Lab Project: OpenStreetMap

Zhenfeng Shi, Hongru Zhu, Chang Zhou

5130309777, 5130309784, 5130309787

Abstract

OpenStreetMap powers map data on thousands of web sites, mobile apps, and hardware devices. It is built by a community of mappers that contribute and maintain data about roads, trails, cafs, railway stations, and much more, all over the world. Our task is to load the OpenStreetMap data into MySQL database system, and make suitable tables, indexes and queries for them.

Keywords: OSM, Database

1. Usage

1.1. Environment

Python 3 + pymysql

1.2. File Structure

The following tree demonstrate the useful files in the folders, other files can be ignored.

```
.
├── DataInsertion
|   ├── dumpin.py
|   └── utils.py
├── Query
|   ├── calc_dist.py
|   ├── mysql2xml.py
|   ├── Query.py
|   └── utils.py
├── readme.md
├── Report
|   └── Group10_FinalReport.pdf
├── SZZ_install.py
├── TableCreation
|   ├── create_database.py
|   └── create_tables.py
└── XML
    ├── text2.xml
    └── text.xml
```

Figure 1: Tree for files

1.3. Install

Enter the root path of this project, run the following command in the shell:

```
python SZZ_install.py [-h] [-c host] [-u user] [-p passwd] [-n dbname] [-i input]

                    -c:  host connect, for instance 'localhost'
                    -u:  username for mysql, for instance 'root'
                    -p:  password for mysql, ignore this if no password
                    -n:  name for the new database
                    -i:  inputfile path, for instance '../shanghai_dump.osm'
```

For instance,

```
python SZZ_install -c localhost -u root -n OSM -i data/shanghai_dump.osm
```

1.4. Queries

2. Database Design

2.1. E-R Model

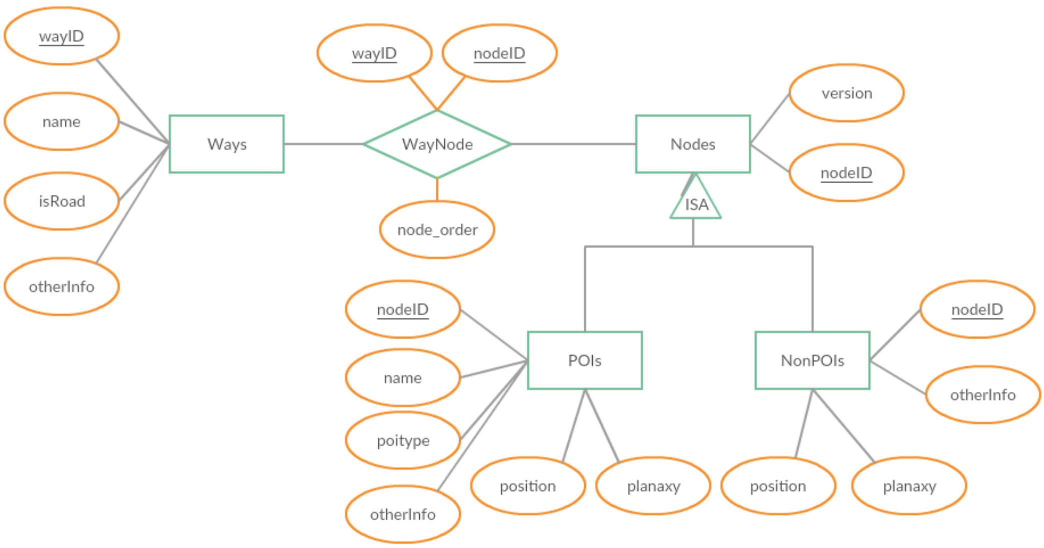

Figure 2: Entity Relationship Diagram

2.2. SQL For Table Creation

Codes are in 'OSM/TableCreation/create_tables.py'

```
CREATE TABLE ways(
           wayID VARCHAR(12),
                    LineString LINESTRING,
                    name VARCHAR(100), INDEX(name),
                    isRoad VARCHAR(100),
                    otherInfo TEXT,
                    PRIMARY KEY(wayID)
                ) ENGINE=MyISAM

CREATE TABLE nodes(
                    nodeID VARCHAR(12),
                    version TINYINT(1), INDEX(version),
                    version BOOLEAN,
                    PRIMARY KEY(nodeID)
                ) ENGINE=MyISAM

CREATE TABLE POIs(
                    nodeID VARCHAR(12),
                    position POINT NOT NULL, SPATIAL INDEX(position),
                    planaxy POINT NOT NULL, SPATIAL INDEX(planaxy),
                    name VARCHAR(100), INDEX(name),
                    poitype VARCHAR(100), INDEX(poitype),
                    otherInfo TEXT,
                    PRIMARY KEY(nodeID)
                ) ENGINE=MyISAM
```

```
create table nonPOIs(
                    nodeID VARCHAR(12),
                    position POINT NOT NULL, SPATIAL INDEX(position),
                    planaxy POINT NOT NULL, SPATIAL INDEX(planaxy),
                    otherInfo TEXT,
                    PRIMARY KEY(nodeID)
                ) ENGINE=MyISAM

create table WayNode(
                     wayID VARCHAR(12), INDEX(wayID),
                     nodeID VARCHAR(12), INDEX(nodeID),
                     node_order INT(2),
                     FOREIGN KEY (nodeID) REFERENCES nodes(nodeID),
                     FOREIGN KEY (wayID) REFERENCES ways(wayID)
                ) ENGINE=MyISAM
```

2.3. Data Insertion

Codes are in ‘OSM/DataInsertion/dumpin.py’

For the data we parsed from XML, we inserted them into corresponding fields of our created tables.

Notably, if we insert the data directly into the table, the insertion time complexity would be $O(log(N))$, where N is the entries already existed in the table, due to the index (primary key) building process.

Therefore, in order to speed up the insertion process, we disable all the keys before the insertion, and enable them after the insertion. This will ensure every row is inserted in time complexity $O(N)$.

The SQL code is as follows:

```
LOCK TABLE ‘nodes‘, ‘pois‘, ‘nonpois‘ WRITE;
ALTER TABLE ‘nodes‘ DISABLE KEYS;
ALTER TABLE ‘pois‘ DISABLE KEYS;
ALTER TABLE ‘nonpois‘ DISABLE KEYS;
/*...insertion...*/
ALTER TABLE ‘nodes‘ ENABLE KEYS;
ALTER TABLE ‘pois‘ ENABLE KEYS;
ALTER TABLE ‘nonpois‘ ENABLE KEYS;
UNLOCK TABLES;
```

The **LOCK TABLE** is to make sure no other users are writing at the same time.

2.4. Index

Besides index for primary keys, we built 8 indexes to accelerate the queries. Especially, in order to speed up the spatial queries, we applied Spatial Index in MySQL. For **MyISAM** tables, Spatial Index creates an R-tree index. The key idea of the R-tree is to group nearby objects and represent them with their minimum bounding rectangle in the next higher level of the tree. For storage engines that support non-spatial indexing of spatial columns, the engine creates a B-tree index. A B-tree index on spatial values is useful for exact-value lookups, but not for range scans. In our cases, the R-tree is more suitable because required query 4, 5, 6 all include range scans.

3. Point Mapping

The longtitude and latitude are used as the absolute coordinates. However, when calculating the distance between two points of given longtitude and latitude, we have to take spherical properties into consideration.

For instance, the distance between (30.4, 122.1) and (30.4, 122.6) is 48.0073 Km. The distance between (32.4, 122.1) and (32.4, 122.6) is 46.995 Km. There would be a error about 1 Km if we ignore the spherical properties of the Earth.

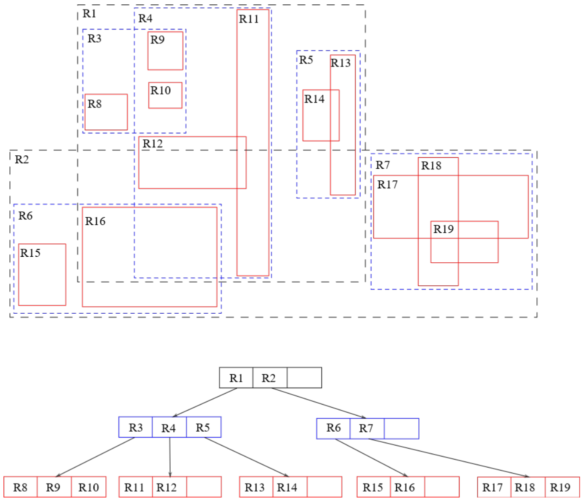

Figure 3: R-tree in 2 dimention

3.1. Different Ways of Calculating Distances Between Two Given Coordinates

a) Vincenty's formulae are two related iterative methods used in geodesy to calculate the distance between two points on the surface of a spheroid, developed by Thaddeus Vincenty (1975). They are based on the assumption that the figure of the Earth is an oblate spheroid, and hence are more accurate than methods that assume a spherical Earth. For simplicity and focus on the course related work, here we only provide the link to the Vincentys paper without further explanation. (http://www.ngs.noaa.gov/PUBS_LIB/inverse.pdf)

b) Another approach is to map latitude-longitude coordinates to plana coordinates and then calculate the distance in between. We used the definition of Millers cylindrical projection, which is more accurate near the equator. Further about the derivation please see the original paper. (http://www.jstor.org/stable/210384)

3.2. Implementation

We randomly sampled the start and the destination and got the distribution of the distance error derived using two methods. We concluded that the Vincenty distance, which is more accurate, was 0.66 1.15 times of the Miller distance. This is further explored in our Query 4 and Query 5 design.

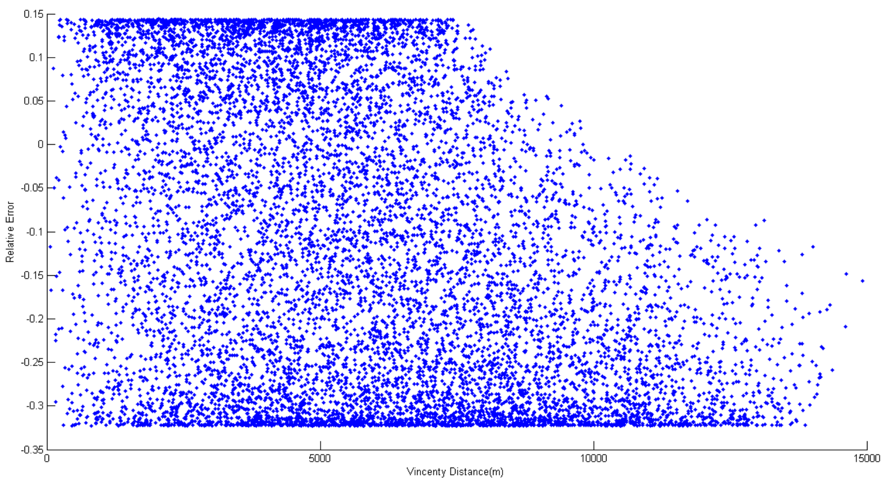

Figure 4: Relative error between miller distance and vincenty distance

4. Solution to Required Queries

Codes are in 'OSM/Query/Query.py'

1. Given a node, return all ways that contain it, and infer whether the node is an intersection of roads, i.e., a crossroad.
   Solution: This query is simple. Based on the given node ID, we go to table 'waynode' to find the corresponding way ID, and extract information about this way ID in table 'ways'.
   We have compared the speed of two queries:

```
SELECT * FROM ways
    WHERE wayID IN
    (SELECT wayID FROM waynode WHERE nodeID=givenNodeID);
or
SELECT wayID, LineString, name, isRoad, otherInfo
    FROM ways NATURAL JOIN waynode
    WHERE waynode.nodeID=givenNodeID;
```

   The corresponding runtimes are $0.72s$ and $8.29s$. Therefore, we choose to use the first solution instead of the second one.
2. Given a way, return all the nodes along the way.
   Solution: To realise this query, we designed a solution to lookup the tables multiple times. Firstly, we go to table 'waynode' to find the corresponding node ID based on the given way ID. Then find the detail information for these nodes in 'POIs' and 'Non-POIs'. Similar to query 1, we find the solution using the following SQL is faster than using JOIN.

```
tmpResult$\leftarrow$SELECT * FROM nodes
                        WHERE nodeID IN
                        (SELECT nodeID
                            FROM waynode
                            WHERE wayID=givenWayID);
result=[];
for row in tmpResult:
  if(row['version']==1):
    tmp$\leftarrow$SELECT nodeID, AsText(position) AS position, name, poitype, otherinfo
                      FROM pois
                      WHERE nodeID=row['nodeID'];
    result.append(tmp);
  else:
    tmp$\leftarrow$SELECT nodeID, AsText(position) AS position, otherInfo
                      FROM nonpois
                      WHERE nodeID=row['nodeID'];
    result.append(tmp);
```

3. Search the name of the road and return information of those matched.
   Solution: This query only involve one table 'ways'. The SQL is simple as follows:

```
SELECT * FROM ways WHERE name LIKE ('%input_string%');
```

   In order to speed up the query process, we built index for 'name' in 'ways'.
4. Query the POIs within a radius of a given location (Longtitude-latitude coordinates).
   Solution: In this query, we first convert given coordinate to 'planaxy', and draw a circle with a larger radius than required, namely 1.33 times of the original radius. This is to ensure that all nodes within the radius under Vincenty distance will always be included. According to our simulation above. Later we use spatial index to scan for all nodes in a polygon that perfectly circumscribe the desired circle. After we got all possible nodes, we use a linear time examination to find all nodes within the original radius. The core SQL commands responsible for this query are:

```
SET @poly='Polygon((x-rad, y+rad,
                    x+rad, y+rad,
                    x+rad, y-rad,
                    x-rad, y-rad,
                    x-rad, y+rad))';
SELECT nodeID, ST_AsText(position), name, poitype
```

```
            FROM POIs
            WHERE MBRContains(ST_GeomFromText(@poly), planaxy);
```

5. Find the closest road to a given GPS coordinate.
   Solution: In this query, we first convert given coordinate to 'planaxy', and draw a circle with a larger radius than required simiar to Query 4. Next we use a iterative method to find at least on NONPOI point which is closest to the tartget GPS coordinate. To improve efficiency and fast our search, we use an exponentially growing raidus, starting from 10 meters as the initial radius. In the i^{th} attempt, we will search in a circle with radius $10e^{i-1}$ meters. We use the same technique as the above to use the spatial index in a polygon and run filters to get desired answer. The core SQL commands responsible for this query are:

```
    SET @poly='Polygon((x-rad, y+rad,
                        x+rad, y+rad,
                        x+rad, y-rad,
                        x-rad, y-rad,
                        x-rad, y+rad))';
    SELECT nodeID, ST_AsText(position)
        FROM nonPOIs
        WHERE MBRContains(ST_GeomFromText(@poly), planaxy);
    SELECT ways.wayid, ways.name, ways.isRoad, ways.otherInfo
        FROM waynode, ways
        WHERE waynode.nodeid=NID and waynode.wayid=ways.wayid and ways.isroad<>'0';
```

6. Implement an API to return the XML in osm format defined in the wiki page, given a rectangular area bounding box (x1, y1, x2, y2) as parameters.
   Solution: In this query, we firstly get all the nodes information from 'nodes', 'pois' and 'nonpois'. Then we go to table 'waynode' and 'ways' to find information of included ways.

```
    SET @poly='Polygon((x1, y1,
                        x1, y2,
                        x2, y2,
                        x2, y1,
                        x1, y1))';
    SELECT nodeID, AsText(position) as position, name, poitype, otherInfo
        FROM pois
        WHERE MBRContains(GeomFromText(@poly), position);
    SELECT nodeID, AsText(position) as position, otherInfo
        FROM nonpois
        WHERE MBRContains(GeomFromText(@poly), position);

    SELECT wayID, name, isRoad, otherInfo
        FROM ways
        WHERE wayID in
        (SELECT DISTINCT wayID
            FROM nonpois
            NATURAL JOIN waynode
            WHERE MBRContains(GeomFromText(@poly), position));
    SELECT wayID, name, isRoad, otherInfo
        FROM ways
        WHERE wayID in
        (SELECT DISTINCT wayID
            FROM pois
            NATURAL JOIN waynode
            WHERE MBRContains(GeomFromText(@poly), position));
```

   Then we follow the format defined in the wiki page to output the xml file. In this process, we query the node IDs corresponding a way ID.

```
SELECT nodeID, node_order FROM waynode
    WHERE wayID=givenWayID order by node_order;
```

5. Extended Queries

6. Human Computer Interaction

7. Division of Work

Zhenfeng Shi: Schema designing; Database, table, index creation; Required query 1, 2, 3, 6.

Hongru Zhu: Algorithms dealing with geo-spatial data designing; Required query 4, 5.

Chang Zhou: Demonstration and interfaces; Extended queries.