OpenStreetMap Project
Data Wrangling with MongoDB

Alexei Nekrassov

Map Area: St Petersburg, Russia

https://mapzen.com/metro-extracts/

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

1. Problems encountered in your map

After analyzing small subset of the file in text editor and running data.py on it, it became clear that many of the nodes are one-line latitude/longitude entries with no address or any additional information. To simplify further data analysis I created a file without these one-line nodes:

```
grep -v "<node.*\/>$" saint-petersburg_russia.osm > saint-petersburg_russia.osm.1
```

[time to run[1]: 3'53'']

This reduced file size from 733 to 253 MB (from 10,324,754 to 7,080,304 lines)
(These one-liners are quite long, but many lines "inside" multi-line nodes are short – that explains disproportion between size reduction in megabytes and lines.)

Analysis of ***postcodes*** showed some invalid or questionable entries:

```
grep "addr:postcode" saint-petersburg_russia.osm.1|c:/cygwin/bin/sort -u |less
```

For example:

```
"+7 (911) 296-59-96"
"121025"       ← postcodes for areas in Moscow region, not near St-Petersburg
"141980"
"143103"
"172002"       ← postcodes for areas in Tver region, between Moscow and St-Petersburg
"172006"
"172009"
"24/7"/>
"8"/>
"RU"/>
"оф.26"/>
```

These postcodes need to be cleaned. Further analysis of data with 121, 142* and 172* codes is needed after loading into MongoDB – these codes are valid but represent areas hundred miles away from St-Petersburg.

As a part of automated cleanup process, I replaced with an empty string every post code that doesn't conform to 6-digit format that is the Russian standard.

[1] Tests timed on laptop with 4-core i7 2.7GHz CPU, 16GB RAM, 128GB SSD

Then I looked for ***over-abbreviated street names***, e.g., for площадь (square):

```
    egrep "пл\.|Пл\.|ПЛ\." saint-petersburg_russia.osm.1
```

resulted in 25 lines. Closer look at these records showed that simple substitution “пл.” → “площадь” is not correct, because some of these “пл.” actually refer to “платформа” (rail station). Also I discovered that addr:street doesn’t have this particular abbreviation; it is only present in other fields. In general, we would have to come up with more sophisticated algorithm than simple string replacement; most likely we would need to look at other fields’ contents to try and distinguish *square* abbreviation from *rail station* abbreviation.

I did find though other instances of street name abbreviations in addr:street: “пр.” for “проспект” (avenue) and “ул.” for “улица” (street).

As a side note, I was pleasantly surprised to see that Cygwin’s grep works correctly with Cyrillic characters – these two commands produced the same result, showing that “-i” switch works on Cyrillic characters:

```
    egrep "пл\.|Пл\.|ПЛ\." saint-petersburg_russia.osm.1
    grep -i "ПЛ\." saint-petersburg_russia.osm.1
```

Also, early on I verified that I can process Cyrillic characters correctly in Python and MongoDB: I converted small subset of map data to JSON using data.py; extracted one line with Cyrillic characters and loaded it into MongoDB. Then I visually verified that the text is correct when queried from the DB: Python converted WIN-1251 code page text to Unicode UTF-8, and mongoimport correctly loaded Unicode text into the DB. The query via mongo showed original text as expected.

Next step was cleaning and converting the *full* dataset to JSON by running data-clean.py on saint-petersburg_russia.osm. That’s where I’ve ran into issues with Python working on Cyrillic characters – string.find() and replace() failed on non-ASCII characters. I fixed that problem by converting address string and mapping table to Unicode.

Full file conversion to JSON took 6’06” and resulted in 808 MB file. The DB load took 1’41”.

Disk space used by MongoDB went from 80MB (after db.spb.drop() followed by db.repairDatabase() to reclaim space) to 2080MB. In other words, 808MB of data took approximately 2GB of database storage.

2. Overview of the Data

2.1. Dataset statistics collected before loading into MongoDB

File sizes

saint-petersburg_russia.osm	733 MB
saint-petersburg_russia.osm.json	808 MB

Number of users (users.py)
2005
[time to run: 4’19’’]

2.2. Dataset statistics collected via MongoDB

This section contains basic statistics about the dataset and the MongoDB queries used to gather them.

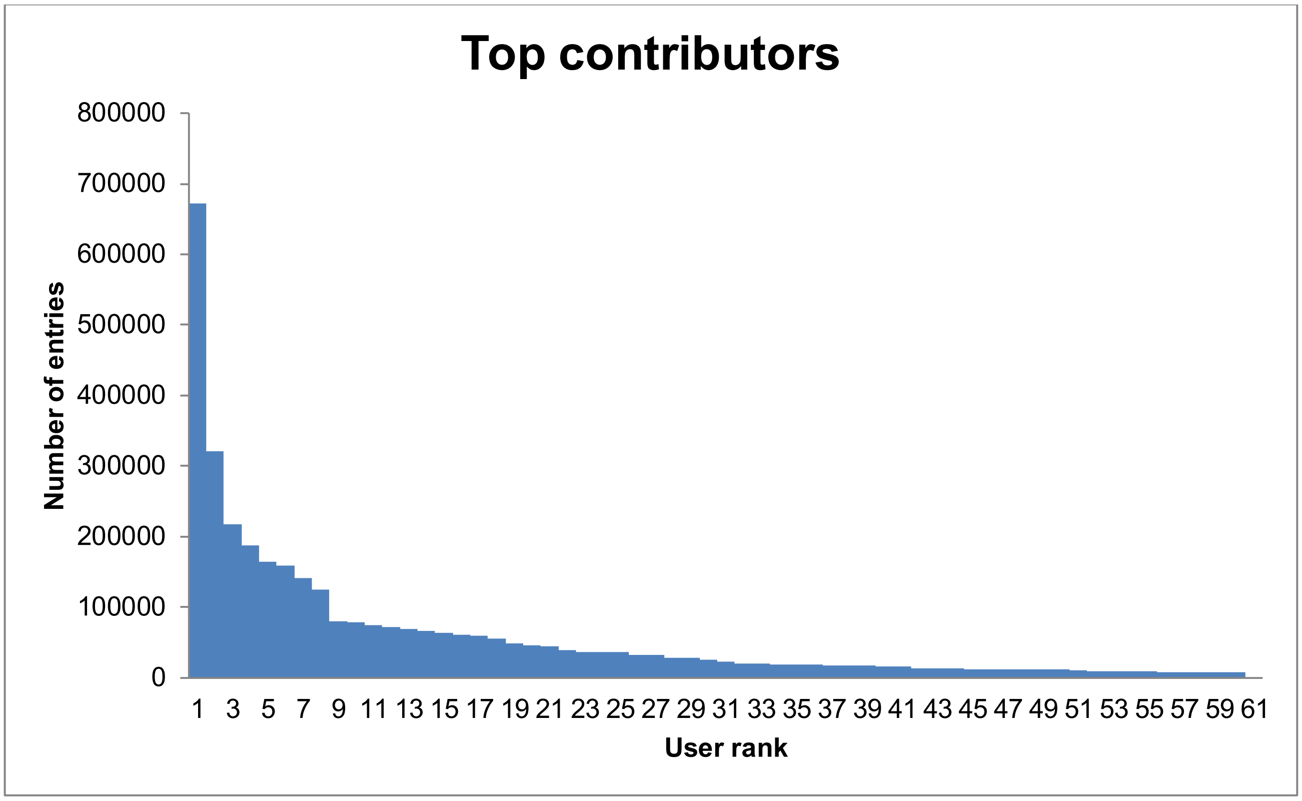
```
# Number of documents
db.spb.find().count()
3,880,541

# Number of nodes
db.spb.find({'type' : 'node'}).count()
3,367,675

# Number of ways
db.spb.find({'type' : 'way'}).count()
512,866

# Top contributing users
db.spb.aggregate([
    {$group:
     {
     _id: "$created.user",
     count: {"$sum":1}
     }
    },
    {$sort: {count : -1}}
])
```

Distribution of top contributing users



To produce this chart I saved output from the previous query in file “top” and ran:

```
awk -F":" '{print $3}' top|sed "s/ //g"|sed "s/\}//g"|grep -v ^$
```

The results were plotted in MS Excel.

It is interesting to note that majority of the entries were created by small number of users.

3. Other ideas about the datasets
Additional data exploration using MongoDB queries

During preliminary analysis we noticed that post code data has some issues. We can analyze this further with the following queries:

```
# Post codes:
# Entries with post code field
db.spb.find({'address.postcode' : {'$exists' : 1}}).count()
11475

# Entries with existing but empty post code field
db.spb.find({'address.postcode' : {'$eq' : ''}}).count()
6
```

```
# Entries with post codes outside of St Petersburg area
# (any post code that doesn’t start with '18' or '19'
db.spb.find({'address.postcode' : {$regex : '^1[^89]'}}).count()
66
```

The data produced by this last query (the data itself can be seen after removal of “count()” call) need to be analyzed further to decide whether it needs to be removed from our dataset.

3.1 Processing international map sets

Data cleanup is locale-specific. For example, street names mapping will be different in different languages. Also, post code format varies between countries. The cleanup is manageable for one locale at a time (e.g., when data only contains US addresses, or only Russia addresses), but would be significantly more complicated if we have to deal with multiple locales at once (e.g., North America with different post codes standards in USA and Canada, and different languages in Mexico and north of it).

One possible way to deal with multi-locale map sets is to use country field to drive checks and conversions. For example when dealing with North American map set and cleaning post codes we can define three formats of post codes:

```
postcode_ca = ...   # define regex for Canada post codes
postcode_us = re.compile(r'^\d{5}(?:[-\s]\d{4})?$')
postcode_mx = ...   # define regex for Mexico post codes
```

The situation is complicated by use of different fields to specify country: addr:country or is_in:country, and potentially abbreviated or full names of the country.

During the data cleanup phase we cannot be sure that the country will be specified before post code in the document. Therefore we need to first extract country and post code, and then after all tags are processed – perform the post code check:

```
# process 2nd level tags
for child in element:
  ...
  # determine country
  if key == “is_in:country” or key == “addr:country”:
    country = val
  ...
  # determine post code
  if key == "addr:postcode"
    zip = val

# after the loop through all tags:

# determine country-specific post code format
if country == “USA”:
  postcode = postcode_us
```

```
elif country == “CA”
  postcode = postcode_ca
elif country == “Mexico”
  postcode = postcode_mx

# remove invalid post codes
if not postcode.match(zip):
    zip = ''
```

Conclusion

Cygwin, Python and MongoDB have good built-in support for international character sets, although the level of support is somewhat uneven – simple conversion worked with Cyrillic text with no extra coding, but more involved tasks like find() and replace() required explicit conversion to Unicode; exact nature of required conversions was not immediately apparent, and took some work to get right.

Data load and queries in MongoDB are very fast – comparable or faster than Unix commands like grep and sort on plain text file; and MongoDB of course provides much more sophisticated query mechanism than what can be achieved by Unix tools. This speed comes at a price of significant storage utilization: in our example the data took 2.5 times more space in the DB than in JSON file.

While basic data cleanup can be done with simple string replacement and such, some of the data cleanup tasks are not trivial. In our example we saw some relatively small number of addresses far outside of St Petersburg area. Technically it is easy to filter out such addresses, but the question arises: “Why these addresses for two regions close to Moscow got included in St Petersburg dataset?” This kind of questions requires further investigation and analysis.