From the Open Street Map to Km4City street graph: a process outline

Version 0.3

Date: 21-01-2018

Ref: info@disit.org

Up to April 2017, the Km4City street graph data (toponyms, street numbers, Public Administration boundaries, traffic restrictions, and so on) was gathered from the Tuscany Region through the employment of appropriate ETL processes that basically imported the Tuscany Region data to appropriate MySQL tables, and translated it to an appropriate set of RDF triples through appropriate executions of an appropriately configured Karma[1]. The intentional repetition of the *appropriate* adjective is aimed at clarifying that even if that process is out of the scope of this document and it is therefore relegated to just a bunch of lines, nothing has been easy within it, nor its design, neither its implementation.

From April 2017 onward, a new need has emerged: the coverage of the Km4City project had to be extended outside of the boundaries of the Tuscany Region. Therefore, other source data had to be identified, and corresponding RDF triples had to be generated.

The gathering of the data from the Public Administration through requests to the Administrations other than the Tuscany Region, would have implied to design and develop a new ETL ingestion for each new data source. Also, bureaucratic matters would have to be faced, which was a threaten both for the time of release and for the availability and affordability of the necessary data over the years.

All such issues have been overcome by adopting the *Open Street Map*[2] as a source of data. Open Street Map (briefly, OSM) is an open source project aimed at creating maps that could include street graphs, street numbers, traffic restrictions, boundaries, points of interest, and so on. The Open Street Map data come from a wide community of voluntaries spread all over the World, weakly coordinated through the OSM Wiki[3].

This way, just one ingestion process had to be designed and implemented, being that the Open Street Map covers the Globe in its entirety. Also, the wide community of the Open Street Map contributors is a guarantee about the availability of updated data over the time. Also, the open license under which the OSM maps are distributed is a guarantee about the affordability of such data. Finally, the Open Street Map comes with a set of tools, the most relevant of those for the purposes of the ingestion and update of the Open Street Map data is the Osmosis[4].

In Figure 1, an outline is provided of the process through which the Open Street Map data is leveraged for populating the Km4City Knowledge Base.

Firstly, an appropriate Open Street Map file has to be downloaded. Different formats are available, and the opportunity of getting just an extract instead of the complete Planet map is also provided by some websites such as Geofabrik[5]. Then, a relational database with an appropriate schema has to be created. A set of SQL scripts is provided by Osmosis for the purpose, that address two alternative schemas. Some of those install optional functionalities, and can be skipped. Appropriate decisions must be taken depending of the purposes of the data ingestion. Then, the relational database can be filled reading from the OSM file

[1] http://usc-isi-i2.github.io/karma/
[2] https://www.openstreetmap.org/
[3] https://wiki.openstreetmap.org/wiki/Main_Page
[4] http://wiki.openstreetmap.org/wiki/Osmosis
[5] https://www.geofabrik.de/

(Osmosis suits for the purpose), and an appropriate SQL script that we at DISIT have developed can be launched that creates indexed tables that will be leveraged in the following and adds appropriate indexes to some of the tables that have been filled through the data import from Open Street Map. Once that the relational database has been filled, the RDF triples can be generated. The triplification of the street graph is performed in two steps: appropriate SQL scripts that we at DISIT have developed are executed on the same database that has been filled with Open Street Map data, and a Sparqlify dump is then launched that reads from the relational database and writes the RDF triples conforming to an appropriate configuration file that we at DISIT have shaped, namely a SML (Sparqlification Mapping Language) file. This way, an increased efficiency is achieved: the SQL queries included within the SML configuration file become long simpler, and their optimization become long faster. At last, the new triples can be loaded to Virtuoso, the NoSQL DBMS in use within the Km4City project, and the process ends. Periodically, the process can be repeated: an OSM file that contains the map updates (namely, an OSC file) can be downloaded, the updates can be applied to the RDB through Osmosis, the efficient SQL scripts aimed at preparing the data for the triplification can be executed, the Sparqlify dump can be launched, and the new triples can be uploaded.

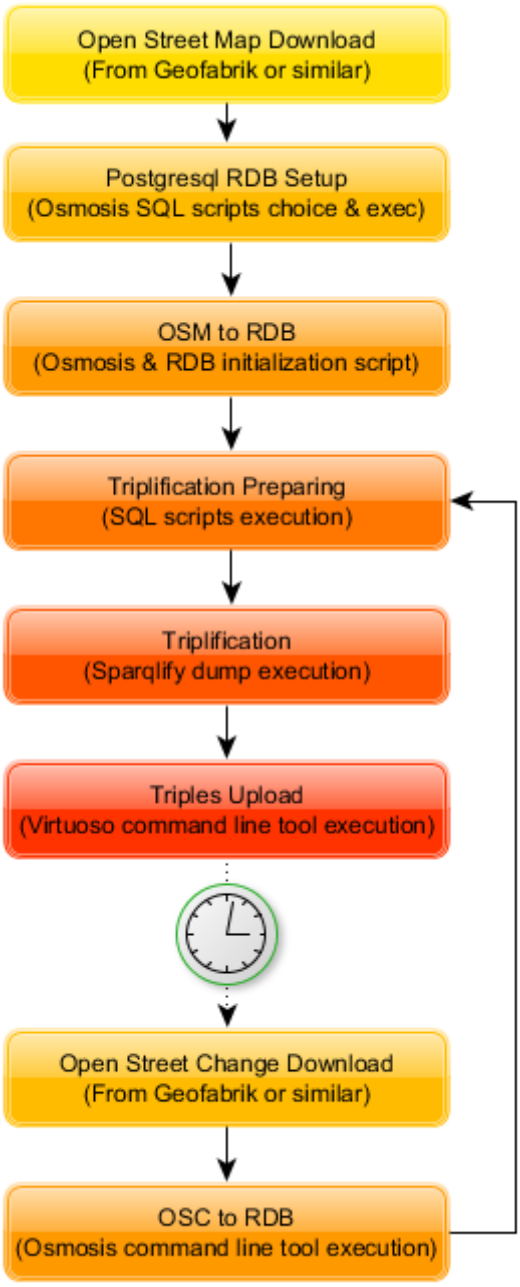

Figure 1 The OSM data ingestion process

In the following, each of the outlined steps is discussed in greater detail.

As for the acquisition of the maps from Open Street Map, it must be said that the Open Street Map data is distributed in several different formats[6]. Indeed, the same OSM map could be found represented in as many as seven different ways. As a result, a decision about the source data format had to be taken. Among the OSM file formats, two are compressed not human-readable file formats thought to be used with specific software tools, and the others are the *OSM XML* (the basic OSM format), the *Overpass JSON* (a JSON literal translation of the OSM XML), and the *Level0L* (a plain text representation generated by the Level0 map editor). In any case, none of those could be directly used for generating RDF triples, for a wide set of motivations. Firstly, the mapping of the Open Street Map data model to the Km4City data model is very intricated, and good chances were that a very hard to maintain mapping tool could result from an attempt of performing a one-step implementation of such mapping. Also, due to the complexity of the mapping, a sequential access to the OSM source file was not a choice, but the OSM files are so large that loading them in memory for enabling a random access would have easily lead to memory issues. Also, geospatial functions (such as determining the boundary where a point falls choosing among thousands of candidate boundaries, determining the nearest line given an arbitrary point choosing among millions of candidate lines, and many other) are needed for a flexible and effective data import, but ready-to-use geospatial functions are not available that efficiently spread across a whole XML, JSON or plain text files. Also, an official DTD schema is not available for the OSM XML. Therefore, no assumption can be made while parsing it. As a result, nor a validation neither a safe navigation is possible over the original OSM XML format, and the same applies to the JSON and plain text files derived from it, which leads to an increased complexity of the import tool, where the absence or the corruption of the data would have to be handled. All this has led to the choice of making a preliminary import to a relational database, and using the relational database as a source, instead of the OSM files directly. The RDBMS that Open Street Map recommends[7] is Postgresql[8], also considering its very useful PostGIS[9] extension (**Errore. L'origine riferimento non è stata trovata.**). As a result, it has been our choice. Osmosis is suitable for filling a relational database reading from an OSM file[10], but the destination database must exist and it is required to have an appropriate schema[11]. Osmosis supports two different schemas, and provides appropriate sets of scripts for building each of them. Within each set, optional scripts are included, aimed at supporting actions, bboxes, and linestrings.

[6] http://wiki.openstreetmap.org/wiki/OSM_file_formats
[7] http://wiki.openstreetmap.org/wiki/Databases_and_data_access_APIs#Choice_of_DBMS
[8] https://www.postgresql.org/
[9] http://postgis.net/
[10] https://wiki.openstreetmap.org/wiki/Osmosis/Detailed_Usage_0.41#--write-pgsimp_.28--ws.29
[11] https://wiki.openstreetmap.org/wiki/Osmosis/PostGIS_Setup

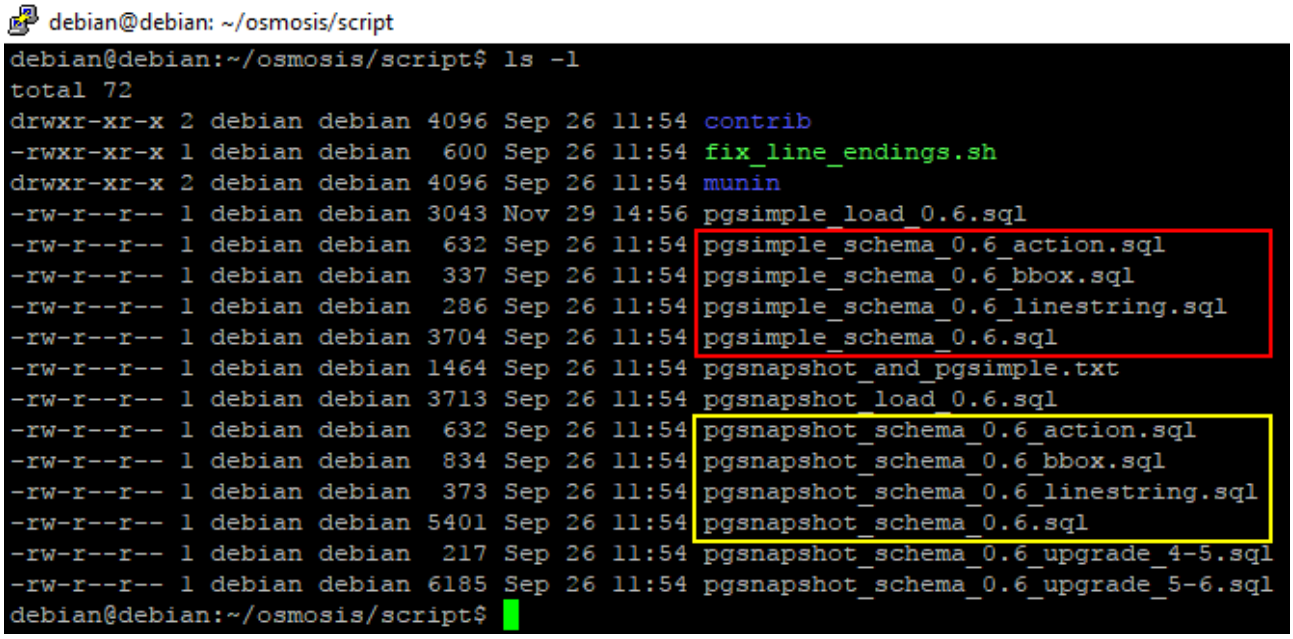

Figure 2 The Osmosis scripts for generating the two supported RDB schemas

If the *snapshot* schema is chosen, the OSM tags, that is the metadata that apply to the OSM relations, ways, and nodes, where most of the information that is necessary for a mapping implementation is wrapped, are concatenated in plain text fields. So, an effective indexing cannot be performed over them, and a (time and resource consuming) parsing is needed for identifying and extracting the key-value pairs. As a result, the older *simple*[12] RDB schema has been preferred, and the SQL scripts wrapped within the red box in Figure 2 are executed for generating such a schema.

After having populated the RDB with the OSM data, the initialization script that we at DISIT have developed can be launched. The initialization script typically need to be launched once per RDB, immediately after that the first filling of the RDB with OSM data has been completed. Further executions of the script are indeed necessary in the only case in which a modification would occur to the boundaries of one or more of the districts, municipalities, provinces and regions that have been imported from the Open Street Map. The initialization script is indeed mainly aimed at performing long running operations on the boundaries (districts, municipalities, provinces, regions) that can be found within the imported Open Street Map, and at defining appropriate indexes both over the newly created tables and over some in particular of the tables and fields that have been introduced and filled by Osmosis during the generation of the schema and the subsequent data import. This way, a remarkable speeding has been achieved of several queries that are executed at each triplification process, such as the ones where the Municipality within which a point falls has to be retrieved, and many other. A fragment of the initialization script is provided in Figure 3.

[12] http://wiki.openstreetmap.org/wiki/Osmosis/Detailed_Usage_0.45#PostGIS_Tasks_.28Simple_Schema.29

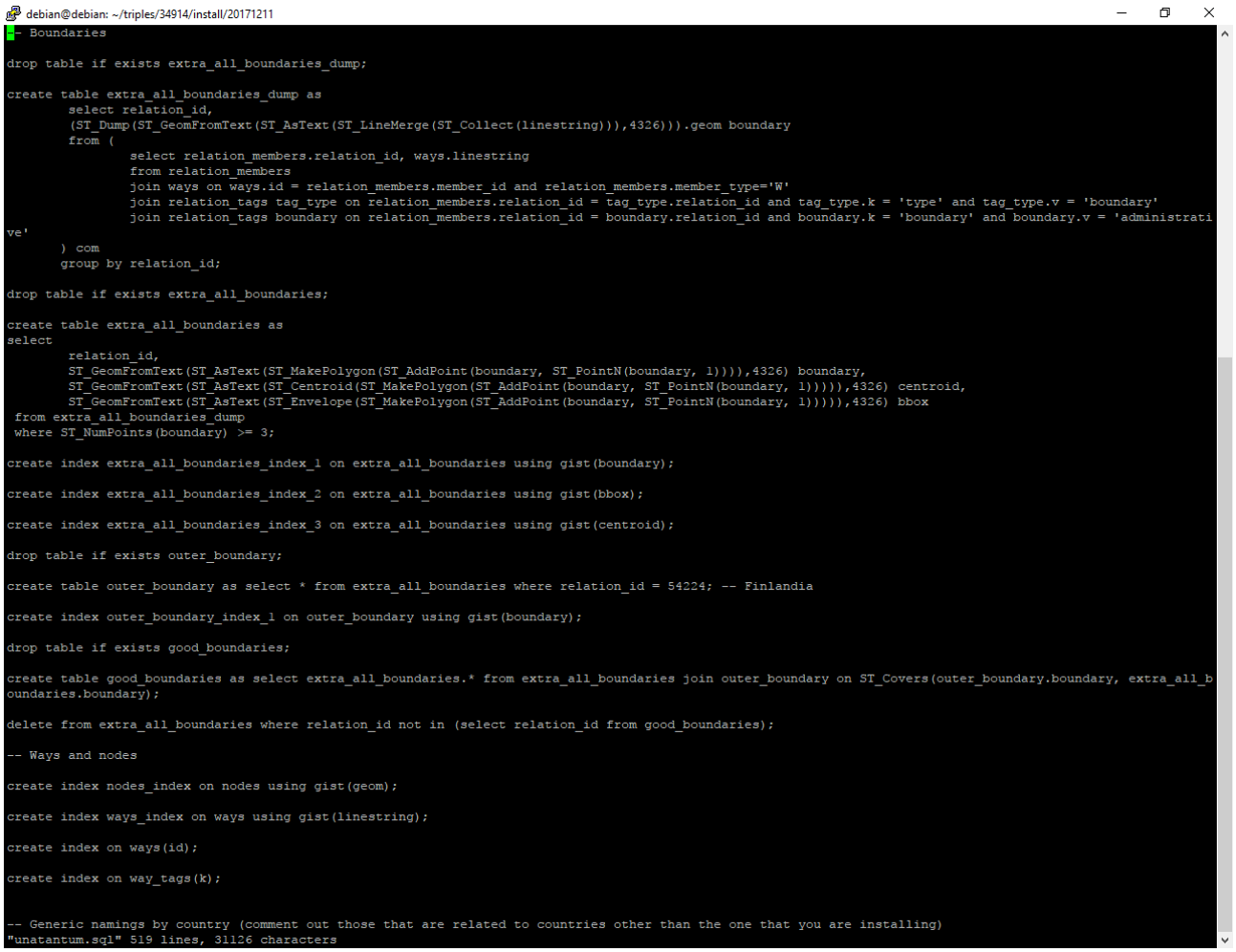

Figure 3 A fragment of the initialization script

Once that the initialization script has been launched, the first triplification process can be started. At now, each triplification process is performed through the execution of a shell script (Figure 4) that we at DISIT have developed, which automatically performs all the necessary operations, from the preparation of the data on the relational database, to the generation of a text file that contains the RDF triples, and the cleaning of such file for that it could be successfully uploaded to Virtuoso. The uploading of the RDF triples to Virtuoso is the only step that is not included within the shell script. This way, some further checks over the generated RDF triples can be performed before their uploading, and possible unexpected anomalies can be detected. Digging into the shell script, the psql command is employed for launching SQL scripts and queries on Postgresql. A database always must be specified. At now, a separated database for each country covered by the project has been generated. The pgsimple_fin, is the database where the Open Street Map of Finland has been uploaded. The script.sql is the preparation script, where an appropriate set of tables is created. Before being created, each table is dropped. In reason of that, all the privileges that had been set on the table go lost. So, a query is needed after that the preparation script execution has completed, for granting the needed privileges to the pgsimple_reader_fin user. Then, the RDF triples are generated through the execution of a Sparqlify dump. The necessary parameters, as it can be seen, are the SML configuration file, and the source relational database host, database, username, and password (obfuscated). Finally, the heading rows are stripped away, the duplicated rows are also stripped away, and the resulting file is typically stored with a filename that matches the name of the province or municipality to which it refers, followed by the n3 extension which is the expected extension for the RDF triples files.

```
psql -d pgsimple_fin -f ./script.sql
psql -d pgsimple_fin -c "grant select on all tables in schema public to pgsimple_fin_reader;"
cd ~/Sparqlify
./sparqlify.sh -m ~/triples/34914/install/20180102/script.sml -h 192.168.0.110 -d pgsimple_fin -U pgsimple_fin_reader -W [redacted] -o ntriples --du
mp > ~/triples/34914/install/20180102/dirty.n3
cd ~/triples/34914/install/20180102
tail -n +3 dirty.n3 > quite_clean.n3
sort quite_clean.n3 | uniq > Helsinki.n3
rm dirty.n3
rm quite_clean.n3
```

Figure 4 A sample shell script for the triplification of the Helsinki street graph

Digging into the SQL script aimed at preparing the triplification, three sections can be identified within it.

The first section, is the configuration sections. It is here that the boundaries of the triplification are set. In the sample fragment provided in Figure 6, as an example, it is stated that the boundary of this triplification is the Open Street Map relation whose OSM ID is equal to 34914 (Figure 5). If it was needed, a set of identifiers could be provided. This way, the subsequent triplification would have been extended over all the identified areas. Expectedly, for that the triples could be generated, such areas had to be confined within the boundaries of the Finland, being that the relational database that we have adopted as a source for our triplification has been filled with the only data from the Finland Open Street Map.

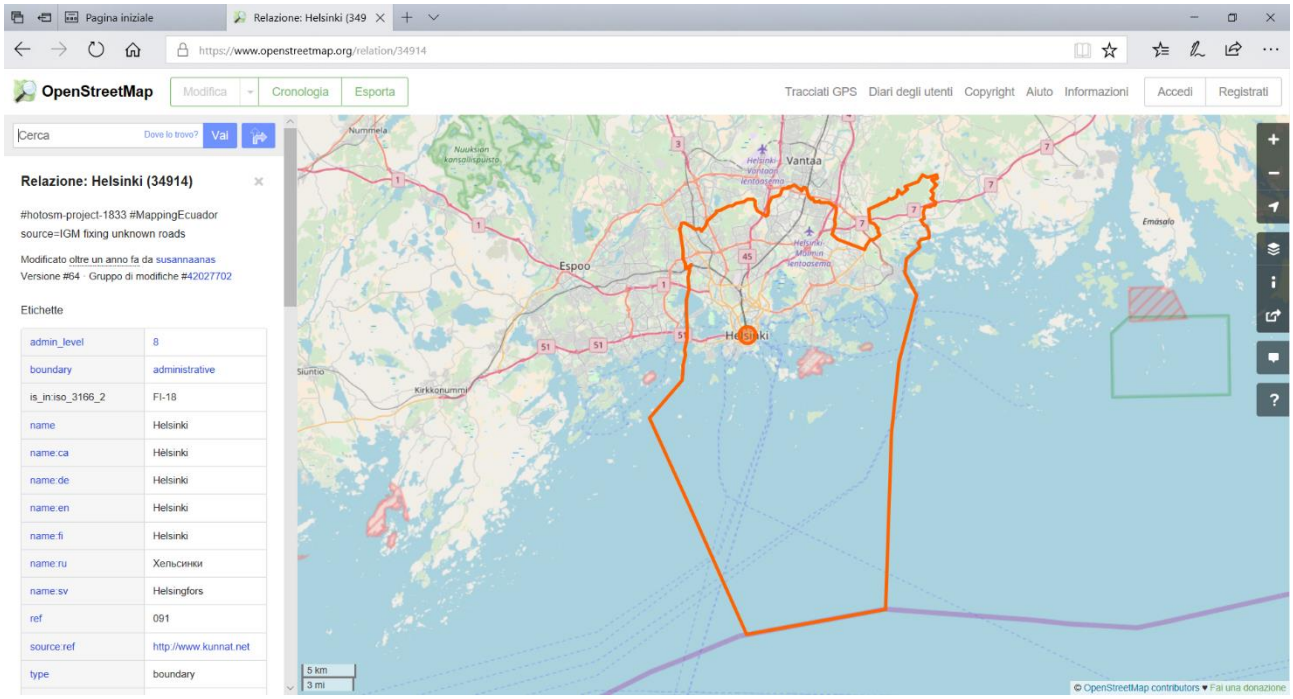

Figure 5 The city of Helsinki on Open Street Map, OSM ID 34914

Also remarkably, the RDF graph can be specified in the configuration section of the preparation script. In this case, we have not cared about it, because we generate RDF triples, where the graph does not appear. It should be noted that it is a choice. Indeed, when a Sparqlify dump is launched, appropriate options can be set for that it produces RDF quads, instead of RDF triples. In this case, the RDF graph appears on each row of the output file, and in this case, this setting about the RDF graph come to be relevant.

Finally, a configuration can be set that is not relevant for the city of Helsinki, but that could be relevant for the city of Florence. Indeed, it can be chosen if the street numbers have to be obtained from the Open Street Map, or from the Tuscany Region, which also provides them. If the Tuscany Region is chosen, the RDF triples that represent the street numbers are not included in the output file, as they are expected to be generated and uploaded separately, through the leveraging of appropriate ETL processes. For the city of Helsinki, we have obviously chosen to leverage the street numbers that are included within the Helsinki Open Street Map.

The second section of the SQL script aimed at preparing the triplification, is where the *support tables* are created. The support tables, are tables that are leveraged for preparation of the data of several properties of several concepts. For this reason, they are grouped at the beginning of the SQL script, without a clear indication of the concept and property to which they are related.

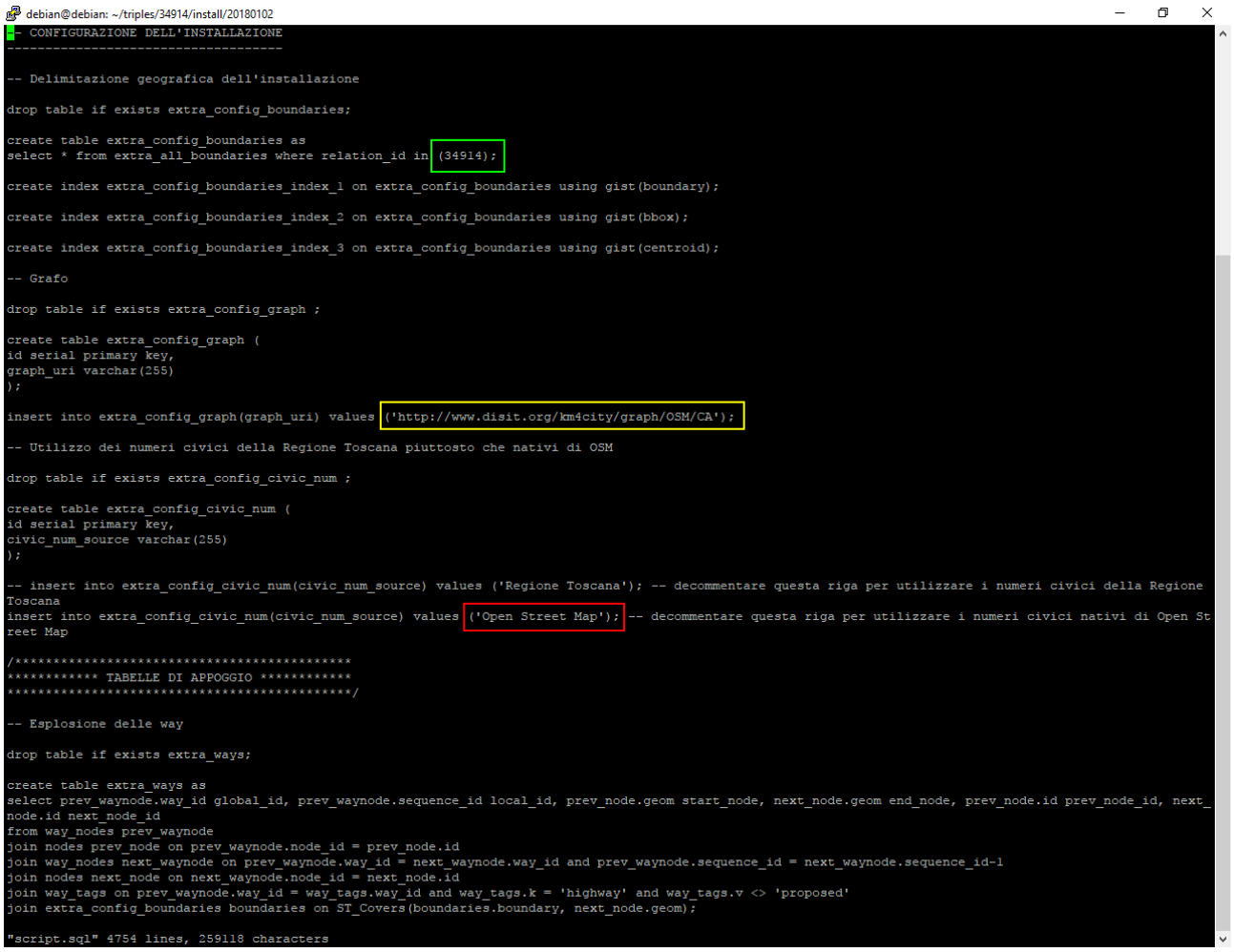

Figure 6 The configuration section of the SQL script aimed at preparing the data for the triplification

The third section of the SQL script aimed at preparing the triplification, is where the specific tables aimed each at containing the data that is needed for producing the RDF triples of a specific property of a specific concept can be found. In Figure 7 a fragment where some tables aimed at containing the data of some properties of the concept *Province*, is provided. As it can be seen, the first table is aimed at containing the URIs of the instances that have to be produced of the concept *Province*, and its target list just includes the RDF graph where the triples has to be included (that will be anyway ignored by the Sparqlify), and a field named *id* where the URI of the instance is contained. Notably, the next table, aimed at preparing the data for the property *identifier*, is identical to the first. Indeed, for all concepts related to the street graph, the property *identifier* is a string whose value is the URI of the instance. Later, in the SML, the two tables will be treated differently: the first will be accessed for producing triples that conform to the template *<uri> rdf:type km4c:Province*, while the second will be accessed for producing triples that conform to the template *<uri> dct:identifier "uri"*. It is true that the same table could be used two times in different ways, so no need was of creating two identical tables, but through the generation of a separated table for each different property of each different concept, we keep the SQL scripts, and the SML scripts, easy to read, and consequently easy to maintain.

In the third table of the fragment, the necessary data for the generation of the RDF triples for the property *name* of the concept *Province* is wrapped in a dedicated table, the *ProvinceName* table. All what is needed for producing of the RDF triples for the *Province name* property, is the URI of the subject, and the filler value. Indeed, the target list of the table simply contains the field *id* (the URI), the field *p_name* (the filler string), and the RDF graph URI.

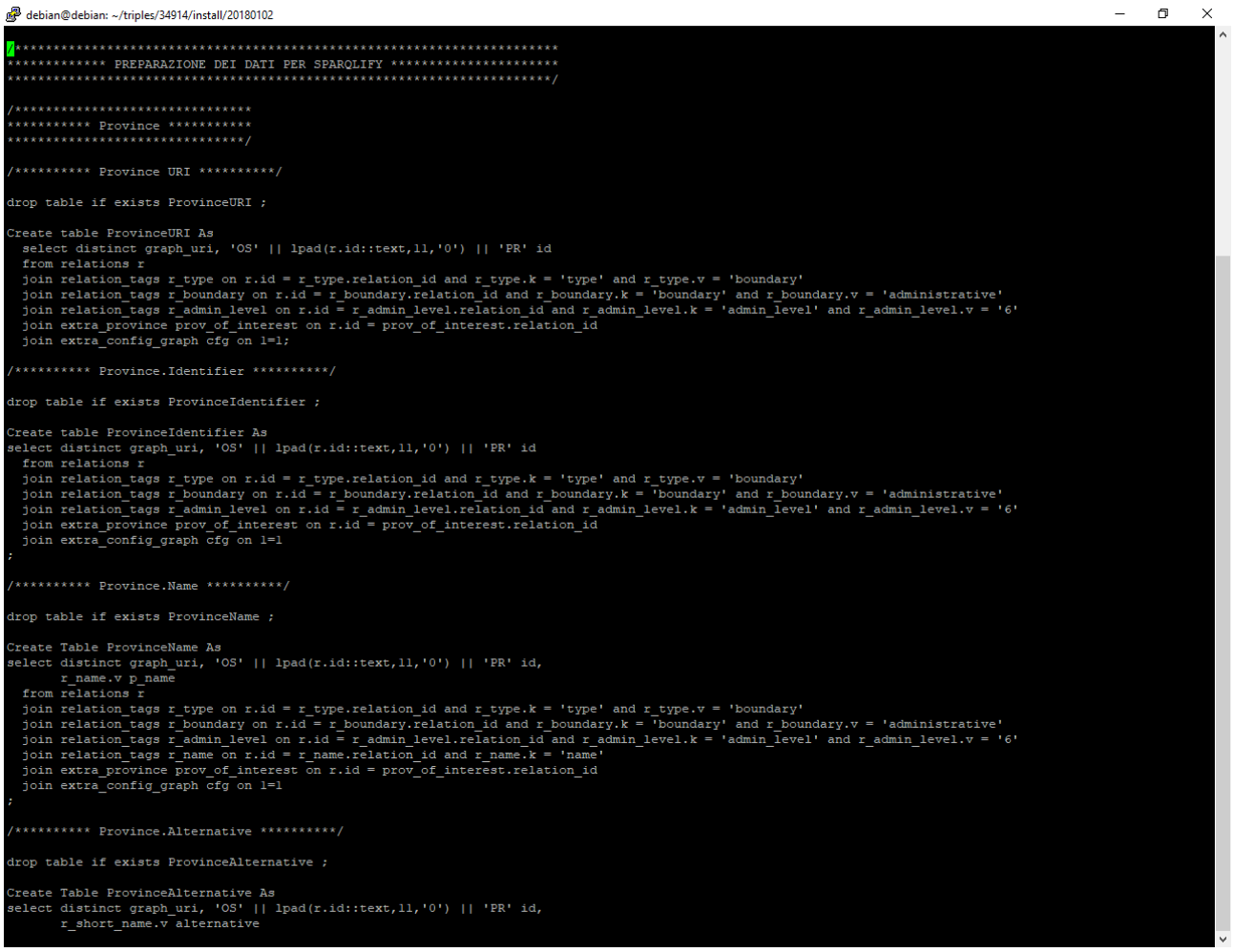

Figure 7 A sample fragment from the third section of the SQL script aimed at preparing the triplification

Digging into the SML script, where the SQL and the SPARQL query are mixed for instructing Sparqlify about what RDF triples have to be generated from which RDB source data, we now analyze the fragment provided in Figure …, where the generation of some RDF triples for the concept *Province* is configured.

In the first fragment, outlined in green, we are instructing the Sparqlify as follows. Through the *From* directive, we are indicating that the necessary data must be read from the table ProvinceURI. Remarkably, it is the simplest possible SQL query: get all rows and all fields from a specified table. It has been made possible, because the complexity of the mapping is wrapped within the SQL script that prepares the triplification, and it is a great advantage in terms of efficiency, because the simpler is the query, the faster is its optimization. Through the *With* directive, we define a set of variables (at the left side of the equality) leveraging the values that are contained within the RDB table. At the right side of the equality indeed, the names of the fields from the RDB table appear, together with native SML functions through which some basic operations such as specifying how a string value has to be interpreted (if it has to be considered a simple string, if it has to be considered an instance URI) or performing a concatenation can be executed. For each row of the RDB table, the properties are set, and they are employed as it is specified in the *Construct* directive. In this case, the *Constraint* directive instructs the Sparqlify to generate just one RDF

triple for each row that can be found in the RDB table, which has to be put in a RDF graph whose URI can be found in the variable *graph*. The triple that has to be generated must have: (1) the value of the variable *s*, which is a URI, as subject; (2) the *rdf:type* as property URI (the reserved word *a* stands indeed for that); (3) the URI that identifies the concept *Province* as a filler. The leading *Create View* directive should be considered a simple label for this configuration fragment.

In the second fragment, outlined in yellow, the RDB table *ProvinceIdentifier* is read, which is identical to the *ProvinceURI* one. The *id* field is employed in two different ways: it is employed for setting the variable *identifier* equal to a string whose content is the URI, and it is employed for setting the variable *s* equal to a URI. In the *Construct* directive, the variable *s* bears the role of the subject of the RDF triple that has to be generated, while the variable *identifier* is the filler. The property URI is statically set equal to *dct:identifier*.

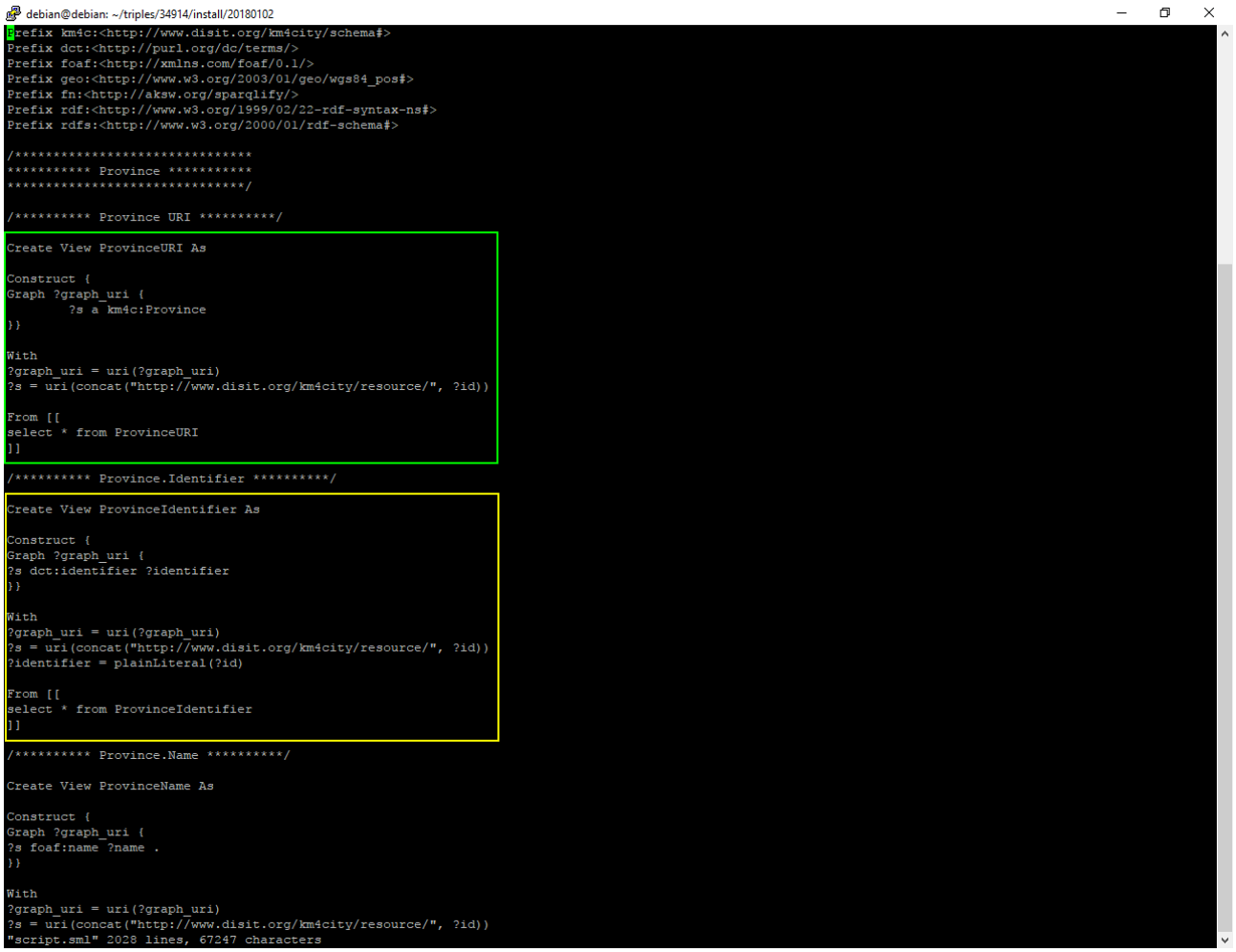

Figure 8 A sample fragment from the SML file where the production of some triples for the concept Province is configured

Finally, in Figure 9 the first lines of the triples file for the city of Helsinki are showed. The file that is showed in the fragment has already been cleaned. Nevertheless, it contains over 10 millions triples. As it can be seen, the rows are incidentally ordered in an alphabetical manner, as a consequence of the sorting that has been applied as a preliminary step for the removal of the duplicate rows. As expected, each row is made up by three parts, separated through a space character: the subject (the angled parenthesis denote that is has to be interpreted as a URI), the property, and the filler. As expected, the graph URI does not appear.

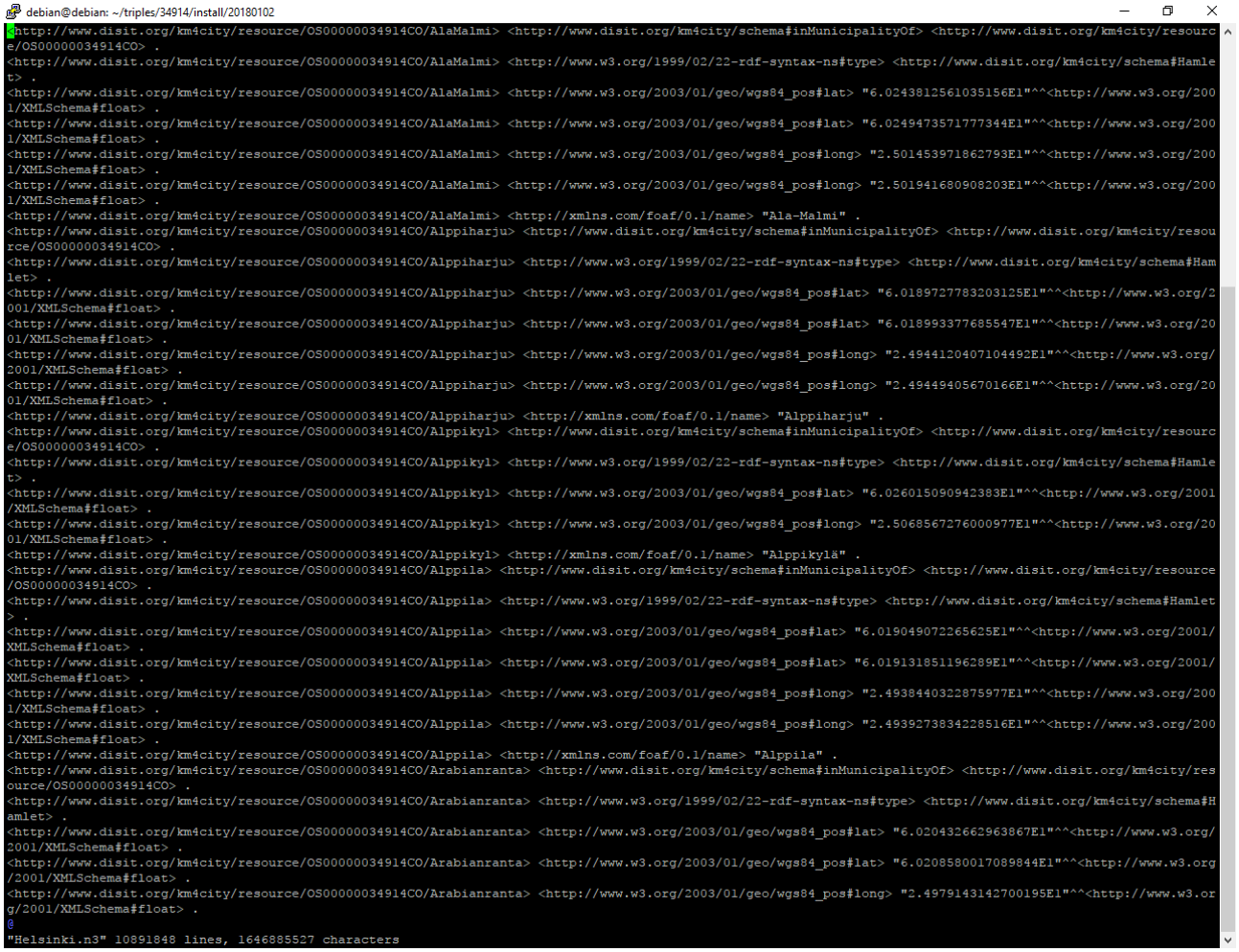

Figure 9 The first few lines from the triples file for the city of Helsinki

Now that the overall process that leads to the generation of the RDF triples has been outlined, an eye can be put at the Open Street Map and Km4City data models mapping, that can be considered the specification for the SQL scripts and configurations that we have introduced above.

For a full understanding of the *data models mapping*, a brief overview about how the street graph is represented within the OSM XML files is necessary. In Open Street Map, all *Public Administrations* (PA) are represented through relation elements that have a set of tag child elements that describe the level and the name of the PA, and a set of member child elements that describe the boundary of the PA. It should be noted that not all relation elements represent a PA: a deep look over the relation child elements is necessary for understanding what it represents. In OSM, *roads* are represented through way elements. Precisely, a road can be represented through a single way element, or through a set of way elements grouped within a relation element. Each way element has a set of *nd* child elements, each of which addresses a node element, so outlining the path of the way. It should be noted that not all the way elements represent a road or a segment of road. Indeed, a way just represents a line on the map. A deep look over the way child elements, and over the relation elements that includes the way, is mandatory for understanding what a way represents. In OSM, the *street numbers* and the related entrances are mainly represented through node elements. Again, not all node elements represent a street number. Indeed, a node element merely represents a point on the map. A deep look over the node child elements is requested for understanding if it really represents a street number, and for efficiently and effectively determining the road where the street number is located. In some cases, a look at the relation elements that include the node is even necessary for identifying the road where the street number is located. Indeed, the node elements that represent the street numbers, rarely coincide with the node elements that outline the path of the road where the same street numbers are located. In OSM, the *lanes* are represented through tag

elements whose key include the lanes word, and that are children of OSM way elements that represent roads or parts of road where such lanes are located. A deep parsing of both the key and the value of such tags is necessary for understanding what a specific tag exactly says about the lanes. Lastly, *traffic restrictions* are spread everywhere, are expressive enough for representing any sort of prohibition and exception, and can be represented differently in different parts of the OSM map.

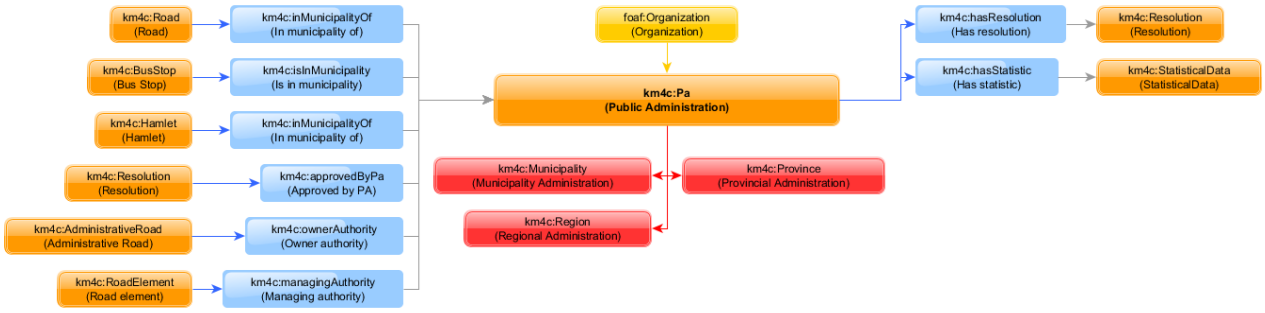

Figure 10 An outline of the Km4City Pa class

Within Km4City, each Public Administration is represented as an instance of the class *Pa* (Figure 10).

A new instance of the class *Pa* is generated for every OSM *relation* tagged with *type* equals to *boundary* and *boundary* equals to *administrative*. A third tag, *admin_level*, is used for determining the tier of the PA, and consequently the specific child class that must be instantiated: if it is set to 4, a *Region* is generated, if it is set to 6 a *Province* is generated, and if it is set to 8 a *Municipality* is generated. At each instance of *Pa* is assigned an URI that includes the id of the OSM *relation* from which the instance is generated. For each *Pa*, three properties are set: the *identifier* (which globally identifies the instance and includes the id of the OSM *relation* from which the *Pa* is created), the *name* (the official name, set equal to the *name* tag of the *relation*), and *alternative* (the abbreviation, set equal to the *short_name* tag of the *relation*). Also, the property *hasProvince* is set for each region, the properties *hasMunicipality* and *isInRegion* are set for each province, and the property *isInProvince* is set for each municipality, for linking the PAs with each other. The PAs are linked with each other on the basis of their boundaries, that are outlined by the OSM *way* elements that are included within each *relation* that represents a Public Administration.

The class *Hamlet* (Figure 11) represents the lowest tiers of sub-national division, below the municipality. Within a city, the instances of *Hamlet* often represent its districts. For each instance of *Hamlet*, five properties are set. The property *identifier* globally identifies the instance. The property *name* provides the name of the hamlet or district. The property *inMunicipalityOf* indicates the municipality where the hamlet or district is located. The properties *lat* and *long* provide the indicative position of the hamlet or district.

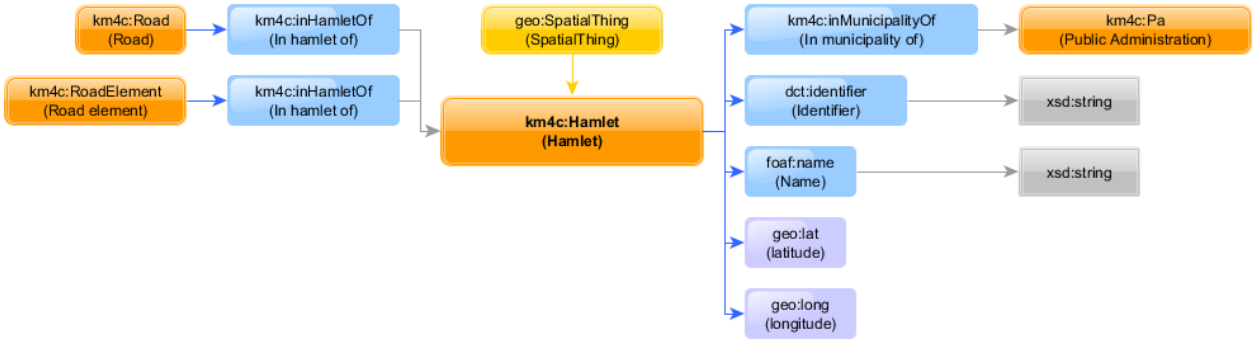

Figure 11 An outline of the Km4City Hamlet class

A new instance of *Hamlet* is generated for every OSM relation tagged with *type* equal to *boundary*, *boundary* equal to *administrative*, and *admin_level* greater than 8. Also, a new instance is generated for every OSM *way* tagged with *boundary* equal to *administrative* and with *admin_level* greater than 8. Lastly, a new instance is generated for every OSM *node* tagged with *place* equal to *suburb*. The *identifier* includes the id of the OSM element from which the instance is generated. The *name* is set equal to the value of the name tag of the element from which the *Hamlet* instance is generated. The *inMunicipalityOf* is set equal to the municipality where the hamlet or district is located, identified by looking at their boundaries. The *lat* and *long* properties are set equal to the respective attributes of the OSM node element from which the *Hamlet* instance is generated, or computing the centroid of the boundary if the *Hamlet* instance is generated from an OSM *way* or *relation*.

The class *Road* (Figure 12) represents every type of road. For each instance of *Road*, eight properties are set. The property *identifier* globally identifies the instance. The property *roadType* provides the street type designation[13], where applicable. The property *roadName* provides the distinguishing name of the road, obtained removing the street type designation from the full name of the road. The property *extendName* provides the full name of the road. The property *alternative* provides an alternative naming for the road, where applicable. The property *containsElement* indicates one of the road segments that make up the road (if the road is composed by several segments, several instances of the property are generated). The property *inMunicipalityOf* indicates the municipality where the road is located, and the property *inHamletOf* similarly indicates the hamlet or district where the road is located (when applicable).

A new instance of *Road* is generated when one of the following is found within OSM:

1. a *way* element tagged with *highway* set equal to any value except for *proposed*, and not referenced by any relation that represents a road;
2. a relation tagged with *type* equal to *route*, *route* equal to *road* (or not instantiated), and *network* different from *e-road*, and that also includes at least one *way* tagged with *highway* set to any value except for *proposed*.

In the first case, the properties of the *Road* instance are set as follows:

- the *identifier* includes the id of the *way* from which the instance is created;
- the *roadType*, *roadName*, and *extendName* are set parsing the *name* tag of the OSM *way* element. Precisely, the value of the tag is written as is on the property *extendName*, while the *roadType* and the *roadName* are set searching the *extendName* for one of the allowed road types, kept in a separate list which varies from country to country. If a match occurs, the matched road type is written into the *roadType*, and the remaining is written into the *roadName*. Otherwise, the *roadType* is not instantiated, and the *roadName* is set equal to the *extendName*;
- the *alternative* is set equal to the *alt_name* tag of the OSM *way*, if available;

[13] https://en.wikipedia.org/wiki/Street_or_road_name#Street_type_designations

- for the identification, instantiation and linking of the segments of road, it should be noted that each OSM *way* represents, in the most general case, a complex path outlined by a set of OSM *node* elements each representing a point on the map. A new *RoadElement* instance is generated for each linear segment outlined by two consecutive nodes, and a new *containsElement* property is instantiated on the *Road* for linking the newly generated road element to the road where it is located;
- the properties *isInMunicipality* and the *isInHamlet* are set equal to the instances of *Municipality* and *Hamlet* that represent the municipality and the hamlet (or district) where the road is located. If a road falls within no district, the *isInHamlet* property is not instantiated. A road is considered to fall within a municipality, hamlet or district, if at least one of the nodes that outline the path of the

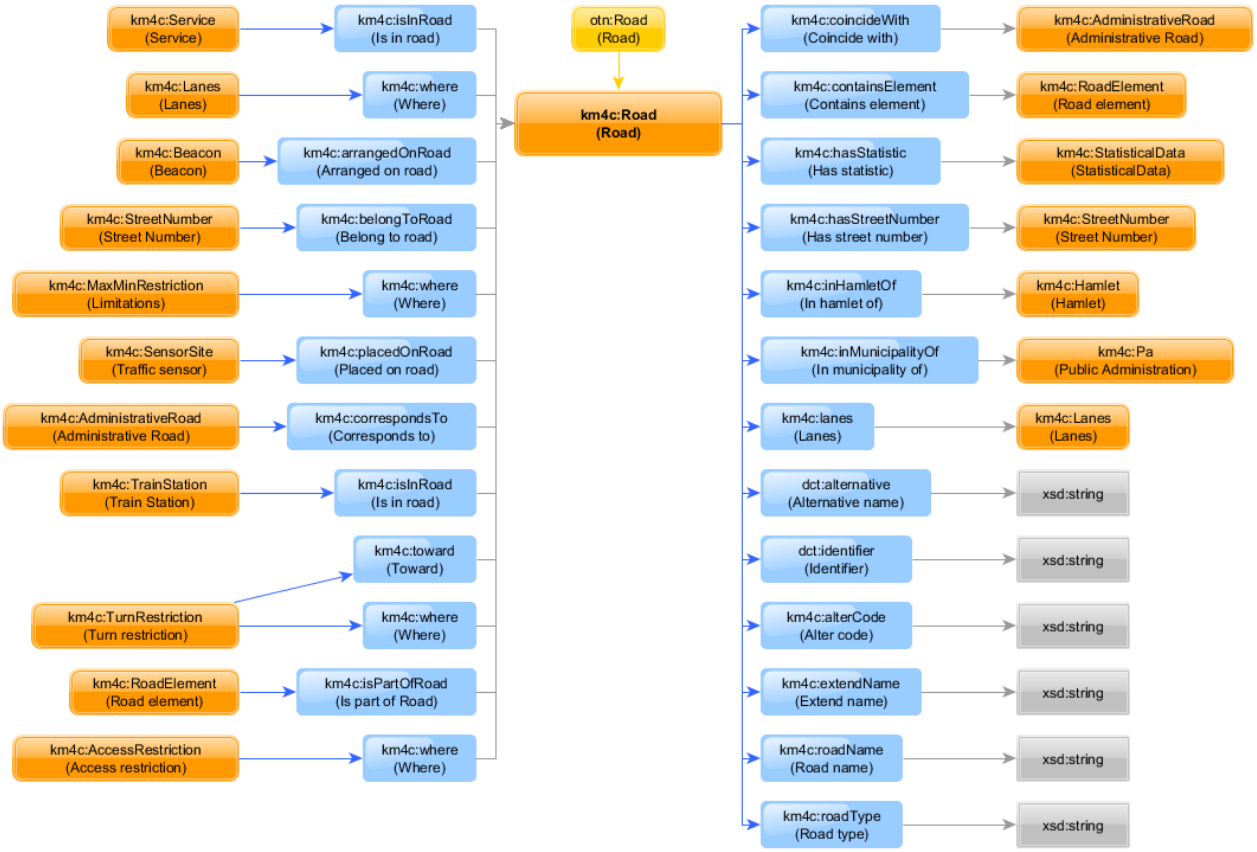

Figure 12 An outline of the Km4City Road class

road falls within the boundary of the municipality, hamlet or district.

In the second case, the properties of the *Road* are set as follows:

- the *identifier* includes the id of the relation from which the instance is created;
- the *roadType*, *roadName*, and *extendName* are set parsing the *name* tag of the *relation*;
- the *alternative* is set equal to the value of the *alt_name* tag of the *relation*;
- for each linear segment outlined by two consecutive nodes within each *way* member of the *relation*, a new instance of the class *RoadElement* is generated;
- the *isInMunicipality* and the *isInHamlet* properties are set so that if at least one node of at least one *way* among the ones included in the *relation*, falls within the boundary of the municipality, hamlet or district, then the road is considered to belong to the municipality, hamlet or district.

The class *RoadElement* (Figure 13) represents a segment of a road. A new *RoadElement* is generated for

each linear segment of each OSM *way* that represents a road or a segment of road.

The main properties set on a *RoadElement* are described below. The *identifier* is composed by concatenating the OSM *way* id with the position of the segment in the OSM *way*. The properties *startsAtNode*, *endsAtNode*, and *route* provide basic information about the segment of road, indicating the two extremities of the segment of road, and providing a geometric representation for the segment. They

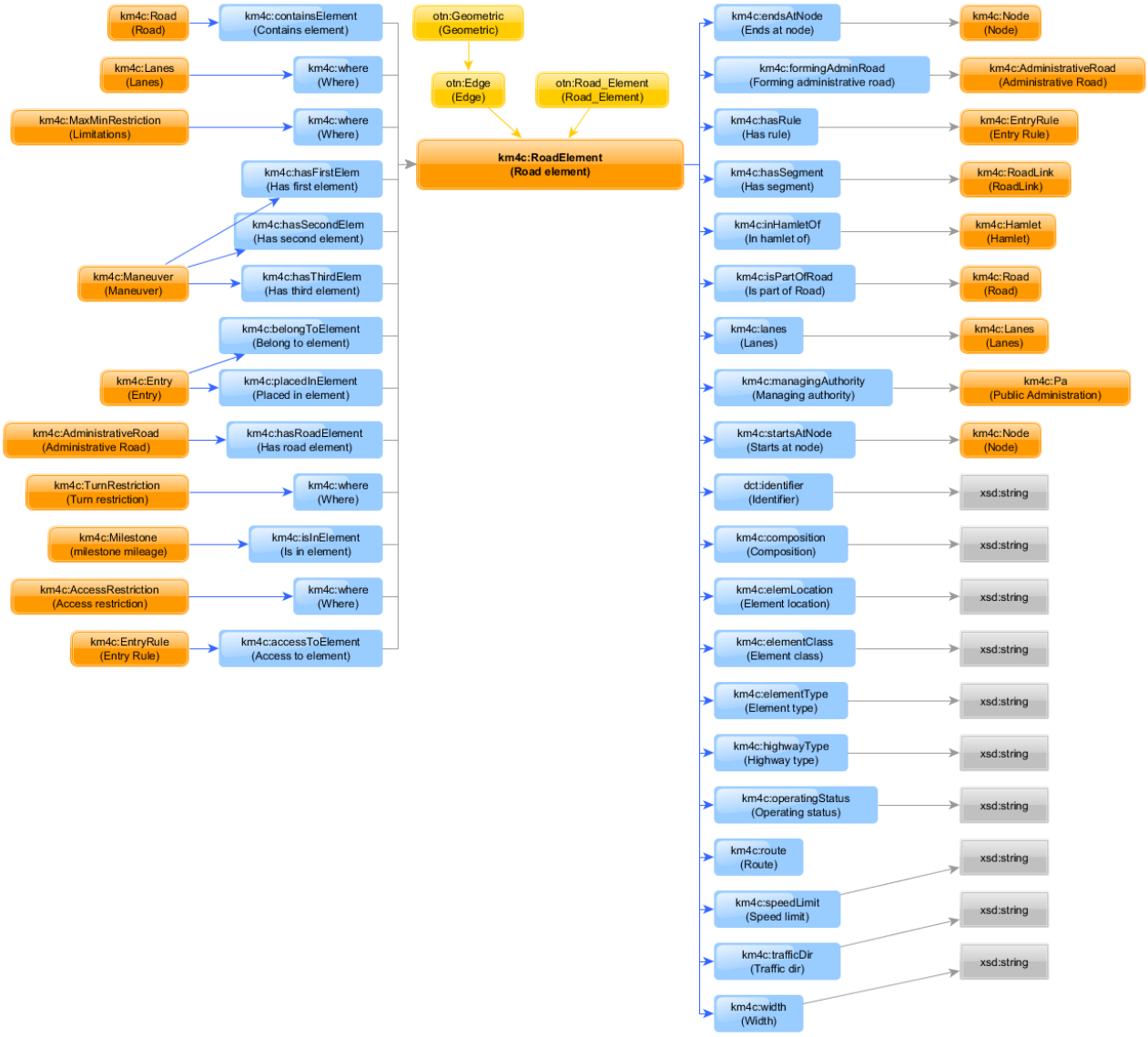

Figure 13 An outline of the Km4City RoadElement class

are set looking at the OSM *node* elements that outline the path of the OSM *way*.

The property *elementType* provides an information about the location and usage of the segment of road. Precisely, it says if the segment of road is located within a parking, a link, a roundabout, a crossroad, an highway barrier, a railway level crossing, a pedestrian area, a road reserved for particular usages (bus stop, rest area, emergency), or if it is a generic segment of road. This property is primarily set on the basis of the *highway* tag of the OSM *way*, together with other tags like service, amenity, junction, barrier, area, railway, lanes. The property *composition* tells if the segment of road is a dual carriage one. It is set to dual carriage if the segment of road belongs to a type of road that always is a dual carriage road for law, or if the way is within a relation tagged with *type* equal to *dual_carriageway*. The property *elemLocation* provides an information about the elevation of the segment of road. Precisely, it says whether the segment of road is located in a tunnel, bridge, link, a union of them, or none of them. The property is set on the basis of the

bridge, *tunnel*, and *highway* tags of the *way*, and on the basis of the *type* tag of the *relation* in those cases where the *way* is contained within a *relation* that represents a road. The property *length* provides the length of the segment of road, which is computed and set equal to the distance between the start and the end node of the segment of road. The property *width* provides the width of the segment of road. It is set equal to the *width* or *est_width* tag of the *way*, if available. Otherwise, the property is not instantiated. The property *operatingStatus* provides an information about the status of the segment of road. Precisely, it says whether the segment of road is under construction, disused, or operating. The property is set looking at the tags *highway* and *disused* of the OSM *way*. The property *highwayType* provides further information about the type of the road to which the segment of road belongs. It is set equal to the value of the tag *highway* of the *way*. The properties *managingAuthority* (and *inHamletOf*) are set equal to the municipality (and the hamlet/district, if applicable) within which the segment of road falls. They are set searching for the municipality (or the hamlet/district) such that at least one of the extremities of the segment of road falls within the boundary of the municipality (or hamlet/district).

Each street number is represented in Km4City through a couple of instances: an instance of the class *StreetNumber* (Figure 14), and an instance of the class *Entry* (**Errore. L'origine riferimento non è stata trovata.**), linked through the property *hasExternalAccess* defined for the *StreetNumber*.

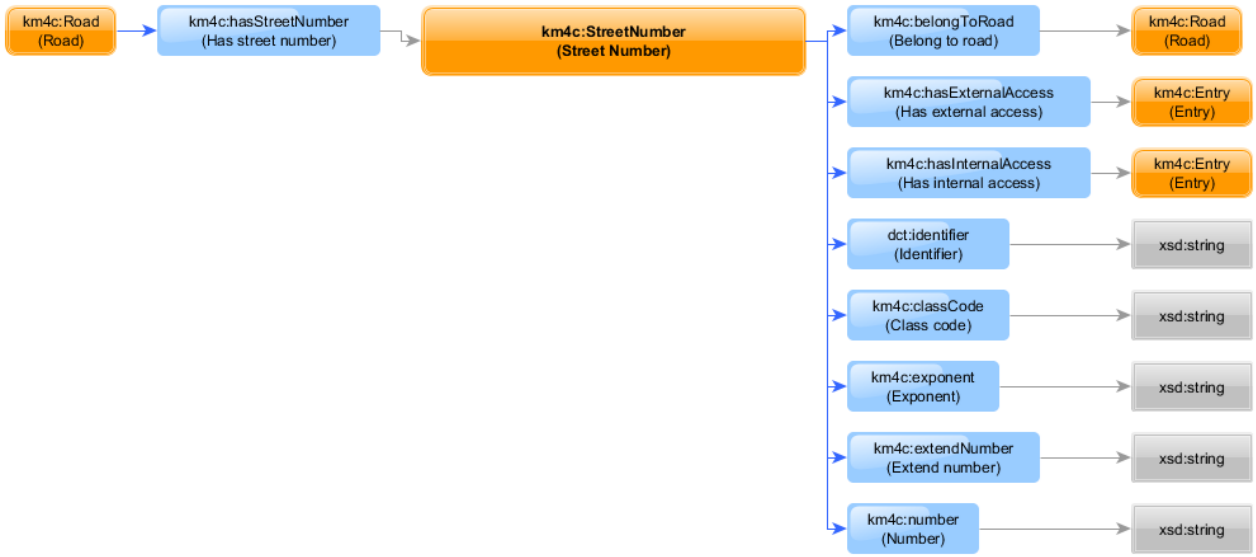

Figure 14 An outline of the Km4City StreetNumber class

A new instance of *StreetNumber* is generated for every OSM *node* tagged with a complete address spread across three different tags: *housenumber*, *street*, *city*. Also, a new *StreetNumber* is created for every OSM *node* that is tagged with a *housenumber* and that is contained within a *relation* tagged with *type* equal to *associatedStreet*. Indeed, such relations exist in OSM for linking the street numbers to the road or segment of road where they are located. Lastly, a new *StreetNumber* is created for every *node* that is tagged with a *housenumber* and that belongs to the path of a *way* that represents a road or a segment of road.

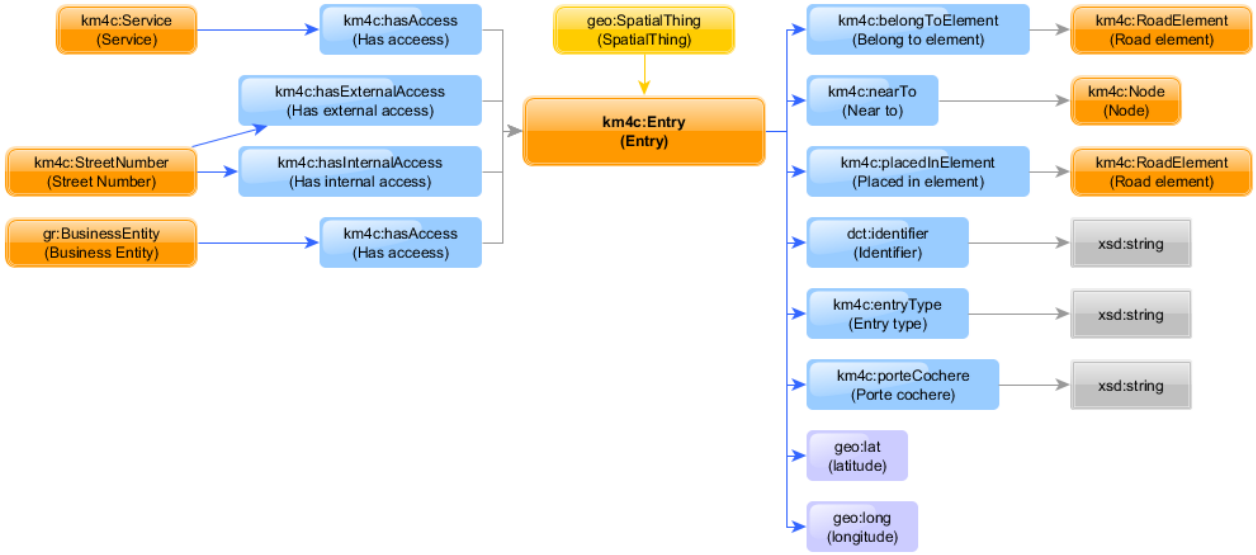

Figure 15 An outline of the Km4City Entry class

The following properties are set for every instance of the class *StreetNumber*. The *identifier* globally identifies the instance, and it includes the OSM *node* id. The property *belongToRoad* indicates the road where the street number is located. The property *classCode* is necessary for three Italian cities (Florence, Genoa, and Savona), where two completely disjoint numbering systems coexist: the red numbers, and the black numbers. So, a property is necessary for discriminating between the two. The property is only set for the street numbers located in one of these cities, by parsing the *housenumber* tag value. The properties *number*, *exponent*, *extendNumber* provide the numeric part, the literal part, and the whole street number. They are set parsing the *housenumber*. For each instance of *Entry*, the following properties are set. The *identifier* globally identifies the entry, and it contains the OSM *node* id. The properties *lat* and *long* provide the exact position of the entry on the map. The property *porteCochere* tells if the motor vehicles can transit through the entry. If *motorcar* equal to *yes* or *motorcycle* equal to *yes* on the *node* that represents the entry, they can. The property *placedInElement* is set equal to the nearest of the segments or road that compose the road where the entry is located.

The class *Lanes* represents a set of lanes drawn on the asphalt of a road. For each instance of *Lanes*, the property *where* indicates the road where the lanes are drawn, and the property *lanesCount* provides a count for the lanes, distinguishing those that are reserved to specific categories of vehicles from the others.

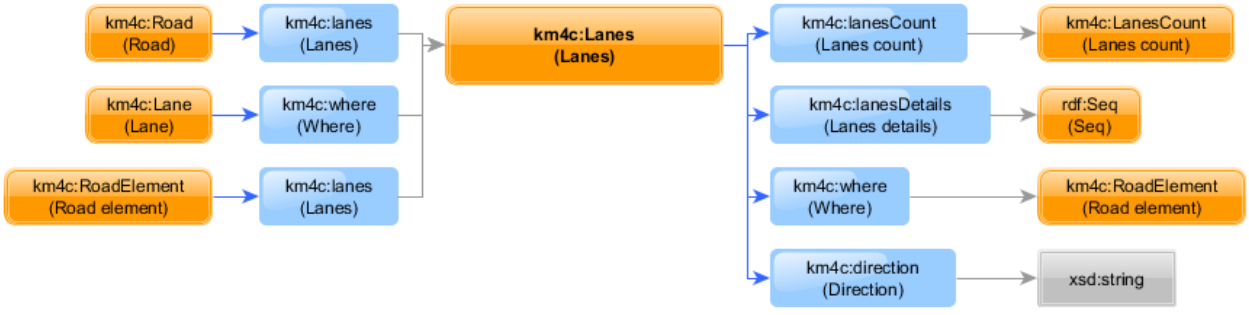

Figure 16 An outline of the Km4City Lanes class

Indeed, the property *lanesCount* is set equal to an instance of *LanesCount* (Figure 17), and each instance of *LanesCount* has a property *undesignated* which provides the count of the lanes that can be traversed by all the types of vehicle, and additional properties which provide the count of the lanes that are reserved to

specific categories of vehicles.

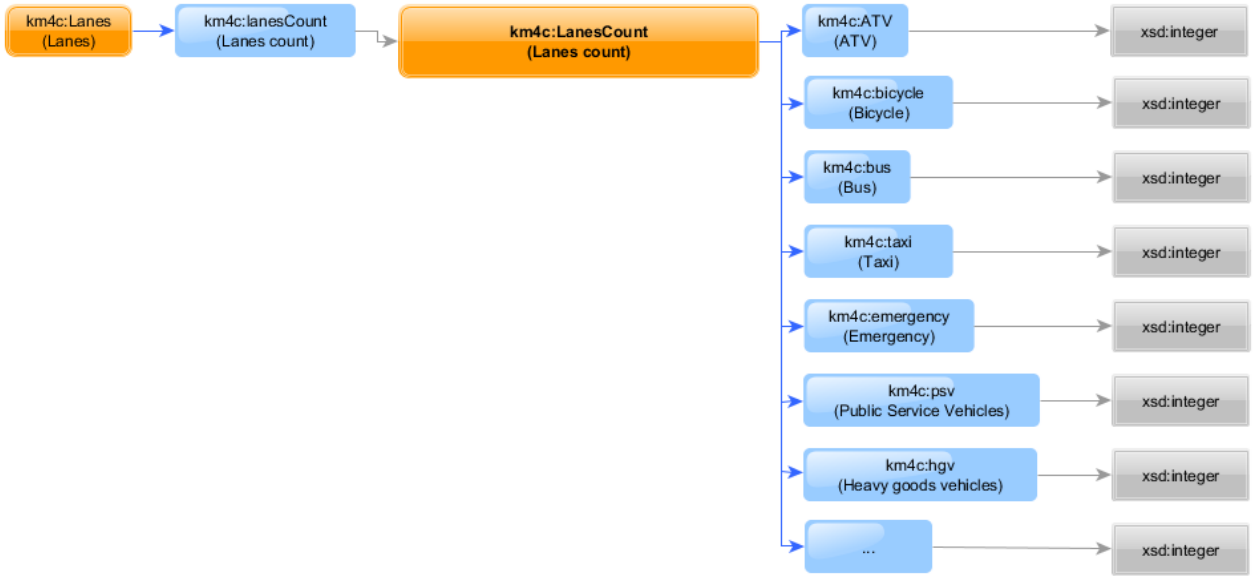

Figure 17 An outline of the Km4City LanesCount class

When specific restrictions apply to at least one of the lanes, the property *lanesDetails* is also instantiated, and it is set equal to a *Seq* of *Lane* (Figure 18) instances, that represent each the restrictions that apply to the specific lane, if any. The lanes are added to the *Seq* in left-to-right order as viewed in their driving direction. For each instance of the class *Lane*, two properties are always set: the *where*, that is set equal to the instance of *Lanes* that represents the set of lanes to which it belongs, and the property *position* that is set equal to the position that the lane occupies within the set. Also, the property *turn* is instantiated if a restriction applies to the lane about the direction that the vehicles will have to take once they will reach the next crossroad. Also, the property *restrictions* is instantiated if other types of restriction apply to the lane, and is set equal to a *Bag* of *Restriction* instances, each representing a specific restriction that applies to the lane.

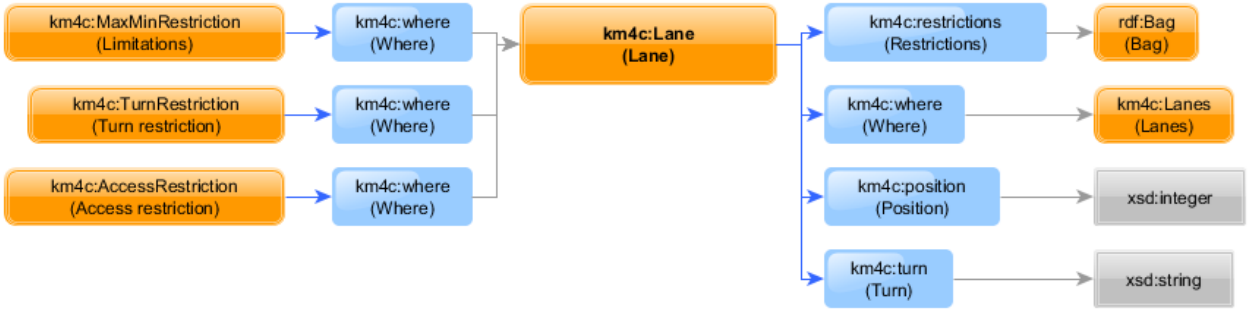

Figure 18 An outline of the Km4City Lane class

In OSM, all the information about the lanes is wrapped within tags whose key includes the word *lanes*, together with other optional specifications. In greater detail, if the key only contains the word *lanes*, the total count of the lanes is provided. Instead, if the key of an OSM *lanes* tag also includes a direction (*forward* or *backward*), it is intended that the tag only applies to the lanes that are traversed in the specified direction. In brackets, the *forward* direction is the positive direction of the OSM *way* where the lanes are drawn, and the positive direction of a OSM *way* goes from the first listed toward the last listed of the child OSM *node* elements that outline the path of the *way*. As an example, it could happen to find the tag *lanes:forward* and the tag *lanes:backward* applied to the same *way*, each set to an integer value. Well, they provide the count of the lanes, separately for the two directions. Also, within an OSM *lanes* tag key, a category of vehicles could be specified. As an example, we could find an OSM tag with the following key:

lanes:forward:bus. In this case, the value of the tag would be the count of the lanes that go forward and that are reserved to the buses. It should be noted here, that if the key of an OSM *lanes* tag includes something other than a direction and a category of vehicles, it is intended that it does not provide a count of the lanes. In greater detail, if the key of an OSM *lanes* tag includes the *turn* word, it is intended that the value of the tag provides an information about the direction that the vehicles will have to take once they will reach the next crossroad. Characters of pipe are used within the value of the tag, for separating the information related to the different lanes, provided from the left as viewed in the driving direction of the lanes. Similarly, an OSM tag could have its key equals to *hgv:access:lanes* and the value set to *no|no|yes*. Well, it would mean that high weight vehicles are allowed on the rightmost lane only. It should be noted here that abbreviated keys are sometimes used, especially for OSM tags that express access restrictions. As an example, it could happen to find the above key abbreviated as *hgv:lanes*, with the tag value unaltered. So, it is not sure that the OSM tags whose key includes the word *lanes* together with at most a direction and a category of vehicles, always provide a count of the lanes. A combined parsing of both the key and the value of OSM *lanes* tags is needed for a correct interpretation.

The class *Restriction* represents a generic traffic restriction, and it is specialized by the following three classes: *TurnRestriction*, *AccessRestriction* and *MaxMinRestriction*. The class *TurnRestriction* represents a restriction concerning the permitted maneuvers at a crossroad.

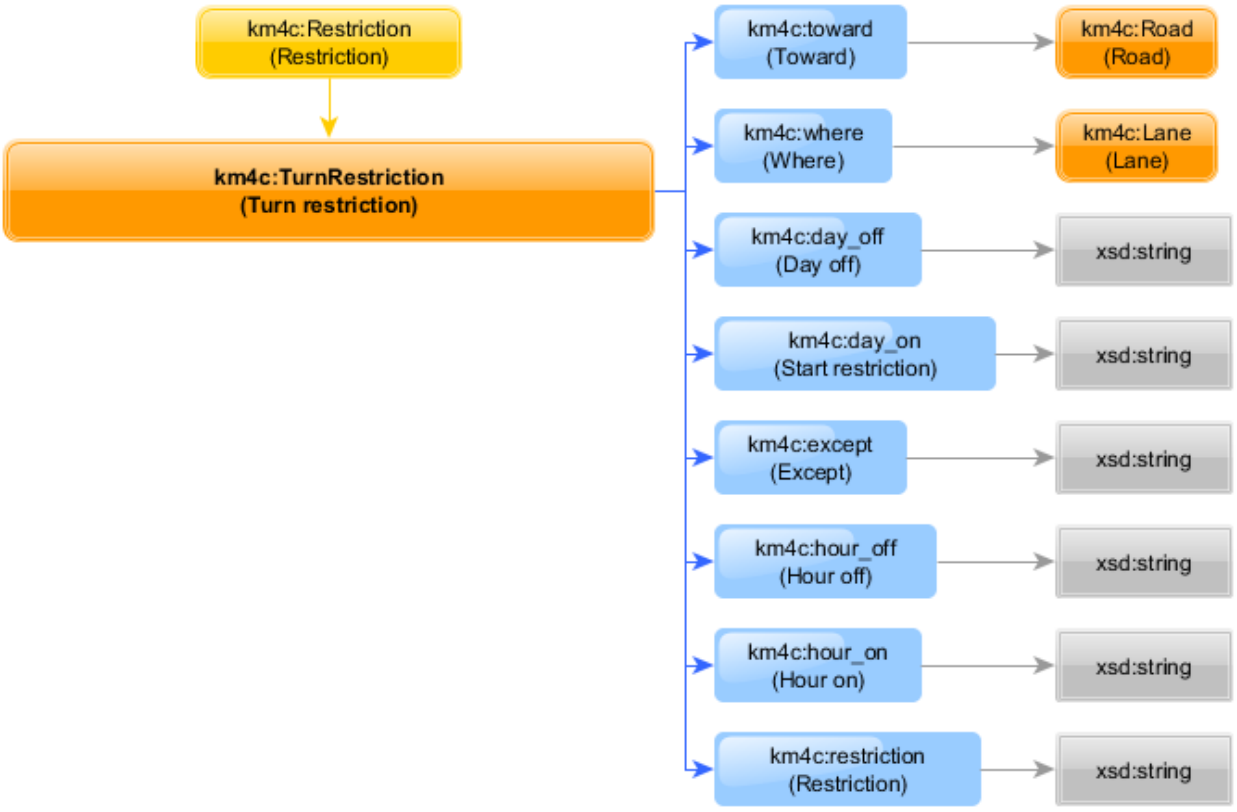

Figure 19 An outline of the Km4City TurnRestriction class

Three properties are always set for a *TurnRestriction* (Figure 19):

- *where*, which is set to the road of origin;
- *toward*, which is set to the road of destination;
- *restriction*, which describes the restriction.

Also, if the restriction only applies for a limited period, one or more of the following properties could be set: *day_on*, *hour_on*, *day_off*, *hour_off*. Also, if the restriction does not apply to a category of vehicles, the property *except* is set.

A new instance of *TurnRestriction* is created for every OSM *relation* on which a tag with key *type* and value *restriction* can be found. The property *where* is set looking at the OSM *way* that is included in the *relation* with *member_role* equal to *from*. Similarly, the property *toward* is set looking at the OSM *way* that is included in the relation with *member_role* equal to *to*. Lastly, the property *restriction* is set equal to the value of the tag *restriction* (always applied to the turn restriction *relations*) which indicates whether the maneuver is mandatory or forbidden. Also, if one or more of the tags *day_on*, *hour_on*, *day_off*, *hour_off*, *except* are applied to the relation, corresponding properties are set on the *TurnRestriction*.

The class *AccessRestriction* (Figure 20) represents the prohibition of accessing a road or a segment of road, or the prohibition of traversing a road or a segment of road in a specific direction. The restriction can apply to a specific category of vehicles or to all the vehicles, and can be subject to a number of conditions. Two properties are always set for an instance of *AccessRestriction*:

- *where*, which indicates the road where the restriction applies;
- *access*, which describes the restriction, which could be a prohibition or a permission for specific categories of vehicles to traverse the road (which implies a prohibition for all of the others).

The property *who* is set when the restriction only applies to a specific category of vehicles. Also, the property *direction* is set when the restriction only applies to one of the two traffic directions. Lastly, the property *condition* is set when the restriction is subject to one or more conditions.

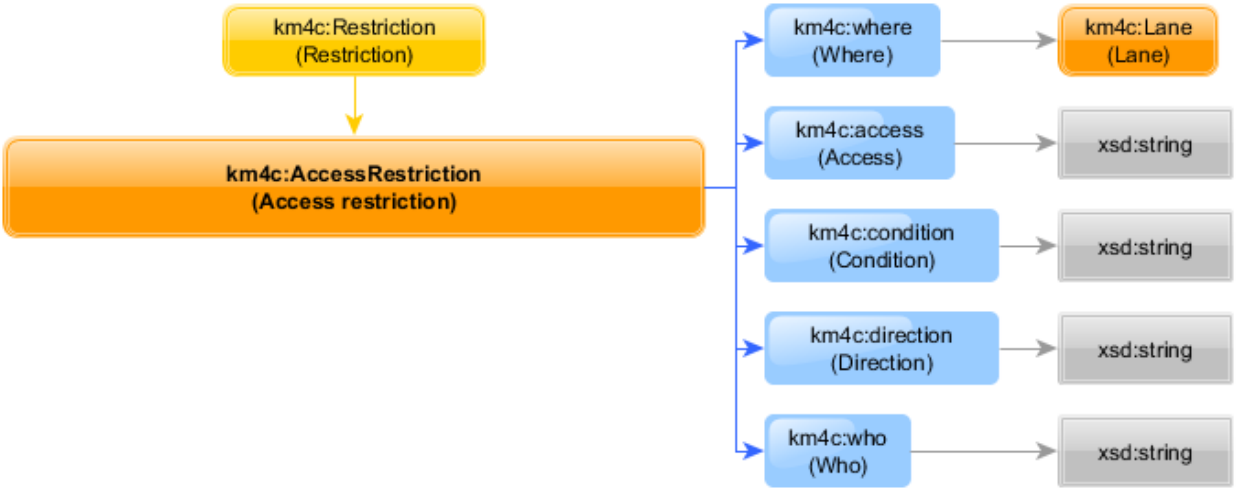

Figure 20 An outline of the Km4City AccessRestriction class

In OSM, most of the information about the traffic restrictions is contained within tags whose key includes the word *access* or *oneway*. Anyway, it could happen to find *day_on*, *hour_on*, *day_off*, *hour_off* tags applied to the same OSM element to which the access tag is applied. They indicate that the restriction is temporary, and they are taken into consideration for properly setting the property *condition*. For a complete understanding of what an *access* tag indicates, the tag key needs to be parsed. Indeed, if the restriction only applies to a specific traffic direction and/or if the restriction only applies to a specific category of vehicles, it is specified within the key of the tag. In truth, a combined parsing of both the key and the value of the tag is even necessary for a correct interpretation of the restriction, because the word *access* is sometimes suppressed within the key of the tag, especially when the restriction only applies to a specific category of vehicles. When the restriction applies to a road, the *access* property is set equal to the value of the *access*

tag. Instead, when the restriction applies to a set of lanes, which is denoted by the word *lanes* included within the key of the OSM *access* tag, the value of the access tag is parsed, a new instance of *AccessRestriction* is generated for each of the lanes, and the *access* property is properly set for each of the lanes. Also, the key of an *access* tag could contain the indication *:conditional*, and the value of the *access* tag could include the symbol @ followed by the condition that must be met so that the restriction could apply. In this case, the property *condition* is consequently set. It might even happen that the value of the *access* tag contains a set of different values, each associated with a specific condition. In this case, the value of the *access* tag is parsed, and a set of instances of *AccessRestriction* is generated, one for each of the value/condition pairs. When a *oneway* tag is applied to an OSM *element*, a restriction applies to all the vehicles about the direction that the traffic must follow. If the value of the tag is set to 1, it means that the positive direction only is allowed, while if the value is set to -1, it means that the negative direction only is allowed. The positive direction of a OSM *way* goes from the first listed toward the last listed of the child *node* elements that outline the path of the *way*.

The class MaxMinRestriction mostly represents restrictions about the maximum and minimum speed, and the size or the weight of the vehicles. Three properties are always set for each instance of the *MaxMinRestriction* class:

- *where*, which indicates the road or segment of road where the restriction applies;
- *what*, which describes the type of restriction (as an example, *maxspeed*);
- *limit*, which provides the imposed limitation (as an example, 50 km/h).

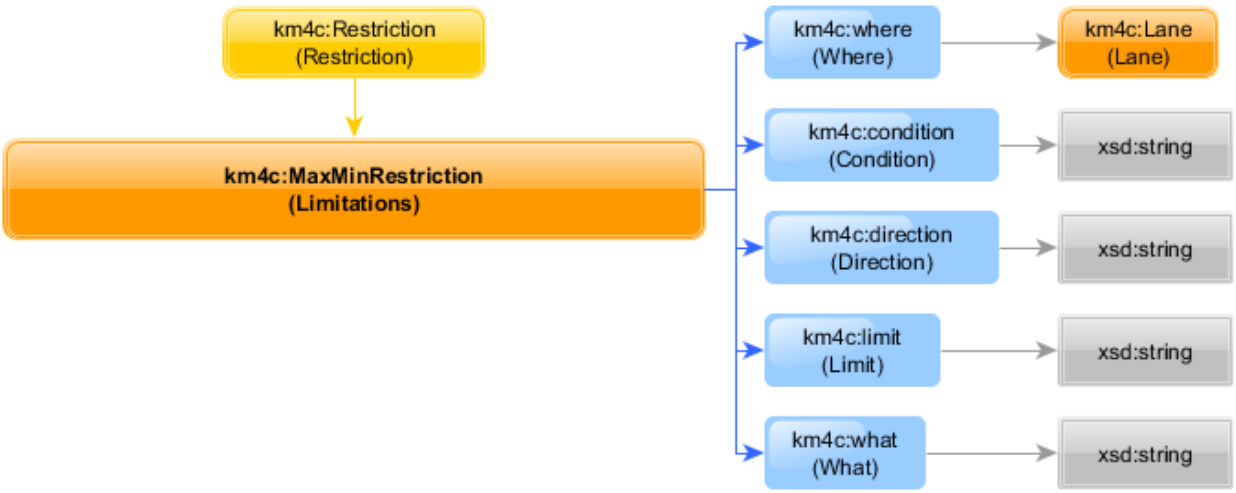

Figure 21 An outline of the Km4City MaxMinRestriction class

The property *condition* is set if the restriction only applies under specific conditions. Also, the property *who* is set if the restriction only applies to a specific category of vehicles. A new instance of the *MaxMinRestriction* class is generated for each OSM tag whose key includes one of the following: *maxweight*, *maxaxleload*, *maxheight*, *maxwidth*, *maxlength*, *maxdraught*, *maxspeed*, *minspeed*, *maxstay*.