OpenStreetMap Data Case Study - Salt Lake City, Utah

Overview

In this study, I will use Python and MongoDB to investigate the structure of XML data. To illustrate how XML data can be investigated with these tools, I will be generating metadata for an extract from the Open Street Map Project (OSM). From studying the metadata, I hope to find areas of interest for data cleaning.

About the OSM Extract

The area chosen for investigation is my home city of Salt Lake City, Utah. This area was chosen as I have some familiarity with the area's addressing and landmarks.

Salt Lake City addresses its streets using a direction based grid system. At the center of the grid, is the LDS Temple. The address provides a distance as a number of city blocks times 100 and a direction of travel away from the temple. For example, 350 East would be three and a half city blocks East of the LDS Temple. The address portion describing the direction of travel away from the LDS Temple will be referred to as "directional".

The map data was obtained from https://www.openstreetmap.org/. From this site, I performed a search for "Salt Lake City". The first item in the search results was selected as the area of interest. From here, I went to the data's export page. A direct link to this export page has been provided below. Due to the size of the data set, I found it best to utilize the prepared Overpass API source.

Salt Lake City, Utah, United States
https://www.openstreetmap.org/export#map=12/40.7765/-111.9206

Overpass API data source
https://overpass-api.de/api/map?bbox=-112.1155,40.6387,-111.7255,40.9140

The resulting map dataset is approximately 105mb in size.

```
λ  gci map

Directory: C:\Users\mdjen\Documents\GitHub\WGU\C750-MongoDB


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        10/15/2018   5:40 AM      108017568 map
```

Included Scripts

What follows is a list of scripts created to Extract, Transform, Load and Query dataset. These scripts were written and tested with Python 3.7. I have made these scripts are available at https://github.com/mdjenkin1/WGU/tree/master/C750-MongoDB

- element_getter.py: Provides a method of element iteration that serves full elements. The functionality of this script was adopted from course material on Udacity.
- describe_xml.py: Metadata generator for XML data. Extracts information about the structure and content of XML data and transforms it into a JSON compatible structure.
- xml_desc_to_mongo.py: The load portion of the ETL process for XML Metadata generation. The metadata generated by describe_xml.py will be loaded to MongoDB for further inquiry.
- xml_metadata_inquiries.py: A collection of functions for making investigations of xml descriptions.
- map_to_mongo.py: First of two scripts to handle OSM data extraction and loading to mongodb.
- slc_street_cleanup.py: Leverages the extraction and loading functionality of the map_to_mongo.py script while also introducing data transformations to provide more uniform data.

Investigation

Parsing and Collecting XML Metadata

To begin my investigation, I took the stance of someone that has no foreknowledge of OSM data. This approach allowed me to explore the data with a method that can be applied to completely foreign data sets. My first question in this process was to obtain information about how the XML was structured. Before I could answer this question, I needed to have a clear understanding of the XML specification.

XML at its most basic is a structured collection of elements. Therefore, it's natural to structure a description of as a list of element descriptions. With this base structure in mind, I just need a structure for describing elements.

Elements naturally lend themselves to Python dictionaries. They can all have a name, attributes, text and nested elements. Only an element's name is mandatory. The element name is unique by element type. Although elements of the same type do not necessarily have the same number of attributes, nested elements or text. As we are constructing metadata, we're not yet interested in actual values. We are interested in the data types used for values in those aspects. So, our structure should include information on what types of data are used for the aspects of elements of a specific type.

This led to the following data structures.

XML Description structure

```
{
    "elements":[
        {elem1},
        ...,
        {elemN}
    ]
}
```

Element Description Structure

```
{
    "name": string,
```

```
    "attribs": {attrib_name:[types...], ...},
    "nested_elements": [types...],
    "text": [types...]
}
```

Structuring metadata in this form provides an ease of loading to a MongoDB engine. Like a SQL engine, a MongoDB engine provides tools for querying datasets. With MongoDB as a target engine for the data extraction and a plan for data structure; what's left is extracting, transforming and loading (ETL) the data.

The describe_xml.py script was written specifically to generate XML metadata in MongoDB friendly format. It has a companion script, xml_desc_to_mongo.py, to facilitate the loading of XML metadata to a local MongoDB instance.

Investigating OSM XML Metadata

A run of xml_desc_to_mongo.py provided OSM XML metadata generation and loaded it to my local MongoDB engine. It's now possible to start the investigation. So I fired up a MongoDB client and opened the newly populated xml_descriptions database. To begin, I took a quick look at the OSM XML metadata in its entirety.

```
> db.osm_map.find()
{ "_id" : ObjectId("5bd9aed24aa1033ccc473753"), "name" : "note", "attribs" : null,
"nested_elements" : null, "text" : [ "str" ] }
{ "_id" : ObjectId("5bd9aed24aa1033ccc473754"), "name" : "meta", "attribs" : {
"osm_base" : [ "datetime" ] }, "nested_elements" : null, "text" : [ null ] }
{ "_id" : ObjectId("5bd9aed24aa1033ccc473755"), "name" : "bounds", "attribs" : {
"minlat" : [ "float" ], "minlon" : [ "float" ], "maxlat" : [ "float" ], "maxlon" :
[ "float" ] }, "nested_elements" : null, "text" : [ null ] }
{ "_id" : ObjectId("5bd9aed24aa1033ccc473756"), "name" : "node", "attribs" : {
"id" : [ "int" ], "lat" : [ "float" ], "lon" : [ "float" ], "version" : [ "int" ],
"timestamp" : [ "datetime" ], "changeset" : [ "int" ], "uid" : [ "int" ], "user" :
[ "str" ] }, "nested_elements" : [ "tag" ], "text" : [ null, "str" ] }
{ "_id" : ObjectId("5bd9aed24aa1033ccc473757"), "name" : "tag", "attribs" : { "k"
: [ "str" ], "v" : [ "str", "datetime", "float", "int" ] }, "nested_elements" :
null, "text" : [ null ] }
{ "_id" : ObjectId("5bd9aed24aa1033ccc473758"), "name" : "nd", "attribs" : { "ref"
: [ "int" ] }, "nested_elements" : null, "text" : [ null ] }
{ "_id" : ObjectId("5bd9aed24aa1033ccc473759"), "name" : "way", "attribs" : { "id"
: [ "int" ], "version" : [ "int" ], "timestamp" : [ "datetime" ], "changeset" : [
"int" ], "uid" : [ "int" ], "user" : [ "str" ] }, "nested_elements" : [ "tag",
"nd" ], "text" : [ "str" ] }
{ "_id" : ObjectId("5bd9aed24aa1033ccc47375a"), "name" : "member", "attribs" : {
"type" : [ "str" ], "ref" : [ "int" ], "role" : [ "str" ] }, "nested_elements" :
null, "text" : [ null ] }
{ "_id" : ObjectId("5bd9aed24aa1033ccc47375b"), "name" : "relation", "attribs" : {
"id" : [ "int" ], "version" : [ "int" ], "timestamp" : [ "datetime" ], "changeset"
: [ "int" ], "uid" : [ "int" ], "user" : [ "str" ] }, "nested_elements" : [
"member", "tag" ], "text" : [ "str" ] }
{ "_id" : ObjectId("5bd9aed24aa1033ccc47375c"), "name" : "osm", "attribs" : null,
"nested_elements" : null, "text" : [ null ] }
>
```

In my initial scan, I'm looking for any values that don't make sense. For instance, the 'v' attribute of the tag element has a variety of data types. This suggests a type of generic element that describes a variety of data types or an element that has attribute values in need of cleaning. Context suggests the former. Either way, this variety of types for tag 'v' attributes was a deciding factor to investigate the data stored in tag elements.

I also noted, there are no elements nested in tag elements. However, tag elements are nested in relation, way and node elements. This may be easier to see by limiting the fields returned by our query.

```
> db.osm_map.find({}, {"_id":0, "name":1, "nested_elements":1})
{ "name" : "note", "nested_elements" : null }
{ "name" : "meta", "nested_elements" : null }
{ "name" : "bounds", "nested_elements" : null }
{ "name" : "node", "nested_elements" : [ "tag" ] }
{ "name" : "tag", "nested_elements" : null }
{ "name" : "nd", "nested_elements" : null }
{ "name" : "way", "nested_elements" : [ "tag", "nd" ] }
{ "name" : "member", "nested_elements" : null }
{ "name" : "relation", "nested_elements" : [ "member", "tag" ] }
{ "name" : "osm", "nested_elements" : null }
>
```

Another thing I noticed are the nested_elements. There are elements that are not nested in other elements and do not have nested elements. It would be valuable to have a query that identifies such elements.

To create that query, I first extracted a list of all element tags

```
> db.osm_map.aggregate([{$project: {"_id": "$name"}}])
{ "_id" : "note" }
{ "_id" : "meta" }
{ "_id" : "bounds" }
{ "_id" : "node" }
{ "_id" : "tag" }
{ "_id" : "nd" }
{ "_id" : "way" }
{ "_id" : "member" }
{ "_id" : "relation" }
{ "_id" : "osm" }
>
```

I then found all elements with nested_elements.

```
> db.osm_map.aggregate([{$match: {"nested_elements": {$ne: null}}}, {$group:
{"_id": "$name"}}])
{ "_id" : "way" }
{ "_id" : "relation" }
{ "_id" : "node" }
>
```

Next, I found all elements that are nested_elements.

```
> db.osm_map.aggregate([{$match: {"nested_elements": {$ne: null}}}, {$unwind:
"$nested_elements"}, {$group: {"_id": "$nested_elements"}}])
{ "_id" : "member" }
{ "_id" : "nd" }
{ "_id" : "tag" }
>
```

It was at this point I hit a limitation of MongoDB that doesn't exist in a traditional SQL based database. There is no way to perform set operations on multiple aggregation results within the MongoDB client. The work around for this limitation is to perform the set operation inside an application layer. Python was more than capable of combining these queries for the answer I was looking for.

```
λ  python .\xml_metadata_inquiries.py
These are the elements that have no nested elements and are not nested elements.
{'note', 'meta', 'bounds', 'osm'}
```

Note: When converting the queries from the Mongo Client to Python Script, it was necessary to encase the aggregation operators in quotes and replace the nulls with Nones.

When it comes time to process actual data stored in the OSM data, these elements without any children or parents will be excluded from this investigation. There may be information of interest in these elements. However, they seem to be lacking the same level of interest as found in elements that have relationships to other elements.

Structuring Data For Investigating

Having investigated the metadata generated from the OSM XML data, we can now turn our attention to the actual data. From studying the metadata, elements with nested tag type elements seemed to have the greatest potential for items requiring cleaning. Before getting started on this next task, it's important to decide how to structure the data we will be extracting.

For structuring the actual data, there's no reason to deviate too far from the XML metadata element structure. The base structure would remain the same. Instead of value types, we store actual values. Instead of one entry to describe all elements of and encountered type, there would be one entry per encountered element. Where there is deviation is in how nested elements and attributes are handled.

Each element has a different set of possible attributes and nested element types. Nested elements need to describe full elements. It only makes sense to include nested elements in a list for each type of nested element. Handling nested elements in this manner means each parent element type will have its own structure. This enables us to expand element attributes to the root level of each element in our dataset. To accommodate this structuring of element data, each parent element type will be added to a collection of elements sharing the same type.

Our investigation of OSM XML metadata has our investigation targeting elements of type: node, way and relations. Also thanks to the metadata, we can extract all the fields needed to store elements of these types. After extracting the information from our metadata and restructuring it for our design, we end with the following structures.

```
nodes: [{
{
    "type" : "node",
    "id" : value,
    "lat" : value,
    "lon" : value,
    "version" : value,
    "timestamp" : value,
    "changeset" : value,
    "uid" : value,
    "user" : value,
    "tags" : [
        {"k": k_value, "v": v_value},
        ...
    ],
    "text" : value
}]

ways: [{
    "type" : "way",
    "id" : value,
    "version" : value,
    "timestamp" :value,
    "changeset" : value,
    "uid" : value,
    "user" : value,
    "nd_ref": [
        values,
        ...
    ],
    "tags" : [
        {"k": k_value, "v": v_value},
        ...
    ],
    "text" : value
}]

relations: [{
    "type" : "relation",
    "id" : value,
    "version" : value,
    "timestamp" : value,
    "changeset" : value,
    "uid" : value,
    "user" : value,
    "tags" : [
        {"k": k_value, "v": v_value},
```

```
        ...
    ],
    "members":[
        {"type": value, "ref": value, "role": value},
        ...
    ],
    "text" : value
}]
```

Raw Data

With it settled on how to structure the data, it was now time to parse the XML and load it to MongoDB. The map_to_mongo.py script did the heavy lifting. With a loaded database and Mongo client, we're now prepared to investigate the data set.

Tag Type Counts

First thing, I took a look at is how many unique tag keys actually exist in the dataset.

```
> db.ways.aggregate([{$unwind: "$tags"}, {$project: {"tag_key": "tags.k"}},
{$group: {"_id": "$tag_key", "count": {$sum: 1}}}, {"$sort": {"count" : -1}}])
{ "_id" : "tags.k", "count" : 472374 }
> db.nodes.aggregate([{$unwind: "$tags"}, {$project: {"tag_key": "tags.k"}},
{$group: {"_id": "$tag_key", "count": {$sum: 1}}}, {"$sort": {"count" : -1}}])
{ "_id" : "tags.k", "count" : 46414 }
> db.relations.aggregate([{$unwind: "$tags"}, {$project: {"tag_key": "tags.k"}},
{$group: {"_id": "$tag_key", "count": {$sum: 1}}}, {"$sort": {"count" : -1}}])
{ "_id" : "tags.k", "count" : 3160 }
>
```

For every tag in the relations data, there's more than ten in the node data. There's almost ten times again as many tags in way data as there is in node data. With so few tags in relation data, it might be best to see what its most common tag types are.

```
> db.relations.aggregate([{$unwind: "$tags"}, {$project: {"tag_key": "$tags.k"}},
{$group: {"_id": "$tag_key", "count": {$sum: 1}}}, {"$sort": {"count" :
-1}}]).pretty()
{ "_id" : "type", "count" : 1017 }
{ "_id" : "restriction", "count" : 733 }
{ "_id" : "name", "count" : 130 }
{ "_id" : "building", "count" : 91 }
{ "_id" : "route", "count" : 70 }
{ "_id" : "ref", "count" : 61 }
{ "_id" : "public_transport:version", "count" : 46 }
{ "_id" : "network", "count" : 46 }
{ "_id" : "natural", "count" : 38 }
{ "_id" : "wikidata", "count" : 36 }
{ "_id" : "operator", "count" : 35 }
```

```
{ "_id" : "from", "count" : 30 }
{ "_id" : "addr:street", "count" : 29 }
{ "_id" : "addr:postcode", "count" : 29 }
{ "_id" : "addr:housenumber", "count" : 29 }
{ "_id" : "addr:city", "count" : 29 }
{ "_id" : "to", "count" : 29 }
{ "_id" : "wikipedia", "count" : 26 }
{ "_id" : "addr:state", "count" : 24 }
{ "_id" : "boundary", "count" : 21 }
Type "it" for more
>
```

Approximately a third of the tags in relations being of type "type", is a curious thing. This suggests a case for an nested element that might be better served as an attribute. It could be an interesting exercise to determine what value of type are being tagged in relations.

For now, we're instead going to look at the most common tags in node and way elements. With such a greater number of available tag elements, we're sure to find something in need of cleaning.

```
> db.nodes.aggregate([{$unwind: "$tags"}, {$project: {"tag_key": "$tags.k"}},
{$group: {"_id": "$tag_key", "count": {$sum: 1}}}, {"$sort": {"count" :
-1}}]).pretty()
{ "_id" : "highway", "count" : 4985 }
{ "_id" : "name", "count" : 3995 }
{ "_id" : "emergency", "count" : 3598 }
{ "_id" : "ele", "count" : 2434 }
{ "_id" : "amenity", "count" : 1999 }
{ "_id" : "natural", "count" : 1908 }
{ "_id" : "crossing", "count" : 1309 }
{ "_id" : "railway", "count" : 1265 }
{ "_id" : "gnis:Class", "count" : 1249 }
{ "_id" : "gnis:County", "count" : 1248 }
{ "_id" : "gnis:County_num", "count" : 1243 }
{ "_id" : "gnis:ST_alpha", "count" : 1241 }
{ "_id" : "gnis:ST_num", "count" : 1238 }
{ "_id" : "gnis:id", "count" : 1235 }
{ "_id" : "import_uuid", "count" : 1230 }
{ "_id" : "is_in", "count" : 1209 }
{ "_id" : "source", "count" : 1156 }
{ "_id" : "place", "count" : 1044 }
{ "_id" : "power", "count" : 946 }
{ "_id" : "time", "count" : 776 }
Type "it" for more
> it
{ "_id" : "shop", "count" : 674 }
{ "_id" : "cuisine", "count" : 522 }
{ "_id" : "addr:street", "count" : 486 }
{ "_id" : "addr:housenumber", "count" : 478 }
{ "_id" : "gnis:feature_id", "count" : 396 }
{ "_id" : "barrier", "count" : 385 }
{ "_id" : "addr:city", "count" : 366 }
```

```
{ "_id" : "gnis:created", "count" : 361 }
{ "_id" : "gnis:county_id", "count" : 349 }
{ "_id" : "gnis:state_id", "count" : 346 }
{ "_id" : "addr:postcode", "count" : 334 }
{ "_id" : "operator", "count" : 309 }
{ "_id" : "addr:state", "count" : 304 }
{ "_id" : "building", "count" : 276 }
{ "_id" : "religion", "count" : 228 }
{ "_id" : "ref", "count" : 221 }
{ "_id" : "denomination", "count" : 188 }
{ "_id" : "opening_hours", "count" : 184 }
{ "_id" : "website", "count" : 178 }
{ "_id" : "access", "count" : 165 }
Type "it" for more
>
```

In comparison to relation elements, node elements seem to have more diverse tags. Highway is the most common type of node tag and makes up little more than one tenth of all node tags. The next two most common tag types each account for approximately 8% of all node tags.

```
> db.ways.aggregate([{$unwind: "$tags"}, {$project: {"tag_key": "$tags.k"}},
{$group: {"_id": "$tag_key", "count": {$sum: 1}}}, {"$sort": {"count" :
-1}}]).pretty()
{ "_id" : "building", "count" : 87344 }
{ "_id" : "addr:housenumber", "count" : 35200 }
{ "_id" : "addr:postcode", "count" : 35100 }
{ "_id" : "addr:street", "count" : 35072 }
{ "_id" : "addr:city", "count" : 34179 }
{ "_id" : "addr:state", "count" : 33683 }
{ "_id" : "utahagrc:parcelid", "count" : 31214 }
{ "_id" : "highway", "count" : 29062 }
{ "_id" : "name", "count" : 16757 }
{ "_id" : "tiger:county", "count" : 10504 }
{ "_id" : "tiger:cfcc", "count" : 10392 }
{ "_id" : "tiger:name_base", "count" : 9872 }
{ "_id" : "tiger:reviewed", "count" : 9256 }
{ "_id" : "tiger:name_type", "count" : 5947 }
{ "_id" : "tiger:name_direction_prefix", "count" : 5314 }
{ "_id" : "surface", "count" : 5129 }
{ "_id" : "tiger:name_base_1", "count" : 3986 }
{ "_id" : "name:full", "count" : 3870 }
{ "_id" : "name:prefix", "count" : 3743 }
{ "_id" : "service", "count" : 3571 }
Type "it" for more
>
```

Way tags, like relation tags, are a bit lopsided towards one type. The with building tags being the most common at approximately 20 percent of all way tags. More than double the number of addr:housenumber tags.

Tiger Data

One type of way tag that I found curious are the "tiger:" tags. There seems to be an awful lot of them.

```
> db.ways.aggregate([{$unwind: "$tags"}, {$project: {"tag_key": "$tags.k",
"tag_value": "$tags.v"}}, {"$match": {"tag_key": {$regex: "^tiger"}}}, {$group:
{"_id": "$tag_key", "count": {$sum: 1}}}, {"$sort": {"count" : -1}}])
{ "_id" : "tiger:county", "count" : 10504 }
{ "_id" : "tiger:cfcc", "count" : 10392 }
{ "_id" : "tiger:name_base", "count" : 9872 }
{ "_id" : "tiger:reviewed", "count" : 9256 }
{ "_id" : "tiger:name_type", "count" : 5947 }
{ "_id" : "tiger:name_direction_prefix", "count" : 5314 }
{ "_id" : "tiger:name_base_1", "count" : 3986 }
{ "_id" : "tiger:name_direction_suffix", "count" : 2883 }
{ "_id" : "tiger:name_direction_prefix_1", "count" : 2579 }
{ "_id" : "tiger:name_direction_suffix_1", "count" : 1688 }
{ "_id" : "tiger:name_type_1", "count" : 1138 }
{ "_id" : "tiger:name_base_2", "count" : 899 }
{ "_id" : "tiger:name_direction_prefix_2", "count" : 682 }
{ "_id" : "tiger:tlid", "count" : 426 }
{ "_id" : "tiger:source", "count" : 426 }
{ "_id" : "tiger:separated", "count" : 399 }
{ "_id" : "tiger:upload_uuid", "count" : 258 }
{ "_id" : "tiger:name_type_2", "count" : 255 }
{ "_id" : "tiger:name_base_3", "count" : 213 }
{ "_id" : "tiger:name_direction_suffix_2", "count" : 190 }
Type "it" for more
>
```

Some of these Tiger tags have intuitive types. Others not so much. Maybe a quick look at the values of the "tiger:cfcc" tags will be helpful.

```
> db.ways.aggregate([{$unwind: "$tags"}, {$project: {"tag_key": "$tags.k",
"tag_value": "$tags.v"}}, {"$match": {"tag_key": {$regex: "^tiger:cfcc$"}}},
{$group: {"_id": "$tag_value", "count": {$sum: 1}}}, {"$sort": {"count" : -1}}])
{ "_id" : "A41", "count" : 9485 }
{ "_id" : "A21", "count" : 356 }
{ "_id" : "A31", "count" : 179 }
{ "_id" : "A63", "count" : 132 }
{ "_id" : "A11", "count" : 94 }
{ "_id" : "A74", "count" : 67 }
{ "_id" : "B11", "count" : 26 }
{ "_id" : "A15", "count" : 18 }
{ "_id" : "A73", "count" : 10 }
{ "_id" : "B11;B21", "count" : 4 }
{ "_id" : "A25", "count" : 4 }
{ "_id" : "A31:A33", "count" : 3 }
{ "_id" : "A39", "count" : 3 }
{ "_id" : "A31;A41", "count" : 2 }
```

```
{ "_id" : "A45", "count" : 2 }
{ "_id" : "A31:A41", "count" : 2 }
{ "_id" : "A15:A63", "count" : 2 }
{ "_id" : "A51", "count" : 2 }
{ "_id" : "C10", "count" : 1 }
>
```

Nope, not helpful at all. That query only left me with more questions. So it's time to take to the internet and find out what the situation is on Tiger data. A quick search and I found answers:
https://wiki.openstreetmap.org/wiki/TIGER

Based on the information found in the OSM wiki, Tiger data is a prime candidate for cleaning. There's even a communal effort documenting known issues and methods for correcting it. As is, there's already quite a bit of automation to clean tiger data. Current efforts for cleaning imported Tiger data is of the manual type. Researching the import automation and manual cleaning of Tiger is out of scope for this project.

This exploration of raw data returns a lot of areas worth investigating. Although, it also presents an issue with selection paralysis. In selecting an area to begin cleaning, I noticed that all three types of elements have address tags. As correcting issues with addressing is very much in scope with this course, I decided to focus on uniform addressing of Salt Lake City data.

Street Abbreviation Investigation.

Having lived my entire life in Salt Lake, I've found it curious how differently other cities manage addresses. In Salt Lake City, the house and street address can be broken further into two parts; a number and a direction. In other words, Salt Lake uses a coordinate system. Each number and directional combination describes how many blocks, in which direction away from the Mormon temple the address is. To avoid the need of decimals, city blocks are numbered as hundreds. As an example, 200 South is a street that runs East/West and is 2 blocks South of the Mormon Temple. 450 East is half a block between 400 East and 500 East.

What I suspect has happened with OSM data is a mixing of directional abbreviations in both house and street addresses. It's common for people to provide a single letter for the directional rather than the full word when handing out an address. With "addr:housenumber" and "addr:street" tags existing in all of our parent element types, these tag types would be a good exercise in applying uniform data cleanup to different data loads.

```
> db.ways.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k": /^addr:str/},
{"tags.k": /^addr:hou/}]}}, {$group: {"_id": "tags.k", "count": {$sum: 1}}}])
{ "_id" : "tags.k", "count" : 70296 }
> db.nodes.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k": /^addr:str/},
{"tags.k": /^addr:hou/}]}}, {$group: {"_id": "tags.k", "count": {$sum: 1}}}])
{ "_id" : "tags.k", "count" : 973 }
> db.relations.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k":
/^addr:str/},{"tags.k": /^addr:hou/}]}}, {$group: {"_id": "tags.k", "count":
{$sum: 1}}}])
{ "_id" : "tags.k", "count" : 58 }
>
```

A straight count shows the way elements accounting for a disproportionate number of all street and housenumber address tags. A straight count doesn't tell us how many are using abbreviated directionals. To get an idea of how many will need to be cleaned, we'll need to introduce some filtering by way of regular expression.

```
> db.ways.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k": /^addr:str/},
{"tags.k": /^addr:hou/}]}}, {$match: {"tags.v": /\b[NESWnesw]\b/}}, {$group:
{"_id": "tags.k", "count": {$sum: 1}}}])
{ "_id" : "tags.k", "count" : 473 }
> db.nodes.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k": /^addr:str/},
{"tags.k": /^addr:hou/}]}}, {$match: {"tags.v": /\b[NESWnesw]\b/}}, {$group:
{"_id": "tags.k", "count": {$sum: 1}}}])
{ "_id" : "tags.k", "count" : 109 }
> db.relations.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k":
/^addr:str/},{"tags.k": /^addr:hou/}]}}, {$match: {"tags.v": /\b[NESWnesw]\b/}},
{$group: {"_id": "tags.k", "count": {$sum: 1}}}])
{ "_id" : "tags.k", "count" : 7 }
>
>
```

Without even looking for false positives, these numbers are very low compared to the total number of addr:street and addr:housenumber tags.

```
> db.ways.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k": /^addr:str/},
{"tags.k": /^addr:hou/}]}}, {$match: {"tags.v": /\b[NESWnesw]\b/}}, {$project:
{"addr": "$tags.v"}}])
{ "_id" : ObjectId("5bdf759186e7792414502bda"), "addr" : "300 N" }
{ "_id" : ObjectId("5bdf759186e7792414502d3e"), "addr" : "239 S Main St" }
{ "_id" : ObjectId("5bdf759186e7792414502e31"), "addr" : "S 1400 East" }
{ "_id" : ObjectId("5bdf759186e7792414502e32"), "addr" : "S 1400 East" }
{ "_id" : ObjectId("5bdf759186e7792414502e36"), "addr" : "S 1400 East" }
{ "_id" : ObjectId("5bdf759186e7792414502e3b"), "addr" : "S 1452 East" }
{ "_id" : ObjectId("5bdf759186e7792414502e87"), "addr" : "N Terrace Hills Drive" }
{ "_id" : ObjectId("5bdf759186e7792414503bc9"), "addr" : "720 N" }
{ "_id" : ObjectId("5bdf759186e77924145040a0"), "addr" : "3289 E" }
{ "_id" : ObjectId("5bdf759186e779241450418c"), "addr" : "2354 S" }
{ "_id" : ObjectId("5bdf759186e779241450418d"), "addr" : "2309 S" }
{ "_id" : ObjectId("5bdf759186e7792414504190"), "addr" : "2375 S" }
{ "_id" : ObjectId("5bdf759186e77924145041ad"), "addr" : "23 E" }
{ "_id" : ObjectId("5bdf759186e77924145041af"), "addr" : "2120 S" }
{ "_id" : ObjectId("5bdf759186e77924145041b4"), "addr" : "2200 S" }
{ "_id" : ObjectId("5bdf759186e77924145041be"), "addr" : "2101 S" }
{ "_id" : ObjectId("5bdf759186e779241450444a"), "addr" : "1255 W" }
{ "_id" : ObjectId("5bdf759186e7792414504b2e"), "addr" : "1840 S" }
{ "_id" : ObjectId("5bdf759186e7792414504c4b"), "addr" : "361/363 N" }
{ "_id" : ObjectId("5bdf759186e7792414504cf4"), "addr" : "4400 S 700 E" }
Type "it" for more
> db.nodes.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k": /^addr:str/},
{"tags.k": /^addr:hou/}]}}, {$match: {"tags.v": /\b[NESWnesw]\b/}]}, {$project:
{"addr": "$tags.v"}}])
```

```
{ "_id" : ObjectId("5bdf758286e779241442a3b6"), "addr" : "4408 S" }
{ "_id" : ObjectId("5bdf758286e77924144390e6"), "addr" : "2100 S" }
{ "_id" : ObjectId("5bdf758286e779241443985e"), "addr" : "1264 W" }
{ "_id" : ObjectId("5bdf758386e779241444024c"), "addr" : "2211 W" }
{ "_id" : ObjectId("5bdf758386e779241444024c"), "addr" : "2300 S" }
{ "_id" : ObjectId("5bdf758386e779241444024d"), "addr" : "2211 W" }
{ "_id" : ObjectId("5bdf758386e779241444024d"), "addr" : "2300 S" }
{ "_id" : ObjectId("5bdf758386e779241444048f"), "addr" : "W 400 S" }
{ "_id" : ObjectId("5bdf758386e7792414440673"), "addr" : "1320 E." }
{ "_id" : ObjectId("5bdf758386e7792414442834"), "addr" : "344  S" }
{ "_id" : ObjectId("5bdf758386e7792414443b9b"), "addr" : "895 E 4500 S" }
{ "_id" : ObjectId("5bdf758386e7792414444e5d"), "addr" : "3167 E" }
{ "_id" : ObjectId("5bdf758386e7792414448094"), "addr" : "28 S" }
{ "_id" : ObjectId("5bdf758386e77924144482a5"), "addr" : "2227 S" }
{ "_id" : ObjectId("5bdf758386e7792414448593"), "addr" : "1854 S" }
{ "_id" : ObjectId("5bdf758386e7792414448593"), "addr" : "1955 W" }
{ "_id" : ObjectId("5bdf758386e77924144485cb"), "addr" : "836 W" }
{ "_id" : ObjectId("5bdf758386e77924144485cb"), "addr" : "1100 N" }
{ "_id" : ObjectId("5bdf758386e7792414448643"), "addr" : "1309 S" }
{ "_id" : ObjectId("5bdf758386e7792414448b5f"), "addr" : "307 W 600 S" }
Type "it" for more
> db.relations.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k":
/^addr:str/},{"tags.k": /^addr:hou/}]}}, {$match: {"tags.v": /\b[NESWnesw]\b/}},
{$project: {"addr": "$tags.v"}}])
{ "_id" : ObjectId("5bdf759586e779241451f349"), "addr" : "2210 E" }
{ "_id" : ObjectId("5bdf759586e779241451f34a"), "addr" : "2170 E" }
{ "_id" : ObjectId("5bdf759586e779241451f34b"), "addr" : "2251 E" }
{ "_id" : ObjectId("5bdf759586e779241451f34c"), "addr" : "2159 E" }
{ "_id" : ObjectId("5bdf759586e779241451f34e"), "addr" : "463 S" }
{ "_id" : ObjectId("5bdf759586e779241451f35d"), "addr" : "200 E" }
{ "_id" : ObjectId("5bdf759586e779241451f35e"), "addr" : "307 E" }
>
```

The initial selection looks good. However, false positives can be hiding deeper in the data. Grabbing a few more entries from the way elements returned our first false positives. It appears our regex is improperly retrieving possessives.

```
{ "_id" : ObjectId("5bdf759186e7792414504f08"), "addr" : "Carl's Jr." }
...
{ "_id" : ObjectId("5bdf759186e779241450f928"), "addr" : "Saint Mary's Drive" }
{ "_id" : ObjectId("5bdf759186e779241450f92a"), "addr" : "Saint Mary's Drive" }
{ "_id" : ObjectId("5bdf759186e779241450f92b"), "addr" : "Saint Mary's Drive" }
{ "_id" : ObjectId("5bdf759186e779241450f92c"), "addr" : "Saint Mary's Drive" }
{ "_id" : ObjectId("5bdf759186e779241450f92d"), "addr" : "Saint Mary's Drive" }
{ "_id" : ObjectId("5bdf759186e779241450f92e"), "addr" : "Saint Mary's Drive" }
{ "_id" : ObjectId("5bdf759186e779241450f92f"), "addr" : "Saint Mary's Drive" }
{ "_id" : ObjectId("5bdf759186e779241450f930"), "addr" : "Saint Mary's Drive" }
```

Not satisfied that we've found all the false positives, I continue my scan of the hits. My diligence is met with success. There are streets in an area known as 'The Avenues'. The streets running North and South in 'The

Avenues' are named with letters. Some of these streets have been caught in our search.

```
{ "_id" : ObjectId("5bdf759186e779241451780c"), "addr" : "N Street" }
{ "_id" : ObjectId("5bdf759186e779241451781c"), "addr" : "N Street" }
{ "_id" : ObjectId("5bdf759186e7792414517826"), "addr" : "S Street" }
{ "_id" : ObjectId("5bdf759186e7792414517835"), "addr" : "N Street" }
{ "_id" : ObjectId("5bdf759186e77924145178a6"), "addr" : "S Street" }
{ "_id" : ObjectId("5bdf759186e77924145178b4"), "addr" : "N Street" }
```

Additional scanning for false positives doesn't yield any more. There's now enough information to modify our selection of values to clean by excluding our false positives. Adding exclusion matches to our query to eliminate the false positives provides more accurate counts.

```
> db.ways.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k": /^addr:str/},
{"tags.k": /^addr:hou/}]}}, {$match: {$and: [{"tags.v": /\b[NESWnesw]\b/},
{"tags.v": {$not: /[NESWnesw] Street$/}}, {"tags.v": {$not: /'[NESWnesw]/}}]}},
{$group: {"_id": "tags.k", "count": {$sum: 1}}}])
{ "_id" : "tags.k", "count" : 206 }
> db.nodes.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k": /^addr:str/},
{"tags.k": /^addr:hou/}]}}, {$match: {$and: [{"tags.v": /\b[NESWnesw]\b/},
{"tags.v": {$not: /[NESWnesw] Street$/}}, {"tags.v": {$not: /'[NESWnesw]/}}]}},
{$group: {"_id": "tags.k", "count": {$sum: 1}}}])
{ "_id" : "tags.k", "count" : 108 }
> db.relations.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k":
/^addr:str/},{"tags.k": /^addr:hou/}]}}, {$match: {$and: [{"tags.v":
/\b[NESWnesw]\b/}, {"tags.v": {$not: /[NESWnesw] Street$/}}, {"tags.v": {$not:
/'[NESWnesw]/}}]}}, {$group: {"_id": "tags.k", "count": {$sum: 1}}}])
{ "_id" : "tags.k", "count" : 7 }
>
```

It's worth noting there are abbreviated directionals utilizing periods. These are being identified by our regular expression. The cleaning solution will need to handle these stray periods.

Another consideration are abbreviations that are abutted to the number portion of the address. These will need to be separated from the number with a space before they're expanded. Including regex to catch these for cleaning produces the final counts for cleaning.

```
> db.ways.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k": /^addr:str/},
{"tags.k": /^addr:hou/}]}}, {$match: {$and: [{$or: [{"tags.v": /[0-9]+
[NSEWnsew]\b/},{"tags.v": /\b[NESWnesw]\b/}]}, {"tags.v": {$not: /[NESWnesw]
Street$/}}, {"tags.v": {$not: /'[NESWnesw]/}}]}}, {$project: {"addr": "$tags.v"}},
{$group: {"_id": "$tag_key", "count": {"$sum": 1}}}])
{ "_id" : null, "count" : 209 }
> db.nodes.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k": /^addr:str/},
{"tags.k": /^addr:hou/}]}}, {$match: {$and: [{$or: [{"tags.v": /[0-9]+
[NSEWnsew]\b/},{"tags.v": /\b[NESWnesw]\b/}]}, {"tags.v": {$not: /[NESWnesw]
Street$/}}, {"tags.v": {$not: /'[NESWnesw]/}}]}}, {$project: {"addr": "$tags.v"}},
{$group: {"_id": "$tag_key", "count": {"$sum": 1}}}])
```

```
{ "_id" : null, "count" : 149 }
> db.relations.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k":
/^addr:str/},{"tags.k": /^addr:hou/}]}}, {$match: {$and: [{$or: [{"tags.v": /[0-
9]+[NSEWnsew]\b/},{"tags.v": /\b[NESWnesw]\b/}]}, {"tags.v": {$not: /[NESWnesw]
Street$/}}, {"tags.v": {$not: /'[NESWnesw]/}}]}}, {$project: {"addr": "$tags.v"}},
{$group: {"_id": "$tag_key", "count": {"$sum": 1}}}])
{ "_id" : null, "count" : 7 }
>
```

There's still a decision to make about casing for the expanded directional abbreviations. Should the directionals be all caps or title cased? I think it best to follow what is most common in the unmodified dataset. A quick look into the way tags shows there is only one all caps entry in the nodes dataset. The clear convention is to use title case for the directionals.

Also included in the casing queries are an exclusion of items that will be changed in our clean up. This allows a dual purposing of these counts. First, it assists us to determine casing convention. Second, it gives us a count of items that are not expected to be cleaned. With a count of items expected to be changed and a count of items expected to not change, we can set expected values for these queries post cleaning. Any difference between actual values and expected values will provide insight to the accuracy of the cleanup script.

```
> db.ways.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k": /^addr:str/},
{"tags.k": /^addr:hou/}]}}, {$match: {$and: [{"tags.v":
/\bSouth|North|East|West\b/}]}}, {$match: {"tags.v": {"$not": /\b[NESWnesw]\b/}}},
{$group: {"_id": "tags.k", "count": {$sum: 1}}}])
{ "_id" : "tags.k", "count" : 11288 }
> db.nodes.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k": /^addr:str/},
{"tags.k": /^addr:hou/}]}}, {$match: {$and: [{"tags.v":
/\bSouth|North|East|West\b/}]}}, {$match: {"tags.v": {"$not": /\b[NESWnesw]\b/}}},
{$group: {"_id": "tags.k", "count": {$sum: 1}}}])
{ "_id" : "tags.k", "count" : 348 }
> db.relations.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k":
/^addr:str/},{"tags.k": /^addr:hou/}]}}, {$match: {$and: [{"tags.v":
/\bSouth|North|East|West\b/}]}}, {$match: {"tags.v": {"$not": /\b[NESWnesw]\b/}}},
{$group: {"_id": "tags.k", "count": {$sum: 1}}}])
{ "_id" : "tags.k", "count" : 21 }
> db.ways.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k": /^addr:str/},
{"tags.k": /^addr:hou/}]}}, {$match: {$and: [{"tags.v":
/\bSOUTH|NORTH|EAST|WEST\b/}]}}, {$match: {"tags.v": {"$not": /\b[NESWnesw]\b/}}},
{$group: {"_id": "tags.k", "count": {$sum: 1}}}])
> db.nodes.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k": /^addr:str/},
{"tags.k": /^addr:hou/}]}}, {$match: {$and: [{"tags.v":
/\bSOUTH|NORTH|EAST|WEST\b/}]}}, {$match: {"tags.v": {"$not": /\b[NESWnesw]\b/}}},
{$group: {"_id": "tags.k", "count": {$sum: 1}}}])
{ "_id" : "tags.k", "count" : 1 }
> db.relations.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k":
/^addr:str/},{"tags.k": /^addr:hou/}]}}, {$match: {$and: [{"tags.v":
/\bSOUTH|NORTH|EAST|WEST\b/}]}}, {$match: {"tags.v": {"$not": /\b[NESWnesw]\b/}}},
{$group: {"_id": "tags.k", "count": {$sum: 1}}}])
>
```

With this information, we can expect the the following number of counts in each collection after cleanup. Specifically, the number of tags with only full directional names plus the number of tags identified for abbreviation expansion should produce the total number of tags with full directionals after cleaning.

- Ways: 11288 + 209 = 11,497
- Nodes: 348 + 149 = 497
- Relations: 21 + 7 = 28

Street Abbreviation Clean Up

Now that we have an idea of what addressing data exists and plan on how to clean it, there's nothing left to do but do it. Extracting and loading the data is something we've solved with the map_to_mongo.py script. I could reinvent that process. Instead, I've expanded on it with a transformation script (slc_street_cleanup.py) to modify the data between the extract and load steps. Running this script produces a more uniform dataset with expanded street directionals.

```
> db.ways.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k": /^addr:str/},
{"tags.k": /^addr:hou/}]}}, {$match: {$and: [{"tags.v":
/\bSouth|North|East|West\b/}]}}, {$group: {"_id": "tags.k", "count": {$sum: 1}}}])
> db.ways.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k": /^addr:str/},
{"tags.k": /^addr:hou/}]}}, {$match: {$and: [{"tags.v":
/\bSouth|North|East|West\b/}]}}, {$group: {"_id": "tags.k", "count": {$sum: 1}}}])
{ "_id" : "tags.k", "count" : 11497 }
> db.nodes.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k": /^addr:str/},
{"tags.k": /^addr:hou/}]}}, {$match: {$and: [{"tags.v":
/\bSouth|North|East|West\b/}]}}, {$group: {"_id": "tags.k", "count": {$sum: 1}}}])
{ "_id" : "tags.k", "count" : 496 }
> db.relations.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k":
/^addr:str/},{"tags.k": /^addr:hou/}]}}, {$match: {$and: [{"tags.v":
/\bSouth|North|East|West\b/}]}}, {$group: {"_id": "tags.k", "count": {$sum: 1}}}])
{ "_id" : "tags.k", "count" : 28 }
>
```

Except for node data, the counts are at the expected values. As the node data is off by only one, it would be good to take a look at this single value and determine why it wasn't cleaned.

```
> db.nodes.aggregate([{$unwind: "$tags"}, {$match: {$or: [{"tags.k": /^addr:str/},
{"tags.k": /^addr:hou/}]}}, {$match: {$and: [{$or: [{"tags.v": /[0-9]+
[NSEWnsew]\b/},{"tags.v": /\b[NESWnesw]\b/}]}, {"tags.v": {$not: /[NESWnesw]
Street$/}}, {"tags.v": {$not: /'[NESWnesw]/}}]}}, {$project: {"addr":
"$tags.v"}}])
{ "_id" : ObjectId("5be1990086e7793790470312"), "addr" : "3540 2200 W\nSalt Lake
City, UT 84119" }
>
```

There is more wrong with this entry than just the abbreviated directional. I suspect it is not being caught because of differences between how MongoDB and Python handle string interpretations. In this case I suspect

that Python is interpreting '\n' as a CRLF character instead of a literal backslash + n. A quick jump into a Python shell and my suspicion was validated.

```
>>> import re
>>> victim = "3540 2200 W\nSalt Lake City, UT 84119"
>>> victim_raw = r'3540 2200 W\nSalt Lake City, UT 84119'
>>> direction_abbr_re = re.compile(r'(.*?)\b([NESW])\b(.*$)', re.IGNORECASE)
>>> str_sections = direction_abbr_re.search(victim)
>>> str_sections_raw = direction_abbr_re.search(victim_raw)
>>> print(victim)
3540 2200 W
Salt Lake City, UT 84119
>>> print(victim_raw)
3540 2200 W\nSalt Lake City, UT 84119
>>> print(str_sections)
None
>>> print(str_sections_raw)
<re.Match object; span=(0, 37), match='3540 2200 W\\nSalt Lake City, UT 84119'>
>>>
```

I've decided to leave this string handling behavior in the script as is. My intent is to illustrate an issue that can be addressed in future iterations of the script. It would be simple to cast address values as raw strings before transforming them. That solution would surely take care of this one edge case.

Instead, there's value to be gained in considering why this CRLF character is included and interpreted as it is. There could be other non-printed control characters encountered in future datasets. A better solution would take into consideration why these control characters might be included in the data and manipulate the data accordingly when those characters are encountered. If strings containing control characters were converted to raw strings before cleaning, unhandled control characters could be missed.

For example, this unaltered data point. I'm guessing the CRLF character was included as part of a mailing address import. With mailing addresses, the house and street numbers are on one line with city, state, and zip code on the next. That's clearly what we have here. If we were to interpret this data point as a raw string, we would end up including city, state and zip code data in addr:housenumber or addr:street data. A better solution would be stripping out the extraneous data and encoding it to the proper tag types.

Conclusion/Additional Ideas

Finally, we've reached the end of this case study. If there's one thing I would take away from this experience, it's the knowledge that even a simply constructed XML document can have many layers for data investigation. Even in this brief peak into OSM data uncovered a number of items that could be explored for cleaning and restructuring. Items like incorporating Tiger import data or better handling of relation "type" tags.

Another area of this case study that could be improved upon are the tools developed for data transformation and cleaning. Care was taken to ensure some of these tools could be adapted to dissecting future XML datasets. In that vein, the queries for interpreting XML metadata could be better utilized in an automated fashion for generating reports on a variety of XML structures. As the base script structures were designed with this expansion in mind, I fully expect I will be performing such refinements in the future.

For final thoughts on utilizing MongoDB as a storage and query engine for data; it is different from traditional SQL engines. That isn't to say one type of engine is better than the other. I'm sure there are places where pipeline based queries would do better than set manipulation queries and vice versa. MongoDB will be a powerful tool for my tool chest going forward. This is only a start of what I will do.