Project: Wrangle OpenStreetMap Data

By Wayne Radinsky

For this project, I got data for the municipality of Lyon, France. I thought about doing someplace familiar, like my hometown of Denver, but then I thought, as a data analyst, I'll practically never be working with familiar data. So I got data for Toulouse, France. It turned out to be too small (36 MB vs the required 50+). So that's why I decided to move up to a larger city and went with Lyon. I've never been to Lyon (or Toulouse, or to anywhere in France for that matter, or to anywhere in Europe for that matter), so this is totally unfamiliar territory. I got enough data to capture the entire municipality of Lyon (at least as Google Maps outlined it), which is still considerably less that the entire metropolitan area of Lyon. The file is 249.8 MB. The query to retrieve the file, specifically, using the Overpass API, was:

(node(45.707041, 4.770233, 45.809566, 4.899329);<;);out meta;

First I looked to see what tags were in the file. They were (in order of first appearance in the file):

- osm
- note
- meta
- node
- tag
- way
- nd
- relation
- member

Well, that was useful but didn't give me an idea how prevalent the various tags were, so I did it again and counted.

- osm: 1
- note: 1
- meta: 1
- node: 970,659
- tag: 706,689
- way: 147, 911
- nd: 1,286,512
- relation: 5,212
- member: 195,920

So we can see, the bulk of the data is in node, tag, way, nd, and member tags, with a little bit in relation tags.

Next, I decided to see what tags are within other tags, to get some sense of the hierarchical structure. This is how that came out looking:

- osm: note, meta, node, way, and relation
- note: -
- meta: -
- node: tag
- tag: -
- way: nd and tag
- nd: -
- relation: member and tag
- member: -

What this means is that the “osm” tag has “note”, “meta”, “node”, “way”, and “relation” tags within it. The dashes just mean there are no tags within those, so there are no tags within “note”, “meta”, “tag”, “nd”, or “member” tags. “node” tags can contain “tag” tags. “way” tags can contain “nd” and “tag” tags. “relation” tags can contain “member” tags and “tag” tags. So “tag” is kind of interesting as it appears in multiple places. A “tag” can be attached to a “node”, a “way”, or a “relation”.

We can also see that only “node” and “relation” are top-level because they are children of the “osm” tag. I'm ignoring at this point the “note” and “meta” tags, as visual inspection of the head of the file shows they just contain some metadata about the OSM file (that it comes from www.openstreetmap.org, is made available under ODbL, and the date it was generated).

Next I looked at the attributes of each tag.

- osm: version and generator
- note: -
- meta: osm_base
- node: id, lat, lon, version, timestamp, changeset, uid, and user
- tag: k and v
- way: id, version, timestamp, changeset, uid, and user
- nd: ref
- relation: id, version, timestamp, changeset, uid, and user
- member: type, ref, and role

XML tag and attribute meanings

At this point I started thinking about what relational schema to convert this into. I knew from the coursework that the “k” and “v” attributes on tags are for key/value pairs, and that's where a lot of the interesting “human-meaningful” data is. But if I was coming at this project afresh, I wouldn't know that, so I'd need to consult the documentation for further meaning of the various tags and attributes.

The documentation for the file format is on the OpenStreetMap project's wiki at:

https://wiki.openstreetmap.org/wiki/OSM_XML

It explains, among many other things, that:

- A node represents a specific point on the earth's surface defined by its latitude and longitude.
- A way is an ordered list of between 2 and 2,000 nodes that define a polyline. Ways are used to represent linear features such as rivers and roads. Ways can also represent the boundaries of areas (solid polygons) such as buildings or forests. [...] Note that closed ways occasionally represent loops, such as roundabouts on highways, rather than solid areas.
- A relation is a multi-purpose data structure that documents a relationship between two or more data elements (nodes, ways, and/or other relations). These objects are known as the relation's members.
- All types of data element (nodes, ways and relations), as well as changesets, can have tags. Tags describe the meaning of the particular element to which they are attached.

The tag system can have any key and value combinations. However they've worked to define a commonly used set of key names for amenities (restaurants, schools, parking lots, ATM locations, health clinics, cinemas), barriers (walls, fences, ditches), boundaries (postal code boundaries, etc), buildings, crafts (meaning places where industrial goods are produced), emergency facilities (fire stations, etc), geological features (glacial moraines, etc), highways (and other roads), historic sites, landuse (commercial, residential, etc), leisure (nightclubs, casinos, etc), man_made features (e.g. communications towers), military facilities (airfields, barracks, etc), natural features (grassland, forest, etc), offices, places (city names, etc), power structures (power plants, etc), public transport facilities (e.g. bus stops), railway features (tracks, stations, stops), routes (bicycle, bus, canoe, ferry, etc, routes), shops (food, general stores, clothing stores, etc), sports facilities (soccer fields, ice rinks, etc), telecom locations (exchanges, data centers), tourist attractions (campsites, theme parks, zoos), aerialways (e.g. chair lifts), aeroways (airports, gates, runways), waterways (rivers, streams, etc), and other common things one finds on maps.

The top 10 keys (“k” attributes), along with the number of occurrences, are:

1. source : 181,756
2. building : 105,062
3. addr:housenumber : 60,980
4. highway : 35,915
5. name : 29,684
6. wall : 22,435
7. amenity : 14,316
8. natural : 13,919
9. addr:street : 13,745
10. surface : 9,261

SQL or MongoDB?

To put this in a relational database, there are two ways we could do it. We could do separate tables for the tags for nodes, tags for ways, and tags for relations. Or we could do one table for tags. It could have the foreign key field have IDs for nodes, ways, or relations, and we could have a type field to indicate which type of ID it is. Or we could have 3 separate foreign key fields, one for node IDs, one for way IDs , and one for relations.

There are advantages and disadvantages to each approach. The advantage of doing it the first way, where we have separate tables, is that would could establish foreign key relationships in the database system itself. That enables the database system itself to do things like cascading deletes. However, since we're only using the data for exploration, and not for an on-line system with continual inserts and deletes (though the OpenStreetMap website itself probably does), this consideration might not be too important. The advantage of doing it the second way is that we have all the tags in one table and can search it all at once. In practice, since we would need to do joins with different tables, depending on whether a tag belongs to a node, way, or relation, that is not as easy as it sounds. But semantically, in the minds of the end users, the people using the maps, there might not be a meaningful distinction between a “railway” tag that links to a node (say, for a station) vs one that links to a way (say, for railroad tracks). They're all “railway”.

In this case, we don't have to decide because we have the schema given to us as part of the project. So let's go look at that.

Before I do that I should probably comment on the SQL vs MongoDB decision. I went through all the course material on MongoDB because I'm familiar with SQL already (from my day job). I ended up with mixed feelings about it. It seemed like the “schemaless” philosophy of MongoDB is similar to the distinction in programming languages between statically typed languages and dynamically typed languages. What a dynamically typed languages makes easy to do is make variables that contain data structures of arbitrary complexity – dictionaries with lists that themselves are dictionaries, that themselves contain lists, etc, to any level of depth. And what MongoDB makes easy to do is to take those variables and stuff them directly into the database. That's probably a useful thing to do, and maybe especially useful in the context of interactively exploring data. SQL databases, with the rigidity of their precisely defined tables with all the types specified and all the primary key/foreign key relationships specified, are probably better suited to production code, like statically typed languages, where everything needs to be rock solid. I know there's an argument on the other side about how with really *really* huge databases, like those giant companies like Google have, there's the issue that when you change the schema, it doesn't change instantly, but there's a period of time when different parts of the same database have different schemas and everything gets out of sync. Google from what I understand uses a noSQL system called BigTable for most of their data.

Even so, I decided to go with SQL because the relational model makes the most sense to me conceptually. The relational model is based on set theory and is a powerful model that can model any data, and I like the precise control over types and relationships. I liked the MongoDB “aggregation pipeline”. SQL can do the same thing, though nesting SELECTs can get cumbersome. But in the end I decided to go with SQL because I like the relational model conceptually, and MongoDB with it's “stuff any variable no matter how complex its structure into the database” approach to data doesn't use it. I know you all recommended using whichever we're least familiar with, but it didn't seem like an absolute requirement, so I decided to go with SQL anyway. I'll keep MongoDB in my back pocket and maybe find a use for it in the future.

Ok, let's have a look at the data schema we're going to use. The data schema is at

https://gist.github.com/swwelch/f1144229848b407e0a5d13fcb7fbbd6f

and I can see there are separate tables for nodes_tags and ways_tags, with the foreign key relationships explicitly declared. Ok, so that's how we'll do it. Relations (and members) are omitted, so I guess we're not supposed to do anything with that data.

To parse the XML, I have to give credit to Fredrik Lundh of effbot who explained the “root.clear()” trick for parsing XML files too large to fit in memory. (Actually, my computer has 32 GB of memory, so I'm pretty sure it *can* fit in memory, but let's go ahead and assume we're dealing with a large enough dataset that we don't want to fit it in memory for this project.)

http://effbot.org/elementtree/iterparse.htm

Once we've parsed the XML into CSVs and in turn parsed the CSVs into SQL (and by the way, I would have just had the XML parser do INSERTS into the DB and skip the intermediate CSV files, but making the CSV files was required by the project description, so you have them), we count the number of records and see if the totals match above:

- Number of nodes: 970,659
- Number of ways: 147,911
- Number of node tags: 286,030
- Number of ways tags: 398,394
- Number of ways nodes: 1,286,512

The totals for tags don't match. We have 286,030 + 398,394 = 684,424 tags, but our previous number was 706,689. In fact we didn't bring the relations tags into the SQL database, so we wouldn't expect the totals to add up. The missing 22,265 tags probably belong to relations.

The rest of the totals match. We saw 970,659 nodes in the XML and see the same number here. We saw 147, 911 ways in the XML and we see the same number here. We saw 1,286,512 nd tags in the XML and we see the same number of ways nodes here. We didn't import the relations and members into the SQL database. This is all what we would expect and a nice sanity check.

Unlike the numbers above, which were tallied up in the code that handles the XML parsing, these numbers were done from database queries. The project description says I need to list the queries, so here are the queries used to generate the above numbers:

- `SELECT COUNT(*) FROM nodes WHERE 1;`
- `SELECT COUNT(*) FROM ways WHERE 1;`
- `SELECT COUNT(*) FROM nodes_tags WHERE 1;`
- `SELECT COUNT(*) FROM ways_tags WHERE 1;`
- `SELECT COUNT(*) FROM ways_nodes WHERE 1;`

Data Cleaning

So I looked at cleaning the addr:street field. It turns out that in France, the type of street is at the beginning of the name rather than the end. In this country, we would say “Mirabeau Street”, or “Mirabeau St”, but in France they say, “Rue Mirabeau”. And by the way, that “St” abbreviation – I expected to see the same thing in France, lots of street type abbreviations to correct. Not so. Instead what they do in France is follow the “Rue” with “du”, “de”, “d'”, “des”, “de la”, or “de l'”, or just a name without any “de” form. The “de” basically means “of”. Some examples: Rue du Château de la

Duchère, Rue de Montcharmont, Rue d'Avignon, Rue de la Martinière, Rue de l'Etoile, Rue des Rivières, Rue Notre-Dame. I didn't "correct" for any of these, but I could imagine there may be scenarios where you'd need to strip off the "de" forms to get at just the street name. In a real-world scenario, I'd be cleaning the data for some specific purpose and the data cleaning operations would be geared towards that. For the moment I'm just going to make note that this potential issue exists.

Besides "Rue", the French also use: Boulevard, Avenue, Passage, Route (road), Chemin (path), Place (square), Cours (course), Cour (court), Montée (climb), Quai (dock), Allée (driveway), Espace (space), Passerelle (bridge), Promenade (walk), Carrefour (crossroads), Ruelle (alley – used only once for Ruelle du Vieux Moulin ("Old Mill Alley"), Les Jardins (gardens – used only once for Les Jardins d'Arcadie ("Arcadia Gardens"), Lotissement (allotment – used only once for Lotissement des Sabines ("Sabines's Allotment"), Impasse (cul-de-sac – a French term we use in English but ironically they don't use), Square (in English, used a few times despite them having the French "Place"), Côte (coast – and no, Lyon is nowhere near any coastline), Cité (city), Ancienne Route (old road), Ancienne Montée (old climb), and Parvis (square in front of the entrance to a church).

They also use "Grande Rue" and "Petite Rue" – meaning "big street" and "little street" – though "Petite Rue" was used for exactly one street, Petite Rue de Monplaisir ("My Pleasure Little Road"). "Grande Rue" was used only 5 times – 6 if you count the street named "Grande Rue" – just "Grande Rue", nothing after it. I suspect this is how to say "Main Street" in French.

There's also a street named "Rue", nothing after it, just "Rue". (That's like us having a street named "Street", nothing else.) I suspected this was an error and tried to fix by using the latitude and longitude to find the street on another map, in this case, Google Maps. It turned out this didn't refer to a node, it referred to a way, and the way was apparently a footpath invisible from the street because it was *inside* a building. (Punch in "45.7430031,4.8388860" into Google Maps and switch to the aerial photo view if you want to see it yourself.) I decided to "Fix" this by putting in the name of the nearest major street, "Rue Lortet", in the parser. I subsequently fixed it more properly by removing the "addr:street" record from the database after the parsing was done.

```
DELETE FROM ways_tags WHERE (id = 44895025) AND (type = 'addr') AND
(key = 'street');
```

Note that this causes the "ways tags" count listed above to decrement by 1! (From 398,394 to 398,393.)

Corrections

For "/25 Grand Rue", I used the latitude and longitude (45.7390562, 4.7721181) to determine that the street was just "Grande Rue". I assumed the /25 was a typo. For "37", I used the latitude and longitude (45.7740714, 4.8081164) to determine that the street was "Grande Rue de Vaise".

By the way, in case you're wondering why I'm not listing queries for finding the latitudes and longitudes of these nodes, it's because I did it the old-fashioned way, by grepping the OSM file. That was because I did this before the SQL database was created. In retrospect it might have been easier to find the coordinates by doing SQL queries. You query for a tag with the error you are looking for and follow the node_id field to the node in question and see what its latitude and longitude are. For

example:

```
SELECT ways.id FROM ways, ways_tags WHERE (ways_tags.type = 'addr')
AND (ways_tags.key = 'street') AND (ways_tags.value = 'Rue') AND
(ways_tags.id = ways.id);
```

This yields the same ID as I found manually from the OSM file shown above, 44895025.

For most of the other corrections, they were just fixing the case. In this project I just made a simple mapping of original text to corrections, because it was a simple way to do all the corrections in one swoop. A more sophisticated way to do it would be to make a system that found uncapitalized versions of common words, such as “rue”, and turn them into their capitalized versions (“Rue”). Other case corrections would still have to be found and fixed by hand. For example, “Boulevard du 11 novembre 1918”. In French, month names are normally not capitalized (making this probably an easy mistake to make in French), but if it's part of a street name, it probably is supposed to be capitalized, so I correct it to “Boulevard du 11 Novembre 1918”.

“Rue moliere”, besides getting the capitalization corrected, needed an accent mark, “Rue Molière”. It would not surprise me if there are hundreds of entries in this data set that are missing accent marks – I just happened to catch this one by chance because I searched on the name in Google Maps. Since I have no general-purpose way of detecting missing accent marks, this is the only one I fixed. A more sophisticated system that checks names against a dictionary would probably catch a lot – but wouldn't work in this case because “Molière” isn't a regular French word that's in the dictionary.

For “Cours DOCTEUR LONG”, I discovered there was another entry, “Cours du Docteur Long”. Before correcting “Cours DOCTEUR LONG” to “Cours du Docteur Long”, I looked to see if “du” is normally used with “Docteur” in street names. I found it is used both ways:

Avenue du Docteur Georges Lévy
Avenue du Docteur Terver
Cours Docteur Jean Damidot
Cours du Docteur Long
Montée du Docteur Mastier
Place Docteurs Charles et Christophe Mérieux
Place du Docteur Frédéric Dugoujon
Place du Docteur Lazare Goujon
Quai du Docteur Gailleton
Rue Docteur Alexis Carrel
Rue Docteur Edmond Locard
Rue Docteur Jean Barbier
Rue Docteur Jean Sauzay
Rue Docteur Laënnec
Rue du Docteur Albéric Pont
Rue du Docteur Crestin
Rue du Docteur Dolard
Rue du Docteur Edmond Locard
Rue du Docteur Fleury-Pierre Papillon
Rue du Docteur Ollier
Rue du Docteur Pierre-Fleury Papillon

Rue du Docteur Pravaz
Rue du Docteur Rollet
Rue du Docteur Salvat

So I decided to check Google Maps to see if “du” was used with *this* particular street name. Google Maps said, “Cours Dr Long”. Interesting that Google Maps used the “Dr” abbreviation where OpenStreetMap never does. Maybe American OpenStreetMap contributors use abbreviations while French OpenStreetMap contributors don't? Anyway, there wasn't any “du” so I corrected both “Cours DOCTEUR LONG” and “Cours du Docteur Long” to “Cours Docteur Long”.

Another perhaps noteworthy correction is “Rue du Docteur Fleury-Pierre Papillon” to “Rue du Docteur Pierre-Fleury Papillon”. Both versions appeared in the dataset and I gambled they refer to the same person. So I checked Google (not Google Maps, just regular Google) and Google seemed to think this fellow's name was Pierre-Fleury. So I corrected Fleury-Pierre to Pierre-Fleury.

I used the latitude and longitude to find where this “Galerie Soufflot” was, and found the “Galerie Soufflot” entries were along a street called “Quai Jules Courmont”. I don't know why someone (I won't say his name to protect the guilty but user=“Olyon”, uid 1443767) put “Galerie Soufflot” in for the street. Maybe there's a gallery with the name Soufflot somewhere around there.

Finally, there's “Caluire-et-Cuire”. Google Maps seemed to think “Caluire-et-Cuire” was the name of a city in the Lyon metropolitan area, not the name of a street. In this dataset “Caluire-et-Cuire” appears in the “addr:city” field a lot, but only once in the “addr:street” field. So the “addr:street” entry is probably an error. Once again, latitude & longitude to the rescue. It turns out that the street is “Quai Clemenceau”. Also, this addr:street is a tag on a relation, not a node or way, so we do this correction for nothing – it gets discarded when we make the CSVs for our SQL database.

One finale finale note before I list the complete set of corrections. I corrected “A 7” to “A7”. A7 is the major freeway that goes through Lyon. Google Maps also shows an A6, but weirdly, that doesn't appear in this dataset.

In the interest of brevity, I'm going to omit the full list of corrections. For the complete set of corrections, consult the Python code file starting on line 66.

Questions & answers

We're asked to report the size of the file, number of unique users, number of nodes and ways, and number of chosen type of nodes, like cafes, shops etc. The size of the file, number of nodes and ways, and some other numbers have already been shown. We haven't seen the number of unique users. We could make a “users” table and make an entry every time we encounter a new user, and then count the size of that table when we're done. If we were going to use that users table for other stuff later on, that might be worth doing. But since we're not going to do that, we're not going to bother, and will just do it with a query. It turns out to be a bit tricky because we have users in both our “nodes” and “ways” tables. If we do an inner join, we'll only get the users in both tables. If we do a right or left join, we'll get all the users in one table, but not the other. We have to do a full outer join to count all the users. I have been using SQL databases for years and have never had an occasion to do a full outer join until

now. We further have to specify we want only unique users. Then we have to do a count on the result of that query. When we put it all together, this is what we get:

```
SELECT COUNT(*)
FROM (SELECT DISTINCT nodes.uid, ways.uid FROM nodes FULL OUTER JOIN
ways ON (nodes.uid = ways.uid) ) sub
WHERE 1;
```

This came back with `sqlite3.OperationalError: RIGHT and FULL OUTER JOINs are not currently supported`.

Well. So much for that. So I tried an alternative, the UNION keyword.

```
SELECT COUNT(*)
FROM (
    SELECT uid FROM nodes
    UNION
    SELECT uid FROM ways
) sub
WHERE 1;
```

This was instant and came back with 1,376.

Of course, one further flaw in this analysis is that we excluded the “relations” table. By modifying the XML parser to count unique users in all tables we can see that the total number of unique users in the original OSM file is 1,405.

I'm not sure what the number of users, or the users themselves, are useful for. Maybe different users enter data with different degrees of accuracy, and it might be possible to figure out which users are the most accurate. I'm not trying to do anything like that here.

Now we have to look at the number of chosen type of nodes, like cafes, shops, etc. Well, those sound like amenities to me, so let's count amenities.

1. bench: 1,886
2. bicycle_parking: 1,615
3. parking_space: 1,563
4. restaurant: 1,100
5. recycling: 774
6. waste_basket: 706
7. post_box: 352
8. bicycle_rental: 331
9. cafe: 306
10. fast_food: 277

The query we used to get this was:

```
SELECT value, COUNT(*)
FROM nodes_tags WHERE (type = 'regular') AND (key = 'amenity')
GROUP BY value
ORDER BY COUNT(*) DESC;
```

But wait! That's just the number of amenities in the nodes tags. There are more in the ways tags! Let's have a look.

```
SELECT value, COUNT(*)
FROM ways_tags
WHERE (type = 'regular') AND (key = 'amenity')
GROUP BY value
ORDER BY COUNT(*) DESC;
```

Top 10 amenities ways with number of occurrences:

1. parking: 11,68
2. school: 282
3. place_of_worship: 144
4. kindergarten: 68
5. bench: 49
6. parking_space: 44
7. shelter: 44
8. toilets: 35
9. restaurant: 34
10. hospital: 31

So, some similarities but some differences, too. More kindergartens.

Just for kicks, and because it's now so easy, let's do the same thing for “building”, “highway”, “name”, “wall”, “natural”, and “surface”. (I'm skipping addr:housenumber and addr:street from the top 10 list above, as well as “source”, which isn't about things in the real world but about where data came from, things like “Grand Lyon - 10/2011” and “Métropole de Lyon 2018”). All we're going here is replacing “amenities” with the new category, e.g. “building”, in the SQL queries above, for example:

```
SELECT value, COUNT(*)
FROM nodes_tags
WHERE (type = 'regular') AND (key = 'building')
GROUP BY value
ORDER BY COUNT(*) DESC;
```

Top 10 buildings nodes with number of occurrences:

1. yes: 25
2. college: 5
3. public: 5
4. chapel: 4
5. house: 4
6. dormitory: 3

7. garage: 2
8. residential: 2
9. school: 2
10. service: 2

Top 10 buildings ways with number of occurrences:

1. yes: 99,208
2. apartments: 2,286
3. house: 419
4. garages: 352
5. residential: 323
6. school: 229
7. roof: 156
8. office: 131
9. church: 95
10. retail: 89

Ok, so if something is a building and you don't want to say what kind, I guess you just say “yes”? That's kind of funny. Most buildings, almost all, are ways not nodes. In the OpenStreetMap system, anyone can put tags on anything, though, so you can't say putting the “building” tag on ways is the standard way. But it looks like it's pretty close to the standard way.

Top 10 highways nodes with number of occurrences:

1. crossing: 6,850
2. traffic_signals: 1,518
3. bus_stop: 1,340
4. street_lamp: 572
5. stop: 268
6. give_way: 254
7. distance_marker: 88
8. turning_circle: 81
9. mini_roundabout: 55
10. elevator: 45

Top 10 highways ways with number of occurrences:

1. service: 6,069
2. footway: 5,571
3. residential: 3,589
4. unclassified: 1,645
5. steps: 1,306
6. primary: 1,113
7. tertiary: 939
8. secondary: 849
9. pedestrian: 835
10. path: 638

So nodes and ways both use the “highways” tag a lot, with ways using it a bit more. Nodes and ways use them differently, however, with nodes having tags for crossings, traffic signals, bus stops, street lamps, and the like, while ways use it for, well, I'm not sure what “service” means, but after that we have footways, residential highways (I'm guessing that means residential streets, not literally highways), unclassified highways (whatever those are), steps (does Lyon really have that many steps?), as well as “primary”, “secondary”, and “tertiary”, which I'm guessing distinguish major highways from minor ones.

Now for names... I'm going to skip names because the names were all names I didn't know, like “Panneau Affichage Libre”, “Station Bluely”, and “Gare de Vaise”. So I can't get any meaningful insight from that without doing research into each of the names.

Moving on to walls.

Top 10 walls nodes with number of occurrences:

(none!)

Top 10 walls ways with number of occurrences:

1. no: 22,408
2. yes: 5
3. stone: 3
4. noise_barrier: 2
5. castle_wall: 1

So walls is kind of funny. No nodes have “wall” tags. 22,408 ways have “wall” tags that are simply “no”. So the reason “wall” even showed up in our top 10 list is because of the 22,408 “no”s. How whacky is that? Then there's 5 “yes”, 3 “stone”, 2 “noise_barrier”, and one “castle_wall”. So, hardly any real walls.

Top 10 natural features nodes with number of occurrences:

1. tree: 13,534
2. stone: 1
3. wild_animals: 1

Top 10 natural features ways with number of occurrences:

1. tree_row: 135
2. wood: 83
3. scrub: 61
4. water: 59
5. grassland: 13
6. cliff: 12
7. lakebank: 5
8. heath: 2
9. beach: 1
10. tree: 1

Ok, so “natural” made the top 10 list because people bothered to tag 13,534 individual trees.

Some of the ways tagged with “scrub” or “grassland” might be more interesting.

Finally, let's look at “surface”.

Top 10 surfaces nodes with number of occurrences:

1. asphalt: 24
2. concrete: 13
3. compacted: 3
4. grass: 1
5. gravel: 1
6. paved: 1
7. unpaved: 1

Top 10 surfaces ways with number of occurrences:

1. asphalt: 7,531
2. concrete: 452
3. paved: 194
4. paving_stones: 166
5. gravel: 157
6. compacted: 115
7. sett: 112
8. fine_gravel: 92
9. grass: 77
10. ground: 71

I never heard of “sett” so I looked it up. The OpenStreetMap wiki says, “Sett paving, formed from natural stones cut to a regular shape. The stones do not cover the surface completely, unlike paving_stones.” It links to Wikipedia which says, “A sett, usually referred to in the plural and known in some places as a Belgian block or sampietrino, is a broadly rectangular quarried stone used for paving roads. Formerly in widespread use, particularly on steeper streets because setts provided horses' hooves with better grip than a smooth surface, they are now encountered rather as decorative stone paving in landscape architecture. Setts are often referred to as 'cobbles', although a sett is distinct from a cobblestone in that it is quarried or worked to a regular shape, whereas the latter is generally a small, naturally-rounded rock. Setts are usually made of granite.” And the Wikipedia page has some pictures.

https://wiki.openstreetmap.org/wiki/Key:surface

https://en.wikipedia.org/wiki/Sett_(paving)

Validity, Uniformity, Consistency, Accuracy, Completeness

In the project description, it says question responses should take about 3-6 pages, but I couldn't find any questions that would take that much space to answer. Most of the questions I've done so far are questions about the data I basically made up myself (with the exception of a few like the number of unique users). (If those count, I'm good, because I'm at 5 pages.) So I thought, maybe the project description is referring to validity, uniformity, consistency, accuracy, and completeness. (If the whole thing is supposed to be limited to 6 pages, I'm going to get in trouble because I'm going to run over.)

Validity – validity here means "conforms to a schema". I would say this data definitely conforms to a schema. We were able to parse 249.8 MB of data without running into anything that violated the schema. So, all good here.

Uniformity – uniformity here means "all units the same". Well, the only numerical units here are latitude and longitudes. If any latitude or longitude numbers were in different units, the numbers would be outside the box we asked for. So lets find the minimum and maximum latitudes and longitudes and see if anything fell outside the box.

- Minimum latitude: 45.7070412
- Maximum latitude: 45.809566

- Minimum longitude: 4.770233
- Maximum longitude: 4.899329

That all fits with the box of the query we started with, which if you don't recall, was (node(45.707041, 4.770233, 45.809566, 4.899329);<;);out meta;

The queries used to get the numbers were:

```
SELECT MIN(lat) FROM nodes WHERE 1;
SELECT MAX(lat) FROM nodes WHERE 1;
SELECT MIN(lon) FROM nodes WHERE 1;
SELECT MAX(lon) FROM nodes WHERE 1;
```

Consistency – consistency here means "data in one part of dataset matches relevant other part of dataset". The data here seems to be structured in such a way that there isn't any duplication of data, so there isn't any opportunity for two pieces of data to be in conflict and to require a human to determine which is more "authoritative". In database terms, we would say it is properly normalized. So I would say this data is 100% consistent.

However, in a less strict sense of the idea of "consistency", I would say this data isn't very consistent. Because so many different users entered the data, and the OpenStreetMaps project lets people enter data in such a very open-ended way, with no corporate overload, that the data is not entered in a consistent way. You see some of that in the observations above, but where I think you see it a lot more is when, instead of looking at the "top 10" things, were there has to be more consistency for the numbers to add up, you look at the bottom of the lists. There's tags (keys) like "sleeping", "highway_1", "abandoned:highway" (which should probably be "highway" = "abandoned"), "lanes:psv:forward", and "parking:condition:right:fee_per_hour". So in this less technical sense of the

term “consistency”, this dataset is inconsistent.

Accuracy – accuracy here means testing against a “gold standard”. I've mentioned Google Maps numerous times, and I've used it for years, and while it doesn't seem perfect – it is possible to encounter mistakes – and it's pretty hard to encounter mistakes and what's on the maps fits my experience of the “real world” where I live. So I'm going to treat Google Maps as the “gold standard” and look up a few points from OpenStreetMaps and see if I see what I expect when I switch to the arial view or Street View in Google Maps.

Let's start with the “castle_wall”. I don't know if you noticed in the lists above that a “castle_wall” showed up, but I sure did. It made me wonder, is there really a castle wall in Lyon? Let's get some latitude and longitude coordinates and see what it looks like on Google Maps. First, we need to find the way ID of the alleged castle wall. We do that with this query:

```
SELECT id FROM ways_tags WHERE (type = 'regular') AND (key = 'wall')
AND (value = 'castle_wall');
```

This quickly reveals the way ID is 264154511. Next we plug that into a query to get nodes that belong to the way and the latitude and longitude coordinates from those nodes. We do that with this query:

```
SELECT nodes.lat, nodes.lon FROM ways_nodes, nodes WHERE
(ways_nodes.id = 264154511) AND (ways_nodes.node_id = nodes.id) ORDER
BY ways_nodes.position;
```

Which gives us these latitude and longitude coordinates for the castle wall:

1. 45.7363943, 4.7702885
2. 45.7363944, 4.7703374
3. 45.7364345, 4.7703465
4. 45.7364886, 4.7703532
5. 45.7365617, 4.7703392
6. 45.7366143, 4.7702874

Incidentally, the way has 11 points in the database, but we only get 6 back. I think that's because the nodes for the missing points are just below our longitude minimum. That means part of the wall is just west of our bounding box. It appears that when we do the original query with the Overpass API, and get the original OSM file, we give it a bounding box, and it will only give us nodes that are within the box, but it will give us ways if *any* of its nodes are within the box.

Nonetheless, we have enough coordinates to find on Google Maps. I grabbed the first one and plugged it in. As expected it is on the very western end of our bounding box – and near the southern end, too. It's in the corner and not technically in the city of Lyon. It loos like it's in a town called Francheville le Bas. Let's zoom in and flip to aerial view. Holy moly Batman, it looks like a castle wall! It looks like maybe it's extending up from the roof of a building. But when we switch to Street View and have a look from the nearby street (Grand Rue), we see that the castle wall is just on top of a pile of rock.

You should be able to just bang this link and see it:

https://www.google.com/maps/place/45%C2%B044'11.0%22N+4%C2%B046'13.0%22E/@45.7364904,4.7701021,62m/data=!3m1!1e3!4m5!3m4!1s0x0:0x0!8m2!3d45.7363943!4d4.7702885

Let's do one more and call it a day. Let's do a tunnel. I first looked to see if there were any nodes tagged with “tunnel” but there weren't. So I looked for ways and found some. Here's the query:

```
SELECT id, value FROM ways_tags WHERE (type = 'regular') AND (key =
'tunnel');
```

There were 689 tunnels found, so I won't include them here. But I will mention that the “tunnel” key had 3 values, “yes”, “building_passage”, and “culvert”. (Plus three “no” entries – weird).

I decided to generate a random number from 1 to 689, and I put the list in a file, so when I got the number I could just go to that line in the file. The number was 556. Line 556 was way id 442928017. This might not happen when you run the code because I didn't use an “ORDER BY” clause, so the arrangement of the records when returned from the database is indeterminate. Usually you don't want that, but since I'm picking at random, it's fine here. The “tunnel” key for this entry has the value “building_passage”.

Ok, let's get the nodes and have a look and see what's there. Here's the query to get the nodes:

```
SELECT nodes.lat, nodes.lon FROM ways_nodes, nodes WHERE
(ways_nodes.id = 442928017) AND (ways_nodes.node_id = nodes.id) ORDER
BY ways_nodes.position;
```

Here's the latitude and longitude coordinates for the tunnel we hand-picked:

1. 45.7637991, 4.8284233
2. 45.7637891, 4.828538

Since there are just two, I'm guessing they're for each end of the tunnel, and it is a short tunnel. Let's plug them into Google Maps and see what's there.

Well. It looks like the “tunnel” just goes from one side of a building to another. It looks like all it does is allow people from the street to enter a courtyard type area in the middle of some buildings where there are some restaurants, without technically entering the building. Could be a nice place to have lunch away from the hustle and bustle.

Not a very dramatic tunnel but I guess it counts as a tunnel and the “tunnel” key was marked “building_passage”, so I guess that's what a building passage looks like.

https://www.google.com/maps/@45.7638385,4.8285698,3a,75y,22.39h,82.75t/data=!3m8!1e1!3m6!1sAF1QipNaMyM6DUr30IvlRySydaLTwbcuPn7ZdLgx42rG!2e10!3e11!6shttps:%2F%2Flh5.googleusercontent.com%2Fp%2FAF1QipNaMyM6DUr30IvlRySydaLTwbcuPn7ZdLgx42rG%3Dw203-h100-k-no-pi-0-ya111.23999-ro-0-fo100!7i13000!8i6500

Completeness – completeness here means, “do we have all the records we need?” This is the hardest to answer, and I feel at this point, the only truly honest answer I can give is “I don't know.” Certainly we downloaded a boatload of data, it subjectively it *seems* to have a lot of detail.

In the interest of time, I'm going to call it quits here. But before I sign off, I should probably give some thoughts as to what it would take to determine completeness. One way to determine completeness would be to do what we've been doing with Google Maps but in reverse – find things on Google Maps and see if they exist (and are accurate) in this OpenStreetMaps dataset. Google Maps has businesses, so we could go down a variety of streets and look up every single business shown in Google Maps to see if it is also in the OpenStreetMap dataset. One place where we've noticed this kind of discrepancy already is seeing the A7 highway in the dataset but not the A6.

Another thing to consider is the question, “do we have all the records we need?” What we need depends on what we're using the data for. Here, we're not really using it for anything, except practice with data wrangling. If we had a particular use in mind, we would want to scrutinize the data with regard to that specific need. If, for example, we were doing something with tunnels, we might really need to know if we have *all* the tunnels.

Final thoughts

The OpenStreetMaps dataset was a fun dataset. And while I have compared it with Google Maps in this project report, it is clear that it is different from Google Maps. For one thing, it has an extraordinarily simple schema. Google Maps probably has a very complex schema, with layers and layers of data it can unfold over base maps. It is probably full of complex optimizations for speed. (Google Maps is noticeably faster than OpenStreetMaps's website). However, because anybody can put tags for anything in OpenStreetMaps, there's a lot of stuff in it that's not in Google Maps at all. Stuff like building passage tunnels. You can search Google Maps for nearby places to get lunch or go to the cinema. But if you need to know where all the building passage tunnels are, you're out of luck. But with OpenStreetMaps, you can get that data – or contribute it yourself. Very cool.

The other thing about this data, though, besides the bits of random errors that we would expect in a dataset made by 1,000 or so people, is just the weirdness of certain aspects of it. Like having 22,408 tags with “wall” = “no”. Huh? That's whacky. Does every “real world” dataset not only have errors but also inexplicable whackynesses?

In the case of a way (for a castle wall) having 11 points but us only having 6 of them, that seems whacky at first but it turns out there is a good reason for it – some nodes outside the bounding box of our query. Similarly, maybe setting “wall” = “no” has a good explanation if we only knew it?