrib[attr]
            elif attr == "lat":
                lat = float(element.attrib[attr])
            elif attr == "lon":
                lon = float(element.attrib[attr])
            else:
                node[attr] = element.attrib[attr]

        if lat != 0 and lon != 0:
            node["pos"] = [lat, lon]

    for tag in element.iter("tag"):
        key = tag.attrib["k"].lower()
        if audit_CanNOTBeUsedAsKey(key):
            continue
        elif key == "address":
            node["address"]["fulladdress"] = tag.attrib["v"]
        elif key.startswith("addr:"):
            #if there is a second ":" that separates the type/direction of a
            street, the tag should be ignored
            if key.find(":", 5) < 0:
                k = key[5:]
                if len(k) < 1:
                    continue
```



```

        if k in node["address"].keys():
            print("-- Dupl address (%s): %s [id=%s]" % (k,
tag.attrib["v"], element.attrib["id"]))

        v = tag.attrib["v"]
        if (k == "street"):
            v = update_streetname(v)
        elif (k == "postcode"):
            v = update_postcode(v)

        if k in node["address"].keys():
            node["address"][k] += " " + v
        else:
            node["address"][k] = v

    else:
        node[key] = tag.attrib["v"]

for nd in element.iter("nd"):
    if "ref" in nd.attrib:
        if "node_refs" not in node.keys():
            node["node_refs"] = []
        node["node_refs"].append(nd.attrib["ref"])

if len(node["created"]) == 0:
    del node["created"]
if len(node["address"]) == 0:
    del node["address"]
return node
else:
    return None

```

Section 3. MongoDB

Loading Data to MongoDB

File size of the generated JSON file is about 497M (497,779,327). Loading such file from the client (python) application consumes a lot of memory on the remote computer. Therefore I used **mongoimport**:

```
$ mongoimport --db OSM --collection Boston --drop --file
/home/alex/Public/boston_ma.osm.json

.... imported 2182658 objects
```

To verify number of loaded objects I ran:

```
> use OSM
> db.Boston.find().count()
2182658
```

Another way of loading large file is to split it into multiple smaller files. Though it will require loading multiple files into the database as well. The easiest way to do it in this project is when we generate JSON file. A simple function that produces multiple files may look like this:

```
def osm_json_inChunks(file_in, chunk_rows=100000):
    ''' Transforms osm file into JSON file
    Do it in chunks where each file will have not more than chunk_rows elements
    File names are: <file_in>.json.1, <file_in>.json.2...
    '''
    chunk = 1
    i = 0
    fo = None
    for _, element in ET.iterparse(file_in):
        el = shape_element(element)
        if el:
            if (i == 0 or i >= chunk_rows):
                if fo:
                    fo.write("\n]")
                    file_out = "{0}.json.{1}".format(file_in, chunk)
                    fo = codecs.open(file_out, "w")
                    fo.write("[\n")
                    chunk += 1
                    i = 0
            else:
                fo.write(",\n")

        fo.write(json.dumps(el))
        i += 1
    return
```

Querying the Data

NOTE: I ran all listed below MongoDB queries via python application (P3_BostonMA_db.py), here they are listed as a console calls for simplicity.

Completeness Check

Based on the previous analysis, I want to make sure that all nodes are imported to the database. I have total number for each high level node listed in the boston_ma_struct.txt file. The two nodes we are importing are:

- Node – 1,888,262 nodes
- Way – 294,396 nodes

```
> db.Boston.aggregate([
  {"$group": {"_id": "$node_type",
    "count": {"$sum": 1}}},
  {"$sort": {"_id": 1}}
])
{u'_id': u'node', u'count': 1888262}
{u'_id': u'way', u'count': 294396}
```

Total numbers match, we can proceed.

Data Source

Working with dataset I noticed that many records have attribute “source” populated and it is often look like a file name. It is interesting to check how much data points were actually imported from some other data sources:

```
> db.Boston.aggregate([
  {"$group": {"_id": "$source",
    "count": {"$sum": 1}}},
  {"$sort": {"count": -1}},
  {"$limit": 10}
])

{u'_id': None, u'count': 2063773}
{u'_id': u'massgis_import_v0.1_20071008193615', u'count': 52647}
{u'_id': u'massgis_import_v0.1_20071009092358', u'count': 11778}
{u'_id': u'massgis_import_v0.1_20071009101959', u'count': 8477}
{u'_id': u'massgis_import_v0.1_20071009091249', u'count': 6097}
{u'_id': u'massgis_import_v0.1_20071009091627', u'count': 5792}
{u'_id': u'massdot_import_081211', u'count': 4568}
{u'_id': u'massgis_import_v0.1_20071009102616', u'count': 4520}
{u'_id': u'Bing', u'count': 3101}
{u'_id': u'massgis_import_v0.1_20071009092815', u'count': 2824}
```

The top 5 records indicate that the largest subset of records where source populated came from “massgis” files, and the sixth came from MASS DOT. To dig dipper I ran:

```
> db.Boston.aggregate([
  {"$match": {"source": {"$regex": "^(mass)"}}},
  {"$project": {
    "source": {"$substr": ["$source", 0, 7]},
    "node_type": "$node_type"
  }},
  {"$group": {"_id": {"source": "$source", "type": "$node_type"},
    "count": {"$sum": 1}}},
  {"$sort": {"count": -1}},
  {"$limit": 10}
])
{u'_id': {u'source': u'massgis', u'type': u'node'}, u'count': 84317}
{u'_id': {u'source': u'massgis', u'type': u'way'}, u'count': 22333}
{u'_id': {u'source': u'massdot', u'type': u'node'}, u'count': 4641}
{u'_id': {u'source': u'massdot', u'type': u'way'}, u'count': 42}
{u'_id': {u'source': u'mass.go', u'type': u'way'}, u'count': 1}
```

There are 88,958 “node” nodes out of total of 1,888,262 (roughly 4.71%) and about 7.6% (22,375 out of 294,396) of “way” nodes came from files from Massachusetts Office of Geographic and Environmental Information (MassGIS) and Massachusetts Department of Transportation (MassDOT).

Data Contributors

It is interesting to find out when majority of the data was loaded:

```
> db.Boston.aggregate([
  {"$match": {"created.timestamp": {"$exists": 1}}},
  {"$project": {
    "date_year": {"$substr": ["$created.timestamp", 0, 4]}
  }},
  {"$group": {"_id": "$date_year",
    "count": {"$sum": 1}}},
  {"$sort": {"count": -1}},
  {"$limit": 10}
])
{u'_id': u'2009', u'count': 1246290}
{u'_id': u'2013', u'count': 508169}
{u'_id': u'2010', u'count': 98808}
{u'_id': u'2012', u'count': 78848}
{u'_id': u'2011', u'count': 70633}
{u'_id': u'2014', u'count': 66854}
{u'_id': u'2007', u'count': 66310}
{u'_id': u'2015', u'count': 43195}
{u'_id': u'2008', u'count': 3551}
```

So the big chunk of data was entered in 2009 – about 57%. And the top 10 contributors in 2009 are:

```
> db.Boston.aggregate([
  {"$match": {"created.user": {"$exists": 1}}},
  {"$project": {
    "user": "$created.user",
```

```

    "date_year": {"$substr": ["$created.timestamp", 0, 4]}
  }},
  {"$match": {"date_year": "2009"}},
  {"$group": {"_id": "$user",
    "count": {"$sum": 1}}},
  {"$sort": {"count": -1}},
  {"$limit": 10}
])
{u'_id': u'crschmidt', u'count': 1228878}
{u'_id': u'fiveisalive', u'count': 3412}
{u'_id': u'Ahlzen', u'count': 3087}
{u'_id': u'nkhall', u'count': 2464}
{u'_id': u'PeterEastern', u'count': 1216}
{u'_id': u'Prithason', u'count': 1109}
{u'_id': u'Pouletic', u'count': 936}
{u'_id': u'JasonWoof', u'count': 819}
{u'_id': u'landeess', u'count': 618}
{u'_id': u'Bill Ricker', u'count': 568}

```

This is absolutely astonishing how much data was entered by individual users. I hope that at least some of the data was actually imported from somewhere. To check this hypothesis I can limit data by year 2009 and check the source of the user “crschmidt” entries:

```

> db.Boston.aggregate([
  {"$match": {"created.user": {"$exists": 1}}},
  {"$project": {
    "user": "$created.user",
    "date_year": {"$substr": ["$created.timestamp", 0, 4]},
    "source" : "$source"
  }},
  {"$match": {"date_year": "2009", "user": "crschmidt"}},
  {"$group": {"_id": "$source",
    "count": {"$sum": 1}}},
  {"$sort": {"count": -1}},
  {"$limit": 5}
])
{u'_id': None, u'count': 1227555}
{u'_id': u'massgis_import_v0.1_20090107171258', u'count': 631}
{u'_id': u'MassGIS OpenSpace (http://www.mass.gov/mgis/osp.htm)',
  u'count': 558}
{u'_id': u'massgis_import_v0.1_20071009090921', u'count': 36}
{u'_id': u'massgis_import_v0.1_20071008165629', u'count': 35}

```

Well, majority of the data entries do not have the source identified, but there are certain number of data points imported from files and, as suspected, from the MassGIS web site:

<http://www.mass.gov/mgis/osp.htm>

City Stats and Data Cleanup

Let's find top 20 cities based on the total number of entries:

```
> db.Boston.aggregate([
  {"$match": {"address.city": {"$exists": 1}}},
  {"$group": {"_id": "$address.city", "count": {"$sum": 1}}},
  {"$sort": {"count": -1}},
  {"$limit": 20}
])
{u'_id': u'Boston', u'count': 601}
{u'_id': u'Malden', u'count': 413}
{u'_id': u'Cambridge', u'count': 375}
{u'_id': u'Somerville', u'count': 166}
{u'_id': u'Quincy', u'count': 51}
{u'_id': u'Brookline', u'count': 33}
{u'_id': u'Weymouth', u'count': 33}
{u'_id': u'Medford', u'count': 30}
{u'_id': u'Arlington', u'count': 22}
{u'_id': u'West Roxbury', u'count': 20}
{u'_id': u'Chestnut Hill', u'count': 19}
{u'_id': u'Watertown', u'count': 17}
{u'_id': u'Jamaica Plain', u'count': 16}
{u'_id': u'Arlington, MA', u'count': 14}
{u'_id': u'Allston', u'count': 11}
{u'_id': u'Brookline, MA', u'count': 11}
{u'_id': u'Dorchester', u'count': 11}
{u'_id': u'Roslindale', u'count': 10}
{u'_id': u'Milton', u'count': 9}
{u'_id': u'Boston, MA', u'count': 9}
```

Apparently there is some consistency issues with city names: city of Boston listed twice with names “Boston” and “Boston, MA”. To get a sense of how bad the issue is, let's see what other types of “Boston” entries we have:

```
> db.Boston.aggregate([
  {"$match": {"address.city": {"$exists": 1}}},
  {"$match": {"address.city": {"$regex": "Boston", "$options": "i"}}},
  {"$group": {"_id": "$address.city", "count": {"$sum": 1}}},
  {"$sort": {"count": -1}},
  {"$limit": 20}
])
{u'_id': u'Boston', u'count': 601}
{u'_id': u'Boston, MA', u'count': 9}
{u'_id': u'East Boston', u'count': 8}
{u'_id': u'South Boston', u'count': 2}
{u'_id': u'BOSTON', u'count': 1}
{u'_id': u'boston', u'count': 1}
```

I think all of those records except “East Boston” and “South Boston” must be replaced with “Boston”:

```
> db.Boston.update(  
  {"address.city": {"$exists": 1}, "address.city": {"$in": ["Boston,  
MA", "BOSTON", "boston"]}},  
  {"$set": {"address.city": "Boston"}},  
  {multi: true}  
)  
WriteResult({"nMatched": 11, "nUpserted": 0, "nModified": 11})
```

Of course, similar type of analysis and clean up should be done for all other cities in the dataset.

Other Ideas about the datasets

Quality of Data

There are at least two distinctly different types of validation could be performed on OpenStreetMap data: “object descriptions” and “map coordinates” validation. I think they should be addressed differently, because solutions could be quite different.

By “object description” I mean everything that describes each data point, such as name, address, amenity and other such properties of a data point. “Map coordinates” are geospatial coordinates and properties that describe relationship between objects, such as way intersections, references and such.

Improving “object description” quality

It seems that some of the basic checks, such as implemented in this project, are relatively easy to implement and support on a regular basis. The best way to do it is by having staging database, where all data changes collected, validated and corrected, and only after that goes to production database. The algorithms are rather simple, but may require human intervention. For example, city name posted as “Boston” and “Boston MA” are very similar and could be easily identified as such programmatically. It is also easy to make a suggestion to use name “Boston”. The question is can we apply this suggestion automatically or user confirmation would be preferred? For bulk load mode, confirmation mechanism could be difficult to implement and it may require several iterations.

Another issue I see there is related to data structure. Current xml schema is very flexible and it allows user to add new tags. I think it was extremely beneficial at the beginning of the project, but the more mature the project become, the more restrictions and controls should be applied. For instance, a simple data normalization may significantly improve data quality and guide users to put data elements into correct placements, select value from an existing list instead of typing it and such. Though it may require new tools and API that will support better normalization and could be tricky to implement for bulk load operations.

Improving “map coordinates” quality

Actual maps data on OpenStreetMap is not perfect. There are several applications available that perform checks for “non-closed areas”, ways without tags, missing tags, “floating islands” and such. Some of them are listed in the “Mapping Helpers” sections in the wiki web site:
<http://wiki.openstreetmap.org/wiki/Applications>

It is difficult for me to understand why OpenStreetMap does not collaborate with the authors of those apps – it seems that they have pretty good knowledge and instruments to detect problems. Perhaps detecting a map data inconsistency has been resolved already, but fixing the data is really challenging.

There are also several blog posts on [openstreetmap.org](https://www.openstreetmap.org) web site in the section “User Diaries” - <https://www.openstreetmap.org/diary>, where active users share their experience in maintaining and improving data quality. Many of them developed their own tools and scripts that help automating the tedious process of data fixing. I think an extra effort of analyzing their experience and combining codes into one library, may give a good starting point to a set of instruments for validation and improving data. So OpenStreetMap may become an open source project not only for the data but also for the code, which supports this data.

Ask users

When I see dozens of iOS and Android applications listed on OpenStreetMap and using its data, I see passionate users who know the data and know how to use it. I think it is worth to ask them to help. Perhaps in a form of competition.

For instance, each application developer who wants to use OpenStreetData, encouraged to provide feedback about data inconsistencies found. Feedback could be fully automated and if well structured, may be used to improve data automatically or in a semi-automated fashion. In return, OpenStreetMap may list the best contributors on their web site (which will benefit contributors in promoting their apps). A simple competition, announcing winners every quarter or something like that, may make this process even more fun.

I think, it is difficult to validate map data quality without actually being in that exact intersection that is not well documented. And if we can collect this data directly from the end user, who is actually there, on the ground and reporting an issue, that could be quite beneficial and unique. I think it is somewhat similar to what Waze did with their navigation app – they made UI so easy to use, so users can send traffic reports while driving! Would be nice to have an app which will allow user to report map issues in a similar fashion.

Conclusions

OpenStreetMap data for the Boston area is in a relatively good shape, however some extra cleaning and formatting might be done to improve data quality. For instance, keeping address in a tag “address” makes it difficult to discover when majority of records have all address related information in the tags “addr:*”. Many other attributes could be updated as well. City name, for example, sometimes reported in a several different ways and making it consistent will improve quality of data.

Discovering that a chunk of data (about 5% of the total data entries) came from MassGIS and MassDOT, makes me think that the government of Massachusetts made significant effort to provide geographical data in electronic format and OpenStreetMap users find a way to import this data. Perhaps they can find a way to collaborate and improve each other’s data services even more.

Individual contributions to the OpenStreetMap data are enormous. It is fascinating to see how individual contributors are committed to this project. Would be interesting to check top contributors across different regions – I suspect that we may see there the same people working on the map data in different regions.

References

- https://wiki.openstreetmap.org/wiki/OSM_XML
- https://wiki.openstreetmap.org/wiki/Map_Features
- <https://docs.mongodb.org/manual/reference/mongo-shell/>
- <https://docs.mongodb.org/manual/tutorial/configure-linux-iptables-firewall/>
- <https://docs.mongodb.org/manual/reference/operator/query/regex/>