Open Street Map Project

June 21, 2017

1 Open Street Map Data Wrangling Project

1.1 Location: Edinburgh Scotland

By complete chance my wife and two good friends ended up here on the same days in the same hotel. The best I can do is this map.

The data file for the whole city of Edinburgh was rather large so I took a manually selected subset of the city and exported it using the Overpass API.

2 Problems Encountered in the Map

After loading the data and performing some exploratory querying I noticed a few issues worth addressing.

Cleaning and Auditing

- Compound values in the key attribute of tags separated by a colon (e.g. disused:amenity, bicycle:repair, addr:street)
- The use of the key values **postal_code** and **postcode**, which both have values representing postal codes
- The use of the key values **url** and **website**, which both have values representing web addresses
- Value attributes with semicolon delimited lists and Key = 'source'
- Value attributes containing commas were truncated when being loaded into the database

Other Issues

- Use of reserved words in the table definitions
- Skipped nodes records on import to MySQL

2.0.1 Compound values in key attributes

Upon reviewing the initial data loaded into MySQL I noticed that many keys contained compound values where the parts where separated by a colon (e.g. addr:housenumber, recycling:glass, diet:vegan)

```
SELECT `key`, COUNT(*) as cnt
FROM nodes_tags
WHERE `key` REGEXP '[A-Za-z]*:[A-Za-z]*'
GROUP BY `key`
ORDER BY cnt DESC;
```

In looking at the query results, it was clear that keys with the prefix **addr** were the vast majority of all keys with compound values. Similar to the course case study, I decided to split the compound value and use **addr** as the value for the **type** field and the value after the colon as the **key**. In fact, the majority of compound values would benefit from being split in the same manner so I applied this change to all compound key values.

```
if key == 'k':
    if not is_compound_value(value):
        tag_detail['key'] = value
        tag_detail['type'] = 'regular'
    else:
        # split key on : and assign second element as the key
        # and the first (and third if it exists) elements as the type
        key_strs = value.split(':')
        if len(key_strs) == 3:
            tag_detail['key'] = key_strs[1]
            tag_detail['type'] = key_strs[0] + ':' + key_strs[2]
        else:
            tag_detail['key'] = key_strs[1]
            tag_detail['type'] = key_strs[0]
```

2.0.2 postcode and postal_code

The data uses both **postcode** and **postal_code** as keys. In both cases the value attribute is a properly formed postal code. Since postcode was the far more common value I converted any postal_code key values to postcode.

The query below was used to determine that each field contained valid postal codes.

```
SELECT `key`, `value`, type
FROM nodes_tags
WHERE `key` in ('postcode', 'postal_code') AND `value` REGEXP 'EH([0-9]|1[0-7]) [0-9][A-Z]{2}'
GROUP BY `key`, `value`, type
```

2.0.3 website and url

The data uses both **website** and **url** as keys, but both contain values that are web addresses. To correct this (as well as the postcode / postal_code) I wrote two functions. The first checks if a given key (e.g. postal_code) is in the list of keys that we want to update to another value. The second function uses a dictionary containing a mapping to update the key.

```
synonym_mapping = {'postal_code' : 'postcode',
                   'url' : 'website'}

synonym_keys = ['postal_code', 'url']
```

```
def has_synonym_key(value):
    return value in synonym_keys

def update_synonym_key_value(name, mapping):
    name = name.replace(name, mapping[name])
    return name
```

2.0.4 Semicolon delimited lists in value attribute when k='source'

While exploring the nodes_tags table I noticed that **35,693** out of **159,704** or **22.35%** of the key values were source.

```
SELECT `key`, COUNT(*) as cnt
FROM nodes_tags
GROUP BY `key`
ORDER BY cnt DESC;
```

Looking at the distribution of values within the keys we see that the vast majority of values, **35,043** are **survey** (**19,901**) and **Bing** (**15,142**). In the remainder of the list there are some minor issues. For example, there are records with values of naptan_import;survey, naptan_import; survey, naptan_import/survey, and naptan_import;survet. Additionally, there were a few combinations of Bing;survey and NLS_OS_Edinburgh_map_YYYY;Bing;survey.

Show all keys with value == 'source'

```
SELECT `value`, COUNT(*) as cnt
FROM nodes_tags
WHERE `key`='source'
GROUP BY `value`
ORDER BY cnt DESC;
```

Show any value containing 'naptan_import' where key == 'source'

```
SELECT `value`, COUNT(`value`) AS cnt
FROM nodes_tags
WHERE `key` = 'source' AND `value` REGEXP "naptan_import"
GROUP BY `value`
ORDER BY cnt DESC;
```

Show any value containing 'Bing' where key == 'source'

```
SELECT `value`, COUNT(`value`) AS cnt
FROM nodes_tags
WHERE `key` = 'source' AND `value` REGEXP "Bing"
GROUP BY `value`
ORDER BY cnt DESC;
```

To handle these minor variances I decided to use only the first entry in any record with ';'. This change was only applied to values where key='source'.

```
if is_source_key(tag_detail):
    val_strs = value.split(';')[0]
    tag_detail['value'] = val_strs
else:
    tag_detail['value'] = value
```

2.0.5 Truncated values after importing to MySQL

There were values being truncated during the import process due to there being commas in the value. After some research I added the --fields-optionally-enclosed-by=′"′ to allow those fields to go into the table including the comma.

From command line (repeat for each of the csv files) to import into the the database:

```
mysqlimport --ignore-lines=1 --fields-terminated-by=, --fields-optionally-enclosed-by='"', --v
```

Import csv to MySQL
Allow commas

2.0.6 Importing the schema to MySQL

The provided .sql had to be modified for use with MySQL because the column names **key** and **value** are reserved words in MySQL. They simply need to be escaped by surrounding them in back-ticks (e.g. **‘key‘** and **‘value‘**) in the .sql file.

From the command line:

```
mysql -u username -p database_name < schema_file.sql
```

StackOverflow

2.0.7 Skipped node records on import to MySQL

- Using the --verbose flag during import I was able to see that ***165,602*** out of ***195,524*** nodes records were skipped (i.e. not imported into the table). After some investigation of the raw nodes.csv file I noticed that the id values were up to 10 digits long. In the open_map_project_schema.sql file the id fields were set to INT. I then looked at the MySQL documentation and was able to determine that the INT type can only store values up to ***4,294,967,295*** assuming the field is UNSIGNED. Since the schema file only specified INTEGER the limit was ***2,147,483,647***. The largest node ids were larger than the unsigned limit.
  I changed all of the columns that were related to ids to the BIGINT type. After that change all nodes loaded successfully.

```
In [2]: %%html
        <style>
        table {float:left}
        </style>
```

```
<IPython.core.display.HTML object>
```

3 Overview of the Data

3.0.1 File Sizes

File	Size
endinburgh_scotland.osm	55 MB
nodes.csv	16 MB
nodes_tags.csv	5.3 MB
ways.csv	2.2 MB
ways_nodes.csv	7.4 MB
ways_tags.csv	3.6 MB

3.0.2 Unique Users

Since both the nodes table and ways table contain the field **uid** and there is no foreign key relationship the number of unique users will be a union of the unique users from each table. Using `UNION` automatically removes duplicates so all we need to do is count the number of records from the result of the union.

```
SELECT COUNT(*)
FROM (SELECT uid FROM nodes
UNION
SELECT uid FROM ways) as users;
```

There are **291** unique users in the dataset.

3.0.3 Number of Nodes and Ways

According the the schema, nodes are unique so each record in the nodes table has a unique id

```
SELECT COUNT(*)
FROM nodes;
```

There are **195,524** nodes.
Similarly, ways records are unique

```
SELECT COUNT(*)
FROM ways;
```

There are **37,167** ways.

3.0.4 Tree Friendly

There are **14,667** trees included in this subset of the Edinburgh data.

```
SELECT `key`, `value`, COUNT(*) as cnt
FROM nodes_tags
WHERE `key` = 'natural'
GROUP BY `key`, `value`
ORDER BY cnt DESC;
```

Key	Value	Count
natural	tree	14,467
natural	peak	7
natural	spring	2
natural	cliff	1
natural	mud	1

3.0.5 Bicycle Friendly too

The most frequently listed **amenity** is **bicycle_parking**.

```
SELECT `key`, `value`, COUNT(*) as cnt
FROM nodes_tags
WHERE `key` = 'amenity'
GROUP BY `key`, `value`
ORDER BY cnt DESC
LIMIT 10;
```

Key	Value	Count
amenity	bicycle_parking	394
amenity	restaurant	386
amenity	bench	298
amenity	cafe	284
amenity	fast_food	188
amenity	pub	168
amenity	motorcycle_parking	131
amenity	telephone	128
amenity	post_box	108
amenity	atm	103

3.0.6 Sources

I was initially surprised by the frequency of **survey** and **Bing** as source values in the nodes_tags, but after looking at the OpenStreetMap Wiki these listed among the most common values for **human mappers** in the **Usage Statistics** section.

```
SELECT `key`, `value`, `type`, COUNT(*) as cnt
FROM nodes_tags
GROUP BY `key`, `value`, `type`
ORDER BY cnt DESC
LIMIT 10;
```

Key	Value	Count
source	survey	19,901

Key	Value	Count
city	Edinburgh	15,155
source	Bing	15,142
natural	tree	14,667
country	GB	14,176
denotation	urban	12,663
entrance	yes	1,158
barrier	gate	927
barrier	bollard	513
housenumber	1	509

3.0.7 Top Contributors

To get a view of what users were making the most contributions I combined all of the activity from nodes, nodes_tags, ways, ways_nodes, and ways_tags. Every entry in these tables is attributed to a user.

```
SELECT user, COUNT(*) as cnt
FROM
    ((SELECT n.user FROM nodes n
        JOIN
            (SELECT id FROM nodes
                UNION ALL
            SELECT id FROM nodes_tags) AS na
        ON n.id = na.id)
    UNION ALL
    (SELECT w.user FROM ways w
        JOIN
            (SELECT id FROM ways
                UNION ALL
            SELECT id FROM ways_nodes
                UNION ALL
            SELECT id FROM ways_tags) AS wa
        ON w.id = wa.id))
    AS all_activity
GROUP BY user
ORDER BY cnt DESC
LIMIT 10;
```

User	Records
eric_	284,268
sophiemccallum	262,818
eisa	90,274
leilaz	55,668
drnoble	23,543
saintam1	12,090

User	Records
rob_michel	10,793
sairfeet	9,982
c0d0	6,801
Rostranimin	5,906

4 Other Ideas

The biggest issue with this dataset is that when a node has only a single node_tag it is a k="source" attribute, which refers to the source of the node information. This is different from additional node_tags which are descriptive of the location or item being identified (e.g. k="highway" v="traffic_signals"). Though both are node_tags these seem like fundamentally different information and any analysis should probably consider them as separate.

4.0.1 User concentrations and area of operation

It would be interesting to understand why the records in the data set are so heavily concentrated in only two users. Would a map of another area have similarly skewed distributions of user contributions? I would also be curious to expand the map area and see how far these users contributions extend. One potential way to start to answer these questions would be to update the data set so that every node or way contained a tag with the appropriate postal code. With all of the postal codes available we could then begin to map user contributions against the postal codes. This would allow us to see if users are local or if they operate across broad areas. I think if the data for postal codes could be expressed in terms of the latitudinal and longitudinal boundaries they apply to it would be fairly straightforward for a competent programmer to update all available nodes.

If such a list of postal codes could not be expressed by latitudinal and longitudinal boundaries then the process of incorporating this information into the nodes would be far more difficult.

4.0.2 More thorough treatment of compound values

As discussed above, key values containing multiple pieces of information were split. The value after the : became the key and the value before the : became the type. Because the majority of these values were combinations of **addr:value** and other values looked to similarly benefit from this change it was done accross the board. It is likely not the case that all such values shoud be transformed and it would require additional auditing and manual review to decide where to apply the transformation.

4.0.3 Clean up phone numbers

There is some inconsistency in the format of the phone numbers that should be addressed. The results of the query below show varying lengths. Some of these are due to multiple phone numbers being stored, while others are differences in formatting. A review of the OpenStreetMap documentation may reveal the appropriate convention(s) for this area, in which case any deviant phone numbers could be adjusted to the proper format.

```
SELECT `key`, length(`value`) as len, COUNT(*) as cnt
FROM nodes_tags
WHERE `key` = 'phone'
GROUP BY `key`, len
ORDER BY len DESC
```

```
In [ ]:
```