Data Wrangling with MongoDB (Final Project)

Author	OpenStreetMap Area
Thomas Wirtz	Minneapolis/St. Paul, MN United States

1. Problems Encountered in my Map

Auditing and Cleaning Overview

Auditing individual fields in the dataset (cities, streets, postcodes, phones, amenities, cuisines, and religious denominations) uncovered the following general issues:

- Obvious typos
- Extraneous blanks and other invalid characters
- Inconsistencies with capitalizations (e.g., 'Elk River' and 'elk river')
- Inconsistencies with abbreviations (e.g., 'Saint Paul', 'St. Paul', and 'St Paul')
- Inconsistencies with formatting (e.g., parentheses and separators in phone numbers)
- Misplaced data (e.g., cities with appended state names, postcodes containing house numbers, streets containing full addresses)
- Inconsistencies with street types: (e.g., Street, St., St)
- Missing street types (e.g., 'Arkwright' instead of 'Arkwright St')
- Inconsistencies with ordinals in streets (e.g., 'First' and '1st')
- Inconsistencies with compass point identifiers in streets (e.g., 'Northeast', 'N.E.', 'N. E.', and 'NE') and their placement at the beginning or end of the string
- Missing digits in phone numbers
- Vanity names in phone numbers (e.g., '952-746-SNAP')
- Multiple phone numbers in phone fields

My project code addressed each of these items in its cleanup logic. To standardize street values, for example, I employed logic to:

- perform general string cleanup and fix capitalization issues
- update street types from a canonical set (abbreviations without periods): Ave, Blvd, etc.
- update compass points from a canonical set: N, S, E, W, NE, NW, SE, and SW
- move compass points from the beginning to the end of strings
- update ordinal names to their shortened forms (e.g., change First to 1st)
- update all forms of Saint to St (abbreviated without the period)

Having attended University of Minnesota, my accumulated "domain knowledge" of the Minneapolis and St. Paul areas help in resolving a few of the observed inconsistencies in the data (e.g., changing "Inver Grove" to "Inver Grove Heights"). Checking Google Maps, visiting

websites of local businesses, and (in a couple of cases) reading street signs on Google Street View all helped to resolve other identified inconsistencies.

Transformation Example

Let's examine an untransformed way element from the input map area file (which describes a police station in Apple Valley, MN), as well as its representation as a cleaned and transformed JSON document. This is an example of a closed way (whose first and last node references are the same). The node list describes a counterclockwise, closed path around the building's perimeter. Here's a link to the way on the OSM site:
https://www.openstreetmap.org/way/31064153

Untransformed Element (Input)

```
<way id="31064153" version="3"
timestamp="2010-09-03T02:02:44Z"
changeset="5667306" uid="219968"
user="DCJoeS">
  <nd ref="345369052"/>
  <nd ref="345369055"/>
  <nd ref="345369057"/>
  <nd ref="345379092"/>
  <nd ref="345379095"/>
  <nd ref="345379099"/>
  <nd ref="345379102"/>
  <nd ref="345379107"/>
  <nd ref="345379111"/>
  <nd ref="345379114"/>
  <nd ref="345379118"/>
  <nd ref="345379121"/>
  <nd ref="345379124"/>
  <nd ref="345379127"/>
  <nd ref="345379130"/>
  <nd ref="345379133"/>
  <nd ref="345379135"/>
  <nd ref="345379138"/>
  <nd ref="345369052"/>
  <tag k="amenity" v="police"/>
  <tag k="building" v="yes"/>
  <tag k="addr:city" v="apple valley"/>
  <tag k="addr:street" v="147th Street West"/>
  <tag k="addr:postcode" v="55124"/>
  <tag k="addr:housenumber" v="7100"/>
</way>
```

Transformed Document (Output)

```
{
  "building": "yes",
  "amenity": "police",
  "node_refs": [
    "345369052",
    "345369055",
    "345369057",
    "345379092",
    "345379095",
    "345379099",
    "345379102",
    "345379107",
    "345379111",
    "345379114",
    "345379118",
    "345379121",
    "345379124",
    "345379127",
    "345379130",
    "345379133",
    "345379135",
    "345379138",
    "345369052"
  ],
  "created": {
        "changeset": "5667306",
        "user": "DCJoeS",
        "version": "3",
        "uid": "219968",
        "timestamp":
"2010-09-03T02:02:44Z"
  },
  "osm_type": "way",
  "address": {
        "city": "Apple Valley",
        "street": "147th St W",
        "housenumber": "7100",
        "postcode": "55124"
  },
  "id": "31064153"
}
```

The city and street values are highlighted in both the untransformed (dirty) and transformed (clean) data in the above table. In the cleaning logic of my project code, the following programmatic steps caused string updates to occur:

- Invoked the *capwords* function in the string module to update “apple valley” to “Apple Valley”
- Invoked the *compile* and *sub* functions in the re module to perform pattern matching and string substitution, updating the type value of “Street” to its canonical value of “St”
- Invoked the re module methods again to update the compass point value of “West” to its canonical value of “W”

Missing and Faulty Data

I also discovered (unsurprisingly) a fair amount of incomplete data (e.g., my dataset contained 34 nodes whose addresses have cities but not streets). While I did not attempt to fix these issues, I do suggest a possible solution in Section 3.

The following table provides two examples of nodes with missing street types. In the first example, “Arkwright” should be “Arkwright St”. In the second example, “Maryland” should be “Maryland Ave E”. I discovered these cases by observation and resolved them manually. My code implements the fixes programmatically. I should note here that these are also great examples of incomplete address data (e.g., each is missing city and state values).

Missing Street Type Example #1	**Missing Street Type Example #2**

`<node id="1831043797" lat="44.9862267" lon="-93.0839601" version="1" timestamp="2012-07-19T15:10:49Z" changeset="12333562" uid="745262" user="Pheng Yang">`
`  <tag k="building" v="yes"/>`
`  <tag k="addr:street" v="Arkwright"/>`
`  <tag k="addr:postcode" v="55130"/>`
`  <tag k="addr:housenumber" v="1495"/>`
`</node>`
	`<node id="1565910628" lat="44.9773068" lon="-93.0626045" version="1" timestamp="2011-12-28T19:31:29Z" changeset="10229457" uid="575943" user="PaulPeine">`
	`  <tag k="name" v="Golden Harvest Food"/>`
	`  <tag k="shop" v="supermarket"/>`
	`  <tag k="source" v="local knowledge"/>`
	`  <tag k="addr:street" v="Maryland"/>`
	`  <tag k="addr:postcode" v="55106"/>`
	`  <tag k="addr:housenumber" v="900"/>`
	`</node>`

Finally, I discovered 187 faulty ways in the dataset (i.e., ways having only one node reference). Two examples are provided in the table below.

Faulty Way Example #1	**Faulty Way Example #2**

`<way id="18067877" version="4" timestamp="2015-01-03T08:07:30Z" changeset="27880333" uid="33255" user="jaded">`
`  <nd ref="186774276"/>`
	`<way id="18084585" version="7" timestamp="2013-09-04T03:20:44Z" changeset="17664506" uid="502142" user="Mulad">`
	`  <nd ref="1412513847"/>`
	`  <tag k="power" v="line"/>`

```
    <tag k="name" v="Slater Road"/>
    <tag k="highway" v="residential"/>
    <tag k="tiger:cfcc" v="A41"/>
    <tag k="tiger:county" v="Dakota, MN"/>
    <tag k="tiger:reviewed" v="no"/>
    <tag k="tiger:name_base" v="Slater"/>
    <tag k="tiger:name_type" v="Rd"/>
</way>
```

```
    <tag k="cables" v="3"/>
    <tag k="voltage" v="69000"/>
    <tag k="tiger:cfcc" v="C20"/>
    <tag k="tiger:county" v="Goodhue,
MN"/>
  <tag k="tiger:reviewed" v="no"/>
</way>
```

2. Overview of the Data

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

File Sizes

Data File	Description	Uncompressed Size (bytes)
minneapolis-saint-paul_minnesota.osm	OSM metro extract provided by Mapzen (downloaded on May 12th, 2015)	652858628
data.json	Output file of data.py (pretty=True); input file for mongoimport command	955497356

How many total documents?

```
db.nodes.find().count()
```

```
3328978
```

How many nodes and ways?

```
db.nodes.aggregate(
      { "$group" : { "_id" : "$osm_type",
                     "count" : { "$sum" : 1 } } },
      { "$sort" : { "count" : -1 } }
).pretty()
```

```
{ "_id" : "node", "count" : 3019782 }
{ "_id" : "way", "count" : 309196 }
```

How many unique users?

```
db.nodes.aggregate(
  [
     $match :
     {
       "created.uid" : { $exists : 1 }
     }
  },
  {
     $group :
     {
       _id : "unique users",
       unique_uids : { $addToSet : "$created.uid" }
     }
  },
  {
     $unwind : "$unique_uids"
  },
  {
     $group :
     {
       _id : "$_id",
       count : { $sum : 1 }
     }
  }
).pretty()

{ "_id" : "unique users", "count" : 1197 }
```

How many prisons?

```
db.nodes.find(
  [
     amenity : "prison"
  }
).count()
```

3. Other Ideas About the Datasets

Reverse Geocoding

Each of the nodes in the dataset contained latitude and longitude values, but many of those nodes contained incomplete address information (e.g., thirty-four nodes were present with city values but with missing street values). Reverse geocoding could be used to populate missing address information or to validate existing address data.

Several reverse geocoding API providers are available online, offering RESTful services that can be invoked easily from a Python script, a command line curl request, or a web browser. Google is one such provider, offering the reverse geocoding capability from its Geocoding API. As an example, I used the following URL (with my private API key value) to retrieve address information for a node with missing street value, corresponding to “The Boy Scout Store” in Burnsville, MN. The untransformed node and an excerpt from the API invocation results (showing the house number, street, and full formatted address) are shown in the table immediately below the URL.
https://maps.googleapis.com/maps/api/geocode/json?latlng=44.7435722,-93.2730174&key=*API_KEY*

Untransformed Node Element	**Google Geocoding API Output (Excerpt)**
<node id="1992142135" lat="44.7435722" lon="-93.2730174" version="2" timestamp="2012-11-23T00:26:05Z" changeset="13993078" uid="933061" user="PrometheusAvV"> <tag k="name" v="The Boy Scout Store"/> <tag k="shop" v="clothes"/> <tag k="addr:city" v="Burnsville"/> <tag k="addr:state" v="MN"/> <tag k="addr:country" v="US"/> <tag k="addr:postcode" v="55337"/> >	{ "results" : [{ "address_components" : [{ "long_name" : "14250", "short_name" : "14250", "types" : ["street_number"] }, { "long_name" : "Plymouth Avenue", "short_name" : "Plymouth Ave", "types" : ["route"] }, { "long_name" : "Burnsville", "short_name" : "Burnsville", "types" : ["locality", "political"] }, { "long_name" : "Minnesota", "short_name" : "MN", "types" : [

```
"administrative_area_level_1", "political"
]
        },
        {
          "long_name" : "55337",
          "short_name" : "55337",
          "types" : [ "postal_code" ]
        },
        {
          "long_name" : "5785",
          "short_name" : "5785",
          "types" : [ "postal_code_suffix"
]
        }
       ],
       "formatted_address" : "14250
Plymouth Avenue, Burnsville, MN 55337,
USA",
...
   ],
   "status" : "OK"
}
```


Several potential challenges would require ample consideration in order to fully implement this idea. First, it would still be necessary to assess the data quality of the reverse geocoder service provider's results and to perform data wrangling. Second, service providers may place rate limits on the number of free requests (e.g., Google allows a maximum of 2500 requests per 24 hour period using the same API key). Third, performance and throughput may be throttled by the provider (e.g., Google allows no more than 5 requests per second). I did find one provider (Texas A&M University website) that offers a batch request service, which might potentially address some of these concerns.

Additional data exploration using MongoDB queries

Find Vincent Hall (Math building at University of Minnesota):

```
db.nodes.find(
  {
     "name:en" : "Vincent Hall",
     "address.city" : "Minneapolis"
  },
  {
     _id : 0,
     osm_type : 1,
     "name:en" : 1,
     description : 1,
     operator : 1,
     "umn:BuildingCenterXYLongitude" : 1,
```

```
      "umn:BuildingCenterXYLatitude" :1
  }
).pretty()

{
      "name:en" : "Vincent Hall",
      "description" : "Department of Mathematics",
      "osm_type" : "way",
      "umn:BuildingCenterXYLatitude" : "44.97451",
      "operator" : "University of Minnesota",
      "umn:BuildingCenterXYLongitude" : "-93.234684"
}
```

Find the bar with an address nearest the Math building:

```
db.nodes.findOne(
  {
      amenity : "bar",
      address : { $exists : 1 },
      pos:
      {
        $near :
        {
          $geometry: { type: "Point",  coordinates: [ -93.234684,
44.97451 ] },
          $minDistance: 0,
          $maxDistance: 1500
        }
      }
  },
  {
      _id : 0,
      name : 1,
      address : 1,
      pos : 1
  }
)

{
      "name" : "Grumpy's Bar & Grill",
      "pos" : [
                -93.2528319,
```

```
                44.9750485
        ],
        "address" : {
                "city" : "Minneapolis",
                "street" : "Washington Ave S",
                "housenumber" : "1111",
                "postcode" : "55415"
        }
}
```

Which city has the most restaurants serving Chinese food?

```
db.nodes.aggregate(
  {
     $match :
     {
       amenity : { $in : [ "restaurant", "fast_food" ] },
       cuisine : { $regex : "chinese" },
       "address.city" : { $exists : 1 }
     }
  },
  {
     $group :
     {
       _id : "$address.city",
       count : { $sum : 1 }
     }
  },
  {
     $sort : { count : -1 }
  },
  {
     "$limit" : 1
  }
).pretty()

{ "_id" : "Cottage Grove", "count" : 3 }
```

Which two cities have the most restaurants serving the same type of food, excluding American and pizza?

```
db.nodes.aggregate(
  {
      $match :
      {
        amenity : { $in : [ "restaurant", "fast_food" ] },
        cuisine : { $exists : 1, $nin : [ "american", "pizza" ] },
        "address.city" : { $exists : 1 }
      }
  },
  {
      $group :
      {
        _id : { cuisine : "$cuisine", "address.city" :
"$address.city" },
        count : { "$sum" : 1 }
      }
  },
  {
      $sort : { "count" : -1 }
  },
  {
      "$limit" : 2
  }
).pretty()

{
      "_id" : {
                "cuisine" : "mexican",
                "address.city" : "Minneapolis"
      },
      "count" : 9
}
{
      "_id" : {
                "cuisine" : "sandwich",
                "address.city" : "St Paul"
      },
      "count" : 5
}
```