# OpenStreetMap Data Wrangling with MongoDB

***Ben Mescher***

## Map Area: Milwaukee, Wisconsin

https://www.openstreetmap.org/relation/251075 (https://www.openstreetmap.org/relation/251075)

https://mapzen.com/data/metro-extracts/#milwaukee-wisconsin (https://mapzen.com/data/metro-extracts/#milwaukee-wisconsin)

### Table of Contents ¶

# 1. Overview of the Project and OpenStreetMap Data

For this project, I downloaded a compressed XML file from MapZen (www.mapzen.com/data/metro-extracts/), which provides weekly extracts of preselected metro areas in OpenStreetMap. The data represents approximately 2,500 square miles of OpenStreetMap data in and around the Milwaukee, Wisconsin metro area. OpenStreetMap is a contributor maintained map of the world. OpenStreetMap (OSM) users are encouraged to contribute local knowledge such as store opening hours, number of floors in an apartment building, whether or not a business has a drive thru, etc.

After running validation checks on the data to ensure the XML fit the OSM data model (described here: http://wiki.openstreetmap.org/wiki/OSM_XML (http://wiki.openstreetmap.org/wiki/OSM_XML)), Python cElementTree was used to iteratively parse the data for export to JSON and load into a local MongoDB instance.

### Data Overview

- Map bounds - latitude 42.656 to 43.389, longitude -87.522 to -88.511
- Map Area - approximately 2,500 sq miles (6,500 km2)
- Data - user contributed OpenStreetMap nodes (lat-lon points) and ways (ordered lists of nodes)
- Source - MapZen.com, https://mapzen.com/data/metro-extracts/#milwaukee-wisconsin (https://mapzen.com/data/metro-extracts/#milwaukee-wisconsin)
- Requested - 2/13/2016, OSM API v0.6, generated by osmconvert v0.7T

### File Sizes

- Compressed MapZen XML - **11.9 MB**
- Uncompressed MapZen XML - **166.2 MB**
- Python Exported JSON (Nodes and Ways) - **213.2 MB**
- MongoDB Collection (Nodes and Ways) - **64 MB**

### XML Counts

- Nodes - **738,460**
  - Nested Tag Attributes for Nodes - **48,834**
- Ways - **83,158**
  - Nested Tag Attributes for Ways - **415,334**
  - Nested Nodes listed on Ways - **911,537**

### MongoDB Counts > `with MongoShell Queries`

```
> db.p3.count()
```

- MongoDB Documents (Nodes + Ways) - **821,618**

```
> db.p3.find({'document_tag_type':'node'}).count()
```

- Nodes - **738,460**

```
> db.p3.find({'document_tag_type':'node','k_v_tag_count':0}).count()
```

- Nodes without Tags - **715,267 (97%)**

```
> db.p3.aggregate([{$match:{'document_tag_type':'node','k_v_tag_count':
{$ne:0}}},{$group:{'_id':'total','cnt':{$sum:1},"sum":
{$sum:"$k_v_tag_count"},"avg":{$avg:'$k_v_tag_count'}}}])
```

- Average Tags per Node, Tagged Nodes Only - **2.06**

```
> db.p3.find({'document_tag_type':'way'}).count()
```

- Ways - **83,158**

```
> db.p3.find({'document_tag_type':'way','k_v_tag_count':0}).count()
```

- Ways without Tags - **489 (0.5%)**

```
> db.p3.aggregate([{$match:{'document_tag_type':'way'}},{$group:
{'_id':'total','cnt':{$sum:1},'sum':{$sum:'$k_v_tag_count'},'avg':
{$avg:'$k_v_tag_count'}}}])
```

- Average Tags per Way - **4.95**

## 2. Problems Encountered in the Data

### A. Empty Nodes

The vast majority of nodes in the dataset did not have any nested tags ("key-value" attributes) whatsoever! Of the 738,460 nodes in the MapZen Milwaukee dataset, 97% were "empty nodes" (nodes with zero tags). Rather than exclude the empty nodes from the JSON and Mongo loads altogether, I instead added a field ("k_v_tag_count") to each document so that MongoDB queries could easily identify the number of nested tags for a given node or way. The "k_v_tag_count" field, created during data wrangling, makes it easier to perform analysis on only non-empty nodes (to calculate the average tags per node without skewing our results downward, for instance).

The remaining 3% of nodes had a more normal-seeming average of 2.06 tags per node. As expected, nearly all of the 83,158 ways in the dataset also contained at least one tag. It is only the 97% of nodes that are empty which invite further investigation.

A look at the users contributing the highest number of empty nodes gives hints on their origin. In MongoShell, we can see the top 3 empty node contributors:

```
> db.p3.aggregate([{'$match':{'document_tag_type':'node','k_v_tag_coun
t':0}},{'$group':{'_id':{'user':'$created.user'},'cnt':{'$sum':1}}},
{'$sort':{'cnt':-1}}]).pretty()

{ "_id" : { "user" : "woodpeck_fixbot" }, "cnt" : 180775 }
{ "_id" : { "user" : "ItalianMustache" }, "cnt" : 68445 }
{ "_id" : { "user" : "shuui" }, "cnt" : 47326 }
```

These three users are responsible for over 40% of the empty nodes in our dataset, nearly 300,000 nodes in total! The user OSM wiki pages for both ItalianMustache and shuui suggest they are both avid human contributors to Milwaukee OpenStreetMap (just 11 days prior to this analysis, user shuui added a few dozen city park trees to the downtown Milwaukee OpenStreetMap - some of which are just 50 feet from my front porch door). But the user OSM wiki page for woodpeck_fixbot (http://wiki.openstreetmap.org/wiki/Fixbot (http://wiki.openstreetmap.org/wiki/Fixbot)) reveals this user to be a bot which was tasked with removing unwanted tags from 170 million nodes across the United States between 2009 and 2010. More information on the objective and reasoning of the 2009-2010 woodpeck_fixbot tag cleaning operation can be found here: http://wiki.openstreetmap.org/wiki/TIGER_fixup/node_tags (http://wiki.openstreetmap.org/wiki/TIGER_fixup/node_tags)

## B. cElementTree "Start" and "End" Event IterParsing

When parsing the 166MB MapZen Milwaukee XML data file, I came across an issue with the iterparse incremental XML parser provided in Python cElementTree that does not usually appear when working with smaller OSM XML files, such as in the example code used for the Udacity Lesson 6 Data Wrangling exercises (reference here:
https://docs.python.org/2/library/xml.etree.elementtree.html#xml.etree.ElementTree.iterparse (https://docs.python.org/2/library/xml.etree.elementtree.html#xml.etree.ElementTree.iterparse)).

Essentially, I had to have the cElementTree iterparse iterator stop at both `<start>` *and* `</end>` XML tags throughout the OSM file. I needed the iterator to pause at the `<start>` XML tags to retrieve the XML attributes stored in node and way `<start>` tags (information such as "created user" and "creation timestamp"), but needed to hold off on iterating through the node or way's nested tags until reaching the `</end>` tag of the node or way. The reason for this is that the contents nested within a node or way (i.e. the nested tags) are **not guaranteed** to be found by the iterator if cElementTree.iterparse() has not yet reached the `</end>` tag of the node or way. Waiting until the end of a way element (`</way>`) to iterate over the tags nested under ways **increased the number of tags located by cElementTree iterparse from 403,778 to 415,334 (3%)**. Further illustration and sources on this topic can be found in the iPython Notebook detailing the code used for this project.

## C. Cleaning Tag Values

Originally, I had attempted to clean tag values (nested tag "v" attributes) by replacing characters I believed could cause issues during either JSON export and MongoDb load. Working first with a small sample of ~20,000 nodes, I would print any tag values believed to have potential issues. An interesting finding from this was that one tag value was actually written using the Russian alphabet.

Based on this initial ~20,000 node investigation, I ultimately decided to **not** clean tag values with ":", ";", or " " because the majority of the tag values with these strange characters had legitimate reasons for using them (the most common by far use of ":" was for web addresses for local businesses, as in "https://www.gloriosos.com/ (https://www.gloriosos.com/)")

Lastly, it should be noted that MongoDB handled the loading of strange characters in fields very well. The aforementioned Russian alphabet characters threw Python errors when attempting to print out in iPython Notebook, but loaded into MongoDB without issue: [the Russian alphabet tag value is saved here under the 'bad keys' dictionary]

```
> db.p3.find({'id':'873099231'}).pretty()
{
    "_id" : ObjectId("56c11afd9ca669021466a1b5"),
    "k_v_tag_count" : 5,
    "name" : "Milwaukee",
    "created" : {
        "changeset" : "26972112",
        "user" : "Aleks-Berlin",
        "version" : "4",
        "uid" : "85218",
        "timestamp" : "2014-11-23T12:38:25Z"
    },
    "wikipedia" : "en:Milwaukee",
    "document_tag_type" : "node",
    "pos" : [
        43.0349931,
        -87.922497
    ],
    "ele" : "188",
    "place" : "city",
    "bad_keys" : {
        "name--ru" : "Милуоки",
        "is_in--continent" : "North America",
        "is_in--state" : "Wisconsin"
    },
    "id" : "873099231",
    "population" : "605013"
}
```

## D. Cleaning Tag Keys

Similar to the Lesson 6 example code, I had originally intended to save all tag keys that contained colons, ":", into their own dictionaries, right at the document-level. Tag keys of "addr:street_name" and "addr:house_number" would be saved in a dictionary as {"address": {"street_name":"ABCDEF","house_number":"123456"}}.

A quick exploration of all the colon-containing keys, however, revealed that there were many more of these kinds of keys than I had anticipated. Further investigation into the "subkeys" (by subkey I mean "street_name" in "addr:street_name") was needed to ensure that I shouldn't actually be combining the subkeys from different colon-containing keys (perhaps there is a "gnis:street_name" tag key that also holds street_name information?)

Ultimately, I was able to confirm that the majority (91% of ~13k keys) of the colon-containing keys in the dataset's node tags started with either "addr:" or "gnis:". Furthurmore, I was able to confirm that the subkeys for these two did not overlap- "addr:" held more street and building level address information, while "gnis:" held more county and city level information. For nodes' nested tags, I decided to embed keys that started with "addr:" or "gnis:" into separate "address" and "gnis" document level dictionaries.

I also decided to add document level dictionaries for node tag keys that started with "seaport:". These tags are nautical markers, mostly in and around the Lake Michigan lakefront and Milwaukee marina. While not a large fraction of the total colon-containing keys in our dataset's node tags, I added these as document level dictionaries simply because I live very close to the lakefront and might be interested in examining these nodes in the future.

For ways' nested tags, the majority (98% of ~229k keys) of the colon-containing keys in our dataset belonged to keys that started "tiger:", which I believe are related to the keys that woodpeck_fixbot worked on (see: section 2A above).

For all other colon-containing keys in tags nested under either nodes or ways, I replaced the ":" with "--" and saved these values in a document-level dictionary "bad_keys". Saving the data in this manner allows MongoDB to either ignore these tags completely, or investigate their properties to develop a strategy to integrate them easier (see: Ideas for Future Work).

## 3. Ideas for Future Work

One thing I would like to do is to extract more information from the colon-containing tags. For example, there is a tag with key "census:population" and values "#######;yyyy". These are attached to city and neighborhood nodes, with ####### equal to the census population value and yyyy equal to the year of the census. Extracting these into a more usable format that would be easier to query in Mongo would be a good future project. There are many colon-containing keys in the dataset (currently saved in the "bad_keys" document-level dictionary), and each requires its own investigation and strategy to transform into a more easily queried format.

Another project to look into would be the removal of the "tiger:" tags on the dataset's ways. It would be interesting to research why woodpeck_fixbot removed these tags were from nodes but not from ways.

Finally, as I mentioned previously, prolific Milwaukee OSM contributor shuii added a bunch of trees as nodes onto the park that sits outside of my apartment less than two weeks before my starting this project. I actually realized this only by chance after taking a break from my data wrangling to browse around on the OSM map area outside of my apartment. A neat future project with this database could combine 2D MongoDB geospatial indexes with the timestamps of contributor edits to create visualizations of OSM user contributions by time and place. Would be very cool to see which areas of Milwaukee are worked on the most frequently, and how that has changed over time.