P3 – Wrangle OpenStreetMap Data

Map Area – Toronto, Ontario, Canada

https://www.openstreetmap.org/relation/324211

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

1. Initial map

```
{'minlat': '43.6234' S, 'maxlon': '-79.2815' E, 'minlon': '-79.5681' W,
'maxlat': '43.7451' N}

C:\Mathlab\CV\Courses\DAND\P3 - Data Wrangling with MongoDB\P3
Project\OSM Data\toronto_canada.osm>dir map_toronto2
 Volume in drive [redacted]
 Volume Serial Number [redacted]

 Directory of C:\Mathlab\CV\Courses\DAND\P3 - Data Wrangling with
MongoDB\P3 Project\OSM Data\toronto_canada.osm

02/22/2016  04:46 PM       161,971,337 map_toronto2
               1 File(s)    161,971,337 bytes
               0 Dir(s)  676,885,475,328 bytes free
```

The file size for map_toronto2.osm is 158MB
Note: See "Load Database" section for the document file size in MongoDB.

2. Map Statistics (Basic)

To get a comprehensive assessment of any problems with the map, the major top level tags <node> and <way> as well as child tag <tag> were treated similar to columns in a table. That is any possible statistics and visualization were done to discover any anomalies and opportunities for auditing.

Number Types of Tags

Tag	Count
bounds	1
meta	1
nd	718,197
node	612,894
note	1
osm	1
relation	1,330

tag	783,394
way	111,687
total	2,265,182

XML Structure

```
{'bounds': set(['None']),
 'member': set(['None']),
 'meta': set(['None']),
 'nd': set(['None']),
 'node': set(['None', 'tag']),
 'note': set(['None']),
 'osm': set(['bounds', 'meta', 'node', 'note', 'relation', 'way']),
 'relation': set(['member', 'tag']),
 'tag': set(['None']),
 'way': set(['nd', 'tag'])}
```

Tag Attributes

```
{'bounds': set(['maxlat', 'maxlon', 'minlat', 'minlon']),
 'member': set(['ref', 'role', 'type']),
 'meta': set(['osm_base']),
 'nd': set(['ref']),
 'node': set(['changeset',
             'id',
             'lat',
             'lon',
             'timestamp',
             'uid',
             'user',
             'version']),
 'note': set([]),
 'osm': set(['generator', 'version']),
 'relation': set(['changeset', 'id', 'timestamp', 'uid', 'user',
'version']),
 'tag': set(['k', 'v']),
 'way': set(['changeset', 'id', 'timestamp', 'uid', 'user',
'version'])}
```

In an OSM file all attribute datatypes are strings. However in some instances datatypes need to be change and validated as other datatypes. In this case the timestamp attribute will be validated and converted to timestamp type, and lat and lon will be validated and converted to floats. In addition the id attribute in a node, way or relation should be unique.

3. Problems Encountered in the Map

Latitude and Longitude

All lat and lon values were valid as data types. However 11,901 coordinates are out of range.

One option is to skip these nodes, but the decision was made to do a closer inspection of the coordinates.
The difference in geocode values are very small and hence the coordinate error could be due to the application or method used to obtain the latitude and longitude.
On even closer inspection three types of nodes were found:

- Empty <node> tags (child tags) that may have been leftover when a way was deleted, accidentally created by clicking on the map,or currently in the process of being created. Due to the fact that lat_diff and long_diff was small these nodes were retained.
- Traffic related nodes. E.g. highways, crossings, train stations, traffic lights, stop signs etc. These were retained as the difference between their lat/lon values and the bounding box were very small.
- Nodes created by software JOSM or CanVec6.0 – NRCan (retained).

<u><tag> k, v attribute pairs</u>
These are not very descriptive from observation, hence the type of keys used in this tag is separated out for further details.
These were checked for two reasons:

- to determine the best JSON structure to adopt based on the key (k) attribute and k values with multiple ":". k atrributes were stored in a csv file for further analysis
- to check the data types stored in the value (v) attribute. Two types of data are stored
    i. single value
    ii. multiple values separated by ';' (list)

<u>Street name addresses – addr:street</u>
Results of invalid address endings.
The ending associated with the address field was checked. This was done to determine valid and invalid addresses.

Problem	**Solution**
Blvd, Ave/Ave., E/E., W/W., Rd/Rd., Grv, Hts	Change to correct word using mapping directory
5700	100 King St West, Suite 5700. Manual edit o addr:suite = 5700 o adder:housenumber = 100 o addr: street = King Street West
Callowhill Drive; Farley Crescent	Valid intersection. Change to list [Calloway Drive, Farley Crescent]
Dixon, Dovercourt, Dundas, Italia, Jarvis, Millway, Terace, Tremont, Vitoway	Dixon → Kipling Avenue; Dixon Road. Edit manually Dovercourt → Dovercourt Road Dundas. <way>. Add to expected list Italia. Via Italia is correct Jarvis → Jarvis Street Terace → Terrace

	Vitoway → Vito Way. Consistency with google maps. Fix manually. Tremont. <way>. Add to expected list Chole Millway → Cole Millway. Fix via code
E/E.: e.g. Queen St E, St.Clair Ave E, King St E, Front St East	E.g. change to Queen Street East. Need to write code to check both last and second last words in address.
STREET, JARVIS	For all addr:street values change to first letter capitalize
West	E.g. adelaide Street West, dundas Street West – see STREET, E/E.
avenue, street	Ryerson avenue and Dundas street – see STREET. Change manually

Results of capitalizing first letter of each word in street address

1st/ 4	1st Avenue and 4 Oaks Gate change to First and Four. Fix via code
983, 519, 211	Store number in addr:housenumber. Manual edit ○ 983 Queen Street East ○ 519 Church Street ○ 211 Queen Street East
1st Floor	Lawrence Avenue West, 1st Floor. Place 1st Floor in addr:floor. Manual edit

4. Prepare for Database Load

Data Cleaning Plan

i. Corrections
    a. Manually edit smaller corrections
    b. Automate editing for all others
ii. Update expected list with additional valid addresses
iii. Update mapping dictionary with additional abbreviations manually
iv. Edit code to check both the last and second last word for each street address
v. Standardized St, St. and Saint to St at the beginning of all street names
vi. Remove apostrophe from addr:street (node) and name (way) attributes
vii. Special processing for name attribute
    - Attribute "name" is used in both node and ways subtags
        - node – used similar to address if "highway" or "guidepost" attributes are also present. Otherwise used to give the designation of a place. Change name into list
        - way – used as an address if "highway" attribute is also present. Change "name" into list
    a. Only edit name attribute if "highway" or "guidepost" attributes exists, but no "addr:street" attribute exists

    b. Also check if attribute “name” or “addr:street” has multiple street names separated by “,;&/” and store as a list.

viii. Street names with multiple addresses separated by ‘;’ convert to a list

ix. Edit remaining addresses manually

x. Replace city of Toronto with Toronto. addr:city

xi. Add a space after the first 3 characters and capitalize all addr:postcode

'Load Database

Load data using mongoimport.

```
mongoimport /d osm /c toronto /u username /p password /file
map_toronto2.json"
```

724581 documents imported. The size of the toronto collection in the osm database is 224MB as shown below..

```
> db.toronto.stats({"scale": 1048576})
{
        "ns" : "osm.toronto",
        "count" : 724581,
        "size" : 224,
        "avgObjSize" : 324,
        "numExtents" : 14,
        "storageSize" : 320,
        "lastExtentSize" : 88.296875,
        "paddingFactor" : 1,
        "paddingFactorNote" : "paddingFactor is unused and unmaintained
in 3.0.
It remains hard coded to 1.0 for compatibility only.",
        "userFlags" : 1,
        "capped" : false,
        "nindexes" : 2,
        "totalIndexSize" : 37,
        "indexSizes" : {
                "_id_" : 22,
                "pos_2d" : 15
        },
        "ok" : 1
}
```

5. Overview of the Data

1. Number of documents

```
db.toronto.find().count()
724581
```

2. Number of nodes

```
db.toronto.find({"type": "node"}).count()
612651
```

3. Number of ways

```
db.toronto.find({"type": "way"}).count()
111657
```

4. Number of unique users

```
db.toronto.distinct("created.user").length
742
```

5. Top 10 users

```
db.toronto.aggregate([{"$group":{"_id":
"$created.user", "count": {"$sum": 1} }}, {"$sort":
{"count": -1},{"$limit": 10}])


[{u'_id': u'andrewpmk', u'count': 579106},
 {u'_id': u'Kevo', u'count': 56808},
 {u'_id': u'andrewpmk_imports', u'count': 14363},
 {u'_id': u'TristanA', u'count': 8498},
 {u'_id': u'Bootprint', u'count': 6651},
 {u'_id': u'rw__', u'count': 3318},
 {u'_id': u'ansis', u'count': 3115},
 {u'_id': u'Nate_Wessel', u'count': 3022},
 {u'_id': u'Nate_Wessel (consulting)', u'count': 2953},
 {u'_id': u'Shrinks99', u'count': 2831}]
```

User andrewpmk has done the bulk of the work with 80% of the edits

6. Top 10 types of cuisine

```
db.toronto.aggregate([{"$match":{"cuisine":{"$exists":
1}}}, {"$group": {"_id": "$cuisine", "count": {"$sum":
1}}}, {"$sort": {"count": -1}}, {"$limit": 10}])

[{u'_id': u'coffee_shop', u'count': 397},
 {u'_id': u'pizza', u'count': 171},
 {u'_id': u'sandwich', u'count': 158},
 {u'_id': u'burger', u'count': 119},
 {u'_id': u'japanese', u'count': 49},
 {u'_id': u'chinese', u'count': 41},
 {u'_id': u'sushi', u'count': 37},
 {u'_id': u'italian', u'count': 34},
 {u'_id': u'ice_cream', u'count': 30},
 {u'_id': u'thai', u'count': 28}]
```

Coffee shops rule in Toronto.

7. Top 10 religions by church

```
db.toronto.aggregate([{"$match":{"amenity":
"place_of_worship"}}, {"$group": {"_id": "$religion",
"count": {"$sum": 1}}}, {"$sort": {"count": -1}},
{"$limit": 10}])
```

```
[{u'_id': u'christian', u'count': 414},
 {u'_id': None, u'count': 28},
 {u'_id': u'jewish', u'count': 21},
 {u'_id': u'muslim', u'count': 8},
 {u'_id': u'buddhist', u'count': 3},
 {u'_id': u'sikh', u'count': 2},
 {u'_id': u'hindu', u'count': 1},
 {u'_id': u'bahai', u'count': 1},
 {u'_id': u'scientologist', u'count': 1},
 {u'_id': u'eckankar', u'count': 1}]
```

8. Of these Christian churches which are the 10 most popular types

```
db.toronto.aggregate([{"$match": {"amenity":
"place_of_worship", "religion": "christian",
"denomination": {"$exists": 1}}}, {"$group":
{"_id": "$denomination", "count": {"$sum": 1}}},
{"$sort": {"count": -1}}, {"$limit": 10}])
 [{u'_id': u'catholic', u'count': 46},
  {u'_id': u'united', u'count': 40},
  {u'_id': u'anglican', u'count': 37},
  {u'_id': u'presbyterian', u'count': 30},
  {u'_id': u'baptist', u'count': 30},
  {u'_id': u'jehovahs_witness', u'count': 15},
  {u'_id': u'roman_catholic', u'count': 15},
  {u'_id': u'lutheran', u'count': 10},
  {u'_id': u'orthodox', u'count': 9},
  {u'_id': u'gospel', u'count': 7}]
```

9. Top 10 Building types

```
db.toronto.aggregate([{"$match": {"building":
{"$exists": 1}}}, {"$group":
{"_id": "$building", "count": {"$sum": 1}}},
{"$sort": {"count": -1}}, {"$limit": 10}])
```

```
 [{u'_id': u'yes', u'count': 13270},
  {u'_id': u'house', u'count': 5681},
  {u'_id': u'apartments', u'count': 2701},
  {u'_id': u'retail', u'count': 1276},
  {u'_id': u'residential', u'count': 893},
  {u'_id': u'garage', u'count': 388},
  {u'_id': u'office', u'count': 387},
  {u'_id': u'school', u'count': 361},
  {u'_id': u'industrial', u'count': 318},
  {u'_id': u'church', u'count': 260}]
```

Documents with “building” = “yes” had insufficient data to draw any conclusions about building type. This is where APIs offered by other location/map services will be useful. The queries can be done to determine the type of buildings at these locations based on latitude and longitude or street address. This is the only way to “clean” this type of data.
While an attempt was made to do so programmatically using Google maps free version of the API, there were too many documents that required editing to complete the project on time.

10. Top 10 Land uses

```
db.toronto.aggregate([{"$match": {"building":
{"$exists": 1}}}, {"$group":
{"_id": "$building", "count": {"$sum": 1}}},
{"$sort": {"count": -1}}, {"$limit": 10}])
```

```
 [{u'_id': u'residential', u'count': 2318},
  {u'_id': u'grass', u'count': 481},
  {u'_id': u'retail', u'count': 377},
  {u'_id': u'construction', u'count': 132},
  {u'_id': u'industrial', u'count': 67},
  {u'_id': u'recreation_ground', u'count': 50},
  {u'_id': u'brownfield', u'count': 48},
  {u'_id': u'commercial', u'count': 38},
{u'_id': u'cemetery', u'count': 29},
{u'_id': u'reservoir', u'count': 14}]
```

Likewise landuse can be added to any node or way using location/map API services to update the collection with a more accurate count.

The data is useful to give an overview of the city, but it really comes alive when location is taken into account. Toronto is known as the city of neighborhoods. A separate database was created called “neighb”.

- Fields came from geonames.org and contained 102 documents.
- Fields include – district (e.g. Downtown Toronto), place name (e.g. Church and Wellesley), postal code, latitude and longitude of all neighborhoods in Toronto (and Mississauga), province and country
- Note not every node/way has a postal code attached. 228, 622 documents in the toronto collection have no postal code information. However this does not affect any location searches at all as proximity of any neighborhood is done using longitude/latitude.
- To search these neighborhoods a geospatial index was created on “pos”.

11. <u>Create geospatial index</u>
    The documents are using legacy coordinates, hence a 2d index was created.

    Create index on nodes with pos (longitude/latitude) field

    ```
    db.toronto.createIndex({ "pos": "2d"} )
    ```

    Create index on neighb collection with pos (longitude/latitude) field. Note pos here is assumed to be the center of the neighborhood, therefore any points close to that neighborhood are given within a radius or distance.

    ```
    db.neighb.createIndex({ "pos": "2d"} )
    ```

12. <u>Find 10 restaurants within a 2km of a neighborhood</u>

```
def query_db(db):
   neighborhood = db.neighb.find_one({ "pos": [ -
     79.4521, 43.6469] }, {"_id":0, "pos": 1, "name":
     1})
```

```
print "Neighborhood is"
pprint.pprint(neighborhood["name"])

return [doc for doc in db.toronto.find(
{ "pos": { "$geoWithin": { "$centerSphere":
[neighborhood["pos"], 2 / 6378.1] } }, "name":
{"$exists": 1}, "amenity": {"$in": ["restaurant",
"cafe", "fast_food"]}}, {"_id": 0, "amenity": 1,
"name": 1} ).limit(10)]

if __name__ == "__main__":
    db = get_db('osm')
    result = query_db(db)
    print "Printing the first 10 results:"
    pprint.pprint(result)
```

```
Neighborhood is
[u'Parkdale ', u'Roncesvalles Village']

Printing the first 10 results:
[{u'amenity': u'fast_food', u'name': u'Pizza Pizza'},
 {u'amenity': u'fast_food', u'name': u'Subway'},
 {u'amenity': u'restaurant', u'name': u'Pavao'},
 {u'amenity': u'fast_food', u'name': u"McDonald's"},
 {u'amenity': u'cafe', u'name': u'Coffee Time'},
 {u'amenity': u'fast_food', u'name': u'Pizza Pizza'},
 {u'amenity': u'fast_food', u'name': u"McDonald's"},
 {u'amenity': u'cafe', u'name': u'Starbucks Coffee'},
 {u'amenity': u'fast_food', u'name': u"McDonald's"},
 {u'amenity': u'cafe', u'name': u'Coffee Time'}]
```

Within 2km we have 3 McDonald's, 2 Pizza Pizza's and 3 coffee shops.

This gives any 10 restaurants within 2km of the center of the Roncesvalles Village and Parkdale neighbourhoods. It does not however gives the 10 closest restaurants. For that see no. 14.

13. Find the 10 closest restaurants within a 2km distance of a point in a neighborhood

```
def query_db(db):
    neighborhood = db.neighb.find_one({ "pos": [ -79.4521,
43.6469] }, {"_id":0, "pos": 1, "name": 1})
```

```
    print "Neighborhood is"
    pprint.pprint(neighborhood["name"])

    return [doc for doc in db.toronto.find({ "pos": SON([(
"$nearSphere", neighborhood["pos"]), ("$maxDistance", 2 / 6378.1
)]), "name": {"$exists": 1}, "amenity": {"$in": ["restaurant",
"cafe", "fast_food"]}}, {"_id": 0, "amenity": 1, "name":
1}).limit(10)]
```

```
Neighborhood is
[u'Parkdale ', u'Roncesvalles Village']

Printing the first 10 results:
[{u'amenity': u'restaurant', u'name': u'Barque Smokehouse'},
 {u'amenity': u'restaurant', u'name': u'Defina Pizzeria'},
 {u'amenity': u'cafe', u'name': u"Timothy's World Coffee"},
 {u'amenity': u'cafe', u'name': u'Lit Espresso Bar'},
 {u'amenity': u'restaurant', u'name': u'The Friendly Thai'},
 {u'amenity': u'restaurant', u'name': u'Fat Cat Wine Bar'},
 {u'amenity': u'cafe', u'name': u'Alternative Grounds'},
_{u'amenity': u'restaurant', u'name': u'Cafe Polonez'},
 {u'amenity': u'fast_food', u'name': u'Subway'},
 {u'amenity': u'cafe', u'name': u"Granowska's"}]
```

The lists of restaurants have completely changed as we now have the 10 closest restaurants to the point [-79.4521, 43.6469].

14. Find the number of coffee shops within 3km of a point (neighborhood center)

```
def query_db(db):
    neighborhood = db.neighb.find_one({ "pos": [ -79.4521,
43.6469] }, {"_id":0, "pos": 1, "name": 1})
    print "Neighborhood is"
    pprint.pprint(neighborhood["name"])
    # return [doc for doc in
    return db.toronto.find({ "pos": { "$geoWithin": {
"$centerSphere": [neighborhood["pos"], 3 / 6378.1] } }, "name":
{"$exists": 1}, "$or": [{"amenity": "cafe"}, {"cuisine":
"coffe_shop"}]}, {"_id": 0, "name": 1} ).count()
```

```
Neighborhood is
[u'Parkdale ', u'Roncesvalles Village']

No of coffee shops:
72
```

15. Find the number of Tim Hortons and Starbucks within 1km of a point (neighborhood center)

```
def query_db(db):
    neighborhood = db.neighb.find_one({ "pos": [ -79.386,
43.6564] }, {"_id":0, "pos": 1, "name": 1})

    print "Neighborhood is"
    pprint.pprint(neighborhood["name"])

      return db.toronto.find({ "pos": { "$geoWithin": {
  "$centerSphere": [neighborhood["pos"], 1 / 6378.1] } }, "name":
  {"$exists": 1}, "$or": [ { "name": {"$regex": "^Tim Horton",
  "$options": "i"}}, { "name": {"$regex": "^Starbucks",
  "$options": "i"}} ], "amenity": "cafe"}, {"_id": 0, "name": 1}
  ).count()
```

```
Neighborhood is
[u'Central Bay Street']

No of coffee shops:
65
```

Query for amenity = café and name beginning with Tim Horton and Starbucks.

16. No of Tim Hortons and Starbucks coffee shops in 1km distance

```
def make_pipeline():

    pipeline = [{"$geoNear": { "near": [ -79.386, 43.6564 ],
                 "maxDistance": 1 / 6378.1,
                 "query": {
                     "name": {"$exists": 1},
                     "$or": [ { "name": {"$regex": "^Tim
Horton", "$options": "i"}},
                              { "name": {"$regex":
"^Starbucks", "$options": "i"}} ],
                     "amenity": "cafe"},
                 "distanceField": "distance", "includeLocs":
"loc", "spherical": "true"
                }},

                {"$group":
                     {"_id": "$name",
                      "count": {"$sum": 1}
                      }
```

```
                    },
                   {"$sort": {"count": -1}
                   }

                   ]

    return pipeline

def aggregate(db, pipeline):
    print "Neighborhood is"
    pprint.pprint(neighborhood["name"])
    return [doc for doc in db.toronto.aggregate(pipeline)]

if __name__ == "__main__":
    db = get_db('osm')
    neighborhood = db.neighb.find_one({ "pos": [ -79.386,
43.6564] }, {"_id":0, "pos": 1, "name": 1})
    pipeline = make_pipeline()
    result = aggregate(db, pipeline)

    print "No of coffee shops:"
    pprint.pprint(result)
```

```
Neighborhood is
[u'Central Bay Street']

No of coffee shops:
[{u'_id': u'Tim Hortons', u'count': 36},

 {u'_id': u'Starbucks Coffee', u'count': 29}]
```

6. Other Ideas about the Dataset

- Each individual node and way can be verified using the vast array of location/geocoding APIs available. The only constraints are for those that use daily/monthly rate limits and/or costs.
- Get bounding box or polygon for neighborhoods from Bing/Google map APIs. Use this information to define neighborhoods as shapes and not as points. This makes for more effective query of places within a defined area.
- Nodes and ways without postal codes are not a huge obstacle due the ability to query using geospatial indexes, standard query operators and $geoNear in the aggregation framework.
- Geospatial queries opens a whole new world of building apps or solving data science problems using OSM. This is especially true when using open data provided by the various governments across the globe and the many datasets made public by Airbnb, Uber and the New York City taxi data. Some sample applications/problems include
    - Investigating the airbnb data to see who is most likely to use it and where (i.e. proximity to certain amenities)

- Using the New York City taxi data to answer many questions. Where people typically go after work (5pm)?
- Using the Uber data to do the same as above
- Using get Toronto Transit Commission (TTC) real time next vehicle arrival and by extension the best routes to take during rush hour (real time)
- Real estate – check neighborhood for hospitals, schools, public transportation, and other popular amenities. Combine this with rent and housing costs and demographic info.
- Map wifi spots in the city as open/"closed".
- Elections, voters and results. Combine with census data to track how voters have changed over time in voting districts.
- Transportation – accidents. This can be done by neighborhood, intersection, proximity to schools etc. and any similarities can be highlighted

In short there are many opportunities for story telling using spatial data due to the large amounts of data already in the public domain.

The biggest challenge is in cleaning the data. Does a perfectly clean dataset exist? I doubt it. What matters is that the data is audited thoroughly and is clean enough to conduct a valid analysis.