Construction of a Road Network Graph from OpenStreetMap Data

Ralf Hartmut Güting, September 2016

1 Introduction

In this document we explain the process of constructing a road network as a directed graph from OpenStreetMap data. The related script in `secondo/bin/Scripts` is `ORGfromOSM.sec`.

2 Input Data and Their Structure

Data can be downloaded from GeoFabrik at `http://download.geofabrik.de/` in the format `.osm.bz2`. Place a file such as `germany-latest.osm.bz2` into a directory, e.g. *Daten* in your home directory and unpack it with the command

```
gunzip germany-latest.osm.bz2
```

resulting in the file `germany-latest.osm`.

The input file is an XML file which contains as a nested structure three kinds of objects:

- nodes
- ways
- relations

A *node* defines a location in geographic space in terms of latitude and longitude. It also has a unique node identifier. A node may have an associated list of node tags. A *node tag* is a (key, value) pair.

A *way* describes a linear geometry, has its own way identifier, and is defined as a sequence of node identifiers. A way may describe a road or the boundary of a park or building, for example. Note that geographic locations are present only in nodes. Hence two roads meeting in a junction would share a node with a unique location. Ways may have associated *way tags* (= (key, value) pairs) as well.

A *relation* has its own identifier and is defined as a list of references to nodes, ways, or relations. Hence a relation can describe a set of anything, even in a nested manner. Relations may also have relation tags.

3 Overview

The script proceeds in the following major steps which are shown and explained below:

1. Process the input file creating a relational representation, consisting of six relations for nodes, node tags, ways, way tags, relations, and relation tags.

2. Create a copy of the node relation such that latitude and longitude are translated into *point* attributes. Further, assign new numeric identifiers that are clustered spatially (similar numbers will lie in similar locations).
3. From the six relations, reconstruct *Ways* as a nested relation. Each way tuple contains a subrelation containing the sequence of nodes defining the geometry of the way. The geometry is also contained as an attribute of type *line*. The way tuple further contains a subrelation for any tags, i.e., (key, value) pairs.
4. Based on tag information, select *Roads* as a subset of Ways.
5. Construct the *Nodes* of the road network graph. Nodes are either *junctions* of two distinct roads, or first or last locations of the way geometry (called *terminal nodes* here).
6. Construct the directed edges of the road network graph. Edges are pieces of roads between two nodes, e.g. between two junction nodes, or between a junction node and a terminal node. This is done in three steps:
    a. Construct the directed edges in the direction the original way is defined over nodes. If a road is a one-way road, then the allowed direction of traversing it corresponds to this order of nodes.
    b. Construct the directed edges in the opposite direction if the road is not a one-way.
    c. Form the union of the two sets of edges constructed in (a) and (b).

4 The Script

4.1 Data Import: Creating Six Relations

This is done by the following query:

```
query fullosmimport('/home/ralf/Daten/germany-latest.osm', "City")
```

The **fullosmimport** operator performs a single scan of the (possibly huge) input file creating six relations simultaneously with schemas:

```
CityNodes(NodeId: longint, Lat real, Lon: real)
CityNodeTags(NodeIdInTag: longint, NodeTagKey: text, NodeTagValue: text)
CityWays(WayId: longint, NodeCounter: int, NodeRef: longint)
CityWayTags(WayIdInTag: longint, WayTagKey: text, WayTagValue: text)
CityRelations(RelId: longint, RefCounter: int, MemberType: text,
  MemberRef: longint, MemberRole: text)
CityRelationTags(RelIdInTag: longint, RelTagKey: text, RelTagValue: text)
```

Several tuples in `CityNodeTags` may refer to the same `NodeId` via `NodeIdInTag`. A way is split into tuples such that each tuple contains one node (`NodeRef`) as well as the order number of this node in the sequence (`NodeCounter`). Again, any number of `CityWayTags` may refer to the same `WayId`. Similar strategies apply to `CityRelations` and their tags. For the construction of the road network we only need `CityNodes`, `CityWays`, and `CityWayTags`.

4.2 Create Spatially Clustered NodesNew

The road network we construct later consists of *Nodes* and *Edges*. Nodes are given by node identifiers and edges are tuples with two attributes *Source* and *Target*, describing a directed edge from node *Source* to node *Target*. We will store edges clustered by *Source* node identifier to enable efficient graph traversal. Nodes of the graph are based on nodes of OpenStreetMap.

At the same time, an interesting query is to retrieve edges in a certain spatial area. If we use the original node identifiers from OpenStreetMap for *Nodes* of the graph, then these numbers are arbitrarily scattered over the entire geographical area. This means that *Edges* from a small area are stored scattered over the entire set of edges. If we retrieve edges from a small area, we need one page access per edge and retrieval is slow.

This is the reason why we assign new node identifiers which are clustered spatially. Hence nodes whose numbers are similar are likely to be close in the geographical space. The following query constructs the new node identifiers in attribute NodeIdNew which is added to node tuples.

```
let CityNodesNew = CityNodes feed
  extend[Easting: .Lon * 1000000, Northing: .Lat * 1000000]
  extend[Box: rectangle2(.Easting, .Easting, .Northing, .Northing)]
  sortby[Box]
  projectextend[NodeId;  Pos: makepoint(.Lon, .Lat)]
  addcounter[NodeIdNew, 1]
  consume
```

The query first adds to each tuple of *CityNodes* attributes *Easting* and *Northing* which are scaled-up from *Lon* and *Lat*, respectively. It then adds a bounding box *Box* for Easting and Northing (a degenerated rectangle). The stream of tuples is then sorted by this *Box* attribute.

What does it mean to sort a stream of tuples by a rectangle attribute? Rectangles are mapped back into points (e.g. the center) and point values are sorted into *z*-order (see e.g. [Or90]). With the **addcounter** operator we add a numbering attribute to the ordered stream of tuples and use these values as new node identifiers; these are clustered spatially.

Note that sorting coordinates into z-order only works on the integer part of the coordinates. This is why scaling up the numbers initially is necessary.

4.3 Create Ways

In the next step we reconstruct the linear features, called *Ways*.

```
let Ways =
  CityNodesNew feed
  CityWays feed itHashJoin[NodeId, NodeRef] sortby[WayId, NodeCounter]
    nest[WayId; NodeList]
    extend[Curve  : .NodeList afeed projecttransformstream[Pos]
      collect_line[TRUE]]
  CityWayTags feed nest[WayIdInTag; WayInfo] itHashJoin[WayId,
    WayIdInTag]
  extend[Box: bbox(.Curve scale[1000000.0])]
  sortby[Box] remove[Box]
  consume
```

CityWays tuples contain references to nodes and they are now joined with the nodes containing the geographic points. They are ordered by WayId and sequence number of the node. This means, all tuples for a given way now appear consecutively in the stream of tuples and in the correct order of nodes. They are now **nest**-ed by *WayId* which means that for a given *WayId* only one tuple with the *WayId* attribute is kept at the top level and all the remaining attributes (the sequence of nodes, in this case) go into a subrelation called *NodeList*. From the *NodeList* a *line* value is computed as attribute *Curve*; the line is simply given by the sequence of node positions.

From the *CityWayTags* in a second step a nested relation is computed which has for each *WayId* a subrelation for the tags. This nested relation is joined with the one resulting from the first step via *WayId*.

Finally, the resulting tuples (one per *Wayid*) are sorted into z-order similarly to the previous subsection.

4.4 Select Roads

The next step is simple: from the *Ways* we select *Roads* by the property that a *highway* tag exists.

```
let Roads = Ways feed
  filter[.WayInfo afeed filter[.WayTagKey = "highway"] count > 0]
  consume
```

4.5 Construct Nodes

Remember that the *nodes* making up a way describe its complete geometry. In contrast, we now want to determine a subset of the nodes as *Nodes* of the graph, namely junction nodes of two roads, start nodes of a road, and end nodes of a road.

```
let Nodes =
  CityWays feed
  CityWays feed {h2}
  itHashJoin[NodeRef, NodeRef_h2]
  filter[.WayId # .WayId_h2]
  CityNodesNew feed
  itHashJoin[NodeRef, NodeId]
  Roads feed project[WayId] {r1} itHashJoin[WayId, WayId_r1]
  Roads feed project[WayId] {r2} itHashJoin[WayId_h2, WayId_r2]
  project[WayId, NodeCounter, NodeIdNew, Pos]
Roads feed
  projectextend[WayId; Node: .NodeList afeed filter[.NodeCounter = 0]
    aconsume]
  unnest[Node]
  project[WayId, NodeCounter, NodeIdNew, Pos]
  concat
Roads feed
  extend[HighNodeNo: (.NodeList afeed count) - 1]
  projectextend[WayId; Node:  fun(t: TUPLE)
    attr(t, NodeList) afeed filter[.NodeCounter = attr(t, HighNodeNo)]
    aconsume]
  unnest[Node]
  project[WayId, NodeCounter, NodeIdNew, Pos]
```

```
        concat
        sortby[WayId, NodeCounter]
        rdup
        consume
```

This query consists of three parts, namely

```
  1. CityWays feed ...
  2. Roads feed ...
  3. Roads feed ...
```

computing these three sets. The first part finds pairs of *CityWays* tuples with the same node identifier but different way identifiers. Such pairs are joined with the node (new version). They are further joined with Roads on each involved way identifier to make sure that it is actually a junction of roads, not of arbitrary ways. The second part finds start nodes of roads and the third part end nodes, respectively. All found *Nodes* are brought into the same format, sorted by *WayId* and *NodeCounter*, and duplicates are eliminated.

Note that junction nodes are created twice, once for each involved *Wayid*. Hence for each *Wayid* we now have a sequence *start node*, *junction node* 1, ..., *junction node n*, *end node* (where junction nodes may be missing or coincide wwith end nodes)

4.6 Construct Edges

The task is now to construct edges of the road network, that is, pieces of road between start/end points and junctions. For a given way (*WayId*), we have on the one hand the sequence of *nodes* (coming from *CityNodes*) and the sequence of *Nodes* derived in the previous step (see Figure 1).

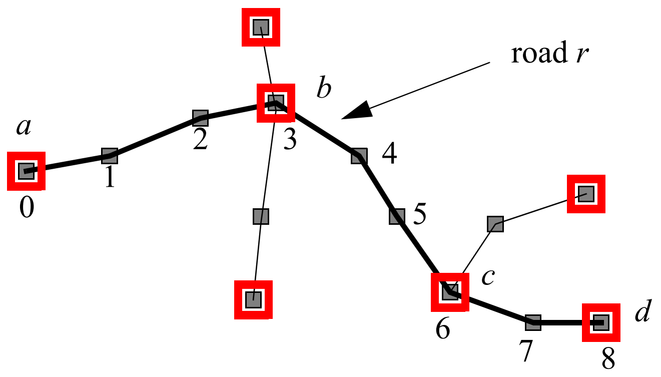

Figure 1: A way/road (drawn fat), its nodes (small, black) and its Nodes (large, red). Labels *a*, *b*, *c*, *d* represent node identifiers.

For the road *r* shown in Figure 1, we need to construct three directed edges in *Up* direction (the direction of increasing node numbers), namely *a-b*, *b-c*, and *c-d* as well as the reverse edges in *Down* direction, if this road does not happen to be a one-way.

4.6.1 Up Edges

```
    let EdgesUp =
      Nodes feed nest[WayId; SectionNodes]
        projectextend[WayId; Sections: .SectionNodes afeed
    extend_last[Source: ..NodeIdNew::0, Target: .NodeIdNew::0,
      SourcePos: ..Pos::[const point value undef],
      TargetPos: .Pos::[const point value undef],
      SourceNodeCounter: ..NodeCounter::0,
      TargetNodeCounter: .NodeCounter::0]
    filter[.Source # 0]
    project[Source, Target, SourcePos, TargetPos,
      SourceNodeCounter, TargetNodeCounter]
    aconsume]
```

The query processes the stream of *Node* tuples which is ordered by *Wayid* and *NodeCounter* from the previous step. It is nested by *WayId* so that we have one tuple for the road with subrelation *SectionNodes*. For the road *r* of Figure 1, the subrelation would have four tuples. In the **projectextend** operator, the subrelation *SectionNodes* is processed. The **extend_last** operator allows one to construct new attributes based on attributes of the current tuple (accessed by `.<attr>`) as well as attributes of the preceding tuple (accessed by `..<attr>`) within a stream of tuples. For example, for the second tuple of the stream for road *r*, *Source* is set to *a*, *Target* to *b*, *SourcePos* to the position (*point*) of *a*, *TargetPos* to the position of *b*, *SourceNodeCounter* to 0, and *TargetNodeCounter* to 3. The first tuple is a special case, as there is no preceding tuple in the stream; in this case the new attribute values are taken from the constant specified behind the :: notation. The first tuple is later eliminated by the filter condition. Hence as a result, three tuples are constructed and stored in a new subrelation *Sections*, each describing an edge.

```
Roads feed {r}
itHashJoin[WayId, WayId_r]
projectextend[WayId; Sections: fun(t:TUPLE)
  attr(t, Sections) afeed
  extend[
    Curve: fun(u: TUPLE)
    attr(t, NodeList_r) afeed
      filter[.NodeCounter_r between[attr(u, SourceNodeCounter),
        attr(u, TargetNodeCounter)] ]
      projecttransformstream[Pos_r]
      collect_sline[TRUE],
    RoadName: attr(t, WayInfo_r) afeed filter[.WayTagKey_r = "name"]
      extract [WayTagValue_r],
    RoadType: attr(t, WayInfo_r) afeed filter[.WayTagKey_r = "highway"]
      extract [WayTagValue_r]
  ]
  aconsume ]
unnest[Sections]
consume
```

Continuing the query, the *Roads* are joined by *WayId* so that now we have a tuple with the complete road information as well as the sequence of edges for this road. In the **projectextend** operator, the *Sections* subrelation is transformed: for each edge (input *Section* tuple), the geometric *line* value for this piece of road is constructed in attribute *Curve*; this is done by scanning the node list of the road between the *SourceNodeCounter* and the *TargetNodeCounter*, feeding node positions into the **collect_line** operator. Moreover, the edge is extended by the name of the road and its type. Finally, the stream of tuples, each with a subrelation *Sections*, is **unnested** so that the output are flat tuples that can be collected into a regular relation.

4.6.2 Down Edges

```
let EdgesDown =
  Nodes feed nest[WayId; SectionNodes]
    projectextend[WayId; Sections: .SectionNodes afeed
      sortby[NodeCounter desc]
      extend_last[Source: ..NodeIdNew::0, Target: .NodeIdNew::0,
        SourcePos: ..Pos::[const point value undef],
        TargetPos: .Pos::[const point value undef],
        SourceNodeCounter: ..NodeCounter::0,
        TargetNodeCounter: .NodeCounter::0]
      filter[.Source # 0]
      project[Source, Target, SourcePos, TargetPos,
        SourceNodeCounter, TargetNodeCounter]
      aconsume]
  Roads feed
    filter[.WayInfo afeed filter[.WayTagKey = "oneway"]
      filter[(.WayTagValue = "yes")] count = 0] {r}
  itHashJoin[WayId, WayId_r]
  projectextend[WayId; Sections: fun(t:TUPLE)
    attr(t, Sections) afeed extend[Curve: fun(u: TUPLE)
      attr(t, NodeList_r) afeed sortby[NodeCounter_r desc]
        filter[.NodeCounter_r between[attr(u, TargetNodeCounter),
          attr(u, SourceNodeCounter)] ]
        projecttransformstream[Pos_r]
        collect_sline[TRUE],
      RoadName: attr(t, WayInfo_r) afeed filter[.WayTagKey_r = "name"]
        extract [WayTagValue_r],
     RoadType: attr(t, WayInfo_r) afeed filter[.WayTagKey_r = "highway"]
        extract [WayTagValue_r]
    ]
    aconsume ]
  unnest[Sections]
  consume
```

The construction of reverse (*Down*) edges is almost the same. The difference is that *Nodes* are now processed in the opposite order (via `sortby[NodeCounter desc])` and *nodes* as well. Furthermore, *Roads* are checked to not contain a (key, value) pair (*oneway*, *yes*) in which case *Down* edges must not be constructed.

4.6.3 Edges

```
let Edges = EdgesUp feed EdgesDown feed concat
  projectextend[Source, Target, SourcePos, TargetPos, SourceNodeCounter,
TargetNodeCounter, Curve, RoadName,
    RoadType; WayId: .WayId]
  consume
```

Finally, the set of *Edges* is constructed as the union of *Up* and *Down* edges, putting the *WayId* attribute to the end of tuples, in order to have *Source* and *Target* as the first two attributes.

References

[Or90] Jack A. Orenstein: A Comparison of Spatial Query Processing Techniques for Native and Parameter Spaces. SIGMOD Conference 1990: 343-352.