REPORT ON DELIVERABLE D2.4

Final Mapping and Transformation Software

SLIPO

PROJECT NUMBER: 731581
START DATE OF PROJECT: 01/01/2017
DURATION: 36 months

Horizon 2020

SLIPO is a research project funded by European Commission's Horizon 2020 Programme.

Dissemination Level	Public
Due Date of Deliverable	Month 36 (31/12/2019)
Actual Submission Date	20/12/2019
Work Package	WP2 – POI Modelling, Identifiers, and Transformation
Tasks	Tasks 2.1-4
Type	Other
Approval Status	Submitted for approval
Version	1.0
Number of Pages	173
Filename	D2.4_Final_mapping_and_transformation_software.pdf

Abstract

This report presents the final version of the POI mapping and transformation software of the SLIPO system. First, we provide an overview of the SLIPO POI ontology and its extensions. In the following, we present the supported POI data harvesting and transformation services for third-pary data. Next, we provide an overview of TripleGeo, thoroughly analyse its components and explain its support for attribute mappings and hierarchical classification schemes for POIs. We also describe its functionality for reverse transformation from RDF to de facto geospatial formats. Finally, we report performance results from a comprehensive evaluation of the software that confirms its efficiency against a variety of input POI representations and formats, underscoring its efficent and scalable operation.

History

version	date	reason	revised by
0.1	02/09/2019	First draft	Kostas Patroumpas
0.2	22/09/2019	Contributions in all sections	Kostas Patroumpas, Dimitris Skoutas, Mohamed Sherif, Simon Scerri, Pantelis Mitropoulos, Simon Scerri
0.3	01/10/2019	Revisions in Sections 2, 3 and 4	Kostas Patroumpas, Michalis Alexakis
0.4	15/10/2019	Revised structure and introductions	Spiros Athanasiou
0.5	17/10/2019	Contributions in all sections	Kostas Patroumpas, Yannis Kouvaras
0.6	02/11/2019	Contributions in all sections	Kostas Patroumpas
0.8	20/11/2019	Updated experimental evaluation	Kostas Patroumpas, Michalis Alexakis
0.9	05/12/2019	Reviewed version; minor edits and improvements in all sections; updated Annexes	Dimitris Skoutas, Yannis Kouvaras, Michalis Alexakis
1.0	20/12/2019	Completed version for submission	Spiros Athanasiou

Author list

organization	name	contact information
ATHENA RC	Spiros Athanasiou	spathan@imis.athena-innovation.gr
ATHENA RC	Kostas Patroumpas	kpatro@imis.athena-innovation.gr
ATHENA RC	Dimitris Skoutas	dskoutas@imis.athena-innovation.gr
ATHENA RC	Yannis Kouvaras	jkouvar@imis.athena-innovation.gr
ATHENA RC	Michalis Alexakis	alexakis@imis.athena-innovation.gr
InfAI	Mohamed Sherif	sherif@informatik.uni-leipzig.de
IAIS	Simon Scerri	simon.scerri@iais.fraunhofer.de
GET	Pantelis Mitropoulos	pmitropoulos@getmap.gr
TomTom	Pauline Baudens	pauline.baudens@tomtom.com
WIGeoGIS	Kai Barenscher	kb@wigeogis.com

Executive Summary

This report presents presents the final version of the mapping and transformation software of the SLIPO system, comprising the SLIPO POI ontology, a series of data harvesting and transformation services for third-party data, and TripleGeo, our state-of-the art ETL software for geospatial features, capable of transforming millions of POIs to RDF in a few minutes.

TripleGeo is the *entry point* for POI datasets in the SLIPO lifecycle. Such data assets may come from a variety of spatial repositories (*databases, geographical files, Web APIs, etc.*) and may have been maintained under diverse schema representations. Hence, ETL processes, tools, protocols, and workflows are required for mapping and transforming such POI data and metadata into RDF adhering to the common, vendor-agnostic, and extendable SLIPO POI ontology.

TripleGeo implements interfaces for accessing POI data in diverse geospatial file formats and DBMSs and extracting them into RDF with geometries under the OGC GeoSPARQL standard. It also incorporates facilities that enable their mapping to an ontology according to well-established vocabularies, as well as assignment of POI entities to categories under a user-specified classification scheme. Since POI data need to be accessible and exploitable by existing software and services, TripleGeo also includes facilities for the *reverse transformation* of RDF triples into POI formats.

Compared to its interim release (ver.1.4) presented in [SLIPO-D2.2], its current final version v2.0 has improved support for a great variety of input POI representations and formats, can handle all thematic (i.e., non-spatial) attributes available per POI, and is able to transform millions of POIs in a few minutes. This testifies the ability of this state-of-the-art software to cope with very large POI datasets in scalable fashion.

The layout of this document is as follows.

In Section 1, we introduce the setting of the transformation task in the context of SLIPO. We describe the objectives of this task and offer a survey of related work on transformation of geospatial information to RDF. Finally, we briefly report our achievements regarding the development of TripleGeo until M36, and the features available in its final version 2.0.

In Section 2, we outline the OWL ontology defined and utilized in the context of SLIPO specifically for representing POI data assets. Applied in several real-world use cases, this agile, comprehensive, and vendor agnostic ontology is also extensible, as verified with two extensions utilized in the course of the project.

In Section 3, we present our software for POI data harvesting and transformation from third-party data (Web pages, Web APIs and data catalogues), which is complemented with a data exploration toolkit allowing to explore the data, compute statistics, identify patterns and analyse the distribution of values in POI attributes.

In Section 4, we discuss our overall progress in developing the TripleGeo software through its nine releases until the final version (ver.2.0), describing its improved functionality, efficiency and scalability. Next, we detail the transformation process as carried out by the final version of TripleGeo. This includes support for attribute mappings and classification schemes, an API for registering POIs in the SLIPO Identifiers Registry, as well as support for achieving scalability over large geospatial data assets via (a) multiple concurrent Java threads and (b) distributed processing over Apache Spark. Finally, we outline the processing flow of reverse

transformation and explain how it is possible to reconstruct RDF data of POIs with geometries into records stored in a geospatial file.

In Section 5, we evaluate the efficiency and scalability of TripleGeo against large POI datasets extracted from OpenStreetMap and stored in various geospatial repositories. This experimental evaluation confirms the versatility of the software and its effiency for handling Big POI data assets at global scale.

Finally, the Annexes we provide: (a) indicative examples on configuring TripleGeo execution, as well as sample mappings and classification schemes applied on POI datasets, (b) a user guide for TripleGeo along with indicative execution examples that demonstrate its operation against POI datasets, and (c) a user guide for our PDH and PHE toolkits.

Abbreviations and Acronyms

API	Application Programming Interface
CRS	Coordinate Reference System
CSV	Comma Separated Values
DBMS	DataBase Management System
EPSG	European Petroleum Survey Group
ETL	Extract-Transform-Load
GIS	Geographical Information Systems
GML	Geography Markup Language
GPS	Global Positioning System
GPX	GPS Exchange Format
GUI	Graphical User Interface
HDFS	Hadoop Distributed File System
INSPIRE	INfrastructure for SPatial InfoRmation in Europe
ISO	International Organization for Standardization
JDBC	Java DataBase Connectivity
JSON	JavaScript Object Notation
KML	Keyhole Markup Language
MBR	Minimum Bounding Rectangle
OGC	Open Geospatial Consortium
OSM	OpenStreetMap
OWL	Web Ontology Language
POI	Point of Interest
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
REST	REpresentational State Transfer
RML	RDF Mapping Language
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language

UML	Unified Modelling Language
URI	Uniform Resource Identifier
UUID	Universally Unique IDentifier
VM	Virtual Machine
WFS	Web Feature Service Interface (OGC Standard)
WMS	Web Map Service (OGC Standard)
WKT	Well Known Text
WGS84	World Geodetic System 1984 (EPSG:4326)
W3C	World Wide Web Consortium
XML	Extensible Markup Language
XSD	XML Schema Definition
XSL	EXtensible Stylesheet Language
XSLT	Extensible Stylesheet Language Transformation
YAML	YAML Ain't Markup Language

Table of Contents

1. Introduction 13

1.1. POI Data Transformation 13

1.1.1. Task Definition 13

1.1.2. Importance and Challenges 14

1.2. State-of-the-Art 14

1.3. Achievements – TripleGeo ver.2.0 16

1.3.1. New/Improved Functionality 17

1.3.2. Performance 17

2. The SLIPO POI Ontology 19

2.1. Overview 19

2.2. Classes 21

2.3. Object Properties 23

2.4. Datatype Properties 24

2.5. Extensions 28

2.5.1. Fusion 28

2.5.2. EV Charging Stations 30

3. POI Harvesting and Transformation 32

3.1. Introduction 32

3.1.1. Scope 32
3.1.2. Importance 35
3.1.3. Challenges 36
3.2. POI Data Harvesting 37
3.2.1. Features and Functionality 37
3.2.2. Implementation 39
3.3. POI Data Exploration 41
3.3.1. Implementation 41
3.3.2. Datasets and Field Types 43
4. The TripleGeo Software 47
4.1. TripleGeo ver.1.1 48
4.1.1. Legacy Implementation 48
4.1.2. Basic Features and Functionality 48
4.1.3. Performance 49
4.2. TripleGeo ver.2.0 50
4.2.1. New Features 50
4.2.2. Scalability 51
4.2.3. Powering POI data services 52
4.3. Features and Functionality 54
4.3.1. Libraries and Frameworks 59
4.3.2. Licensing 60
4.3.3. Documentation 60
4.4. Transformation to RDF 61

4.4.1. Architecture 61
4.4.2. Input and Output 63
4.4.3. Core Modules 65
4.4.4. Scalable Transformation 83
4.5. Reverse Transformation from RDF 84
4.5.1. Architecture 85
4.5.2. Input and Output 86
5. TripleGeo Experimental Evaluation 88
5.1. Datasets 88
5.1.1. Real POI Data extracted from OSM 88
5.1.2. Generation of synthetic POI data 92
5.2. Experimental Setup 92
5.3. Results 93
5.3.1. Transformation to RDF 93
5.3.2. Reverse Transformation 102
6. References 104
7. Annex I 108
7.1. Cardinality restrictions in POI ontology 108
7.2. Triple Geo Configuration Settings 111
7.2.1. TripleGeo Configuration for Transformation 111
7.2.2. TripleGeo Configuration for Reverse Transformation 114
7.3. TripleGeo Sample Mappings 114

7.3.1. Sample RML Mapping 114
7.3.2. Sample YAML Mapping 121
7.4. TripleGeo Sample Classification Schemes 123
7.4.1. Classification Hierarchy in CSV format 123
7.4.2. Classification Hierarchy in YAML Format 131
7.5. Reverse Transformation 139
8. Annex II: PDH and PDE User Manual 141
8.1. Initial Setup 141
8.2. Configuration 141
8.2.1. HTML Harvester Configuration 142
8.2.2. API Harvester Configuration 147
8.2.3. CKAN Harvester Configuration 148
8.2.4. Configuration for POI Data Exploration 149
8.3. Execution 150
8.3.1. Executing PDH 150
8.3.2. Executing PDE 152
9. Annex III: TripleGeo User Manual 158
9.1. Building Installation 158
9.2. Configuration Settings 159
9.2.1. Configuration for Transformation to RDF 160
9.2.2. Configuration for Reverse Transformation 166
9.3. Execution 167

9.3.1. Executing Transformation to RDF 167
9.3.2. Executing Reverse Transformation from RDF 168
9.3.3. Auxiliary Utilities 169
9.3.4. Known Limitations 170
9.3.5. Demonstration 171

1. Introduction

In this Section, we first define the transformation task and its goals regarding the integration of Points of Interest (POI) data. Then, we present the current state of the art in geospatial POI representation and transformation to RDF. Finally, we briefly outline the achievements of the final version 2.0 of the TripleGeo software, which provides the main transformation utility in SLIPO.

1.1. POI Data Transformation

1.1.1. Task Definition

Considering the POI integration lifecycle illustrated in Figure 1, the transformation process in SLIPO is a prerequisite for enabling processing against POIs as Linked Data. Indeed, transformation turns conventional POI and third-party datasets into RDF, thus enabling us to address the POI data integration challenge using the most effective technologies for this task: linked data. The data sources may be DBMSs, files, Web APIs, etc. which may provide a wealth of POI data assets, possibly employing diverse geometry representations, different attribute schemata, and possibly assigning categories to POIs under varying classification schemes. However, by transforming original POI data into a RDF representation according to a consistent, extensible, and rich ontology, all tasks involved in POI data integration (i.e., interlinking, fusion, quality assurance, etc.) are applicable in the Linked Data domain. Once transformation to RDF complete, POI features can be interlinked, fused, and enriched in successive steps that increase the size and the quality of the POI data in a virtuous cycle that implements an iterative workflow as shown in Figure 1.

However, it should be stressed that transformation is actually a *two-way process* that also allows the *reverse transformation* of linked POI data into conventional formats (i.e. de facto POI formats), thus enabling existing products, systems, and services to exploit the integrated POI datasets.

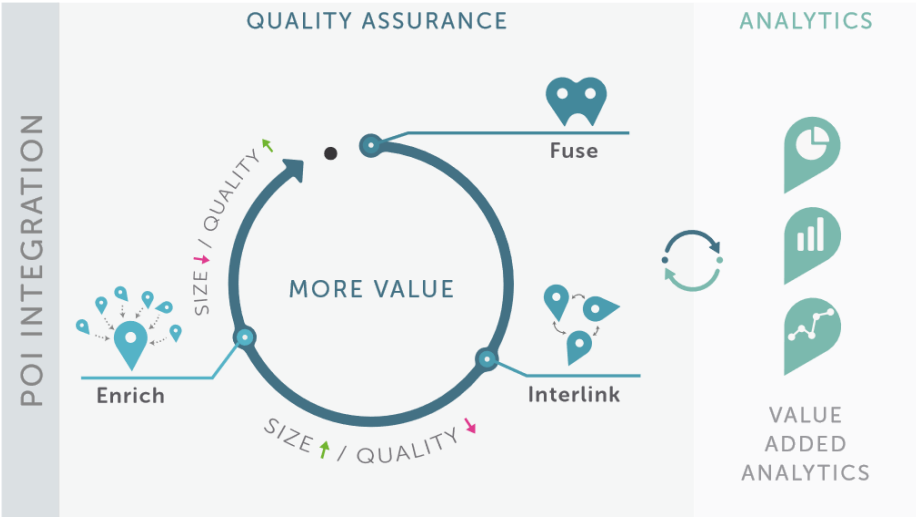


Figure 1: POI integration lifecycle

1.1.2. Importance and Challenges

The primary pillar and underlying idea of SLIPO is to address the POI data integration challenge by applying Linked Data technologies, which are ideally suited to handle the inherent geospatial, thematic, and semantic ambiguities of POIs. Hence, the entry point for POI datasets in the SLIPO lifecycle comprises transforming POI and POI-related Big Data assets into RDF data conforming to common and extendable schemas. This includes all necessary ETL processes, tools, protocols, and workflows for transforming POI data from diverse and disconnected data sources, schemas and formats, to RDF data adhering to a common, vendor-agnostic, well defined, yet agile and extendable SLIPO POI ontology.

Data sources can be distinguished into two main categories: (a) POI entity descriptions and metadata, and (b) auxiliary data that complement, enrich, and contextualize POIs. The first category includes datasets represented by a series of different POI formats, each one carrying its own attribute schema. To handle these datasets, schema mappings that map concepts and attributes of the individual, conventional schemas to classes and properties of the SLIPO POI ontology are applied through data transformation tools. These tools implement interfaces and parsers for extracting POI data from diverse format sources, including specific format files, DBMSs, Web APIs, etc. To handle arbitrary schemas found in data under the second category, the transformation tools incorporate facilities for manual, semi-automatic and automatic schema mapping. To support this functionality, several well-established vocabularies are incorporated so that the mapping process utilizes standardized concepts and terms in the POI ontology. After the transformations are applied, POI data need to be accessible and exploitable by existing software and services (e.g., GIS, spreadsheets), so transformation also includes facilities for the reverse transformation of RDF triples into industry-standard POI formats.

1.2. State-of-the-Art

Creating knowledge from structured (e.g., relational databases, XML) or unstructured sources (e.g., text, images) can be extremely valuable in the Semantic Web. The R2RML Recommendation [R2RML] by W3C specifies an RDF notation for mapping relational tables, views or queries into the RDF data model. An alternative W3C recommendation concerns a direct mapping of relational data to RDF [RDFDirect] defines a simple transformation that can be used to materialize RDF graphs or define virtual graphs, which can be queried by SPARQL. Besides, the more generic RDF Mapping Language RML [DSC+14],[DKF+15] is defined as a superset of R2RML and aims to support not only relational databases, but also other data sources (CSV, XML, or JSON formats). RML can express customized mapping rules from heterogeneous data structures and serializations to RDF.

A variety of tools and scripts [ConverterToRdf] have been proposed for transforming data from an application-specific format into RDF for use with RDF tools and integration with other data, most of them considered as proof-of-concept prototypes [UHAS12]. Some of them are rather "mature" tools for transforming relational databases into RDF, such as Triplify [ADL+09], D2R Server [D2RServer], or Virtuoso's Sponger [Sponger]. During conversion, these tools allow reuse of existing vocabularies and ontologies. However, these tools lack support for geospatial data and operations.

On the other hand, several *Extract-Transform-Load (ETL) tools* can manage the unique characteristics of spatial data. Among them, GDAL/OGR [GDAL] is an open-source translator library implementing the OGC vector model [OGC10] and can handle proprietary storage models for many geospatial DBMSs and files. GeoKettle [GeoKettle] is a metadata-driven spatial ETL tool dedicated to integration of different data sources for building and updating geospatial data warehouses. Finally, FME Workbench is included in ESRI's ArcGIS Data Interoperability extension [FMEworkbench] and enables transformation of geometric and thematic attributes along with schema redefinitions. Currently, such utilities are mainly used for data cleaning, merging, verification or conversion into various formats, but have absolutely no RDF support.

There have been several proposals for geospatial RDF data management such as [GeoJSON],[GeoRDF], [GeoPos84], but none provided a solid framework for developing large-scale applications and services. Since 2012, the OGC GeoSPARQL standard [OGC12] suggests a concrete ontology for representing features and geometries in RDF as Well Known Text (WKT) or Geography Markup Language (GML) literals. GeoSPARQL defines a core set of classes, properties and data types that can be used to construct query patterns in an extension of SPARQL. To cope with incompatible methods for representing and querying spatial data, GeoSPARQL follows other OGC standards [OGC10]. With such standardization, both vendors and users can achieve uniform, transparent, platform-independent access to geospatial RDF data with a rich collection of query operators. RDF stores have been increasingly adhering to implementations of the GeoSPARQL standard (e.g., Oracle Spatial and Graph [Oracle], Parliament [Parliament], or Virtuoso [Virtuoso]), but still some triple stores still maintain their own proprietary geometric representations (e.g., AllegroGraph [AllegroGraph]).

To the best of our knowledge, there have been very few initiatives specifically for converting *geospatial features into RDF resources*. *Geometry2RDF* [geo2rdf] enables geospatial data conversion from different formats (ESRI Shapefile, GML and geospatial DBMS) into RDF [VVS+10] according to the *NeoGeo vocabulary* [NeoGeo] and is not compliant with the GeoSPARQL standard [OGC12]. Data conversion into an appropriate RDF format using a selected ontology is among the functionalities supported by the generic *DataLift* platform [DataLift]. Although initially geometries could be extracted as WKT strings under a custom namespace in order to be queryable with SPARQL, support for GeoSPARQL-compliant geometries was included with its module *GeomRDF* [HABF14]. By default, GeomRDF generates predicates by reusing attribute names, which can be replaced afterwards by predicates from a vocabulary using another module using another Datalift module. In a different approach, *LinkedGeoData* [LGD] aims at adding a spatial dimension to the Semantic Web. It offers a flexible platform for mapping OpenStreetMap (OSM) data [OSM] to RDF, a SPARQL endpoint for making RDF data publicly available, as well as several tools for performing mappings and interlinking of geospatial semantic data. The resulting graph comprises billions of triples interlinked with DBpedia [DBpedia] and GeoNames [GeoNames]. Nevertheless, spatial operations deal strictly with OSM nodes and ways, ignoring any other geographic sources or data types. Specifically for Oracle Spatial and Graph, custom *RDF views* [OracleRDFViews] on relational data can be created via SQL queries or use of non-standard mappings (either with a direct mapping or with an R2RML mapping document) and physically store the generated triples in an RDF model within Oracle. As Oracle supports GeoSPARQL vocabularies, this data is also queryable with topological operations and spatial functions.

GeoTriples [KVS+14],[KSV+18] extends mapping languages R2RML and RML with new constructs specifically for transforming geospatial entities into RDF. Although not adhering to a specific geospatial vocabulary, it

supports the GeoSPARQL standard. It automatically generates and processes such mappings from diverse geospatial formats including relational databases (PostGIS and MonetDB), ESRI shapefiles, XML, GML, KML, JSON and GeoJSON documents and CSV documents. In that respect, GeoTriples seems share many of the capabilities of our transformation tool TripleGeo. However, it lacks support for as many input formats as TripleGeo, it does not include any support for reverse transformation, and its scalability against large geodatasets may suffer because of the complexity of RML mappings. Most importantly, GeoTriples also lacks any specific support for transformation of POI data (e.g., classification schemes). Besides, mappings generated by GeoTriples can be used by the *Ontop-spatial* extension [BK16],[BXK+16] of the Ontology-Based Data Access (OBDA) system *Ontop* [RR15]. This way, users can view their data sources virtually as linked data through on-the-fly GeoSPARQL-to-SQL translation on top of relational databases using ontologies and mappings. This is similar in spirit with Sparqlify, and this process does not require any transformation of data. This is similar in spirit with Sparqlify [Sparqlify] employed in the LinkedGeoData portal [LGD], and does not require any transformation of data.

Towards developing technical specifications for the representation of *Points of Interest on the Web*, a working group was created, initially under the auspices of W3C, but later moved under the OGC. Although still work in progress [OGC-POI], this representation does not cover the issue of transforming POIs from external sources to RDF. In a separate approach, the SDI4Apps project [SDI4Apps] has also developed a data model for POI data and have created a SPARQL endpoint containing millions of POI features, but neither do they explain how this data have been mapped to their ontology nor provide any details on their transformation to RDF. Of course, there are tools that enable extraction of POI entities from datasets. For example, [OsmPoisPbf] scans an OpenStreetMap (OSM) file for nodes and areas (and relations) whose tags indicate them as POIs and extracts those into a text CSV file. To the best of our knowledge, no tool for direct transformation from POI representation to RDF is available.

As we elaborate next, our transformation sotware TripleGeo was initially based on Geometry2RDF [geo2rdf], but since its inception it has been substantially enhanced by integrating access to several external geospatial file formats and DBMSs and providing support for GeoSPARQL data types. In its final release (ver.2.0), TripleGeo includes support for attribute mappings according to a given ontology, assignment of categories according to a classification scheme, customizations specifically targeting transformation of POIs, as well as many other features that establish it as the **state-of-the-art software for transforming geospatial features (and particularly POIs) to RDF [PSM+19]**.

1.3. Achievements – TripleGeo ver.2.0

Version 2.0 is the release of our ETL software TripleGeo available at M36 of the project. Based on the source code of its ver.1.1, which was originally developed in the context of the GeoKnow project [GeoKnow], there have been **nine releases** in the framework of the SLIPO project with particular focus on POI-specific mapping and transformation functionalities. Next, we enumerate the new features, as well as the improved functionality and performance of TripleGeo.

1.3.1. New/Improved Functionality

- **Improved geospatial support**. TripleGeo supports extraction of data from various geospatial repositories, either de facto geographical file formats (e.g., ESRI shapefiles, GML, CSV, etc.), or geospatially-enabled DBMSs (e.g., Oracle Spatial, PostGIS). This includes improved handling not only of primitive geometry types (like points, linestrings, or polygons), but even more complex geometries (MultiPolygons, Geometry Collections), as well as on-the-fly transformation to another coordinate reference system (CRS reprojection).
- **Compliance to standardization initiatives**. This mostly concerns RDF geometries, which can be fully *compliant with the OGC GeoSPARQL standard* (2012), and indirectly to other OGC standards [OGC07],[OGC10]. Moreover, TripleGeo provides support for *INSPIRE-aligned data/metadata* [INSPIRE], abiding by the INSPIRE Directive by the European Commission that sets the legal and technical foundations towards interoperable Spatial Data Infrastructures across Europe.
- **User-defined mappings for transformation of thematic attributes**. Such mappings from original attributes to properties according to a given ontology, allow transformation of all available attributes per input entity. In the SLIPO Workbench, TripleGeo is also coupled with a semi-automatic utility for guiding the user into creating new mappings.
- **Customized URIs**. TripleGeo allows auto-generation of UUIDs for each transformed feature, so that these can be used for assignment of URIs according to namespaces prescribed by an underlying ontology (specified by the user).
- **Reverse transformation from RDF to de facto geographical file formats**. This utility enables users to obtain a certain amount of the semantic information and metadata as attributes in conventional geodata formats (like ESRI shapefiles or CSV).
- **Specialization on POI data**. Although TripleGeo was conceived and remains a *general-purpose* ETL tool to the RDF data model, we have included specific support for transformation of POI data into RDF. So, the software includes support for RML mappings to handle extra thematic attributes into RDF according to the SLIPO POI ontology. For improved efficiency, we have also prepared an alternative mapping facility specifically tailored for SLIPO ontology. TripleGeo also provides support for hierarchical classification schemes regarding POI categories and also yields auxiliary output of basic attributes per POI feature in order to be used by the SLIPO Identifiers Registry.
- **Integration with the SLIPO Workbench**. TripleGeo has been successfully integrated into the SLIPO toolkit and POI data integration workflows. This allows users to interact with the tool in a coherent, user-friendly, and flexible manner under the entire SLIPO data lifecycle. Details on this integration are provided in [SLIPO-D1.4].

1.3.2. Performance

- **Efficiency.** We have conducted a series of tests regarding both transformation and reverse transformation against a variety of geospatial repositories, confirming that TripleGeo can now handle any number of thematic attributes and map them to a given POI ontology. It should be also noted that we have prepared configurations and mappings specifically for transforming datasets

supplied by our SLIPO partners, verifying that TripleGeo can offer consistent RDF representations for POI data from diverse providers.

- **Scalability.** Experimental results testify that TripleGeo ver2.0 achieves orders of magnitude performance gains compared to its original release, and can now efficiently transform millions of POIs in a few minutes. Thanks to POI data partitioning schemes that take advantage of modern cluster infrastructures, TripleGeo offers advanced effiency and scalability against very large POI datasets covering countries, continents, and even the entire planet.

2. The SLIPO POI Ontology

2.1. Overview

Next, we present the *OWL ontology* designed for representing POIs internally in SLIPO. Transforming POI data into this RDF representation, allows us to transfer the tasks involved in POI data integration (i.e., interlinking, fusion, enrichment, etc.) into the Linked Data domain, and thus address the involved challenges using Linked Data technologies.

The POI data model used in SLIPO together with its extensions and intuitive visualizations are publicly available in the project's GitHub repository[1].

Prefix	Namespace
slipo	http://slipo.eu/def
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns
rdfs	http://www.w3.org/2000/01/rdf-schema
xml	http://www.w3.org/XML/1998/namespace
xsd	http://www.w3.org/2001/XMLSchema
geo	http://www.opengis.net/ont/geosparql

Table 1: Namespaces utilized in the OWL ontology specified in SLIPO for POIs

[1] https://github.com/SLIPO-EU/poi-data-model

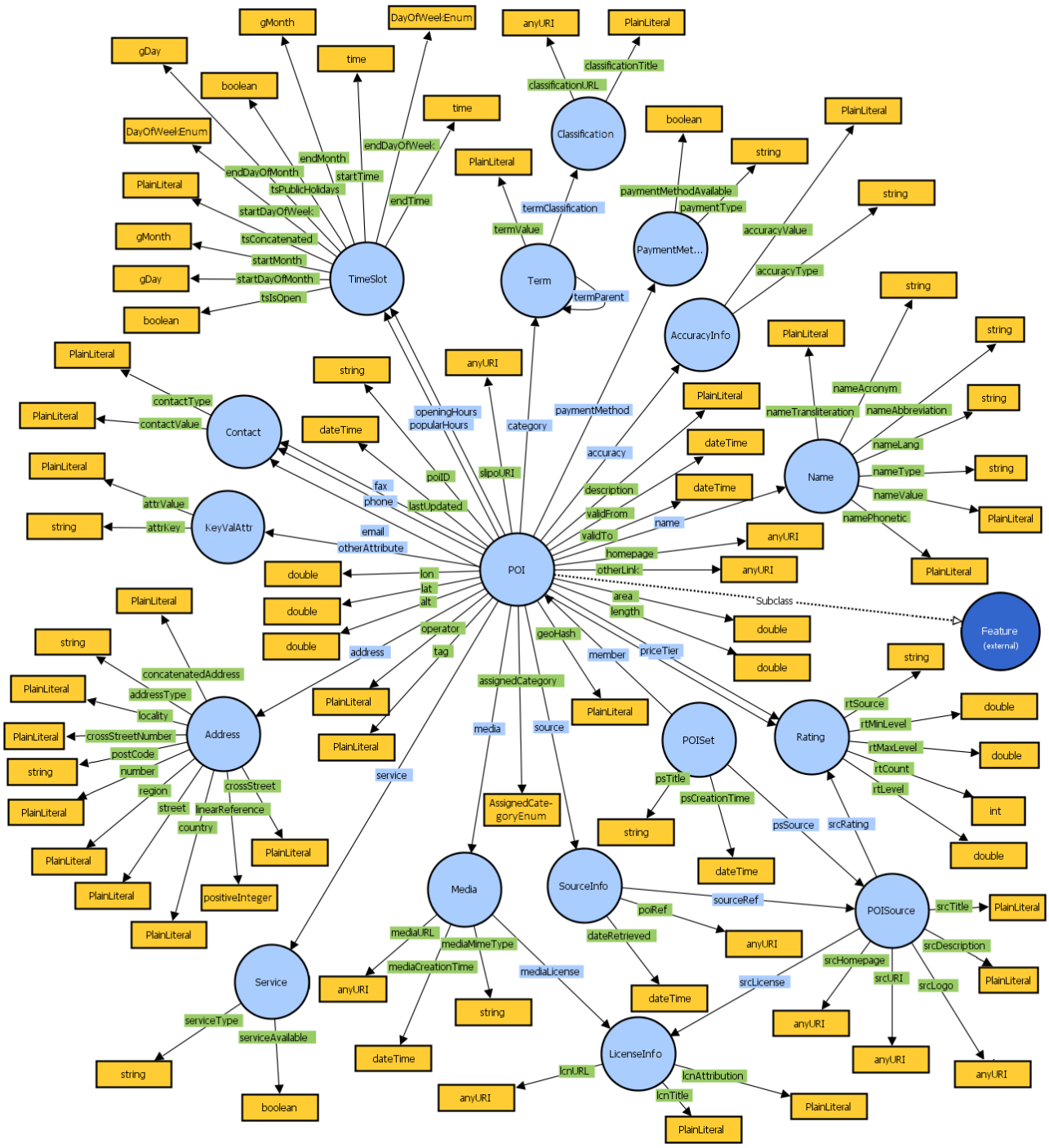


Figure 2: Overview of the SLIPO OWL ontology for POIs

Table 1 lists the namespaces used in designed OWL ontology. Figure 2 illustrates a graph depicting the classes, the object properties, as well as the data properties contained in the SLIPO ontology. The source file for the complete OWL ontology, its corresponding XML schema, as well as several visualizations of the ontology are available online in the project's repository. In the following, we list the classes and properties included in the ontology.

2.2. Classes

The following classes are defined in the OWL ontology:

- *POI* is the main class for POI features and is modelled as subclass of a spatial Feature in GeoSPARQL [OGC12], thus directly inheriting properties regarding their geospatial location. It supports multiple geometric representations, and the type of each geometry (e.g., centroid, navigation point, map pin, boundary) may be specified as well. In addition, the model enables specification of point locations according to the Basic Geo (WGS84 lat/long) Vocabulary [GeoPos84], as well as altitude information (which is not yet included in GeoSPARQL).
- *POISet* represents a collection of POIs (i.e., a POI dataset).
- *POISource*: A source of POI data (e.g., a commercial vendor) may be specified using a *Universal Resource Identifier* (URI), title, homepage, detailed description, license, logo, etc.
- *Classification*: A classification scheme that is applied to a POISet and assigns a *category* (e.g., restaurant, bar, theater, etc.) to each POI. Classification may be possibly hierarchical, having its terms (i.e., categories, subcategories, etc.) in multiple levels with a parent-child relationship. We do not enforce a common, predefined classification to POI data coming from diverse sources, but allow each dataset to retain its own. However, in SLIPO we also employ an embedded classification of 15 broad categories, which greatly facilitates POI data integration, as discussed in Section 4.4.3.4. Matches or conficts of POI categories at any level may be resolved during POI data integration in the Linked Data domain.
- *Term* can be used to specify a category at any level in a classification scheme. Typically, each major category (e.g., food) may be specialized into several subcategories (e.g., restaurant, fast food, pizza, etc.).
- *AccuracyInfo* models accuracy assessment for the properties of POI, including the type of accuracy metric (e.g., positional, geocoding, thematic) and its value.
- *Address* information for a POI may have diverse representations per region or country. The model is flexible to accommodate addresses with street name, house number, postal code, intersection of streets, as linear reference along a road, etc., or even unstructured addresses as string literals.
- *Contact* information may include phone number(s), fax, and email address(es), each with an optional characterization (e.g., mobile, direct line for phones).
- *Name* class supports various naming conventions for POIs: abbreviations, acronyms, phonetic transcriptions, transliterations, etc. as well as characterizations for language and type (e.g., official, alternate, brand, historical).
- *LicenseInfo* covers license information (attribution, title, URL) either for a POI source or for media objects related to POIs.
- *Media* may refer to photo, video or audio associated with a POI and models their URL, MIME type, and creation timestamp, as well as their license.

- *PaymentMethod* indicates whether and which particular forms of payment are accepted (e.g., cash, credit/debit card).
- *Rating* may be used to model the ranking of a POI (with properties such as the rating value, the number of votes, the rating scale, etc.), but also its priceTier as perceived by users.
- *Service* indicates whether and what type of service is offered in a POI (e.g., parking, wifi, air conditioning, room service).
- *SourceInfo* provides information on the data source, including the time this POI was retrieved and its original identifier.
- *TimeSlot* can capture when a POI is open (openingHours) or mostly visited (popularHours) in various temporal granularities (months, weeks, weekdays, hours), and can also handle special cases (e.g., operation in public holidays).
- *KeyValAttribute* can be used to store extra attributes for special cases in the form of key/value pairs.

The following list outlines the aforementioned classes defined in the SLIPO OWL ontology:

Class	Description
AccuracyInfo	Accuracy information for a POI's data.
Address	A POI address.
Classification	A POI classification.
Contact	Contact information.
KeyValAttribute	Placeholder for other attributes in the form of key/value pairs.
LicenseInfo	License information.
Media	Information of a media object.
Name	A POI name.
PaymentMethod	Payment method offered by a POI.
POI	The class of a POI. It is defined as a subclass of *geo:Feature*.
POISet	A set of POIs.
POISource	A POI data source.
Rating	A rating value and scale.
Service	Service offered by a POI.
SourceInfo	Source information for the POI data.
Term	A term of a classification.
TimeSlot	A time slot.

Table 2: Classes specified in the OWL ontology for POIs

2.3. Object Properties

The following object properties are defined:

Property	Domain	Range	Description
accuracy	POI	AccuracyInfo	Accuracy information for a POI's data.
address	POI	Address	Address information for a POI.
category	POI	Term	The category the POI belongs to.
email	POI	Contact	Email address of a POI.
fax	POI	Contact	Fax number of a POI.
media	POI	Media	A media object associated with a POI.
mediaLicense	Media	LicenseInfo	License information for a media object.
member	POISet	POI	Member POI of a POI set.
name	POI	Name	Known names of a POI.
openingHours	POI	TimeSlot	Time slots when a POI is open or closed.
otherAttribute	POI	KeyValAttribute	Other attribute of a POI.
paymentMethod	POI	PaymentMethod	A single payment method accepted by a POI.
phone	POI	Contact	Telephone number of a POI.
popularHours	POI	TimeSlot	Time slots when a POI is mostly visited.
priceTier	POI	Rating	The price tier of a POI.
psSource	POISet	POISource	The source of a set of POIs.
rating	POI	Rating	The rating of a POI.
service	POI	Service	Service provided by a POI.
source	POI	SourceInfo	The source of a POI.
sourceRef	SourceInfo	POISource	A URI identifying the POI source.
srcLicense	POISource	LicenseInfo	License information for a POI source.
srcRating	POISource	Rating	An overall quality rating for a POI source.
termClassification	Term	Classification	The classification the term belongs to.
termParent	Term	Term	A parent of this term.

Table 3: Object properties defined in the OWL ontology for POIs

2.4. Datatype Properties

The datatype properties that had been defined in the initial version of the SLIPO ontology in [SLIPO-D2.1] have proved suitable and versatile to cover all requirements of real-world use cases, as detailed in [SLIPO-D5.2], with only few extra datatype properties being added (*area*, *assignedCategory*, *geoHash*, *length*, *operator*). The fact that only such minor additions were required to the initial model confirms its robustness and flexibility in representing POI data assets and their various features for a wide range of applications.

Overall, the following list indicates the datatype properties defined in the ontology:

Property	Domain	Range	Description
accuracyType	AccuracyInfo	xsd:string	The accuracy metric. E.g., positional accuracy, geocoding accuracy.
accuracyValue	AccuracyInfo	rdf:PlainLiteral	The value of the accuracy metric.
addressType	Address	xsd:string	The type of an address.
alt	POI	xsd:double	The altitude of the POI's location.
area	POI	xsd:double	The area of the POI's geometry. This is only valid for POI entities represented with a surface (e.g., a polygon or multipolygon). By default, the area value is expressed in square meters.
assignedCategory	POI	xsd:string	Specifies an internal category assigned to each POI according to a default (embedded) classification scheme of 15 categories used internally in SLIPO.
attrKey	KeyValAttribute	xsd:string	The key of a custom attribute.
attrValue	KeyValAttribute	rdf:PlainLiteral	The value of a custom attribute.
classificationTitle	Classification	rdf:PlainLiteral	The name of the classification scheme.
classificationURL	Classification	xsd:anyURI	A link to the web page of the classification scheme.
concatenatedAddress	Address	rdf:PlainLiteral	Concatenated, unstructured address information.
contactType	Contact	rdf:PlainLiteral	The type of the contact information.
contactValue	Contact	rdf:PlainLiteral	The contact information itself.

country	Address	rdf:PlainLiteral	Country name or code.
crossStreet	Address	rdf:PlainLiteral	Name of second street (for intersections).
crossStreetNumber	Address	rdf:PlainLiteral	Number of second street (for intersections).
dateRetrieved	SourceInfo	xsd:dateTime	The time when the POI was retrieved from the respective source.
description	POI	rdf:PlainLiteral	A description of a POI.
endDayOfMonth	TimeSlot	xsd:gDay	End day of month of a time slot.
endDayOfWeek	TimeSlot	DayOfWeekEnum	End week day of a time slot.
endMonth	TimeSlot	xsd:gMonth	End month of a time slot.
endTime	TimeSlot	xsd:time	End time of a time slot.
geoHash	POI	rdf:PlainLiteral	Encodes the geographic location of the POI into a short string of letters and digits. By default, a string length (precision) of 8 is used in SLIPO; length values outside of [1,12] are rejected
geomType	geo:Geometry	xsd:string	The kind of geometry. E.g., center, map, navigation, shape.
homepage	POI	xsd:anyURI	A link to the official web page of the POI.
lastUpdated	POI	xsd:dateTime	The time when the information of a POI was last updated.
lat	POI	xsd:double	The latitude of the POI's location.
lcnAttribution	LicenseInfo	rdf:PlainLiteral	Attribution information.
lcnTitle	LicenseInfo	rdf:PlainLiteral	License title.
lcnURL	LicenseInfo	xsd:anyURI	License URL.
length	POI	xsd:double	The length of the POI's geometry. This is only valid for POI entities represented as lines or surfaces (e.g., a polygon or multipolygon); in the latter case, it represents the perimeter of the surface. By default,

			the length value is expressed in meters.
linearReference	Address	xsd:positiveInteger	Distance (in meters) from the start of the street.
locality	Address	rdf:PlainLiteral	City or village.
lon	POI	xsd:double	The longitude of the POI's location.
mediaCreationTime	Media	xsd:dateTime	The time when the media object was created.
mediaMimeType	Media	xsd:string	The MIME type of a media object.
mediaURL	Media	xsd:anyURI	The URL of a media object.
nameAbbreviation	Name	xsd:string	The abbreviated form of the name.
nameAcronym	Name	xsd:string	The acronym form of the name.
nameLang	Name	xsd:string	The language of the name.
namePhonetic	Name	rdf:PlainLiteral	The phonetic transcription of the name.
nameTransliteration	Name	rdf:PlainLiteral	The transliteration of the name to another script.
nameType	Name	xsd:string	The type of the name. E.g.: default, official, alternate, brand, historical.
nameValue	Name	rdf:PlainLiteral	The name.
number	Address	rdf:PlainLiteral	House number.
operator	POI	rdf:PlainLiteral	The name of the POI's operator, such as the company name, the person running the business, etc.
otherLink	POI	xsd:anyURI	Additional link for a POI.
paymentMethodAvailable	PaymentMethod	xsd:boolean	Whether a payment method is available.
paymentType	PaymentMethod	xsd:string	The type of a payment method. E.g.: cash, credit-card, debit-card, prepaid-card, bank-transfer, paypal, check, mobile.
poiID	POI	xsd:string	A unique identifier for a POI.
poiRef	SourceInfo	xsd:anyURI	A URI identifying the POI at the respective source.

postCode	Address	xsd:string	Postal code.
psCreationTime	POISet	xsd:dateTime	The creation time of a POI set.
psTitle	POISet	xsd:string	The title of a POI set.
region	Address	rdf:PlainLiteral	State or province.
rtCount	Rating	xsd:int	The number of votes from which this rating score has been produced.
rtLevel	Rating	xsd:double	The value of a rating.
rtMaxLevel	Rating	xsd:double	The maximum level of a rating scale.
rtMinLevel	Rating	xsd:double	The minimum level of a rating scale.
rtSource	Rating	xsd:string	The source of this rating information.
serviceAvailable	Service	xsd:boolean	Whether the service is available.
serviceType	Service	xsd:string	The type of a service. E.g.: accessible, free-parking, closed-parking, valet-parking, drive-through, free-wifi, wifi, restrooms, reservations, outdoor, airconditioning, delivery.
slipoURI	POI	xsd:anyURI	The SLIPO URL of a POI.
srcDescription	POISource	rdf:PlainLiteral	A description of a POI source.
srcHomepage	POISource	xsd:anyURI	Link to the official web page of a POI source.
srcURI	POISource	xsd:anyURI	A URI identifying a POI source.
srcLogo	POISource	xsd:anyURI	Link to a logo to be displayed for a POI source.
srcTitle	POISource	rdf:PlainLiteral	The title of a POI source.
startDayOfMonth	TimeSlot	xsd:gDay	Start day of month of a time slot.
startDayOfWeek	TimeSlot	DayOfWeekEnum	Start week day of a time slot.
startMonth	TimeSlot	xsd:gMonth	Start month of a time slot.
startTime	TimeSlot	xsd:time	Start time of a time slot.
street	Address	rdf:PlainLiteral	Street name.
tag	POI	rdf:PlainLiteral	A single tag assigned to a POI.
termValue	Term	rdf:PlainLiteral	The term.

tsConcatenated	TimeSlot	rdf:PlainLiteral	Concatenated, unstructured form of opening hours information.
tsIsOpen	TimeSlot	xsd:boolean	Whether a time slot indicates open or closed status.
tsPublicHolidays	TimeSlot	xsd:boolean	Specifies whether the POI is open or closed during public holidays.
validFrom	POI	xsd:dateTime	Starting time of a POI.
validTo	POI	xsd:dateTime	Ending time of a POI.

Table 4: Datatype properties defined in the OWL ontology for POIs

2.5. Extensions

The above presented OWL ontology is a generic, top-level model that can accommodate different types of POIs. But in certain applications that specialize on particular types of POIs, it may be desirable to extend these schemas to include application-specific attributes for those particular POI categories. During the pilot of the project, we have used and occasionally adapted this model in order to address the requirements and needs of the specific applications used for testing and evaluating SLIPO. Next, we discuss two such extensions of the core SLIPO ontology that handle two important aspects: (i) keeping track of the *provenance* of POIs and their properties as they evolve through a data integration process, and (ii) enhancing the model for representing POIs with many extra properties for a *specific use case*. Both these extensions confirm the flexibility of the core OWL ontology to represent POI data assets for a wide range of use cases and practical requirements in real-world applications.

2.5.1. Fusion

During data integration tasks, POIs are linked, enriched, and eventually fused, so it is important to monitor their *provenance*, i.e., changes occuring to them. Since POI entities are by nature volatile with several of their properties changing over time especially after *fusion* with other POI entities, the SLIPO ontology has been extended with new metadata for keeping track of such changes. This includes coverage for several *fusion actions* (e.g., a change in POI metadata due to correction of information), the *pair of POIs* that led to a fused POI, as well as *statistics* regarding the applied fusion actions. Mechanisms for recording the provenance of the data/changes and the individual changes on POI metadata have been supported, so that the full history of fused POI entities can be maintained and queried.

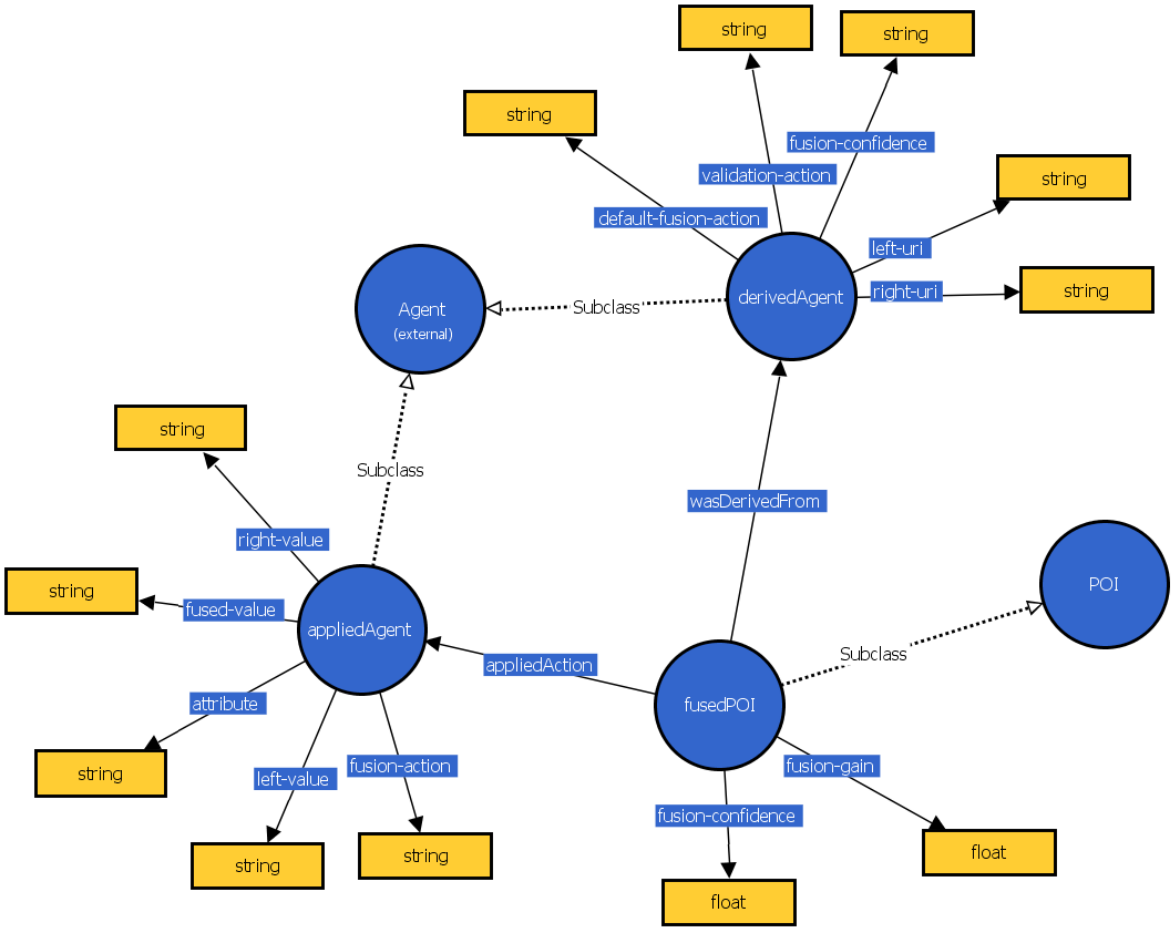


Figure 3: Visualization of the OWL ontology that extends the SLIPO data model to represent provenance of fused POIs

To keep track of changes during fusion, the ontology should be able to represent all relevant value transformations. Towards this, we have made use of the PROV ontology [PROV-O] recommended by W3C to support the interchange of provenance on the Web. As illustrated in Figure 3, our ontology extension in OWL[2] defines two additional classes by subclassing the Agent class from PROV-O:

- The *derivedAgent* class is used to represent how a *fused POI* has been derived, i.e., its association with the pair of POIs involved in its creation and the fusion action applied. It also records the default fusion action, the validation action, as well as the obtained fusion confidence value [SLIPO-D3.7].
- The *appliedAgent* class represents provenance of a particular *property* (i.e., a POI attribute value) after a fusion action has been applied. Therefore, it lists the property name (attribute), the two original values (left/right) and the fused one, as well as the fusion action (e.g., concatenate the two input value, keep the left one).

The resulting *fusedPOI* is represented by a subclass of the main POI class in the SLIPO ontology. It is equipped with properties that serve as links to an instance of the *derivedAgent* class (to keep track of its creation), as well to as many instances of the *appliedAgent* class as the properties changed during fusion. Also note two quality metrics associated with the fused POI, namely *fusion-gain* that estimates the

[2] https://github.com/SLIPO-EU/poi-data-model/blob/master/fused-poi-model.owl

percentage of extra properties compared to the original (e.g., 40% additional attribute values), and *fusion-confidence* that indicates the degree of similarity (in names, geometry, phone number, etc.) between the original features that were fused into a unified one.

2.5.2. EV Charging Stations

As detailed in [SLIPO-D5.1], DINUC C.1 concerns data integration regarding *charging stations for Electric Vehicles* (EV) from various sources. In terms of data modelling, this data includes generic properties as typical POIs (name, category, contact details, etc.), but also provides a wide range of characteristics specific to EV charging stations only.

In order to represent an EV charging station in the SLIPO ontology, we defined it as a subclass of POI, i.e.:

slipo:EVChargingStation rdfs:subClassOf slipo:POI

and has been associated with the additional set of custom properties listed in Table 5. Note that this set of extended properties is not exhaustive, but only covers the most important characteristics of EV-related POIs required in this scenario.

Property Name	Description
EVnumberOfPoints	Number of charge points available.
EVstatusType	The type of the EV charging station, e.g.: residential, public, portable.
EVconnectionType	The type of connection according to relevant standards.
EVconnectionFormalName	Name of the connection.
EVlevel	Charging Level (low, medium, high).
EVquantity	Number of available connections.
EVcurrentType	Supply type (AC Single phase, AC three phase, DC)
EVcurrentTypeDescription	Full description of the supply type.
EVaccessComments	Accessibility information (sometimes including opening hours).
EVcomments	Additional text information concerning other connection characteristics.
EVvoltage	Max Voltage.
EVamps	Max Current (Amps).
EVpowerKW	Max Power (kW).
EVisFastChargeCapable	Flag indicating whether fast change capability is available.
EVisMembershipRequired	Flag indicating whether membership is required for users to access the facility.

Table 5: Extended properties in the SLIPO ontology for EV POI characteristics

This ontology extension was implemented in OWL[3] and has been applied for the transformation of EV-related input data into RDF as detailed in [SLIPO-D5.2]. Figure 4 illustrates the data properties associated with the defined POI subclass for representing EV charging stations.

[3] https://github.com/SLIPO-EU/poi-data-model/blob/master/EV-poi-model.owl

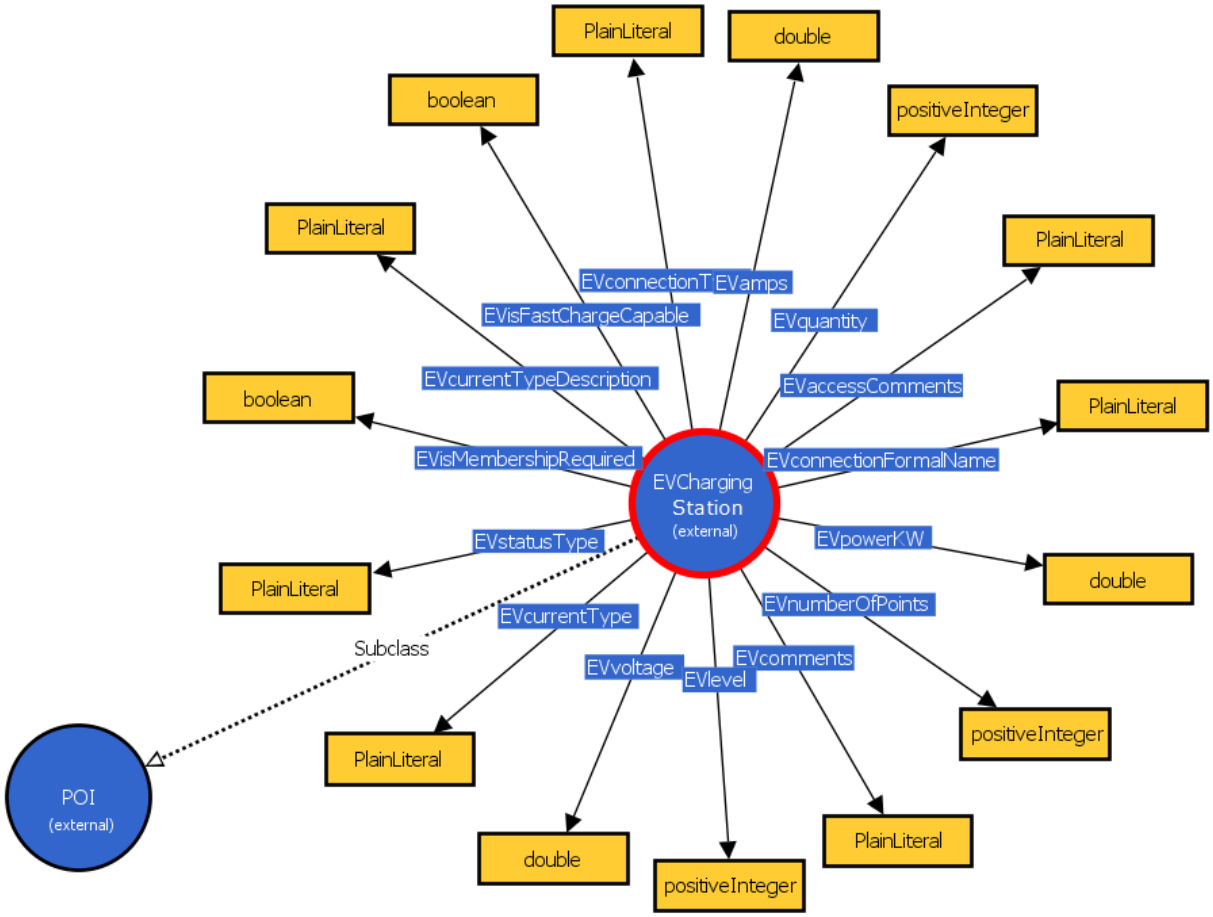


Figure 4: Visualization of the OWL ontology that extends the SLIPO data model with extra properties for EV charging stations

3. POI Harvesting and Transformation

In this section we present POI data harvesting and transformation software for third-party data assets. We have designed and implemented a harvesting toolkit that provides functionalities for extracting and retrieving POI data from Web pages, Web APIs and data catalogues. This is complemented with a data exploration toolkit allowing to explore the data, compute statistics, identify patterns and analyse the distribution of values in POI attributes. This is essential for facilitating the user in generating insights over the harvested data, in order to appropriately steer the subsequent steps of the harvesting and integration workflow when applying third-party data. We present the functionalities provided by these tools, explain how they can be configured and used, and provide some indicative examples.

3.1. Introduction

During the last years, the amount and wealth of information that is available on the Web about locations and Points of Interest (POIs) is constantly growing. This involves a large variety of diverse types of data and sources, including both authoritative sources and user-generated content, which can be downloaded from data catalogues, extracted from Web pages or queried from Web APIs. Thus, collecting and sifting through such available POI data from third-party sources is typically a tedious, time consuming and error-prone process requiring a lot of manual effort. Our goal in this task is to provide a set of tools that can assist the user in this process, reducing the manual effort involved.

Before we delve into the technical details, we explain in this section the scope of our work and give an overview of the developed tools, outlining also their importance and the challenges involved.

3.1.1. Scope

Our efforts in this task have focused on designing and developing two sets of tools. The first involves a framework for *POI data harvesting* (**PDH**), while the second comprises tools for *POI data exploration* (**PDE**).

In the context of the SLIPO POI data integration workflow, the purpose of PDH is to enable and facilitate the extraction and retrieval of third-party POI data, including the extraction of structured, semi-structured and unstructured POI data from heterogeneous sources and formats. The data in question concerns in principle attributes of POIs, as for example the name of a POI, the address or the coordinates, user ratings about the POI, opening hours, etc. However, the framework itself is designed to be agnostic with respect to the nature of the data so that it can be used to extract more kinds of information. This feature could be proven very useful in the extraction of additional information *indirectly* related to POIs, which could even include dynamic information, such as weather data or official documents somehow associated to a POI.

The sources of the data can vary from structured data repositories of a specific and fully defined format, to a completely unstructured Web site. Different tools of the framework apply to each case, with a specific configuration, describing the source type and format, supplied by the user. In this perspective, three

different tools are considered: (i) a generic one, aiming to harvest sources lacking a specific, structured format; (ii) one intended to harvest sites offering a Web API; and (iii) one for collecting data from open data catalogues.

The PDE toolkit aims at complementing the results of the harvesting tools. It addresses data exploration, and in particular the assessment of a given harvested dataset and the provision of insights regarding the presence of outliers and other potential data quality issues. Although PDE is designed to be generic and extensible, so that it can be adapted and applied to other types of data besides POIs, specific functionalities are provided tailored to POI data. Moreover, the usage of the PDE tool is not specifically tied to the output of the harvesters, but it can be applied to any available input data.

PDE can handle two types of data, numerical and strings. In case of numerical data, standard statistical tests are performed, returning the data distribution. For strings, a basic pattern analysis is provided that can give an overview of the dominant patterns inside the dataset, offering to the user an overview of the nature of the dataset. In addition, more specific tests can be performed when specifically dealing with certain POI attributes.

Concluding, PDH and PDE combined provide a service for harvesting and transforming POI data from third-party sources, allowing the user to assess their quality before further integration and analysis takes place within a data integration workflow. In the following, we provide a more elaborate overview of these software components.

3.1.1.1. POI Data Harvesting

The PDH framework consists of three separate tools, with somewhat different functionality to accommodate different types of data sources. Two of them are focused on structured data sources. Usually, the extraction of data from such sources is more straightforward. However, these sources are often dynamic, hence these tools can facilitate the data collection process by being executed periodically to maintain the input data up-to-date.

More specifically, the following tools are included:

- PDH-HTML. This addresses POI data extraction from unstructured sources, in particular, Web pages. It operates in two main stages. For a given Web site, it first generates a list with the URLs of the Web pages containing the data we are trying to collect. Then, the harvester retrieves the HTML code of each Web page, and parses it to extract the relevant POI data. For both stages, configuration provided by the user is required. Inevitably, this requires that the user has some basic knowledge about HTML in order to provide information about the HTML structure and setup of the pages to be parsed. All the required pieces of information are provided to the harvester in terms of a comprehensive configuration file in JSON format.
- PDH-API. This is used to retrieve data from sources offering an HTTP RESTful API. Although the details of each API may of course differ from case to case, we have found that the same generic pattern is observed in several POI data sources. This allows, for example, to implement a specific harvester for retrieving weather information about POIs from https://openweathermap.org, and then it is relatively easy to extend and adapt this same harvester to accommodate a different or an additional weather provider that offers a similar Web API. Typically, a weather provider offers an API

accepting requests about particular places, as well as a list with the supported places and their codes. Thus, regardless of the specific details of the API, the underlying functionality of the harvester will typically involve reading an input list of POIs with their coordinates or place names, then issuing a series of requests to the API to retrieve relevant weather information for each one, and finally updating the results.

- PDH-CKAN. This addresses the case of data catalogues, and in particular, CKAN catalogues [CKAN]. We have chosen to focus on CKAN-based catalogues, since this covers a very large portion of the open data catalogues available online [ODM]. The harvester in this case is configured to search with tags and/or a query string, in order to retrieve data from the catalogue. It can then update any previously collected data with the newly retrieved ones.

3.1.1.2. POI Data Exploration

PDE assists the user in obtaining an overview and insights about an unknown harvested dataset. A dataset usually consists of several fields and each could potentially contain a huge number of different values, numeric or string. Conceptually, a dataset can be represented by a table, with a column for each field and the rows as the actual data. In our current implementation, the input to PDE is a CSV file, with the first row defining the name of the fields, and each value separated from the others with a comma.

PDE can be used to extract insights regarding the entire dataset. The user can retrieve basic information about the shape of the file (e.g., number of rows and number of columns) as well as information about the structure of the fields, e.g., which of them appear to be categorical, and in this case, which are their distinct values.

Moreover, when a specific field (column) is selected, more detailed information about the data contained in this field is computed and presented to the user. This includes several common statistical parameters of the contained values. When dealing with strings, information such as the number of missing values, the number of unique values, their minimum and maximum length, the dominant value and its frequency, are computed. Finally, for numeric types, the distribution parameters (or equivalently the distribution itself) is also computed to provide an overview of the data.

For the case of textual content, we perform a pattern analysis based on regular expressions. First, the tool attempts to identify certain patterns in the input data, and subsequently to compute their frequencies. In this way, the user is provided with a pattern distribution. Each pattern reveals information about the nature of the characters contained in a string, for example if these are Latin or Greek words, numeric and their length, symbols, mixed concatenation of characters, etc. For each pattern, an example is given to be more comprehensible. This case is considered as generic, since the tool is yet completely agnostic about the nature of the data and performs a general analysis based on these identified patterns.

In addition, in the case of categorical data (i.e., when the values of a field are drawn from a relatively small set of distinct values), the tool is able to automatically identify these categories. In fact, it is robust with respect to cases where each category may consist of several words and multiple categories are contained within each record. Once these categories have been found and extracted, their frequency distribution inside the data is calculated and provided to the user.

All functionality described so far is *completely agnostic* to the type of the specific field, which makes our software more broadly useful and generally applicable to various types of data. Nevertheless, since we focus on data concerning POI attributes, the analysis is made more specialized and optimized for field types which are frequently met.

Thus, in this specific analysis, the user can choose the type of the field under consideration. Specifically, the supported field types are the following:

- name
- address
- phone number
- price range
- rating
- opening hours.

For the first three, we apply a predefined set of patterns. Instead of constructing the patterns from the data as in the generic case, this particular set of patterns is used, and as usual their frequency and finally their distribution is calculated. These patterns have been designed to be as informative as possible for the specific field type. For the rest of the supported field types, a different approach is followed since these field types are of numeric nature. Once they have been stripped from any accompanying strings surrounding the numbers (the relative place of text and numbers, however, is taken into consideration), they are classified into appropriate bins, resulting in a distribution of the contained values.

3.1.2. Importance

The importance of the PDH and PDE tools is apparent in practical applications. One can use various sources of third-party POI data to update, enrich or even just confirm, existing POI data collections. A harvesting tool able to extract POI data under various formats and structures is required for this purpose. Subsequently, PDE is necessary for allowing the user to shift through this data to identify patterns, potential errors, and other useful insights to guide the next steps of the integration and analysis. For large datasets containing thousands or millions of POIs, it is clearly not feasible for the user to gain an overview about the usefulness and applicability of the data by just taking a glance at it. Hence, a data exploration tool is indispensable. The collected data often contains errors, such as missing or inconsistent values. As such, the exploration and assessment of the data can provide insights about such errors, also known as *data glitches* [DLS14], and thus guide the user to clean the data before any further processing and analysis on it is executed.

Through PDE, the user can identify potential data glitches, and then decide how to proceed. For example, the user can either clean the outliers from the data or decide that the harvesting procedure could be supplied with a more fine-tuned configuration file. Cleaning the data can take several forms. For instance, one can define a norm for the deviation of each value from what is considered as average, and based on this norm decide whether a value in the original dataset should be discarded or needs to be updated.

3.1.3. Challenges

3.1.3.1. POI Data Harvesting

Harvesting from a data repository offering an API is a relatively straightforward process. However, harvesting data from unstructured sources can be quite challenging. In this case, one needs to parse and analyse HTML documents with the desired data scattered over the entire content. Sometimes, each piece of information is enclosed in a specific tag with a descriptive attribute. Nevertheless, it is not less common that distinct pieces of information intended to be harvested (corresponding in distinct attributes) are not distinguishable in HTML terms.

In order to confront this case, one approach is to demand from the user an extremely detailed configuration file, describing precisely the HTML structure and which pieces should be harvested. Following this path is still an option when designing a tool for specialized users, even though the process of configuration would be nonetheless time consuming. The approach which was chosen, however, follows a different path. Specifically, the HTML structure of the web page is treated as a Document Object Model (DOM). The user has to provide just one characteristic for each piece of information that she wants to harvest. This characteristic could be the id of the parent HTML element or any other unique attribute, but it could also be a constant text from the visible HTML page, for example a title.

Another challenge that had to be confronted during the Web harvester tool design was a way to obtain a list of all the Web pages (URLs) containing the content to be harvested. In practice, harvesting is a two-stage process. In the first stage, it harvests the URLs (supplied with a similar configuration file), while in the second stage the actual harvesting happens, locating and extracting the content of each URL harvested in the previous stage.

3.1.3.2. POI Data Exploration

One common issue in data exploration is the determination of outliers [DFMZ18]. An outlier is defined as an observation that lies an abnormal distance from other values in a random sample from a population. Outliers are not necessarily abnormal values. It is up to the analyst to decide whether to reject or not the outliers. In any case, outliers contain valuable information about the process of data collection, and it is important to understand the reason of their appearance.

When dealing with harvested data, or in general with data containing different types of POI attributes, there is no standard way on defining a distance between values in a population sample. This is the main challenge the current tool had to confront, and at the same time its main goal. Instead of explicitly identifying the outliers themselves, the tool constructs a pattern distribution. By visualizing this distribution, the analyst can then figure out the alienated patterns, and decide whether these patterns match with the specific kind of data or not.

Another main challenge in the process of exploration was the case of categorical fields. It is very common that each value contains more than one categories. This is not a problem as long as the categories are explicitly separated. However, in the absence of a delimiter (which may be missing due to errors in harvesting or because it is indeed missing in the original data), the categories need to be determined automatically. One option is to check the frequency of a particular sequence of words. If a sequence appears always

unseparated, then it is likely that it represents a single category. It is possible, however, although very unlikely, that two categories always appear together, and with the exact same order, thus misleading the tool to recognize this pair of categories as a single category.

3.2. POI Data Harvesting

Next, we describe our POI data harvesting toolkit, giving some technical insights. We present the types of data sources that are supported and outline its features and functionalities. We also present the software's architecture and configuration, giving some corresponding examples.

3.2.1. Features and Functionality

Three different harvesters are provided, as outlined in Section 3.1.1.1, namely PDH-HTML, PDH-API and PDH-CKAN. Each one is responsible for harvesting a specific type of data source, thus jointly covering a large number of data sources existing on the Web. PDH-HTML can be used to extract information from Web pages. PDH-API can be used to retrieve data from sources offering a RESTful Web API. Finally, PDH-CKAN can be used to collected data from open data catalogues operating on the CKAN platform.

In all cases, the results are stored in an output file in JSON format or, for the case of PDH-HTML, in CSV format. The user can also choose whether to include all the results or to write only the updates with respect to a previous run.

The source code for PDH is available at the GitHub repository of the SLIPO project[4] and is available with an Apache License v2.0. In the following, we describe each harvester in more detail.

3.2.1.1. HTML Harvester

PDH-HTML is designed to extract POI data from Web pages by parsing and processing the HTML code of the page, given instructions provided in an accompanying configuration file. It operates in two steps. First, it fetches a list containing the URLs of the Web pages of interest (e.g., obtained by parsing a page within the Web site that lists all available POIs of a certain category). Then, it parses and extracts the content of each page asynchronously.

Before we proceed into describing the two steps separately, we provide some details about the harvesting procedure itself. This procedure, although it presents some variations during the two steps, it has many common features.

First, we assume that the Web site to be harvested contains several HTML pages with a similar structure. Note that this is typically the case in practice. Inside these pages there exist certain elements with the information we would like to collect. The task is to retrieve these pieces of information in a systematic way and assign each one of them to a specific field. In order to achieve our goal, we have to identify the common pieces among all the structures which are associated with the information we are searching for. This procedure, which has to be manually performed by the user, produces the content of the configuration file

[4] https://github.com/SLIPO-EU/poi-data-harvesting

provided to the harvester. This content can be logically separated into two parts, corresponding to the two steps of the harvesting process.

For the first step, the one associated with retrieving the list of page URLs, the most essential element is a starting point, which should be provided by the user. The starting point is a URL from which the harvester will start searching. This initial step yields a list of links, which may be either the final list or an intermediate list. In the latter case, the procedure iterates itself, taking each link as the new starting point, until it finally reaches the complete list with the links of the content that is going to be harvested.

In practice, given a starting point, two pathways can be followed. The harvester requests the starting point URL and tests the response to determine whether it is a JSON string or not. The technique used in the two cases is different. In case it is a JSON string, retrieved by an AJAX request, the response is transformed into an object and is searched by its keys. In the second case, the response is HTML content and HTML harvesting takes place. In both cases, the results are saved as an object and the user has the option either to stop the process at this point, obtaining this list which would be written in a JSON file, or to proceed to the second step, the actual process of POI data extraction. For the former case, the output file can be the input for future harvesting, without having to retrieve each time the URL list.

Once the list is granted, either obtained in a previous harvesting step or given directly in a JSON file, the harvester starts requesting the URLs of the list. The requests occur simultaneously, with the maximum number of simultaneous connections defined by the user. Each time a response is obtained, the HTML content is transformed into a DOM object in order to make parsing and searching easier. The harvester searches the DOM for tags, which have been defined by the user in the configuration file, and it assigns the values it finds to the corresponding fields. These tags could be, for example, titles in the text, the common pieces across all the web pages mentioned previously, or an attribute of an HTML tag (e.g., the id, the class or any other attribute).

3.2.1.2. API Harvester

PDH-API covers a common functionality met on various APIs on the Web. The APIs under discussion have their data structured in a specific way. Each dataset is located behind a specific key (id), so that one could get results in case a key is provided along with the API request. The functionality of this tool shows some similarity with the HTML harvester, in the sense that its first task is to create a list with the URLs corresponding to the total set of keys. This is usually achieved by a specific request prescribed by the user.

When this first step is completed, the harvester proceeds to the second step which is the actual retrieval of the data. In contrast to the HTML harvester, in this second step there are two options. Either it proceeds as usual by iterating over all the URLs of the constructed list, or it executes a query over the list and provides only the relevant results. For example, this harvester can be used to fetch weather data. The keys in this case correspond to the city codes. The user may be interested in specific cities rather than the entire set of results. Moreover, the weather provider may have imposed limits on the number of requests offered free of charge.

3.2.1.3. CKAN Harvester

PDH-CKAN can collect data from CKAN-based data catalogues. CKAN is an open source, web-based management system for open data distribution. CKAN is *the de facto* open data catalogue, in use from the majority of open data catalogues worldwide, justifying the specialized support we provide.

CKAN stores data separated in datasets and resources. A dataset typically comprises two parts. The first is information about the data, known as metadata. Metadata could be, for example, the format of the actual data, the publishing authority or the type of licence under which the data is published. The second is the data itself. The data is stored inside resources. A resource could be, for example, a file or a Web page. The format of the resource is free, it could be a JSON or GeoJSON file or even a flat CSV. Additionally, the resource could be part of the repository system or it could be provided by (and stored in) external sources.

Each dataset may be associated with a specific tag, while each tag may be assigned in multiple datasets. Moreover, a vocabulary is a collection of tags. As a result, each tag may be either a free tag or a tag contained in a vocabulary.

One option offered to the user is to harvest every single dataset contained in a CKAN catalogue. In this case, the application, as in the case of HTML harvesting, proceeds in the two familiar steps. First, it requests a list with the total number of packages (datasets). The CKAN API responds to such requests with a list containing the ids (actually the names, the two terms are used interchangeably by CKAN in this case) of the datasets. Once the list has been fetched, multiple asynchronous requests are issued to fetch the actual dataset corresponding to each item in the list.

Alternatively, the user can issue queries on the datasets. There are two ways of searching inside a CKAN catalogue. One is by searching among the tags attached to each dataset (which could be optionally grouped in vocabularies). The second is by issuing a query over all of the available fields of the datasets. Obviously, one could combine the two methods, e.g., to query the datasets with a free text and filter the query with specific tags (and/or vocabularies).

3.2.2. Implementation

PDH has been developed in PHP, around two fundamental tools. One of them is the cURL project[5], which provides a library for transferring data using various protocols. The other is the DOM extension[6] of PHP, which provides the DOMDocument class allowing to operate on XML documents through the DOM API with PHP. Both tools are essential for the functionality of the harvester. More specifically, cURL is used by all the three harvesting tools, while the DOM extension is required only by the HTML harvester.

To increase the functionality and usability of these tools for the specific needs of the harvesting procedure, two custom classes have been developed. CurlTools provides a couple of simple, yet useful tools for data transferring. One offers the default XML connection options, while the other simplifies the connection with several URLs asynchronously. At the same time, it offers the possibility to monitor the memory usage in order to limit the simultaneous connections if a certain threshold is exceeded (since the processing occurs simultaneously).

[5] https://curl.haxx.se/

[6] http://php.net/manual/en/book.dom.php

The cURL library has the benefit of using a callback function for requests, which is called each time a request has been fetched. This way, the script is being processed in parallel, decreasing significantly the time needed to harvest a whole Web site, although the harvesting time is of course strongly dependent also on the network connection.

DOMTools is a class that offers tools to construct the DOM structure of the HTML page and query for keywords in the tree, as well as to create the list containing the URLs to be harvested and ultimately a tool which provides the actual harvesting from an HTML structure. Optionally, before constructing the XML structure, DOMTools use the tidy PHP extension, which is a binding for the HTML Tidy[7] clean and repair utility. It is called only in the case the extension is available, in order to provide with a correct HTML code, although it is not essential in the workflow (minor errors in the HTML code do not affect the procedure).

For searching efficiently inside the HTML structure, the HTML code is transformed into an XML DOM tree and then the XPATH query language is used [XPATH]. XPATH provides the ability to navigate around the tree representation of the XML document, selecting nodes using various criteria. In practice, when using the HTML harvester, one provides the script with keywords for which the HTML code of the Web pages would be searched for. These keywords specify the node selection criteria, searching for ids or class XML attributes (standard HTML attributes) as well as plain text.

The procedure described above is performed by the method findTag of the DOMTools class. This method is used to provide with an array containing all nodes with the requested characteristics. The harvestData method, on the other hand, iterates over the nodes of the structure as well as their children in order to find the data contained in the requested tags. Finally, constructList iterates in order to extract the href attribute and create the URL list.

The general architecture of the three different harvesters is essentially the same. They are built around a main class, which extends a parent class called Harvester. Since the procedure followed by the three tools is quite different, the functionality of the main class is limited to basic aspects, such as parsing the input arguments, opening the appropriate files and writing the output in the requested format. CurlTools is as well initialized in the parent class and it is saved in a public variable available to its children.

The CKAN and API harvesters utilize only CurlTools since they only need to call an API service. Their main classes, CKANapi and ApiHarvester respectively, are designed to take advantage of the corresponding API functionality. Since the structure of these tools is quite straightforward, we are going to focus below on HTML harvesting. However, in all cases there are two common methods. One is collect and the other is fetchData. Even though for each case the implementation is somewhat different, the former is used to retrieve the list of URLs, while the latter is the actual data extraction method.

For HTML harvesting, two classes have been implemented. One of them, the LinksList, is responsible for fetching the list of URLs (in case no AJAX functionality is provided), while the other, the WebHarvester, performs the actual harvesting on HTML pages, after possibly creating a list of links in case it is provided in JSON format. Although the main functionality for HTML harvesting is provided by DOMTools,the purpose of the WebHarvester is to provide some useful tools, covering the details. For example, WebHarvester

[7] http://www.html-tidy.org/

provides a method for extracting coordinates from a Web page, using regular expressions, as well as a method to detect and fix the Web page encoding.

The main methods of WebHarvester, however, are as in all cases two. The collect method collects all the URLs from a Web site according to some predefined criteria. These criteria are the configuration provided to the script, resulting in the URL list to be harvested. The second method is called fetchData, which does the actual job of data extraction. The output of these tools can be either a CSV file or a JSON document.

Although both methods for list retrieval and data extraction employ similar XPath queries from DOMTools (with the exception of AJAX requests), there is one major difference in their operation. Collect fetches the URL content one by one, while the harvesting works asynchronously. This choice is inevitable, since, for the case of URL collection, the URLs that follow have to be retrieved from the currently requested content. Once the list is constructed, the harvester could in bulk target on the URLs and simultaneously process their content.

The harvesting toolkit is completed with a PHP wrapper script as an example of applying the tools described so far, after selecting the appropriate harvesting method (given by the user). For example, for the case of HTML harvesting, the script reads a configuration file in JSON format, which contains all the information required to create the URL list as well as those pieces of information required to harvest appropriately these URLs. As expected, this script runs in two rounds. In the first one it constructs the list, whereas in the second it feeds the list to the actual harvesting procedure. However, it is possible to adjust the script accordingly. If, for example, the URL list is already provided, the first round is omitted.

3.3. POI Data Exploration

Next, we describe the POI data exploration toolkit. First, we provide an overview of its architecture, emphasizing on a technical description of its parts that could also be reused independently or integrated in other applications. Next, we give a more detailed analysis of each component, starting with the description of pattern and category recognition procedures, followed by the description of the statistical analysis corresponding to each field type. We conclude by giving some examples using these components and demonstrating their functionality.

The source code for PDE is available at the GitHub repository of the SLIPO project[8] and is available under the Apache License v2.0. In the following, we describe each harvester in more detail.

3.3.1. Implementation

The PDE tool has been developed in Python 3. Python is a widely used interpreted programming language, with rich open source library support, especially for applications focusing on numerical analysis and statistics. Version 3 presents many improvements over the previous major version 2. An important one for this project is the native support for UTF data instead of ASCII of version 2. Additionally, the lack of libraries ported to the new version has significantly improved recently.

[8] https://github.com/SLIPO-EU/poi-data-exploration

Further, we exploit a number of third-party libraries. Specifically, we take advantage of the libraries numpy[9] and pandas[10], which are very commonly used in projects dealing with statistics. Python does not natively support array data structures. Numpy is a library that supplements the programming language with the feature of large, multidimensional array data structures, along with a rich ensemble of mathematical functions. Pandas is a software library focusing on data manipulation and analysis. It extends the numpy arrays to the structure of dataframes with integrated indexing. It also supplies its dataframes and series objects with several methods for analysing their data.

Along with these libraries, the tool also takes advantage of some standard third-party modules. It uses the re module to compile and search for regular expressions, similar to those found in Perl. Additionally, it uses the functionality offered by OrderedDict of collections, which is an alternative to Python dictionaries and, also, the unicodedata to manipulate strings in a language independent way. Finally, the json module is used to transform python dictionaries to JSON format.

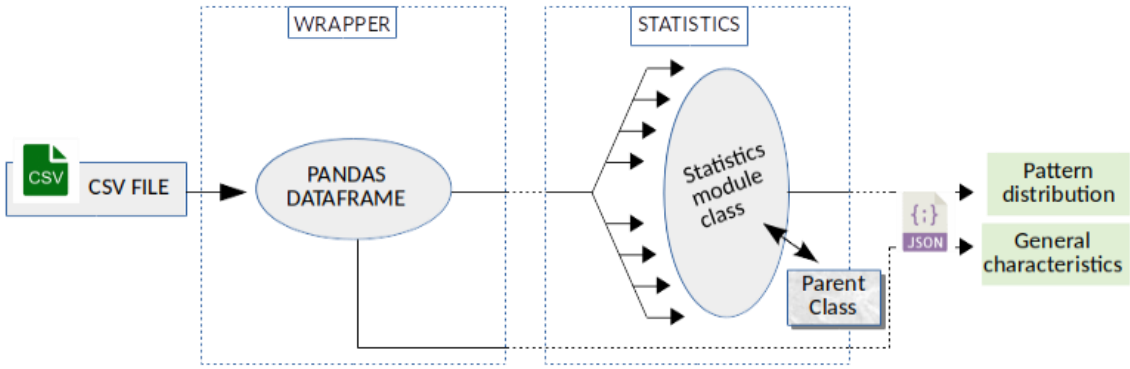


Figure 5: PDE processing flow.

The processing workflow of the PDE toolkit is shown in Figure 5. PDE essentially consists of a Python package called statistics. This package is a collection of eight modules. Each module is a class adding a certain functionality to the parent class of the package. The modules correspond to the categories of the data fields, each performing the respective tests. PDE has been supplemented with a wrapper module, which can also be used as a standalone script.

Each module can be initialized with a pandas series, which typically represents the data of a specific field (i.e., the data collected for a particular attribute for the POIs). During the initialization, all the required methods are called to run the appropriate tests, producing a public dictionary with the final data. This dictionary is accessible from the wrapper which transforms it to JSON and saves it as a JSON file, suitable to directly transform the data into a chart (typically a pie or a bar chart).

[9] http://www.numpy.org/

[10] https://pandas.pydata.org/

Apart from the specific results, the wrapper class offers an independent functionality. The statistics package is dedicated to the analysis of each field separately. The wrapper class prepares the data for analysis, while also offering methods to extract general statistical information about the whole dataset. Briefly, the wrapper class can give the shape of the dataset and, additionally, its structure with the categorical fields and their unique values. This material is, as usual, transformed into JSON and written in another output file, in case the wrapper class is called from the default embedded script.

3.3.2. Datasets and Field Types

The actual input to the PDE toolkit is a pandas dataframe object. However, for a more user-friendly approach, the script asks the user to provide the dataset in the format of a CSV file. In any case, the dataset in either format consists of a number of columns, with each column comprising data for each field (i.e., attribute) of the entity under investigation. These columns of the dataset are internally represented by pandas series objects.

The pandas series object is passed to the appropriate module in order to analyse it. The module selection is accomplished according to the type of the field. A field type may fall into a predefined category or not. In the former case, certain predefined patterns are provided and are searched for in the values of this field to find matches. In the latter case, we attempt to identify whether the values of this field appear to be arbitrary or drawn from a set of distinct values (e.g., POI categories).

3.3.2.1. Identifying Patterns and Categories

Patterns are identified by the module generic of the statistics package. In this module, an iteration over the data series takes place. For each string, a pattern is constructed according to some predefined character groups (character sets, symbols, numeric values, etc.). The patterns, afterwards, are compiled into groups with specific sequence and, given this schema, a corresponding regular expression is generated.

Once we have a regular expression matching the specific string, the entire series is tested against this expression. This test is performed using the pandas regular expression toolkit. The frequency of the regular expression appearance is saved under a user-friendly name to describe the pattern, along with the initial string as a representative example of the pattern. Since the pattern of these strings has been identified, they are removed from the series, and the procedure continues with the next non-matching string.

In case the field is of categorical type, one expects each entry to consist of a sequence of words. Groups of them represent the categories of this field. Even though the number of combinations of the existing categories may be very large, the number of the categories themselves can be sufficiently manipulated. The task of the categorical module is to extract the set of the distinct categories and create the appearance distribution.

In this module, a new series is constructed containing all the words of the original series, one in each entry. The module iterates over the words and registers each word into a separate list. Each time the appearance frequency of the sequential appearance of this list items is calculated. In case a frequency is not equal to the previous value, it means that the previous sequence constituted a category. Otherwise, the procedure is continued to test whether the current sequence combination is a category. In this manner, it is feasible to find all the unique categories of the data, in a reasonable time, even if there is no delimiter to separate them.

3.3.2.2. Statistical Analysis of POI Attributes

Currently, six POI attributes are treated explicitly. Three of them check the ensemble against some predefined patterns, while the rest three transform the entries to numbers, removing the text, although keeping useful information about its position inside the entry. It is important to note, that even though the list of the predefined pattern types has only three representatives, it is straightforward to be extended with new members by defining extra patterns. The specific POI attributes that are handled separately, and the predefined patterns used for each one, are presented in the following and are also listed in Table 6.

- *Name*. The name is usually unique for each POI (although not necessarily). Clearly, an analysis of distinct values is not meaningful here. Moreover, it is possible that any character is allowed inside a name. With these remarks in mind, we conclude that one is usually interested to know the alphabet in which the name is written (e.g., Latin versus Greek characters), the number of words, and any symbols that it potentially contains. As a result, the patterns that are tested inside the Name module include language character sets and symbols. The information of the number of words is also extracted.
- *Address*. The scheme or writing style of addresses often varies. We try to capture various ways of writing an address by compiling a set of predefined regular expressions in the Address module. Special complication emerged for the postal code, since in some countries it may be a 3-digit number, which could also in principle represent the street number. However, taking into account its relative position inside an address string, we believe the module usually makes the correct distinction among them. Concluding, the module tests all the known address patterns, returning their frequency, as well the frequency of remaining cases, i.e., those not matching any of the current patterns. This latter case may represent wrong data or a malformed entry.
- *Phone number*. The writing of a phone number may also vary, but it is possible to identify some common patterns. Notably, it is expected to contain a series of numbers, and may also contain plus (+) or minus (-) signs, or parentheses. The patterns containing combinations of these characters are checked in the corresponding module PhoneNumber, as well as a pattern containing no number at all. This latter case could be considered almost safely as wrong. All remaining cases are returned as being unrecognized, leaving the user to decide about their legitimacy.
- *Price range*. Usually, the price range is given either in the form of two values, which define the range, or in the form of one value, defining the lower or upper limit of the range. Before, between and/or after the numbers, it is very likely to have text or symbols. Since we have to be language agnostic, we have to use some conventions. Thus, if there is text before one single value, we assume this value to denote the upper limit of a range. A common example is phrases such as "up to 20€", which are treated as denoting the range from 0 to 20. On the other hand, if text appears after a single value, this value is considered to represent a lower limit. Of course, these assumptions can be configured and adapted by the user, to fit different scenarios. This procedure is implemented in the PriceRange module, which finally constructs the appropriate bins and distributes the values in them.
- *Rating*. A rating usually is, or can be represented by, a single number. In case there is also text inside the entries, it is omitted. In the rating module, the ratings are distributed to bins, which are automatically created according to the found values.

- *Opening hours*. This field usually presents quite high diversity in its form. This makes it even harder to follow a language-independent approach in this case. However, since day names definition is not difficult in the languages we are interested in, this is not considered as a problem. Actually, what we focus on is not only the name of the days, but also their order inside the week, since for a given format, the missing, in between days of a range (e.g., in phrases such as "Mon. – Fri."), have to be filled in. Inside the Schedule module, the series is turned to lowercase, and all accents are removed. Subsequently, the dominant locale is determined in order to choose the set of day names to use. Once we are done with the locale, the series are split by the opening hours range, using the appropriate regular expression. In this way, a distribution with days and opening hours is constructed.

Table 6: List of predefined regular expressions used to identify patters in the values of specific POI attributes.

Category	Regular expression	Description
Name	(?:[ςερτυθιοπασδφγηξκλζχψωβνμΕΡΤΥΘΙΟΠΑΣΔΦΓΗΞΚΛΖΧΨΩΒΝΜέύίόάήϋΰϊΐΈΎΊΌΆΉΫΪ0-9]+(?:$|\ ))	Greek characters
	(?:[a-zA-Z0-9]+(?:$|\ ))	Latin characters
	(?:[^\(^\[^\]^\ )]+(?:$|\ ))	Other or mixed language
Address	(?:(?:(?:[^0-9]*[0-9]+[^0-9]{1,2}\s[^0-9]+)+)|(?:[^0-9]+))(?=\s|$)	Words regular expression (language independent), e.g. street name, city etc.
	(?:[0-9\-]+[ABCDabcdΑΒΓΔαβγδ]{0,1})	Street number
	(?:\d{5}([\-]\d{4})?)|(?:[0-9]{4})|(?:[0-9]{5}[\-]?[0-9]{3})|(?:[A-Za-z][0-9][A-Za-z] [0-9][A-Za-z][0-9])|(?:[0-9]{3,4})|(?:[1-9][0-9]{3}\s?[a-zA-Z]{2})|(?:[0-9]{5})|(?:\d{3}-\d{4})|(?:(L\s*(-|—|–))\s*?[\d]{4})|(?:[0-9]{2}\-[0-9]{3})|(?:((0[1-9]|5[0-2])|[1-4][0-9])[0-9]{3})|(?:\d{3}\s?\d{2})|(?:[A-Za-z]{1,2}[0-9Rr][0-9A-Za-z]? [0-9][ABD-HJLNP-UW-Zabd-hjlnp-uw-z]{2})	Postal code
	(?:\s+)	A regular expression for space(s)
	[\-\`\~\!\@\#\$\%\^\&*\(\)_\+\[\]\{\}\;\:\"\'\\\,\.\<\>\/\?€]	Symbols
Phone number	^[0-9]+$	Only numbers
	^(?:[0-9]*[\(][0-9]+[\)][0-9]*)+$	Numbers and parentheses
	^(?:[0-9]*[\+][0-9]+)+$	Numbers and + symbol
	^(?:[0-9]*[\-][0-9]+)+$	Numbers and - symbol
	^(?:[0-9]*[+][0-9]+[-][0-9]*)+$	Numbers, + and - symbol
	^[0-9\(\)\+]+$	Numbers, parentheses and + symbol

	^[0-9\(\)\-]+$	Numbers, parentheses and - symbol
	^[0-9\(\)\+\-]+$	Numbers, parentheses, + and - symbol
	^[^0-9]+$	Only non numerical characters

4. The TripleGeo Software

TripleGeo [TripleGeo] aims to bridge the gap between typical geographic representations from a variety of proprietary files, DBMSs, and georeferenced systems with the demands of geospatially-enabled RDF stores. Although it was initially developed based on open-source *geometry2rdf* library [geo2rdf], notable modifications and substantial enhancements have been made in order to meet interoperability needs in RDF stores. In fact, TripleGeo is designed as a *spatial ETL tool*, enabling users to:

- *Extract* spatial data from a source;
- *Transform* this data into a triple format and geometry vocabulary prescribed by the target RDF store;
- *Load* resulting triples into the target RDF store.

Therefore, TripleGeo always preserves data integrity and provides consistent, well-defined geospatial information to end-users. This tool can take as input not only de facto geographical files (e.g., ESRI shapefiles), but may also access spatial tables hosted in major DBMSs (e.g., Oracle Spatial or PostGIS databases). Further, it copes with most common spatial data types, like points, linestrings and polygons, but also more complex geometries (e.g., geometry collections). In addition, TripleGeo can make on-the-fly transformation of a given dataset into another projection system (e.g., data from a national reference system like GreekGrid87 into WGS84). Geometries can be exported in several serialized formats, most typically in WKT as prescribed by the GeoSPARQL standard [OGC12]. Users can control the transformation process through a configuration file that offers ability to parameterize execution and define user-specified settings (e.g., namespaces, serialization format, georeferencing, etc.). Last, but not least, latest versions of the software enable extraction of all thematic (i.e., non-spatial) attributes available in the input data, as well as definition of classification schemes for assigning categories in the input entities.

In the context of SLIPO, several customizations and extensions have been applied in the TripleGeo source code and functionality in order to support transformation of POI data (e.g., support for mappings to the SLIPO ontology [SLIPO-D2.1]). Although these add-ons are focused on POI data transformation, it should be stressed that TripleGeo remains a general-purpose ETL tool that can be used for efficient transformation of any geospatial features in vector format.

In this Section, we first discuss the original ver.1.1 of TripleGeo by briefly outlining its components and processing flow as these were available at the beginning of the SLIPO project (M1). Next, we present the software features available in ver.2.0 available at the end of the project (M36). As will become evident next, TripleGeo ver.2.0 provides advanced functionality and versatility in accessing a multitude of geospatial data formats, as well as orders of magnitude performance improvements in handling large POI datasets. Moreover, the software is being used regularly in transformation tasks against POI data extracted from OSM, effectively powering freely available download services to stakeholders, researchers, and practitioners that need large POI data assets across the entire planet.

4.1. TripleGeo ver.1.1

As discussed in Section 1.2, several ETL tools have been available for converting between geospatial formats, but only a few specifically addressing the emerging needs of geospatially-enabled RDF stores. In 2013, in the context of the GeoKnow project [GeoKnow], we began developing TripleGeo, an open-source ETL utility that can extract geospatial features from various sources and transform them into triples for subsequent loading into RDF stores. At that time, TripleGeo was the first tool that enabled conversion of geospatial features from several sources and formats into GeoSPARQL-compliant serializations according to the OGC GeoSPARQL standard [OGC12].

4.1.1. Legacy Implementation

geometry2rdf [geo2rdf] is an open-source library developed in Java by the Ontology Engineering Group (DIA) of the Facultad de Informática at Universidad Politécnica de Madrid. This tool allows the definition of geometrical information in RDF format. This methodology, also proposed in [VVS+10] for handling linked geodatasets, relies on Oracle's `SDO_UTIL` package for transforming geometrical data into GML format. For geometries stored in a MySQL database, information from the `GEOMETRY` column is extracted in a WKT representation. The next step is to convert the generated GML into RDF. For this purpose, the team has developed the *geometry2rdf* library, which exports a set of RDF triples with geometrical information. Geometries can be available in GML or WKT and are manipulated with GeoTools [GeoTools], not only in order to retrieve features, but also to perform coordinate transformation (if required). Finally, the Jena Semantic Web Framework [Jena] is used to generate the final geospatial RDF. The RDF generated is compliant with the WSG84 RDF vocabulary [GeoPos84] and the GML ontology [OGC07].

It is important to note that in the latest version of *geometry2rdf* (dating back to 2012 and available at [geo2rdf]) offers no support for geometries in the GeoSPARQL standard [OGC12], and no capability to export in formats other than RDF. In addition, there is no support for handling attribute values related to features (e.g., names, types). Concerning interaction with geospatial DBMS platforms, only support for extracting geometries from ESRI shapefiles and Oracle Spatial is available. Despite these important deficiencies, this source code had provided a stable base for developing our initial ver.1.0 of TripleGeo in 2013, mostly geared towards integration of a few geospatial data repositories and support for GeoSPARQL types.

4.1.2. Basic Features and Functionality

The aim of TripleGeo was to bridge the gap between typical geographic representations from a variety of proprietary files, DBMSs, and georeferenced systems with the demands of geospatially-enabled RDF stores.

Figure 6 illustrates the processing flow used for converting geospatial features into RDF triples with TripleGeo ver.1.1. Among its distinctive features, we point out that ver.1.1 could:

- Directly access de facto geographic formats (e.g., ESRI shapefiles, GML, KML) or DBMSs (IBM DB2, MySQL, Oracle Spatial, PostgreSQL/PostGIS).
- Recognize many geometric data types, i.e., not only points, but also (multi)linestrings and (multi)polygons.

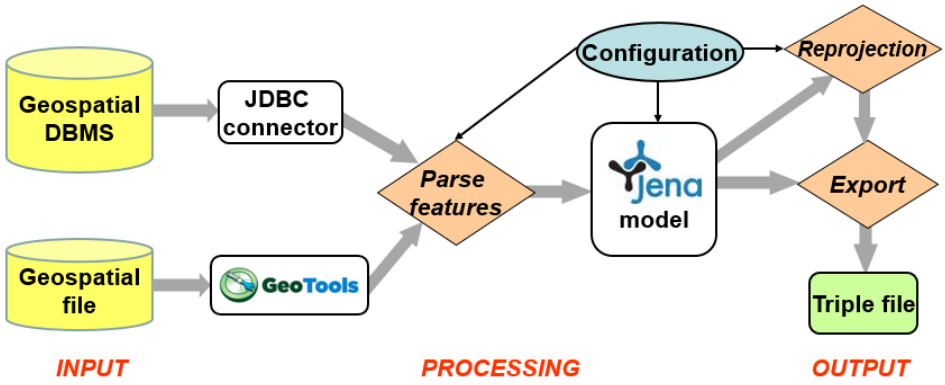


Figure 6: Processing flow diagram for transformation to RDF as applied with TripleGeo ver.1.1

- Extract a *limited* number of *thematic attributes* associated with each feature. In practice, TripleGeo ver.1.1 could handle up to four attributes per feature: its *geometry* (*mandatory*), a unique *identifier* (*mandatory*) for each entity and will be used for identifying the extracted resource, as well as two optional string values concerning its *name* and a *type* that characterizes this entity (e.g., "restaurant").
- Allow on-the-fly reprojection between any established Coordinate Reference Systems (CRS), e.g., transform geometries from GreekGrid87 to WGS84.
- Provide integrated transformation of INSPIRE-aligned spatial data and metadata into RDF using XSL stylesheets for selected INSPIRE data themes [INSPIRE]. This allowed geospatial data (standards-compliant or not) to be transformed to RDF and exposed through GeoSPARQL with limited effort.
- Export triples into various serializations (RDF/XML, NTRIPLES, TTL, etc.) and geometry vocabularies for swift loading into RDF stores.

From a user's perspective, this command-line utility was entirely automated and based on preconfigured settings. A configuration file specified user preferences concerning all stages of the conversion: how input source will be accessed, which data is involved, what geometric representation should be used, whether geometries must be reprojected into another CRS, as well as the output triple notation.

Since its inception, TripleGeo is open source software and it can be redistributed and/or modified under the terms of the GNU General Public License [GPL].

4.1.3. Performance

Regarding ver.1.1 of TripleGeo, we had conducted a performance study in the context of the GeoKnow project [GeoKnow-D2.2]. Indicatively, we had noticed that transformation took less than 3 minutes for an OpenStreetMap layer with around 590,000 point geometries (including cost of writing the RDF triples to disk files). In contrast, 2,600,000 linestring geometries required much more time, about 2.5 hours to conclude the transformation. Such delays should be mostly attributed to memory shortage, as the entire RDF model in ver.1.1 was retained in main memory and grew proportionally to the amount of statements generated per initial record, so it had to spill on disk in case of excessive load. This case signifies that triple extraction for large datasets with millions of features should better be performed in several smaller batches of the original data, as in many modern processing paradigms and exactly as we did in the context of SLIPO.

4.2. TripleGeo ver.2.0

Next, we present the additions to the functionality of TripleGeo, as inferred by the SLIPO requirements, and we discuss some specific issues and targeted actions that guided the implementation of the major release (ver.2.0) of the software delivered at the end of the project.

Thanks to its modular implementation, TripleGeo has been enhanced with more utilities without affecting its existing functionality. Throughout the project, we continued to further extend TripleGeo with several novel features, and most importantly, specific functionalities that can support the scalable transformation of large POI datasets. Next, we present the new features that have been implemented until ver.2.0, as well as the usability and performance improvements to satisfy our scalability goals. We also indicate the time schedule we had set for their implementation in two stages: an *intermediate* functional release available in M15 (see [SLIPO-D2.2]), as well as the final current version (M36) that covers all transformation requirements for SLIPO.

4.2.1. New Features

Next, we brefily present the new functionality provided by TripleGeo, indicating as appopriate the timing of the intermediate and final release.

- Ability to apply *mappings* and *vocabularies* and export both geometric and thematic attributes of the original dataset under a given OWL. This enables incorporating and deploying mappings between identified concepts and properties of the source data and the target ontology into the transformation mechanism. For compatibility, such mappings have been expressed in the widely used mapping language RML [RML], which is an extension of R2RML [R2RML], while also applying best practices in the way original features are mapped into RDF concepts and properties. Ability to accept other mapping schemes like D2RQ [D2RQ] would also be possible, but was not implemented in TripleGeo, since RML is considered to be more advantageous in flexibility and expressiveness for RDF. We also provide a *semi-automatic* functionality for guiding the user into creating new mappings for TripleGeo (e.g., in transforming datasets whose schema is not mapped to an existing POI ontology), which has been integrated in the SLIPO Workbench. [*Intermediate release*: M12, *Final release*: M36]
- Integration with *URI identifier creation*. The methodology developed in SLIPO for creating persistent, unique, vendor and technology independent POI URIs has been integrated in the transformation process performed by TripleGeo. Furthermore, this functionality has become available for any other source dataset (i.e., even non-POI data). This allows users to customize the creation of linked identifiers and to choose a specific naming strategy for URIs (e.g., as a combination of metadata of the original feature). [*Intermediate* release: M12, *Final release*: M36]
- Interaction with other geographic data sources (like GPX, GeoJSON, etc.), de facto POI formats (e.g., CSV) and DBMS platforms (e.g., MS SQL Server spatial, SpatialLite). [*Intermediate* release: M12, *Final release*: M24]

- Support for more complex geometric types (e.g., geometry collections) is equally important for acquiring the most detailed representation of spatial entities instead of point coordinates of centroids. [*Final release*: M12]

The usability options of TripleGeo in the SLIPO POI data integration lifecycle involve:

- *Specialization to POI transformation.* TripleGeo has been extended and optimized to effectively support POIs, with specific vocabularies and operations. TripleGeo is aligned with the POI ontology developed in SLIPO, thus allowing representation of more complex POI metadata and relations to facilitate the POI integration lifecycle. This way, TripleGeo is able to provide support for vocabularies and mappings specifically for POI data handled in SLIPO. Furthermore, we have also successfully employed TripleGeo to handle vocabularies and mappings to existing custom POI schemata (e.g., OSM, schema.org [Place]) for effective manipulation of large-scale open geodatasets. [*Intermediate release*: M12, *Final release*: M36]
- *Integration with the SLIPO Workbench* offers users a GUI and a RESTful API that facilitate customization of the transformation process. Instead of the the standard command-line interface, this GUI exposes the full functionality of TripleGeo for disk-based files, tables in a DBMS or web-accessible data, and offers a unified web interface to extract large POI datasets, convert them and produce their RDF representations. TripleGeo is thus incorporated into the SLIPO toolkit and POI data integration workflows in a coherent, user-friendly, and flexible manner under the entire SLIPO data lifecycle (i.e. transformation, interlinking, fusion, and enrichment of POIs). [*Intermediate release*: M12, *Final release*: M36]
- *Reverse transformation* from RDF into industry-standard geospatial formats. TripleGeo supports reverse transformation of RDF POI data (potentially interlinked or fused in later stages) into de facto POI formats (e.g., shapefiles, GPX, CSV). Of course, there exists an impedance mismatch in this direction, given that the POI ontology is semantically more expressive than the conventional POI schemata, thus POI attributes, relations and metadata may be richer than what can be supported by conventional formats. To address this issue, we defined and implemented the optimal reverse transformations for the various use cases handled in SLIPO. These reverse transformations allow incorporation of the maximum amount of semantic (linked, enriched, fused) POI information and metadata into the available properties of conventional POI formats. [*Intermediate release*: M12, *Final release*: M36]

4.2.2. Scalability

Scalability with increasing data volumes is most challenging. Hence, a parallelization framework in transforming input features and generating RDF triples was considered as most advantageous and has been implemented in SLIPO. Assuming that *n* processing nodes are available in a cluster infrastructure, we opted for solutions that employ data partitioning of the input into *n* disjoint batches, so that TripleGeo can be invoked in *n* separate instances, each one executed in a processing node in isolation from the rest. Such a partitioning may be based on several alternative schemes. For example, the original file may be split into *n* smaller batches with equal number of records in each one. Splitting can also be based on spatial criteria, e.g., employing a subdivision of the space into *n* disjoint regions (tessellation into cells), and creating a batch for each region with all entities contained therein. Of course, each original spatial entity with all its attributes

is included in only one such subset. Such partitioning has absolutely no impact on transformation to RDF, as each entity is transformed independently from the rest. By employing *n* concurrently running instances of TripleGeo to transform each dataset, and emitting triples into flexible RDF storage schemes (e.g., HDFS), we verified that scalability of transformation can be greatly improved. [*Final release*: M36]

As confirmed by extensive empirical tests reported in Section 5, thanks to its data partitioning capabilities and its advanced transformation methods, TripleGeo ver.2.0 has proven able to transform *almost 120 million POIs to RDF in 7.5 minutes* in a modest infrastructure. As a result, it achieves *orders of magnitude* performance gains compared to the edition of the software available at the beginning of the project.

4.2.3. Powering POI data services

OpenStreetMap [OSM] is one of the most valuable global sources of POI data for the industry, innovators and researchers. OSM is crowdsourced by a very large community of volunteers whose open geospatial data power thousands of services and products.

Thanks to TripleGeo, since October 2018 SLIPO provides a free download service available at:

http://download.slipo.eu/results/osm-to-rdf/

for *global POI data extracted from OpenStreetMap in RDF format*[11]. The data is organized in zip archives of RDF triples (N-TRIPLE serialization) per country or region and (as of November 2019) contain more than 18.5 million POIs worldwide. This archive gets updated once a month, so users can download the latest RDF POI data for their line of work.

The reason we provide this service is twofold. First, to demonstrate the scalability and performance of SLIPO at a world-scale. The technologies we have developed for POI data integration work efficiently from the local to the global level. Second, we have received multiple requests from POI value chain stakeholders to streamline and automate the data transformation process of OpenStreetMap POI data to RDF.

Regarding transformation of OpenStreetMap POI data in RDF using TripleGeo:

- A *custom filtering* is being applied against the original OpenStreetMap data in order to select features that represent Points of Interest (POIs).
- The *URIs* of all POI features are generated according to OpenStreetMap specifications in order to be resolvable. This ensures that an HTTP GET request to those URIs, with an appropriate mime type, will result in machine-readable representations of the OpenStreetMap resources they denote.
- All *original tags* are retained per POI and a data property is generated for each key under the OpenStreetMap wiki namespace, so as to be resolvable as well.
- Geometries of OpenStreetMap POIs are represented as *Well-Known Text* (WKT) literals according to the GeoSPARQL standard. Original OpenStreetMap nodes, ways or relations are suitably converted into WKT geometries, i.e., (multi)points, (multi)linestrings, (multi)polygons, geometry collections, etc. for direct cartographic rendering in GIS or geospatial data processing and querying.

[11] http://download.slipo.eu/results/osm-to-rdf/

- For each RDF file, an accompanying JSON file provides useful metadata regarding the transformation process. This JSON file lists all available OpenStreetMap tags concerning POIs in the original data, the spatial extent of the respective area (in WGS84 reference system), as well as the amount of input OpenStreetMap features and output RDF triples.

Due to multiple requests from POI value chain stakeholders, researchers, and practitioners, in November 2019 we launched another free download service for *global OSM POIs in CSV format* available at:

http://download.slipo.eu/results/osm-to-csv/

Extracting this wealth of POI data into a neutral tabular format like CSV (*comma separated values*) can greatly facilitate loading of the file contents in DBMSs and GIS for further analysis, since CSV is widely used for data exchange in the industry. The extracted POI data is organized in zip archives of CSV files per country or region as in the aforementioned RDF download service and the archive is also updated once a month.

We extract POIs from the original OpenStreetMap data using our *OSMWrangle* software[12]. This generic-purpose, open-source ETL tool can be used for extracting features from OpenStreetMap files and transforming them into records in a delimited CSV file. OSMWrangle draws much of its functionality from the source code of TripleGeo, our Extract-Transform-Load tool into RDF developed in the context of SLIPO.

OMSWrangle applies a custom filtering against the original OpenStreetMap data in order to select features that represent POIs. All POI features retain their original OpenStreetMap identifiers. Certain tagged values of the original OSM features are extracted in specific attributes as denoted in the header of each CSV file:

- Detailed geometry (WKT) in WGS84;
- Lon/lat coordinates for the centroid (in WGS84);
- Name (usually in local language);
- International name (usually in English or transliterated in Latin characters);
- Alternative name;
- POI classification (in 15 categories and 261 subcategories) based on OSM tags;
- Address (street, house number, postcode, city, country);
- Contact (phone number, fax number, website, operator);
- Opening hours;
- Full description (if listed in OSM);
- Link to a photo/image;
- Link to Wikipedia article;
- Date of last update,

while the remaining tagged values are listed under the 'OTHER_TAGS' column in the CSV file.

[12] https://github.com/SLIPO-EU/OSMWrangle

4.3. Features and Functionality

Table 7 outlines all major features and functionality available in the successive releases of TripleGeo since the beginning of the project. Note that ver.1.1 is practically the software as inherited from the GeoKnow project, whereas all subsequent releases have been prepared in the context of the SLIPO project. However, note that TripleGeo remains a *general-purpose ETL tool*, and its support for transformation into RDF is not limited to POI datasets only, but may be useful to many other application domains (transport networks, administrative areas, hydrography, cadastre, etc.).

General-purpose features. TripleGeo has evolved considerably during the project and currently supports a wide range of *general-purpose* functionality:

- Access to various *geospatial repositories* for extracting data:
    - Compared to the originally supported four *de facto geographical file formats* (ESRI shapefiles, GML, KML, XML in ver.1.1), more have been added in successive releases: CSV, GeoJSON (in ver.1.2), GPX and OSM XML (in ver.1.3), Json and OSM PBF (in ver. 1.5). In total, 10 de facto geographical file formats are currently supported by TripleGeo.
    - Four *geospatially-enabled DBMSs* were supported in ver.1.1 (namely, IBM DB2, MySQL, Oracle Spatial, PostgreSQL/PostGIS), but more have been added in successive releases: Microsoft SQL Server (in ver.1.2), SpatiaLite (in ver.1.3), as well as ESRI personal geodatabases and Microsoft Access format (in ver.1.4). In total, 8 geospatially-enabled DBMSs are currently supported by TripleGeo.
- Improved handling of *geometries*:
    - Since its inception, TripleGeo aimed to support *various geometry types*, and not only points. Hence, even from ver.1.1, it can process representations of all primitive types for 2-dimensional geometries (Point, MultiPoint, LineString, MultiLineString, Polygon, MultiPolygon), but even more complex geometries (Geometry Collection) are supported since ver.1.2. Therefore, TripleGeo supports all types of geometries as specified by the OGC standards [OGC10],[OGC11],[OGC12].
    - On-the-fly transformation of the coordinate reference systems (*CRS reprojection*) has been one of the distinctive features of TripleGeo, as it allows consistent representation and search over geospatial information that may be possibly collected from various sources using differing georeferences.
    - *Spatial filtering* allows users to specify their area of interest (e.g., a polygon or rectangle) over the input data and thus only transform a subset of the original entities into RDF. This functionality has been introduced in ver.1.8 of the software.

	Functionality	ver.1.1 (M1)	ver.1.2 (M7)	ver.1.3 (M11)	ver.1.4 (M15)	ver.1.5 (M19)	ver.1.6 (M22)	ver.1.7 (M26)	ver.1.8 (M29)	ver.1.9 (M32)	ver.2.0 (M35)
GENERAL-PURPOSE	Geographical file formats supported	4	6	8	8	10	10	10	10	10	10
	Geospatial DBMSs supported	4	5	6	8	8	8	8	8	8	8
	INSPIRE-aligned data/metadata	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES
	Process complex geometries	NO	YES	YES	YES	YES	YES	YES	YES	YES	YES
	CRS reprojection	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES
	Compliance to OGC GeoSPARQL	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES
	Export to various RDF serializations	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES
	Extraction of thematic attributes	MINIMAL	MINIMAL	ALL	ALL	ALL	ALL	ALL	ALL	ALL	ALL
	UUID generation & URI assignment	NO	NO	NO	AUTO	AUTO	USER-SPECS	USER-SPECS	USER-SPECS	USER-SPECS	USER-SPECS
	On-the-fly value check & clean	NO	NO	NO	NO	NO	YES	YES	YES	YES	YES
	Transliteration of string literals	NO	NO	NO	NO	NO	NO	YES	YES	YES	YES
	Spatial filtering	NO	NO	NO	NO	NO	NO	NO	YES	YES	YES
	Thematic filtering	NO	NO	NO	NO	NO	NO	NO	NO	YES	YES
	RDF graph sanity test	NO	NO	NO	NO	NO	NO	NO	NO	NO	YES
	Reverse transformation (export)	NO	NO	NO	YES	YES	YES	YES	YES	YES	YES
	Concurrent multi-threaded execution (+AUTO data partitioning)	NO	NO	YES	YES	YES	AUTO	AUTO	AUTO	AUTO	AUTO
	Distributed execution over Spark/GeoSpark	NO	NO	NO	NO	NO	NO	YES	YES	YES	YES
POI-SPECIFIC	POI Classification schemes	NO	NO	MINIMAL	YES	YES	YES	YES	YES	YES	YES
	Classification scheme validation	NO	NO	NO	NO	NO	NO	NO	NO	NO	YES
	Assignment of SLIPO categories	NO	NO	NO	NO	NO	NO	YES	YES	YES	YES
	RML mappings to SLIPO ontology	NO	NO	YES	YES	YES	YES	YES	YES	YES	YES
	Custom YAML mappings to the SLIPO ontology	NO	NO	NO	YES	YES	YES	MULTI FACET	MULTI FACET	MULTI FACET	MULTI FACET
	Updates to the SLIPO Registry	NO	NO	NO	YES	YES	YES	YES	YES	YES	YES
	Dynamically generated properties	NO	NO	NO	NO	YES	YES	YES	YES	YES	YES

Table 7: Functionality supported by successive versions of TripleGeo

- Improved handling of *thematic properties*:
    - Until ver.1.3, TripleGeo had minimal support for extraction of *thematic attributes*. In fact, at most three attributes could be extracted for a given spatial entity, and these were restricted to a unique identifier, its name, and its category, all represented as string literals with standard RDF properties (like `rdfs:label`, `rdfs:type`) and not allowing any customizations with an OWL ontology. Since ver.1.3, TripleGeo supports *user-defined mappings* from original attributes to properties according to a given ontology, thus allowing transformation of all available attributes per input entity. Although we have specific provision for mappings tailored to the SLIPO ontology (explained next), we stress that such mappings are not limited to POIs, but may be defined for other geospatial data as well (e.g., road networks, administrative areas) and then applied for their transformation to RDF.
    - *Thematic filtering* offers users the ability to specify conditions over attribute values and thus select only those qualifying entities for transformation. For example, the user may specify that only POIs listed under category 'TOURISM' should be transformed.
    - *Value checking and cleansing* offers some limited capabalities to sanitize original values from characters that may be illegal in RDF string literals. This functionality is applied on-the-fly when parsing the original values and takes care of line breaks, double quotes ("), and white space in string literals, backslash characters '\' in URLs, etc.
- Compliance to *standardization* initiatives:
    - Upon its initial release (ver.1.0, June 2013), TripleGeo was the first tool that allowed transformation of geospatial data into RDF geometries fully *compliant with the OGC GeoSPARQL standard* (2012). Of course, such compliance also entails support for geospatial representations according to OGC standards [OGC07], [OGC10], [OGC11], and it is certainly preserved in all subsequent releases of the software.
    - Support for *INSPIRE-aligned data/metadata*: Since ver.1.1, TripleGeo supports extraction of INSPIRE-aligned GML data for several thematic domains, as well as XML metadata and can transform them into RDF triples with geometries. Version 1.4 included only some minor improvements in the usability of this feature, which remains available in all subsequent releases.
- Exporting transformed data into a variety of *RDF serializations* enables their ingestion into RDF repositories. and of course, their utilization in Linked Data integration tasks in SLIPO. However, note that some transformation modes impose restrictions in the serialization format for performance reasons, as will be discussed in Section 4.4.3.4.
- Until ver.1.3, URIs of all entities transformed with TripleGeo were based on original identifiers in the input data; if not present, then there was no guarantee that different entities would not end up having the same URIs. Starting from ver.1.4, and complying with the requirements of SLIPO, TripleGeo *auto-generates UUIDs* per input entity and accordingly assigns a URI using namespaces prescribed by an underlying ontology (specified by the user). Since ver.1.6, TripleGeo also offers several extra options for assigning *user-specified URIs* using built-in functions (e.g., retaining original

identifiers, constructing new ones, or generating random ones). Thus, TripleGeo accomplishes to assign stable, well-formatted, globally unique, persistent, and manageable URIs to the resulting triples.

- Starting from ver.1.4, TripleGeo offers the ability for *reverse transformation* from RDF to de facto geographical file formats. This utility enables users to *export* a certain amount of the semantic information and metadata as attributes in conventional geodata formats (like ESRI shapefiles, CSV, or GeoJSON). Although this is a special requirement within the context of SLIPO for is specific to the POI integration lifecycle, so that semantic data (possibly interlinked, fused, and enriched) can be accessible and exploitable by existing GIS software and services, TripleGeo supports reverse transformation not only for POI datasets, but also for other types of geospatial linked data queryable with SPARQL according to a consistent ontology.
- *RDF graph Sanity Tester* is an auxiliary utility introduced in TripleGeo ver.2.0, which can be used to verify whether the transformed triples are valid and queryable. This optional, *post-processing step* loads the triples obtained from TripleGeo as data file(s) into a disk-based RDF graph and then runs a simple sanity test with a user-specified SELECT query in SPARQL. The RDF graph is considered consistent and usable once this query executes smoothly and returns an answer (e.g., the number of triples in this graph).
- *Scalability*. A large spatial dataset may be *partitioned* into a number of disjoint subsets under diverse schemes, each one ideally having an equivalent number of POIs. Thus, a separate transformation task may be employed for each partition to produce the RDF results, offering dramatic advantages in scalability. TripleGeo offers the following options for transforming large geospatial datasets:
    - *Concurrent execution* takes advantage of Java multi-threading and assigns a portion of the data for transformation to a separate thread. Introduced in ver.1.3, such execution required pre-partitioned files, i.e., the user had to split the data into several disjoint files beforehand and submit them to TripleGeo as a batch. Since ver. 1.6, TripleGeo is able to *automatically partition* certain types of geospatial files (CSV, shapefiles) and directly orchestrate their transformation according to available system resources.
    - *Distributed execution* over Spark/GeoSpark. For parallelized POI data transformation in cluster infrastructures, TripleGeo is able to use Apache Spark and GeoSpark as its underlying partitioning mechanism. This extension was introduced in since ver.1.7 and is available for several spatial formats. In case of GeoJSON and CSV files, it takes advantage of Spark DataFrame, whereas information from ESRI shapefiles is read using GeoSpark. Once data partitioning is done, each portion is assigned to a worker node and transformed with TripleGeo in isolation from the rest offering orders of magnitude performance gains compared to standalone execution.

POI-specific features. TripleGeo support the following features *specifically regarding POIs*:

- *POI classifications*
    - Data providers typically employ diverse *classification* or *tagging schemes* to categorize POIs and describe their type, as this is critical for applications based on POI data (e.g., searching

for restaurants). Since ver.1.4, TripleGeo accepts specification of (possibly hierarchical) classification schemes for POIs, produces RDF triples that fully describe this information along with especially assigned URIs, and introduces extra links between a POI and its respective original category under this scheme.

  - *Validation of a POI classification scheme* can be invoked before transformation in order to verify that the user-specified categorization or hierarchy can be recognized and used in the resulting triples. This auxiliary utility was introduced in TripleGeo ver.2.0 as an optional, *preprocessing step* in order to verify the consistence and suitability of a classification hierarchy where the spatial entities refer to.
  - To facilitate data integration, SLIPO employs an internally-used, vendor-agnostic, two-tier *POI classification scheme*[13] of 15 categories and 261 subcategories. During transformation, each POI is *automatically assigned* to a particular category/subcategory, based on the textual similarity its original classification to that used in SLIPO. Introduced with TripleGeo ver.1.7, this feature can greatly improve the quality of data integration, e.g., in detecting duplicate POIs from two data sources that originally employ diverse classification schemes, as verified with the various DINUCs [SLIPO-D5.2]. Note that the transformed POIs retain links to their original classification, as explained above.

- Regarding extraction and transformation of *thematic attributes* of POIs, TripleGeo currently supports *mappings to the SLIPO ontology* in two alternative formats:
  - *RML mappings* are expressed in the RDF Mapping Language [RML] and specify rules that dictate how to generate RDF triples from input data. This is available since ver.1.3 and can be applied when TripleGeo is executed in the *RML transformation mode* (Section 4.4.3.5.3).
  - *YAML mappings* are less expressive than the aforementioned RML mappings, but they are simpler to specify and also customized for the SLIPO ontology. Most importantly, thanks to their simplicity, they allow very fast transformation of scalable volumes of POI datasets. This king of mappings is available after ver.1.4 and is applicable when TripleGeo is executed either in *GRAPH* (Section 4.4.3.5.1) or *STREAM transformation mode* (Section 4.4.3.5.2). Besides, since ver.1.7, TripleGeo supports mapping for *multi-faceted attributes*, e.g., POI names in several languages can be mapped to distinct properties with the same mapping. Apart from multi-linguality, this extra feature can also support other *multi-valued* properties (e.g., phone numbers, connection points in charging stations for electric vehicles, etc.).
- Since ver.1.5, TripleGeo also supports *dynamically-generated properties*, i.e., values that do not exist in the original data, but can be derived from them (like area and perimeter of polygons, string values transliterated to Latin, etc.). Such properties can be declared in the mapping, invoking certain *built-in functions* that generate their values on-the-fly during transformation.

[13] https://github.com/SLIPO-EU/TripleGeo/blob/master/src/resources/categories.yml

4.3.1. Libraries and Frameworks

Since its inception, TripleGeo includes dependencies to various open-source tools and libraries, all of which are used "as is". The most significant of these libraries and frameworks are:

- ***Apache Jena***. This is a Java framework for building Semantic Web applications. Jena [Jena] provides a collection of tools and Java libraries for developing semantic web and linked-data apps, tools and servers. The Jena Framework includes:
    - an API for reading, processing and writing RDF data in XML, N-triples and Turtle formats;
    - an ontology API for handling OWL and RDFS ontologies;
    - a rule-based inference engine for reasoning with RDF and OWL data sources;
    - stores to allow large numbers of RDF triples to be efficiently stored on disk;
    - a query engine compliant with the latest SPARQL specification; and
    - servers that allow RDF data to be published to other applications using a variety of protocols, including SPARQL.
- ***GeoTools***. GeoTools [GeoTools] is an open source (LGPL) Java library, which provides standards compliant methods for geospatial data management comparable to those implemented in Geographical Information Systems (GIS). The GeoTools library implements Open Geospatial Consortium (OGC) specifications such as ISO 19107 Geometry, Simple Features, Clients for Web Feature Service (WFS) and Web Map Service (WMS), etc. GeoTools is widely used by a number of projects including Web Feature Servers, Web Map Servers, and GIS desktop applications. Among its core features are included:
    - Definition of interfaces for key spatial concepts and data structures, such as Integrated Geometry support provided by Java Topology Suite (JTS), attribute and spatial filters using OGC Filter Encoding specification, etc.
    - A clean data access API supporting feature access in many file formats (like CSV, DXF, edigeo, excel, GeoJSON, Shapefile, WFS) and spatial databases (including DB2, H2, MySQL, Oracle Spatial, PostGIS, SpatiaLite, MS-SQL Server), as well as coordinate reference system and transformation support, an extensive range of map projections, transaction support and locking between threads, etc.
    - A low-memory renderer, to compose and display maps with complex styling.
    - A schema-assisted parsing technology using XML Schema with bindings for many OGC standards including GML, KML, etc.
    - Plug-ins for reading additional raster formats from GDAL.
    - Extensions providing graph and networking support, validation, a web map server client, bindings for XML parsing and encoding, etc.
- ***GDAL/OGR***. The Geospatial Data Abstraction Library [GDAL] is a translator library for raster geospatial data formats, supported by the Open Source Geospatial Foundation (OSGeo). As a

library, it presents a single abstract data model to the calling application for all supported formats. It also comes with a variety of useful command-line utilities for data translation and processing. The related OGR Simple Features Library is a C++ open source library (which lives within the GDAL source tree) and provides similar capabilities (and command-line tools) for read (and sometimes write) access to a variety of *vector* file formats including ESRI Shapefiles, PostGIS, Oracle Spatial, Mapinfo mid/mif and TAB formats, etc.

- ***Java Topology Suite (JTS)***. The JTS Topology Suite [JTS] is an API of 2D spatial predicates and functions, conforming to the OGC Simple Features Specification for SQL. JTS is open source (under the LGPL license) and provides a complete implementation of fundamental 2D spatial algorithms written in Java.
- ***RDF Mapping Language (RML).*** This is a generic mapping language [RML] defined to express customized mapping rules from heterogeneous data structures and serializations to RDF. It is defined as a superset of R2RML, the W3C recommendation for a mapping language from databases [R2RML], and aims at extending its applicability to a broader variety of input sources (principally, CSV, XML, and JSON formats). RML provides a generic way to define mappings easily transferable to cover references to other data structures, combinable with case-specific extensions, and always backward compatible with R2RML. RML is open source and is released under the MIT license.
- ***Apache Spark.*** As a distributed, cluster-computing framework, Apache Spark [Spark] provides APIs in many programming languages, offers a stack of data management libraries (SQL, DataFrames, and Datasets), while also supporting machine learning, graph processing, and streaming applications. Spark abstracts the data in *Resilient Distributed Datasets* (RDD), an immutable distributed collection of data elements that can be stored in memory or disk across a cluster of machines.
- ***GeoSpark*** [GeoSpark] is an extension of Spark core and supports spatial data types, indices (R-tree, Quadtree, grid, etc.), and topological operations at scale. Enriched with a set of out-of-the-box *Spatial RDDs* (SRDDs), it can efficiently load, process, and analyze large-scale spatial data across machines [YZS19].

4.3.2. Licensing

TripleGeo is open source software and its current version (including the Java source code and sample data) are available from [TripleGeo]. It can be redistributed and/or modified under the terms of the GNU General Public License.

4.3.3. Documentation

A JavaDoc with full API documentation has been prepared in HTML format regarding the Java source code of TripleGeo and it is publicly available [TripleGeo]. This documentation covers all classes implemented for TripleGeo with specific details for all their methods and data structures.

4.4. Transformation to RDF

In this Section, we provide a detailed account of the processing flow for transforming POI data to RDF as applied by TripleGeo. We also present its core components, including the implemented functionality that supports attribute mappings and classification schemes, as well as the various transformation modes. Further, we outline the complementary mechanism for registering POIs in the SLIPO Identifiers Registry based on output prepared by TripleGeo. Finally, we discuss our implemented extensions of TripleGeo for achieving scalability over large geospatial datasets using either concurrent multi-threading or parallelization with multiple worker nodes to process disjoint subsets of the input data in parallel and thus dramatically decrease transformation cost.

4.4.1. Architecture

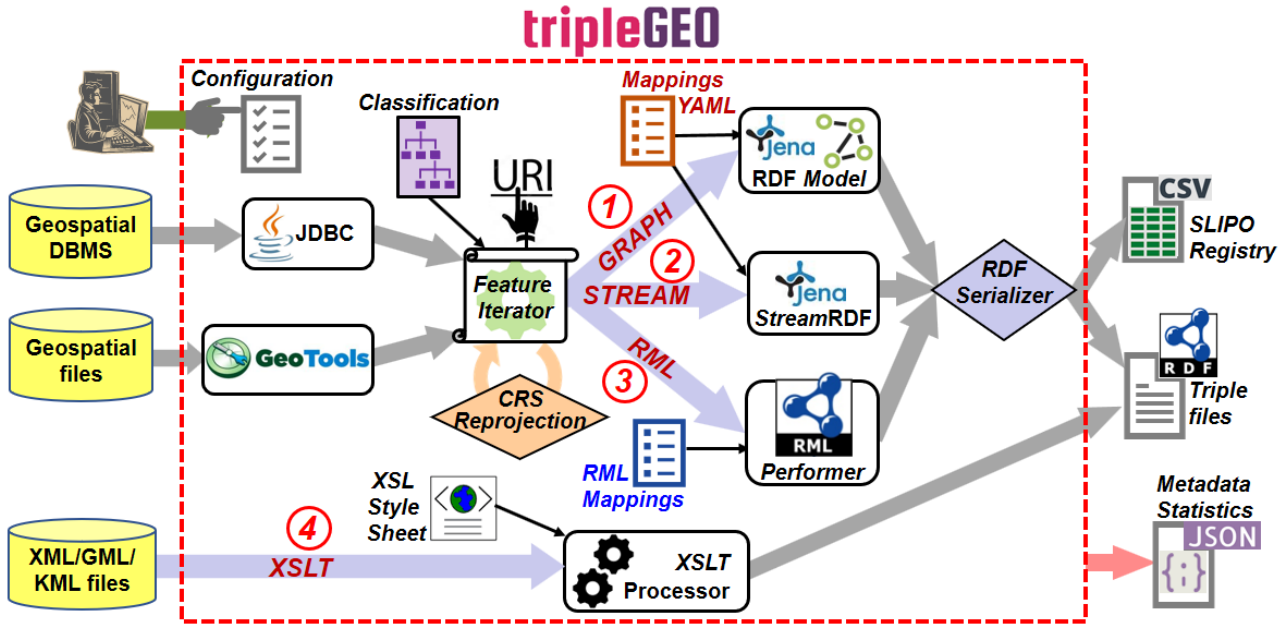


Figure 7: Processing flow for transformation to RDF with TripleGeo

TripleGeo has been implemented with several Java classes that perform specific tasks in a modular fashion. From a user's perspective, the utility works in a straightforward fashion according to some preconfigured settings. Figure 7 illustrates the flow diagram used for converting geospatial features into RDF triples. Next, we outline the basic components of the utility:

Input geospatial data may be obtained from geospatial files either *structured* (e.g., shapefiles, CSV) or *semi-structured* (in XML, GML, or KML), as well as directly from geospatially-enabled DBMSs. Currently, TripleGeo can access features stored in eight DBMS (Oracle Spatial, PostGIS, etc.).

Connectors to source data are required in order to access geometric features. In case of a DBMS, this is possible thanks to suitable JDBC drivers. With respect to shapefiles, the integrated GeoTools library provides all required functionality.

A *configuration file* lists all properties that control the various stages of transformation: how input source will be accessed, which data is involved, what geometric representation should be used, whether geometries must be transformed in another reference system, as well as the output format. All properties that may be specified in this file are explained in Section 9.2.1.

As detailed later in Section 4.4.3.4, transformation to RDF with TripleGeo can be carried out in four different *modes*:

1. *GRAPH*, which makes use of a disk-based Jena model to store all transformed triples;
2. *STREAM*, which applies in-memory conversion with prompt creation of triples per input feature;
3. *RML* for applying on each input feature custom attribute mappings specified by the user in the RDF Mapping language [RML]; or
4. *XSLT* specifically for parsing semi-structured data according to user-provided *XSL style sheets* and generating RDF triples.

Especially for *structured* data obtained either from files or retrieved from a DBMS, a *feature iterator* consumes each input record (i.e., all attributes concerning a POI) and converts its geometry into a suitable representation according to user specifications. Optionally, *reprojection* of geometries into another spatial reference system (CRS) is available. This coordinate transformation is carried out thanks to the integrated GeoTools library and according to user specifications for the source and target CRS.

Regarding *thematic* (i.e., non-spatial) attribute values (e.g., type, name, contact information) of an input feature, TripleGeo emits properly formatted literals as defined in user-specified *attribute mappings*. Depending on the transformation mode, these mappings can be prescribed in two alternative representations, either using RML or in a custom YAML format. In either case, such mappings should reflect the underlying ontology, i.e., the SLIPO POI ontology as introduced in [SLIPO-D2.1] and further extended as discussed in Section 2.

TripleGeo assigns a *URI* to each processed feature according to user specified scheme in the configuration. Namespaces for classes can be also defined in the configuration and used in the transformation according to the mappings. TripleGeo intentionally avoids creating blank nodes by inheriting the URI of the main feature to all its object properties with a suitable suffix.

In case that input data follows a (possibly multi-tier) *classification scheme* into categories, subcategories, etc. assigned to features, this can be also utilized in the transformation by assigning URIs to these categories. The classification scheme is also transformed into triples (according to the hierarchical structure defined in the SLIPO ontology) and each transformed entity (i.e., POI) is linked to its respective category.

Serialization of generated triples into export files is performed by the Jena API. This offers the possibility of writing the output into several triple formats, as detailed next in Section 4.4.2.2. In addition, an extra CSV file is issued, containing basic attributes per input feature (e.g., URI, name, category, geometry of a POI) specifically for registering it in the *SLIPO Registry* [SLIPO-D2.1].

Finally, *metadata statistics* compiled during the entire transformation process are written into another JSON file. This metadata concerns performance, input and output size, number of transformed values per attribute, and the spatial extent of the data.

4.4.2. Input and Output

4.4.2.1. Input

TripleGeo has been implemented for accessing geospatial data from a wide range of file or DBMS repositories. The final release ver.2.0, like all previous ones, has been successfully tested in both MS Windows and Linux environments and can provably deal with *structured or semi-structured geospatial data* stored in one of the following *vector file formats*:

- ESRI *shapefiles* [ESRIshp], one of the most popular geographical data formats supporting primitive geometric data types for (Multi)Points, (Multi)LineStrings, and (Multi)Polygons, along with their thematic attributes;
- *GeoJSON* files in JavaScript Object Notation with geometry features including (Multi)Point, (Multi)LineString, and (Multi)Polygon [GeoJSON];
- *JSON* files in JavaScript Object Notation [JSON] that include location coordinates in a specific CRS, e.g., lon/lat point coordinates in WGS84;
- *GPS Exchange Format* (GPX) that describe waypoints and tracks captured and stored by GPS devices and software [GPX];
- *CSV* containing geometries either as pairs of (x,y) coordinates (for points) or in WKT serialization (for any geometry type);
- OpenStreetMap dump files in either XML or PBF format [OSM];
- *GML* files in Geography Markup Language [GML];
- *KML* files in Keyhole Markup Language [KML];
- *INSPIRE-aligned data* (in GML) and *metadata* (in XML) [INSPIRE].

Since version 1.2, TripleGeo is capable of consuming data from *multiple* geographical files and transforming them into RDF. The only precondition is that all input files must be in the same format (e.g., shapefile), with records in each file having the same attribute schema. TripleGeo handles separately each file and writes its resulting triples into a distinct file according to the user-specified serialization.

TripleGeo is also able to retrieve data stored in the following *geospatially-aware DBMS* platforms:

- *Oracle* Spatial and Graph version 12c or later[Oracle];
- *PostgreSQL* version 9.4 or later with *PostGIS* 2.x [PostGIS];
- *MySQL* version 5.6 or later [mySQL];
- IBM *DB2* version 9.5 or later with Spatial Extender [IBM-DB2];
- SpatiaLite version 4.3.0a or later [SpatiaLite];
- Microsoft *SQL Server* version 2016 or later [MSSQLServer];
- ESRI *Personal Geodatabases* [ESRIGeoDB] in MS Access (.mdb) format; and
- Microsoft Access version 2016 or later [MSAccess].

Geometric data must reside in a *single* table or view in the DBMS. Currently, there is no support for combining information from several sources (e.g., by joining two or more tables). In case this latter is necessary, the user may create a view or a new table to store the combined data before invoking a transformation task with TripleGeo.

4.4.2.2. RDF Output

The main output of TripleGeo is the RDF triples returned after the transformation of the input data. In terms of output serializations, and according to the specifications of the Jena API [JenaDoc] that is used to export the model, the triples can be obtained in one of the following RDF formats:

- *RDF/XML*. This is the *default* output serialization that represents RDF as XML, according to the RDF specifications. Note that an error may occur with this RDF/XML serialization in case of blank nodes in the model. Specifically, a blank node gets a URI reference in this format, and thus it is no longer blank. So, the RDF/XML syntax is not capable of representing all RDF models; for example, it cannot represent a blank node which is the object of two statements.
- *RDF/XML-ABBREV*. This syntax (called *PrettyWriter* by Jena API) takes advantage of features of the RDF/XML abbreviated syntax to write a Jena model more compactly. It is also able to preserve blank nodes where possible. However, it is not suitable for writing very large models, as its performance might not be acceptable for voluminous datasets.
- *N-TRIPLES*. This syntax is most preferable to write large files, and it also preserves blank nodes. This is the *default serialization* available with either *STREAM* or *RML* modes, as this syntax does not use any internal state, it is the fastest to write in a streaming fashion, and data of any size can be output. It also maximises the interoperability with other systems and are useful for database dumps. However, it lacks some of the shortcuts provided by other RDF serialisations (e.g., N3, TTL).
- *N3*. Syntax *Notation3* (or *N3* as it is more commonly known) is a shorthand non-XML serialization of RDF models (not to be confused with N-TRIPLES syntax). N3 has been designed with human-readability in mind; hence, it is much more compact and readable than XML/RDF notation. N3 also offers features beyond a serialization for RDF models, such as support for RDF-based rules.
- *TURTLE* (also abbreviated as *TTL*). This syntax represents a *Terse RDF Triple Language* and provides a way to group three URIs to make a triple. It can also abbreviate such information, for example by factoring out common portions of URIs. Essentially, TURTLE is a simplified, RDF-only subset of N3.

In terms of standardization, the output triples are conformant to W3C standards, thanks to methods provided by the underlying Jena API for creating resources, properties and literals and the statements linking them. Therefore, all output triples are compatible with the most commonly used standards, including RDF, RDFS, OWL, and SPARQL.

With respect to geospatial features, triples can be exported according to the GeoSPARQL standard [OGC12]. In addition, TripleGeo offers the ability to export point geometries into legacy vocabularies for Virtuoso [VirtGeoRDF] and WGS84 RDF Geoposition vocabulary [GeoPos84], but note that this syntax is not compliant to GeoSPARQL.

4.4.2.3. Output to the SLIPO Identifiers Registry

In addition to RDF output, TripleGeo can optionally provide a CSV file having one record for each input feature (i.e., POI) with its basic attributes in order to be registered in the SLIPO Identifiers Registry as discussed in Section 4.4.3.6. Each such record consists of the following attributes:

- *URI* assigned to the POI;
- *Name* of the POI;
- *Category* according to a classification scheme accompanying the original data;
- *Data source* provider of this POI (e.g., OpenStreetMap);
- *Unique identifier* of this POI at the original data;
- *Location* specified as a pair of longitude/latitude coordinates in WGS84.

4.4.2.4. Metadata Statistics

Upon termination of a transformation process, TripleGeo provides *metadata* regarding its execution in a JSON file listed in three main categories:

- *Attribute Statistics*: For each attribute in the original input dataset, a count of *NOT NULL values* on this attribute is given. This reflects the amount of such values that have been actually given to transformation, but note that the number of resulting triples may be inflated depending on the specified mappings.
- *Spatial Extent*: A Minimum Bounding Rectangle (*MBR*) as computed by the spatial extent that covers all transformed geometries. Note that this rectangle is always reported in WGS84 coordinates, irrespective of the spatial reference system (CRS) of input and output datasets.
- *Execution metadata* include the following items:
    - Count of output triples;
    - Count of input records (for structured data) or features (for semi-structured data);
    - Count of input records excluded either due to filtering conditions specified by the user or rejected because of errors (e.g., invalid geometries);
    - End-to-end execution time (in milliseconds) that includes the cost of accessing and fetching the input data, its transformation cost, as well as the cost of writing output to file(s);
    - Path to the output file containing the resulting triples;
    - The RDF serialization of the output triples; and
    - The transformation mode employed in the execution.

4.4.3. Core Modules

In this Section, we provide details concerning the design and implementation of core modules of TripleGeo regarding transformation of geospatial features to RDF.

4.4.3.1. Assignment of URIs

In [SLIPO-D2.1], we discussed extensively the two main approaches regarding the issue of *POI identifiers*, namely either assigning (a) *global shared identifiers* or (b) *local interlinked identifiers*. Since the goal of SLIPO is to address the POI data integration challenges by means of Linked Data technologies, we adopt the second approach for dealing with POI identifiers. Hence, in SLIPO we opt to use *HTTP URIs* as POI identifiers, so that data owners have enough flexibility and full control over creating and managing their own POI identifiers, while still adhering to a uniform format. More specifically, TripleGeo integrates support for constructing such URI identifiers according to recommended best practices. The URI pattern adopted is:

http://{domain}/{type}/{concept}/{reference}[/{attribute}]

and consists of the following components:

- *domain*: As base for the URI in SLIPO, we prescribe `slipo.eu` as the domain.
- *type*: For identifying POIs, value `id` is used to identify the type of features.
- *concept*: In SLIPO, this may take several possible values (`poi`, `poiset`, `poisource`, `classification`, `term`) depending on the respective concept in the ontology where the identified resource adheres to.
- *reference*: This part is typically an alphanumeric string that uniquely distinguishes the given POI from the rest. TripleGeo offers several options for defining such reference using *built-in functions* on URIs that can be invoked in the mappings:
    - This reference can be the indentifier originally used by the data source by specifying the function call *keepOriginalID(<attr-id>)* in the YAML mappings, where *<attr-id>* is the column name for identifiers in the input dataset.
    - A 128-bit long number representing a *Universally Unique Identifier* (*UUID*) can be generated on-the-fly during transformation using function call *getUUID(<data-source>, <attr-id>)*. This function takes as arguments (i) the name of the data source and (ii) the column name for identifiers in the input dataset and returns a *GUID* (*Globally Unique Identifier*) after hashing over the input values. Since creation of such UUIDs is deterministic, transformation of updated POI datasets in the future will assign the same URIs to those POIs that retain their identifiers in the original dataset.
    - Function call *getRandomUUID* assigns a randomly generated *UUID* that represents a 128-bit long value to be used in the URI of a transformed feature. This is a secure random UUID with minimal chance of collisions with existing ones, also qualifying as *GUID* as above.
    - Function call *getNextSerial* returns a long integer value (i.e., a serial number) to be used as reference for POIs not having an identifier in the original dataset (e.g., in GPX files). This option is used by befault in case that POI identifiers in the input data are missing. However, this serial number is not used directly in the URI, but only as a seed for generating a 128-bit *UUID* that will be appended to the URI, as in the previous cases.

- *attribute*: This is an *optional* component, used especially for generating *named URIs* regarding RDF properties of a POI feature, e.g., its name, address, category, etc. This suffix to the URI assigned to a POI is deliberately applied in order to avoid blank nodes in the resulting triples.

4.4.3.2. Attribute Mappings

As already mentioned, TripleGeo offers two different options for specifying mappings from input attributes to RDF, i.e., either *RML mappings* (applicable in RML transformation mode) or custom *YAML mappings* (applicable in GRAPH/STREAM transformation modes). RML mappings are more expressive and are already utilized by other transformation tools (e.g., GeoTriples [KSV+18]). However, YAML mappings are much simpler to specify for industrial users not acquainted with RDF semantics, whereas it can be also customized for the SLIPO ontology. Thanks to their simplicity, YAML mappings allow very fast transformation of large volumes of POI datasets as demonstrated in our evaluation in Section 5. Last, but not least, offering two alternative fashions in specifying attribute mappings enhances the versatility of TripleGeo, as it is able to reuse existing RML mappings, but can also *automatically recommend YAML mappings* for transforming a new dataset as discussed next.

Note that GRAPH/STREAM modes can also manage to transform input data even if *no mapping* has been specified. In this case, the name of an attribute in the input schema is turned into an RDF property (i.e., a predicate) in the ontology namespace defined by the user in the configuration. Certainly, such a *flat mapping* does not abide by a concrete ontology, but nonetheless it provides a valid RDF output that contains links between a central resource (the URI assigned to a POI) with all its known properties.

Such attribute mappings have been principally designed and successfully applied against POI datasets in the context of SLIPO, hence aiming to comply with the underlying POI ontology. However, it should be stressed that their applicability is general. In particular, transformation is not tightly coupled with the SLIPO ontology, but it can be applied against *other POI ontologies* such as schema.org/Place. Although this collaborative model differs a lot from our own ontology, it is rather straightforward to specify suitable mappings and perform transformations[14]. Further, TripleGeo can also be employed to transform spatial entities *other than POIs* (e.g., road network data) to RDF. This feature greatly enhances the ETL functionality of TripleGeo and its usefulness beyond the context of the SLIPO project.

4.4.3.2.1. RML Mappings

In contrast to the R2RML [R2RML] mapping language that can only define mappings of data from relational databases to RDF, the *RDF Mapping Language* (RML) [RML] offers extensions that support mappings also from structured or semi-structured file formats, such as CSV, XML, and JSON. RML is more expressive than R2RML and allows specification of mappings covering joins between input sources (e.g., master/detail tables). In contrast to R2RML, RML provides a vocabulary for defining a generic data source and the iterator pattern that specifies how the source data will be accessed. Such mapping definitions may be reusable across different sources (perhaps after small editing), as well as across different implementations for different source formats. In addition, RML allows cross-references among data from different input sources

[14] Available at https://github.com/SLIPO-EU/TripleGeo/blob/master/test/conf/OSM_Fuel_Berlin_schema.org_mappings_CSV.yml and https://github.com/SLIPO-EU/TripleGeo/blob/master/test/conf/OSM_Fuel_Berlin_schema.org_CSV_options.conf

to be defined already on mapping level, by uniquely defining the pattern that generates a resource and refer to this definition any other time this resource is mapped.

More specifically, an RML mapping specifies rules concerning how input data will be represented in RDF as triples of (*Subject*, *Predicate*, *Object*) statements. This is achieved with a *TriplesMap* construct, which consists of three main parts:

1. The *Logical Source* covers many kinds of input data sources and specifies all necessary properties for accessing a data source and iterating over it.
2. A *Subject Map* defines how URIs will be generated for the specified resource. Such URIs will serve as subjects of all triples transformed according to this TriplesMap.
3. One or more *Predicate-Object Maps* can be defined for a given Subject Map. Each one consists of *Predicate Maps* that control how the respective predicate of triples is composed, as well as *Object Maps* or *Referencing Object Maps* that guide generation of objects in those triples

Note that all Subject, Predicate, or Object Maps are actually *Term Maps*, which control generation of RDF nodes, i.e., URIs or blank nodes or literals. A Term Map can be a *constant-value term map* that always generates the same RDF node, or *a reference-valued term map* that refers to a specific value (e.g., an attribute value) in the original data, or *a template-valued term map* which serves as a blueprint for transforming referenced columns from the source data. Furthermore, RML supports cross-references between Triples Maps, when the subject of a Triples Map is the same as the object generated by a Predicate-Object Map.

An indicative RML mapping for POIs in TURTLE format is given in the Annex (Section 7.3.1). A small excerpt of this RML mapping (`#POIMapping`) is depicted in Figure 8. Since TripleGeo utilizes its own configuration for accessing input data sources, we have simplified the specification of logical sources in RML mappings. So, the actual source (e.g., path to a CSV file) is omitted, whereas of RML's predefined Reference Formulations only the `ql:CSV` is declared. This latter specification actually stands as a proxy for column-oriented data formats, and does not mean that only CSV input is supported. Besides, *no iterator* needs to be specified, as TripleGeo provides one input feature (i.e., a POI) at a time and applies any transformations specified in the RML mapping against its attribute values. In this example, a *Subject Map* is defined in order to generate the URI of each POI according to a *template-valued term map* that concatenates the SLIPO namespace with an attribute value (`UUID`) available in the input feature. Furthermore, extra triples will be generated, which associate this URI with specific nodes, i.e., that it is actually classified as a POI and also as a GeoSPARQL feature. Furthermore, a *Predicate-Object Map* is defined for the name of the respective POI, that will be generated via a separate *ParentTriples Map* specification (`#POIName`) and linked with the URI of that POI using a specific predicate (`slipo:name`). This is an example of *nested mappings*, where one mapping (`#POIMapping`) triggers another (`#POIName`). Another Predicate-Object Map for the timestamp of a POI is defined with a *reference-valued term map* that refers to a specific attribute (`TIMESTAMP`) available in the input record.

```
<#POIMapping>
    rml:logicalSource [
        rml:source "" ;
        rml:referenceFormulation ql:CSV
    ];
    rr:subjectMap [
        rr:template "http://slipo.eu/id/poi/{UUID}";
        rr:class slipo:POI;
        rr:class geo:Feature
    ];
    rr:predicateObjectMap [
        rr:predicate slipo:name;
        rr:objectMap [
            rr:parentTriplesMap <#POIName>
            ]
    ];
    rr:predicateObjectMap [
        rr:predicate slipo:lastUpdated;
        rr:objectMap [
          rml:reference "TIMESTAMP";
          rr:datatype xsd:dateTime
          ]
    ];
```

Figure 8: Fragment of RML mapping for POIs as used in TripleGeo

4.4.3.2.2. YAML Mappings

In TripleGeo, the aforementioned RML mappings are tightly coupled with the RML transformation mode and cannot be applied in any other mode. So, in order to support transformation of an arbitrary number of thematic attributes, we have introduced a *custom mapping* in YAML format to be used in GRAPH and STREAM modes. Our objective was to offer the minimal functionality required for mapping thematic attributes in an input dataset to the properties of SLIPO ontology [SLIPO-D2.1]. Of course, such a mapping may lack the expressiveness and generality of RML mappings, but it is simple and can be easily utilized by users in conversion to RDF under ontologies with moderate complexity like the SLIPO ontology for POIs. Furthermore, as it was specifically aimed to handle POI entities and has been applied against datasets of varied formats and schemata [SLIPO-D5.2], it succeeds to transform them to RDF very fast without missing any input information as demonstrated in Section 5.

An indicative YAML mapping for POIs is given in the Annex (Section 7.3.2). A small excerpt of this mapping is depicted in Figure 9. In practice, for each thematic attribute in an input feature that will be transformed into a resource, this YAML specifies up to eight properties:

- *entity*: This string value is used in creating a new RDF node for the given attribute value. This node inherits the URI of the POI it belongs to, suitably suffixed with this string value. As already mentioned in Section 4.4.3.1, this is a deliberate decision, as it avoids creation of *blank nodes* in the resulting triples, and also provides meaningful (and human readable) URIs for child nodes of POIs.

```yaml
URI:
  entity: uri
  generateWith: getUUID(DATA_SOURCE,osm_id)
osm_id:
  partOf: sourceInfo
  entity: source
  predicate: slipo:poiRef
DATA_SOURCE:
  partOf: sourceInfo
  entity: source
  predicate: slipo:sourceRef
  generateWith: getDataSource
ASSIGNED_CATEGORY:
  entity: assignedCategory
  predicate: slipo:assignedCategory
  generateWith: getEmbeddedCategory
name:
  instanceOf: name
  entity: name
  predicate: slipo:name
  type: official
  language: en
TRANSLIT:
  instanceOf: name
  entity: translit_name
  predicate: slipo:name
  type: transliterated
  language: en
  generateWith: getTransliteration(name)
timestamp:
  entity: lastUpdated
  predicate: slipo:lastUpdated
  datatype: timestamp
GEOHASH:
  entity: geohash
  predicate: slipo:geoHash
  datatype: string
  generateWith: geometry.getGeoHash(10)
```

Figure 9: Fragment of YAML mapping for POIs as used in TripleGeo

- *instanceOf*: This is used to specify RDF properties that are instances of classes. For instance, a name value should be transformed into an instance of the *Name* class in the SLIPO ontology, a phone value to an instance of the *Contact* class in the SLIPO ontology, etc.
- *partOf*: Specifies the name of *parent* (composite) RDF property that this resource is part of. For example, in Figure 9, the `osm_id` identifier will be transformed into a node that is part of the `sourceInfo` property of a POI according to the SLIPO ontology. Similarly, a street name is part of the address property, etc.
- *predicate*: This item specifies the namespace and the RDF property that will be used to link the URI of the POI feature with the RDF node that will be generated for this attribute.
- *generatedWith*: This optional item indicates that the object value will be dynamically created based on a custom, built-in function. TripleGeo offers the following types of ***built-in functions*** for use in YAML mappings:
    - *getDataSource*: Examines the TripleGeo configuration settings and provides the name of the data source as specified by the user. This value is used in the triple(s) that denote the source of each POI according to the underlying ontology.
    - *getEmbeddedCategory*: Classifies the POI into one of 15 broad categories that are used internally in SLIPO as discussed in Section 4.4.3.3. This default categorization is derived from

the textual similarity of the original category to a dictionary of tags that characterize each of the 15 categories embedded in SLIPO and utilized during data integration tasks.

- *getResourceType*(*<attr>*): As explained next, this function call generates a characterization assigned to a resource (representing a POI) based on a given attribute in the input data.
- *getTransliteration*(*<attr>*): This call it takes as argument an attribute of the input data that contains string literals written in a non-Latin alphabet (e.g., Greek, Arabic, Chinese, Cyrillic) and provides its transliteration into phonetically equivalent Latin characters. TripleGeo employs a widely used framework[15] for transliteration to UTF8. Note that *transliteration*, unlike translation, operates on characters without reference to the meanings of words and sentences. Thus, it does not provide a translation from any language to English. As detailed in [SLIPO-D5.2], transliteration was necessary in order to integrate data from diverse sources where names and other string literals were written in different alphabets.
- *getLanguage*: It extracts a language tag from an attribute name, assuming that it concerns a suffix after the ith character in the attribute name (e.g., a column name like *NAME_DE*). This tag *(@de* in this example) will be attached to all values obtained for this attribute in the resulting triples.
- *concatenate*: This function call concatenates a pair or an array of string values into a unified string literal.
- *geometry.getLongitude*: It provides the longitude coordinate (in WGS84) of the given geometry, in order to create a triple with this object value.
- *geometry.getLatitude*: It provides the latitude coordinate (in WGS84) of the given geometry, in order to create a triple with this object value.
- *geometry.getArea*: It calculates the area (always in square meters) of the given geometry in order to create a triple with this object value. This function is only applicable on surface geometries (i.e., polygons and multipolygons).
- *geometry.getLength*: It calculates the length (always in meters) of the given geometry in order to create a triple with this object value. This function is only applicable on surface geometries (i.e., *perimeter* of polygons and multipolygons) or linear geometries (i.e., *length* of linestrings and multilinestrings).
- *geometry.getGeoHash*(*<length>*): Encodes a geographic location into a short string of letters and digits[16] of size *<length>* up to 12 characters (default value is 8), which can assist in identifying nearby locations since their have the same prefix in their geohashed values.

- *type*: This is a characterization assigned as an extra (specialization) RDF property to a given node. For example, this may be used to specify that the name of a POI is "*official*", "*brand name*", "*international*", etc., or that a phone number is "*direct*", "*mobile*", etc. Although this type can be specified in the YML mapping either with a *constant* value (i.e., a string literal as above), it can also

[15] https://unicode-org.github.io/icu-docs/apidoc/released/icu4j/com/ibm/icu/text/Transliterator.html.

[16] http://geohash.org/.

be based on an *attribute* in the input data. In the latter case, its value (e.g., type of phone numbers) can be extracted using the custom built-in function *getResourceType*. For instance, if there is a attribute *TEL_TYPE* that contains the type (call center, sales, etc.) of phone numbers, a call to the built-in function *generateWith.getResourceType(TEL_TYPE)* will extract the respective value per POI and can be used to create a triple. Finally, if the special value *NONE* is specified in the YML mapping for resource type, then no such triple will be issued as its generation is explicitly suppressed by the user.

- *datatype*: An XSD datatype that prescribes the correct interpretation of a string literal in the object of an RDF statement.
- *language*: Specifies the language tag (e.g., "*en*") to be used in string literals for this resource. This user-specified string is validated according to the ISO 693-1 language codes[17].

In transformation under GRAPH or STREAM modes, TripleGeo identifies the YAML mapping defined for a given attribute and accordingly creates RDF triples. The final version of the software offers the following capabilities concerning definition of YAML mappings:

- Ability to *concatenate* attribute values in the original data in order to generate a single literal for an RDF property. For instance, the street name and house number may be listed in separate columns (respectively, STREET and NUMBER), but may need be concatenated into a single value for comparison with those in another dataset. In this case, the a specification like STREET+NUMBER creates a new concatenated value on-the-fly, which is then transformed into triples according to its specified mapping.
- *Wildcard character* **%** is used in YAML mappings in order to specify *multi-faceted attributes*, e.g., a name in various languages available in the same POI dataset (e.g., OpenStreetMap). Specifically for *multi-lingual* attribute values, e.g., specified with attributes like *name_en*, *name_fr*, *name_de*, etc., placeholder **%LANG** should be used for language specifications in YAML mapping (e.g., *name_%LANG* in this example) in order to be recognized in the internal mapping representation. The aforementioned built-in function *getLanguage* can then be used to dynamically infer the language tag from the last part of the attribute name. URIs in these triples will also include a language suffix in order to be distinguishable from the rest.
- *Wildcard character* ***** in the mapping signifies a *multi-valued* property for the same entity, e.g., the number of connections *Connections.*.Quantity* in charging stations for electric vehicles as in [SLIPO-D5.2]. Each original value will be mapped to the same RDF property, but it will generate a separate triple. The original key (attribute name) will serve as suffix in the URI specification for the generated triple.
- *Wildcard character* **_** can be used to denote any attribute in the input dataset not specifically defined in the YAML mapping. Using this convention, any attribute *not specifically mapped to the POI ontology* will be typically transformed "as-is" by issuing a pair of triples, one describing its *key* (i.e., the attribute name) and another one for its *value*. This is supported by the SLIPO ontology using the *slipo:otherAttr* predicate.

[17] https://www.iso.org/iso-639-language-codes.html.

- *Special characters* (like :, -, etc.) appearing in attribute names (allowed in several input formats like CSV or JSON) can be specified in YAML mappings enclosed in *double quotes* to avoid any ambiguity, e.g., "*addr:housenumber*".

4.4.3.2.3. Automatically generated mappings

Once a new POI dataset is acquired and needs to be integrated into the SLIPO ecosystem, it requires transformation. However, specifying attribute mappings for diverse POI datasets with a wide variety in attribute names, data types and contents can be a difficult task for non-experienced users, as it requires some basic understanding of the uncerlying POI ontology. To assist and guide users in creating *attribute mappings* for new datasets, we have extended TripleGeo with a auxiliary *mapping utility*[18] that accepts an input POI file, it examines its schema and contents and automatically recommends a suitable YAML mapping for its transformation.

More specifically, we have built a *Machine Learning* utility that suggests new mappings from a corpus of previously specified ones, available as YAML files from the various use cases of POI data handled in SLIPO as discussed in [SLIPO-D5.2]. This corpus is used as a training set to train *classifiers* that assign the attributes of new datasets to the predicates of our ontology. Specifically, for each predicate of our ontology the utility trains one classifier, which estimates the *probability* that an input attribute corresponds this specific predicate. For each predicate, the utility also stores the names of the attributes in the training set that have been mapped to this particular predicate. The features used in this classification are: i) the set of bigram string similarities between the name of the new attribute and the names of each attribute of the training set that has been mapped to the predicate and ii) several features that describe the contents of the attribute; specifically, if it contains integers, real numbers, characters, specific strings (e.g., *'@', 'www', 'http'*), and repeating entries. The training set is used to construct a *logistic regression classifier* for each predicate. Logistic regression was selected because it can be effectively trained from a small training set of mappings and it can capture the monotonic relations that we expect to exist between our features and the class membership variable. When used on a new dataset, for each attribute of the dataset and each predicate of our ontology, the utility calculates an estimate for the probability that the given attribute corresponds to the predicate. These probability estimates can be used to automatically select the predicate with the highest probability or reported as a suggestion to the user.

As discussed in [SLIPO-D1.4], this functionality can be invoked through the "*Auto*" button when configuring a transformation task in the SLIPO Workbench. The user is presented with automatically generated YAML mappings per attribute for the POI dataset to be transformed into RDF. She can then verify or modify these recommended mappings through a graphical interface before finalizing them and applying them in the transformation task. Overall, this utility relieves users from the burden of defining new mappings from scratch, while it also reuses knowledge from previous mappings. Most importantly, complexity inherent in POI data modelling and the ML classification process is hidden and all functionality is made available through a self-explanatory GUI, thus effectively contributing to a more positive user experience in data integration tasks.

[18] Source code available in https://github.com/SLIPO-EU/triplegeo-ml-mappings

4.4.3.3. Classification Schemes

Typically, POIs available by most data providers are classified to *categories* (e.g., restaurants, bars, theatres, etc.). This classification possibly follows a *hierarchical* scheme in *multiple tiers*, where each major category (e.g., EAT/DRINK) may be specialized into several subcategories (e.g., restaurant, fast food, pizza, etc.), etc.

TripleGeo supports three alternative ways to define a hierarchical classification scheme for POIs:

i. A user-prepared *CSV* file to be employed in assigning categories, subcategories, etc. to each feature in the input dataset. Attributes in this CSV file must be delimited with comma (',') and string values must be enclosed with double-quotes (e.g., `"RESTAURANT"`). Every item in the classification scheme must be defined with a unique identifier and name (e.g., `39,"RESTAURANT"`). Each line of the CSV file specifies a full path from a top-tier category to a bottom-level subcategory. E.g.: `"D","EAT/DRINK",39,"RESTAURANT".` Figure 10 (*left*) depicts a fragment of a CSV-formatted classification scheme customized for OSM data, which is available in the Annex (Section 7.4.1).

ii. A user-prepared *YAML* file with *indentations* used to denote breakdown of a given category into subcategories, i.e., two blank characters in the beginning of a line at each extra level in the hierarchy. The identifier of each category (at any level) is specified after its name and it is preceded with a '#' character. Figure 10 (*right*) depicts a fragment of such a YAML-formatted classification scheme customized for OSM data, which exactly corresponds to the CSV fragment also listed in the same Figure and available in the Annex in full (Section 7.4.2).

iii. Especially for OpenStreetMap input data, a *YAML* file can be used to specify classification of features into categories according to their respective OSM tags. This classification may be employed when converting OSM XML data in either GRAPH or STREAM transformation modes. In TripleGeo, we currently make use of the OSM tags originally proposed in [OsmPoisPbf] , but rearranged and enhanced into a comprehensive *classification hierarchy*[19] during the project in order to classify OSM elements into custom categories.

Our experience from transforming a wide range of POI data [SLIPO-D5.2] indicates individual POIs may arbitrarily refer to any level of the classification hierarchy. For instance, a POI may be originally classified as a restaurant (without any further specification), while another one in the same dataset may be specialized as "fish restaurant" or "pizza" or "French cuisine", depending on the depth of the classification hierarchy. Moreover, some classes may have more levels than others, e.g., "EAT/DRINK" may be further specialized in up to two levels of detail, whereas "AUTOMOTIVE" may have only one level of specialization. During transformation, TripleGeo can identify the level of the originally assigned category and accordingly issues triples with links to its corresponding classification term. Thus, original classification at the finest possible level is fully preserved in the transformed dataset.

[19] Published in https://github.com/SLIPO-EU/TripleGeo/blob/master/test/classification/OSM_POI_sample_classification.yml

```csv
"category_id","category","subcategory_id","subcategory"
"A","ACCOMMODATION",1,"ALPINEHUT"
"A","ACCOMMODATION",2,"CAMPING"
"A","ACCOMMODATION",3,"CARAVAN"
"A","ACCOMMODATION",4,"CHALET"
"A","ACCOMMODATION",5,"GUESTHOUSE"
"A","ACCOMMODATION",6,"HOSTEL"
"A","ACCOMMODATION",7,"HOTEL"
"A","ACCOMMODATION",8,"MOTEL"
"B","BUSINESS",9,"ADVERTISING"
"B","BUSINESS",10,"ATM"
"B","BUSINESS",11,"AUDIOVISUAL"
"B","BUSINESS",12,"BANK"
"B","BUSINESS",13,"CONSTRUCTION"
"B","BUSINESS",14,"CURRENCY"
"B","BUSINESS",15,"EMPLOYMENT"
"B","BUSINESS",16,"ENERGY"
"B","BUSINESS",17,"FINANCIAL"
"B","BUSINESS",18,"INSURANCE"
"B","BUSINESS",19,"IT"
"B","BUSINESS",20,"LEGAL"
"B","BUSINESS",21,"PRESS"
"B","BUSINESS",22,"REALESTATE"
"B","BUSINESS",23,"TELECOM"
"B","BUSINESS",24,"TRAVELAGENCY"
"C","AUTOMOTIVE",25,"CARWASH"
"C","AUTOMOTIVE",26,"CHARGINGSTATION"
"C","AUTOMOTIVE",27,"FUEL"
"C","AUTOMOTIVE",28,"PARKING"
"C","AUTOMOTIVE",29,"RENTALCAR"
"C","AUTOMOTIVE",30,"RESTAREA"
"C","AUTOMOTIVE",31,"TOLLGATE"
"D","EAT/DRINK",32,"BAR"
"D","EAT/DRINK",33,"BIERGARTEN"
"D","EAT/DRINK",34,"CAFE"
"D","EAT/DRINK",35,"FASTFOOD"
"D","EAT/DRINK",36,"ICECREAM"
"D","EAT/DRINK",37,"INTERNETCAFE"
"D","EAT/DRINK",38,"PUB"
"D","EAT/DRINK",39,"RESTAURANT"
```

```yaml
ACCOMMODATION #A
  ALPINEHUT #1
  CAMPING #2
  CARAVAN #3
  CHALET #4
  GUESTHOUSE #5
  HOSTEL #6
  HOTEL #7
  MOTEL #8
BUSINESS #B
  ADVERTISING #9
  ATM #10
  AUDIOVISUAL #11
  BANK #12
  CONSTRUCTION #13
  CURRENCY #14
  EMPLOYMENT #15
  ENERGY #16
  FINANCIAL #17
  INSURANCE #18
  IT #19
  LEGAL #20
  PRESS #21
  REALESTATE #22
  TELECOM #23
  TRAVELAGENCY #24
AUTOMOTIVE #C
  CARWASH #25
  CHARGINGSTATION #26
  FUEL #27
  PARKING #28
  RENTALCAR #29
  RESTAREA #30
  TOLLGATE #31
EAT/DRINK #D
  BAR #32
  BIERGARTEN #33
  CAFE #34
  FASTFOOD #35
  ICECREAM #36
  INTERNETCAFE #37
  PUB #38
  RESTAURANT #39
```

Figure 10: Representations of the same classification scheme: (*left*) in CSV format, (*right*) in YAML format as used in TripleGeo

4.4.3.4. Embedded POI Classification

In most cases, each POI data asset comes with its own proprietary classification scheme that is not necessarily aligned with the classifications utilized by other POI datasets. For instance, a restaurant may have been assigned to the "food" category in dataset A, but to the "French Cuisine" subcategory under the general "Eating & Drinking" category in dataset B. Such discrepancies between classification schemes may have strong impact in the data integration process, since we need to avoid matches between POIs with a similar name in close distance but of different significance and purpose in the real world (e.g., a hotel having the same name with a nearby restaurant). As the classification schemes between two data sources may diverge, it would not easy to overcome mismatches or false misses in the integration process.

In order to deal with this issue in SLIPO, we *internally* employ a simplified, generic classification scheme for POIs that includes *15 broad categories* as listed in Table 8. This *embedded classification* reflects common characterizations identified in most of the datasets handled in the context of SLIPO. Essentially, each POI in the SLIPO ecosystem is automatically assigned to one of these broad categories, depending on its original classification. Intuitively, this automatic classification implies that POIs of different type and purpose most likely will fall under different SLIPO categories, hence reducing the chance of possible mismatches. That's

why it is important that those SLIPO categories are few, wide-ranging and inclusive, subsuming many subcategories and detailed characterization, so that any differences between data sources in the detailed classification of a given POI are reconciled.

Category
ACCOMMODATION
AUTOMOTIVE
BUSINESS
EAT/DRINK
EDUCATION
HEALTH
LANDUSE
NATURAL ATTRACTION
PUBLIC SERVICE
RELIGIOUS
SETTLEMENTS
SHOP
SPORT
TOURISM
TRANSPORT

Table 8: General categories automatically assigned to POIs during data integration

To enable such automatic assignments, each one of the 15 general categories is associated with a set of *tags*, which provide more detailed description of the various types of POIs (i.e., subcategories) belonging to each category. This dictionary of tags under each generic category[20] is used by TripleGeo during transformation of POI datasets into RDF. To do so, it examines the textual similarity of the original class with each of the tags, and finally picks the SLIPO category corresponding to the highest-ranking such tag. Thus, an extra triple is generated for each transformed POI, which specifically provides its assignment to one of the 15 generic categories. It should be noted that this information may only be used internally in SLIPO during data integration without harming the integrity and consistency of all other original information acquired by each data source. Once a data integration task is completed, extra triples with automatically assigned SLIPO categories may be skipped in the final export step to a de facto geographical POI format.

4.4.3.5. Transformation Modes

As already mentioned, TripleGeo offers four different *modes* for transformation of geospatial features to RDF:

1. *GRAPH* mode, which is disk-based and was the only option supported by ver.1.1 of the software;

[20] https://github.com/SLIPO-EU/TripleGeo/blob/master/src/resources/categories.yml

2. *STREAM* mode, which works in main memory was introduced with ver.1.2;
3. *RML* mode also works in memory and introduced with ver.1.3;
4. *XSLT* mode is used only in transformation of semi-structured data, and has not been modified since its launch with ver.1.1.

Next, we provide more details about the specifics of each transformation mode.

4.4.3.5.1. GRAPH Transformation Mode

This transformation mode employs a Jena *Model* [JenaDoc] for collecting all RDF triples produced during transformation and before emitting them to a file. This Model denotes an *RDF graph*, so called because it contains a collection of RDF nodes attached to each other by labelled relations. The Jena Model has a rich API with many methods intended to make it easier to write RDF-based programs and applications. Besides, a Model also provides an abstraction over different ways of storing the RDF nodes and relations: in-memory data structures, disk-based persistent stores and inference engines. In contrast to the initial release ver.1.1 of TripleGeo where models were retained in main memory, in all subsequent versions (until the current ver.2.0) we always employ *disk-based* models when the GRAPH transformation mode is chosen. This is a deliberate decision, in order to account even for very large POI datasets that may not be fully accommodated in main memory, thus incurring excessive computational cost for transformation. This persistent graph essentially stores custom disk-based tuple indices regarding the triples obtained after transforming each input feature (i.e., a POI). In case new triples are being added to the store, the respective indices need to be updated.

In Jena, a single triple is represented as a *Statement* with a *Subject*, a *Predicate*, and an *Object*. According to the RDF specification, only resources can be the subject of an RDF triple, whereas the object can be a resource or a literal. Predicates express the relationship between nodes according to a given ontology; in our case, this is the SLIPO ontology [SLIPO-D2.1] for POIs. Jena models support two distinct types of nodes: *URI* references and *literals*: the former denotes resources for which some assertions are made, whereas the latter denote concrete data values that appear in those assertions. A resource represented as a URI denotes a named thing. In SLIPO, this refers to a POI that has a distinct identity (e.g., a unique identifier in the dataset), which can be used as a direct reference to that resource. Literals representing values other than strings may have an attached *data type*, which helps an RDF processor convert the string representation of the literal into the correct value.

Apart from numeric values, dates, etc. that can be represented by suitable XSD data types, in SLIPO there is the crucial requirement to support *geometries* of POIs. Such geometries denote the location of the POI and can be not only points (i.e., a pair of longitude/latitude coordinates for a restaurant), but also linestrings (e.g., for a scenic route POI), polygons (e.g., the spatial extent of an archaeological site), or even more complex geometries (e.g., a geometry collection with the area, the boundary and the centroid of a building). Such geometry literals are supported according to the GeoSPARQL standard [OGC12], even though the Jena model is *agnostic* of geospatial data types and operations. For instance, topological queries like "find the POI at a given location" (specified by coordinates) cannot be answered with a SPARQL query against the RDF graph in Jena. However, these geometries can be retained in the model and TripleGeo can correctly emit them to the output RDF files with proper WKT serialization according to GeoSPARQL standard.

When the entire input dataset is consumed and its contents have been transformed into statements in the RDF graph, TripleGeo exports the model into a file according to a user-specified serialization format containing all triples in the graph.

4.4.3.5.2. STREAM Transformation Mode

Jena supports processing operations over RDF in a streaming fashion [JenaStream] in cases that applications need to manipulate RDF data at scale. High performance readers and writers for all standard RDF formats are available with the Jena RIOT (RDF I/O technology) API, also extensible with custom formats. N-Triples/N-Quads provide the highest input parsing performance using W3C Standards.

In transforming geospatial data into RDF with the STREAM transformation mode in TripleGeo, we are actually concerned with writing RDF data as a stream. Unfortunately, not all RDF formats are suitable for streamlined writing to files. N-Triples and N-Quads are always written as a stream. Formats that provide pretty printing (for example the default RDFFormat for each of Turtle, TriG and RDF/XML) require analysis of the entire model in order to determine nestable structures of blank nodes and for using specific syntax for RDF lists.

So, for the STREAM transformation mode in TripleGeo, we employ a different strategy. We handle each incoming feature (i.e., a POI with its geometry and all its thematic attributes) in isolation from the rest and we can transform it to triples according to the given YAML mapping to the ontology. Of course, this is implicitly based on the reasonable assumption that all information about a POI is fully included *in a single record*, and there are no properties that need be searched in other entities (e.g., by joins to other input tables or files). This is the typical case in almost all POI datasets available and certainly all those handled in the context of SLIPO. So, when all attribute values of a given POI have been turned into triples, these can be readily emitted and written to the output file. The only restriction is their *serialization*, because they can only be written in N-Triples/N-Quads as a stream.

Regarding writing to output, the user may regulate the rate at which results are stored in the files in order to avoid high I/O interaction with the disk and reduce execution time. More specifically, in the TripleGeo configuration file, the user can prescribe a number of input records that will be transformed in the same *batch* (by default, a batch consists of 10 records), so that their resulting triples are first accumulated in memory and spilled to the disk file together. Ideally, the size of the batch represents a trade-off between not too frequent disk I/O and moderate size of collections of transformed triples retained in memory.

By not maintaining a persistent graph and promptly emitting transformed triples, this STREAM mode can achieve orders of magnitude faster execution, as reported in our experimental results in Section 5.

4.4.3.5.3. RML Transformation Mode

To invoke this transformation mode with TripleGeo, suitable RML mappings for input data features should have been specified in advance (as explained in Section 4.4.3.2.1) and declared in the configuration file.

In aligning RML to work with TripleGeo, we have made slight modifications in source code regarding RML processing. More specifically, an instance of *RML Processor* is created, equipped with as many *RML performers* as the Triples Map constructs defined in the RML mappings. The RML processor extracts in advance all iteration patterns corresponding to *Triples Maps* constructs in the RML mappings. Hence, each

Triples Map activates a dedicated *RML Performer* that is employed to handle specific fragments of an input feature (e.g., specific thematic attributes of a POI regarding its postal address).

In terms of integration within TripleGeo, we have modified RML processing to work in a *streaming* fashion. Note that the original software [RML], consumes input data and creates a materialized Sesame SAIL (Storage and Inference Layer) repository where all transformed triples are stored, and finally exports all output to a user-specified serialization. As this dependence on materialized (in memory or on disk) repositories may easily become a bottleneck in transforming large datasets, we created a wrapper over RML processors for iterating over each input feature and producing transformed triples according to the RML mappings. Hence, for each incoming feature provided by the Feature Iterator over the input data, all activated RML Performers are applied. The defined Subject Map and Predicate-Object Maps are applied against attribute values and the corresponding triples are generated. When necessary, execution of dependent Triples Map is triggered by the appropriately defined *Parent Triples Map* and a *nested mapping* is being applied. All generated triples are kept in an in-memory collection and not materialized on disk. As in the STREAM mode, the user may specify a *batch* size for writing the output in order to minimize I/O cost, so that collected triples are only written to the output file periodically, i.e., once a given number of input records have been processed. Again, as in the STREAM mode, the only restriction is *serialization* of triples, because they can only be streamlined in N-Triples/N-Quads format to the output file.

4.4.3.5.4. XSLT Transformation Mode

This mode can be *exclusively* used for transforming *semi-structured* geographical files into RDF. In particular, it accepts input datasets in GML, KML, and XML formats, as well as *INSPIRE-aligned* data (GML) and metadata (XML). *Geography Markup Language* (GML) is an OGC standard [GML] for representing geospatial information. The basic primitives of GML are spatial features as locations on Earth, and their geometric shapes are modelled with vectors (points, lines, polygons, etc.). The original GML model (version 1.0) was based on RDF/RDFS profiles, but afterwards the OGC introduced XML schemas for interoperability with existing spatial databases. Although this RDF profiling is no longer supported, subsequent GML models (current version is 3.3) still retain certain features of RDF, and most importantly the concept of child elements as properties of a parent resource (RDFS). Thanks to this relaxed "triple-like" binding, GML (as well as similarly structured KML and XML) features may be transformed into RDF using XSLT transformation.

In practice, the user needs to create an application profile for the input dataset taking advantage of a set of predefined *templates* concerning mappings of geometries and thematic attributes for XSLT transformation. Once invoked, such a custom *XSL style sheet* accepts an XML, GML or KML file and maps each input feature into suitable RDF statements according to the mapping. The result is an RDF/XML representation of input GML and KML data or XML metadata. Output RDF files can be readily loaded into a triple store.

TripleGeo also supports *INSPIRE-aligned data* (in GML) and *metadata* (in XML). Regarding such XML metadata, a generic XSL stylesheet (`Metadata2RDF.xsl`) covers all elements and can be reused against any metadata conforming to INSPIRE specifications. Besides, we have introduced one custom XSL stylesheet for each INSPIRE Data Theme (Annex I) [INSPIRE-Themes], practically translating each domain-specific element in the respective GML schema into a suitable RDF resource. Using XSL stylesheets for GML representations of INSPIRE-compliant features seems most preferable and generic, as these scripts can be re-used and can work with any XSLT parser. Regarding handling of 2-dimensional OGC geometries, a

generic stylesheet (`GML2WKT.xsl`) can convert from Geometry Markup Language (GML) to Well Known Text (WKT) representations according to the GeoSPARQL standard [OGC12]. This script can cope with a wide range of both primitive and complex geometric types, including identification of the spatial reference (CRS) of every geometry in order to provide a complete WKT representation for the resulting RDF dataset.

For convenience, all these XSL style sheets have been integrated into TripleGeo since ver.1.1. Users may edit existing style sheets in order to define suitable values for the attributes, namespaces, and other specifications regarding the data or metadata at hand, and then perform invoke TripleGeo to perform an XSLT transformation and finally obtain the resulting RDF files.

4.4.3.6. Identifiers Registry

TripleGeo can support the process of registering POI data to the Identifiers Registry. The Identifiers Registry is a component of the SLIPO architecture that serves a twofold purpose:

- *Creation and assignment of identifiers*. The Identifiers Registry includes a configurable mechanism for creating and assigning identifiers to POIs. As described in Section 4.4.3.1, TripleGeo offers functionality for assigning URIs to POIs during the transformation process. However, in certain cases, it may be desirable or required to decouple the identifier generation process from the transformation process (e.g., if the user wishes to apply a different identifier scheme than that supported by TripleGeo, or in case transformation is handled by another tool that does not generate identifiers according to the expected scheme). In such cases, the user can rely instead on the Identifiers Registry to generate and assign an identifier (URI) to each POI, which substitutes any temporary identifier that may have been assigned to it during transformation.
- *Lookup service for POIs*. The Identifiers Registry maintains a database associating each registered POI with its respective identifier (URI), which may be either the one assigned to it during transformation or the one generated by the Identifiers Registry itself. In addition, it stores certain basic attributes about each registered POI, such as its name, category and location (see Section 4.4.2.3). Thus, the Identifiers Registry can serve as a lookup service for retrieving basic information about a previously imported POI or for checking whether a given POI has already been registered, and which identifier has been assigned to it.

Interaction with the Identifiers Registry takes place via a RESTful API. Next, we list the supported API calls.

4.4.3.6.1. POI Registration

Registration of new POIs can be done in batch mode using the following method:

https://[registry.dev.slipo.eu]/register	(Request type: POST)

The method accepts a list of POIs to be registered, where each entry in the list contains the following parameters:

- *poi_id* (optional). If present, the POI is registered with the specified id, otherwise the Identifiers Registry generates and assigns an identifier to the POI.
- *source*. An identifier specifying the source of this POI.
- *source_id*. The identifier of the POI in its source.

- *names*. A list of known names for the POI.
- *categories*. A list of categories associated with the POI.
- *geom*. The (default) geometry of the POI in GeoJSON format.

The response is a JSON document containing the following information:

- *success*. A Boolean value indicating whether an error has occurred.
- *errors*. In case of an error, an array containing messages for the errors that occurred.
- *result*. In case of success, an array of objects, each one contained the source_id of the POI that was registered and the id that was assigned to it (in case of relying on the Identifiers Registry to assign a new id to the POI).

4.4.3.6.2. POI Deletion

An existing POI entry can be removed from the Identifiers Registry using the following method:

https://[registry.dev.slipo.eu]/delete	(Request type: POST)

Input parameters:

- *poi_id*. The identifier of the POI to be deleted.

Response:

- *success*. A Boolean value indicating whether an error has occurred.
- *errors*. In case of an error, an array containing messages for the errors that occurred.

4.4.3.6.3. POI Search

The following methods are provided for retrieving POI entries from the Identifiers Registry.

https://[*registry.dev.slipo.eu*]/poi/id_search	(Request type: POST)

This method retrieves POI entries by their id.

Input parameter:

- *poi_id*. The URI used to identify the POI.
- *categories* (optional). A list of categories used for filtering.

Response:

- *success*. A Boolean value indicating whether an error has occurred.
- *errors*. In case of an error, an array containing messages for the errors that occurred.
- *result*. A POI entry (comprising the basic information stored in the Identifiers Registry about the POI).

https://[*registry.dev.slipo.eu*]/poi/source_id_search	(Request type: POST)

This method retrieves POI entries by their source and source id.

Input parameter:

- *source*. An identifier specifying the source of the searched POI.
- *source_id*. The identifier of the POI in its source.

Response:

- *success*. A Boolean value indicating whether an error has occurred.
- *errors*. In case of an error, an array containing messages for the errors that occurred.
- *result*. A POI entry.

https://[*registry.dev.slipo.eu*]/poi/name_search (Request type: POST)

This method retrieves POI entries by name.

Input parameters:

- *name*. The name of the POI.
- *categories* (optional). A list of categories used for filtering.

Response:

- *success*. A Boolean value indicating whether an error has occurred.
- *errors*. In case of an error, an array containing messages for the errors that occurred.
- *result*. An array of POI entries.

https://[*registry.dev.slipo.eu*]/poi/bbox_search (Request type: POST)

This method retrieves POI entries by a bounding box.

Input parameters:

- *left*. The minimum x-coordinate.
- *right*. The maximum x-coordinate.
- *bottom*. The minimum y-coordinate.
- *top*. The maximum y-coordinate.
- *categories* (optional). A list of categories used for filtering.

Response:

- *success*. A Boolean value indicating whether an error has occurred.
- *errors*. In case of an error, an array containing messages for the errors that occurred.
- *result*. An array of POI entries.

https://[*registry.dev.slipo.eu*]/poi/radius_search (Request type: POST)

This method retrieves POI entries within a given radius from a given point.

Input parameter:

- *x*. The x-coordinate of the point.
- *y*. The y-coordinate of the point.
- *radius*. The radius of the search.
- *categories* (optional). A list of categories used for filtering.

Response:

- *success*. A Boolean value indicating whether an error has occurred.

- *errors*. In case of an error, an array containing messages for the errors that occurred.
- *result*. An array of POI entries.

4.4.4. Scalable Transformation

As already discussed, each POI is considered as a separate, autonomous entity with its own properties (name, category, locations, etc.), e.g., represented as a single record in a Shapefile or DBMS table. This is the most typical representation of a POI in most data sources. Thus, when it gets transformed into RDF, TripleGeo generates a collection of triples per POI without any links to other POIs. Of course, it may occur that a POI in the input dataset could refer to another POI for expressing *relationships* (e.g., a shop within a shopping centre, an office in a building), but such cases can be handled with suitably defined mappings that explicitly specify how such features will be retained in the RDF output.

Assuming that such conceptual isolation of information concerning each individual POI holds in an input dataset, it opens up opportunities for advanced performance in their transformation to RDF and scalability with large data volumes. Quite simply, a POI dataset may be split into a number of *disjoint subsets*, each one ideally having an equivalent number of POIs. Then, a separate RDF transformation task may be employed for each partition. Naturally, the optimal number of partitions highly depends on the available resources of the cluster or the standalone machine that must perform the RDF transformation. Of course, more partitions mean that each task has to convert a smaller chunk of the original data, but this may not always lead to an overall faster execution. Typically, POI *data partitioning* may be carried out in several fashions:

- *Evenly*, by splitting the dataset into a specified number of partitions, each one holding the same number of POIs.
- *Hash-based*, by employing a hashing algorithm over a partitioning key (e.g., the URIs). A good choice of hashing algorithm can evenly distribute POIs among partitions, giving partitions of approximately the same size.
- *Spatially*, e.g., by employing a uniform grid, a quadtree, or other tessellations.
- *Thematically* after grouping POIs by a specific attribute, e.g., per category or country of origin. In this case, ranges of values may be defined (e.g., over dates), and each such range may be used to map data into a separate partition.

Depending on the strategy, partitioning may yield subsets of varying sizes. For instance, distrubution of POIs in cells of a grid partitioning may not be even, as the original data may be skewed (e.g., more dense POIs in urban areas). But, in any case, each chunk may be processed separately and produce its own RDF output. Finally, these partial results may be merged into a single one, or loaded into a triple store to create a unified RDF representation for the original data.

4.4.4.1. Concurrent Multi-threaded Execution

As a first approach towards scalable RDF transformation, we have adjusted TripleGeo to run concurrently in *multiple concurrent Java threads*. Up to ver.1.5, this method required a external preprocessing step to subdivide the input data and store each partition into a separate file having the same schema as the original. In particular, the user had to specify in the configuration any number of input datasets, provided that they

all complied to the same attribute schema and the same file format. Starting from ver.1.6, this partitioning phase is encapsulated in TripleGeo for POI data in CSV and shapefile format, and can be invoked in the configuration settings of the software.

Once data partitioning is complete, a separate thread of TripleGeo can be launched for transforming each disjoint data chunk, which proceeds totally isolated from the rest. Each thread abides by the same configuration settings, e.g., applies the same classification scheme, attribute mappings, namespaces, as if it worked alone. The entire transformation task is completed once all threads conclude their task, and resulting triples are written into separate files (one per thread) corresponding to its subset of the input data. Depending on the specified serialization, there is the option to merge output files into a unified one (e.g., in case that N-Triples is the chosen serialization) or they can be loaded into a triple store to create a unified RDF representation for the original data.

4.4.4.2. Parallelized Execution over Spark

For parallelized POI data transformation in cluster infrastructures, we have also extended TripleGeo to use Apache Spark [Spark] and GeoSpark [GeoSpark] as its underlying partitioning mechanism. This parallelized extension for TripleGeo is available for several spatial formats. In case of *GeoJSON* and *CSV*, input POI information is parsed into a Spark *DataFrame* that contains all original attributes. Information from ESRI *shapefiles* is read using GeoSpark into a *Spatial Resilient Distributed Dataset (RDD)*, which represents records along with their geometries.

This parallelized process has two stages. First, (Geo)Spark loads the input dataset and partitions it into a user-specified number of partitions. The input data is read and stored in partitions on HDFS accessible by different worker nodes. In the second step, each Spark worker performs an independent transformation task by invoking its own TripleGeo instance against its assigned subset of the POI data. In particular, each input POI instance from the Spatial RDD is mapped to a collection of (key, value) pairs (one key per attribute value) and this is forwarded to TripleGeo for RDF transformation according to the global configuration. Note that there is no need for reshuffling data between workers, since partial transformation results produced by each worker come from disjoint chunks of the original POI data and have no associations to other POIs.

As our experiments in Section 5 confirm, employing multiple concurrent threads or Spark-based partitioning for transforming disjoint pieces of large POI datasets can offer orders of magnitude performance gains compared to standalone execution and testifies the robustness and scalability of TripleGeo. Possible future extensions to TripleGeo may include more advanced data partitioning strategies in order to make estimates about suitable partitions considering available system resources (CPU, memory, disk) and statistics over data characteristics (spatial distribution, number of attributes, etc.).

4.5. Reverse Transformation from RDF

As already mentioned, transformation is actually a *bidirectional* process that should also allow the backward transformation of linked POI data (*potentially interlinked or fused*) into conventional POI formats, and thus enabling existing products, systems, and services to exploit the integrated POI datasets. TripleGeo supports this reverse transformation of RDF POI data into de facto POI formats (like *CSV, GeoJSON,* or *ESRI shapefiles*). Of course, there exists an impedance mismatch in this direction, given that the SLIPO ontology

is semantically more expressive than the conventional POI schemata. In TripleGeo, we have prepared specifications (in SPARQL) that generally retrieve the same attributes as in original geospatial files given to transformation. But generally, POI attributes, relations and metadata in RDF representation will be richer than what can be supported by conventional file formats. Since the SPARQL query that is used to extract records from the RDF data is customizable, TripleGeo provides considerable flexibility in exporting the maximum possible amount of semantic (linked, enriched, fused) POI information and metadata into the resulting attributes of conventional POI formats, as demonstrated in the various use cases [SLIPO-D5.2].

In this Section, we outline the processing flow of reverse transformation and explain how it is possible to reconstruct RDF data on POIs with geometries into records stored in a geospatial file.

4.5.1. Architecture

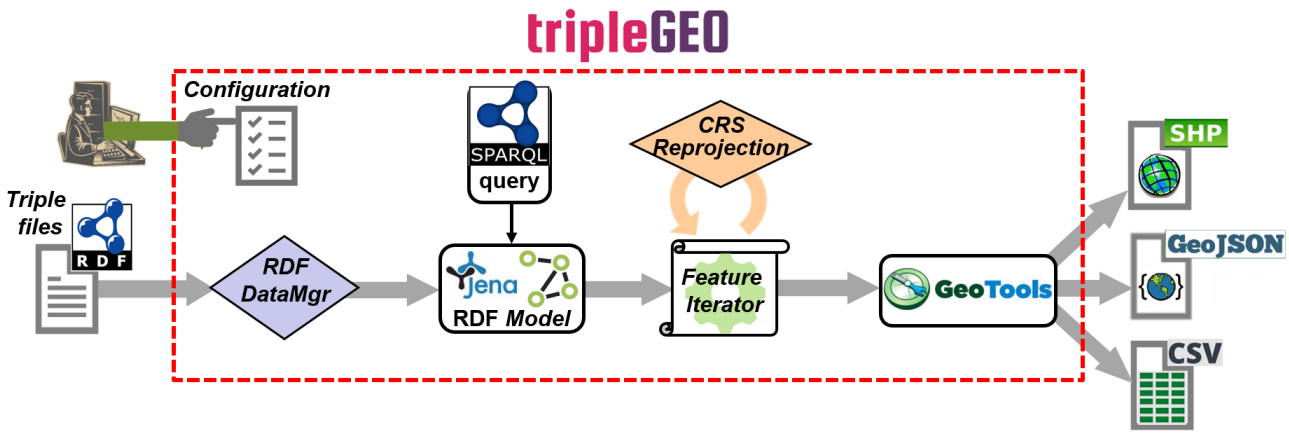


Figure 11: Processing flow of reverse transformation from RDF in TripleGeo

As in the case of transformation to RDF, the *reverse transformation* functionality of TripleGeo works in a straightforward fashion according to some preconfigured settings. Figure 11 illustrates the flow diagram used for reconverting RDF triples with geometries into records in a geospatial file. Next, we outline the basic components of this utility:

Input RDF data may be obtained from files with standard RDF serializations. Multiple RDF files may be specified (with the same serialization and with statements obeying the same ontology) in order to reconstruct a *single* geospatial file with all (geometric and thematic) information.

A *configuration file* lists all properties that are used to control the various stages of reverse transformation: how input data will be accessed, which data is involved, whether geometries must be transformed in another reference system, as well as the output format. All properties that may be specified in this file are explained in Section 9.2.2.

TripleGeo achieves reverse transformation from RDF to geospatial format by creating an intermediate disk-based Jena *model* that stores all input triples (equivalent to the *GRAPH* mode in transformation). Hence, the native *RDFDataMgr* available from Jena API is applied against the input triple files in order to build this RDF model.

The user must specify (in a separate file) a SELECT query in SPARQL that will be used to retrieve results (records) from the constructed model. In order to execute correctly and return meaningful results, this query should conform with the underlying ontology of the input RDF triples. An examply such query is listed in the Annex (Section 7.5).

Once the query is submitted and results are returned, a *feature iterator* consumes each result and recreates a record from it. In particular, each attribute concerning a POI and specified in the SELECT query becomes a column in the resulting file. Optionally, *reprojection* of geometries into another spatial reference system (CRS) is available. This coordinate transformation is carried out thanks to the integrated GeoTools library and according to user specifications for the source and target CRS.

Regarding *thematic* (i.e., non-spatial) attribute values (e.g., type, name, contact information) of an input feature, these are generally stored as strings. This is typical when output is written into *CSV* files. With respect to writing output to *ESRI shapefiles* or in *GeoJSON format*, the integrated GeoTools library provides some functionality for defining data types. So, in case that the data type (e.g., date or numeric) is specified for a value in the RDF input, the respective attribute is defined with the equivalent type available in the shapefile format; otherwise, these are stored as strings.

Finally, note that reverse transformation in TripleGeo does not support parallelization, but works on a *single thread* instantiation only. Even in the case of multiple input RDF files, these are acquired one by one and inserted into a common RDF model so as to generate a single output file containing all reconstructed geospatial features. This strategy guarantees that the resulting records will be complete and all properties in the graph can be properly exported as attribute values in the output geographical file.

4.5.2. Input and Output

4.5.2.1. Input

In terms of input RDF serializations, and according to the specifications of the Jena API [JenaDoc] that is used to build the disk-based model, triples can be obtained in one of the following formats:

- RDF/XML;
- RDF/XML-ABBREV;
- N-TRIPLES;
- N3;
- *TURTLE* (also abbreviated as *TTL*).

Note that these serializations are exactly the same ones supported for the output RDF files produced by the transformation functionality of TripleGeo. In terms of standardization, the input triples are conformant to W3C standards, in order for the compiled model to be queryable via SPARQL. With respect to geometries used as objects in triples, these should be specified as WKT literals according to the GeoSPARQL standard [OGC12].

4.5.2.2. Output

Results from the reverse transformation by TripleGeo can be stored in one of the following *vector file formats*:

- ESRI *shapefiles* [ESRIshp]. Note that only one of the primitive geometric data types for (Multi)Points, (Multi)LineStrings, and (Multi)Polygons is allowed in a single shapefile. In addition, data types for thematic attributes are converted to those supported by shapefiles, e.g., timestamp values must be converted to strings in order to be accepted.
- GeoJSON [GeoJSON]. GeoJSON supports all standard geometry types [OGC11], but their CRS must always be WGS84 (lat/long coordinates). In case that there are mixed geometry types in input geometries (e.g., some are points while others are polygons), TripleGeo properly retains in the output GeoJSON file. Handling data types for thematic attributes is carried out as for shapefiles.
- *CSV* containing geometries either as pairs of (*x,y*) coordinates (for points) or in WKT serialization (for any geometry type). Users must specify a suitable delimiter character (not present in any attribute value), and optionally a quote character for enclosing string values. All thematic attributes in the output file are considered as strings.

5. TripleGeo Experimental Evaluation

In this Section, we examine the efficiency and scalability of the TripleGeo software, as it has evolved and finalized through the course of the project. We report results from a comprehensive validation of TripleGeo against large POI datasets extracted from OpenStreetMap [OSM] and stored in various geospatial repositories. It should be mentioned that TripleGeo has been successfully used to transform diverse POI datasets provided by the SLIPO industrial partners for the various use cases [SLIPO-D5.2].

In Section 5.1, we describe the original OSM datasets and present some of their subtle characteristics, with particular emphasis on their classification into categories, the variety of their geometric representations, as well as the multitude of thematic attributes that may be extracted. We also explain how these OSM data were used to generate several very large synthetic datasets or varying size to be employed in scalability tests. In Section 5.2, we discuss the experimental setup, explaining the performance metrics assessed in our tests. Finally, in Section 5.3, we discuss performance results regarding both transformation to RDF and reverse transformation from RDF, confirming that TripleGeo can now handle any number of thematic attributes and map them to a given POI ontology. Most importantly, these results testify that TripleGeo has achieved *orders of magnitude performance gains* compared to its original release, and can *efficiently transform millions of POIs in a few minutes*. Furthermore, by employing data partitioning schemes and parallelization in modern cluster infrastructures, TripleGeo has confirmed its ability to transform *massive collections of POIs* and generate the resulting RDF triples with *minimal latency*, thus fulfilling the scalability targets set in the beginning of the project.

5.1. Datasets

5.1.1. Real POI Data extracted from OSM

In order to verify the transformation capabilities of TripleGeo, we have extracted POI data concerning all Europe from the OpenStreetMap (OSM) database [OSM], available under the Open Database License (ODbL). In particular, geospatial and thematic information in OSM is distinguished in the following core elements:

- *Nodes*. These are points with a geographic position, stored as coordinate pairs (longitude, latitude) georeferenced in WGS84. They are used to represent any kind of points, from road intersections to points of interest, as well as vertices of more complex spatial entities (lines, polygons, etc.).
- *Ways*. These are ordered lists of nodes, representing a polyline or a polygon. They are used for representing both linear features (e.g., streets, rivers), and areas (e.g., parks, lakes).
- *Relations*. These are ordered lists of nodes, ways and relations. They are used for representing the relationship of existing nodes and ways.
- *Tags*. These are key-value pairs (both arbitrary strings). Even though an ontology [OSMOnto] is recommended for such tags, this is not strictly followed, so free tags may be used in representing metadata about the map objects (such as their type, their name and their physical properties).

Obviously, the data schema and format in OSM is not specifically tailored to representing and describing POIs, but generally any type of map elements. In order to be able to obtain POIs from OSM, we had to extract: (a) the *detailed geometry* of each POI and (b) the complete set of *tags* associated with an OSM feature. Geometry can be more complex than a centroid; for example, in OSM an archaeological site may be represented by its boundary, and a popular scenic route by a polyline (linestring). We have made use of the OGR/GDAL software [GDAL] to extract all features contained in an OSM file, including their detailed geometry (stored in an attribute called *shape*), as well as the full set of tags (*all_tags*) available for each feature, i.e., a list of key-value pairs in *hstore* representation [HSTORE] for all user-specified OSM tags found in a feature. We have extracted such features within the spatial extent of Europe from the OSM database.

Category	Type
POI	HAMLET, VILLAGE, PEAK, CRANE, SUBURB, TOWN, CAVE, TOWERLOOKOUT, BUNKER, EMBASSY, MINE, TOWERCOMMUNICATION, CITY, PEAK1
TRANSPORT	BUSSTOP, FUEL, STATION, TRAMSTOP, RENTALCAR, MARINA, AIRPORT, SUBWAY, LIGHTHOUSE, TERMINAL
SHOP	DEPARTMENTSTORE, SUPERMARKET, CONVENIENCE, CLOTHES, HAIRDRESSER, BAKERY, CARREPAIR, DIY, CAR, FLORIST, BICYCLE, BUTCHER, SHOES, ALCOHOL, HIFI, KIOSK, JEWELRY, MOTORCYCLE, BOOK, PHONE, LAUNDRETTE, CONFECTIONERY, COPYSHOP, GIFT, VENDINGMASCHINE, MARKETPLACE, TOYS, COMPUTER, GARDENCENTRE, GREENGROCER, PET, NEWSPAPER, TOBACCO, FISH, HEARINGAIDS, MUSIC, VIDEORENTAL
FOOD	RESTAURANT, CAFE, FASTFOOD, PUB, BAR, ICECREAM, BIERGARTEN
TOURIST	MEMORIAL, ATTRACTION, FOUNTAIN, THEATRE, ART, MUSEUM, ARCHAELOGICAL, CASTLE2, BEACH, RUINS, MONUMENT, NIGHTCLUB, CINEMA, INFORMATION, WINDMILL, ZOO, THEMEPARK, CASTLE, BATTLEFIELD, WRECK
EDUCATION	SCHOOL, NURSERY, UNIVERSITY, COLLEGE
LANDUSE	GRASS, CONIFEROUSDECIDUOUS, ALLOTMENTS, SWAMP, QUARY, MILITARY, SCRUB, HILLS, DECIDUOUS, CONIFEROUS
AMENITY	PUBLICBUILDING, POSTOFFICE, TOWNHALL, FIRESTATION, LIBRARY, PLAYGROUND, POLICE, COURT, PRISON
POW	CHRISTIAN, ISLAMIC, UNKOWN, JEWISH, BUDDHIST, HINDU, SIKH, BAHAI, JAIN, SHINTO
ACCOMMODATION	HOTEL, CAMPING, CHALET, HOSTEL, ALPINEHUT, CARAVAN, MOTEL
HEALTH	PHARMACY, DOCTORS, HOSPITAL, DENTIST, VETERINARY, HOSPITALEMERGENCY
SPORT	LEISURECENTER, SOCCER, STADIUM, SWIMMING, TENNIS, GOLF, HORSE, CLIMBING, BASKETBALL, SHOOTING, BOWLING, GYM, SNOOKER, GYMNASIUM, SKATING, CRICKET, MOTORRACING, BASEBALL, CANOE, MINIATURGOLF, FOOTBALL, SKIINGDOWNHILL, ICESKATING, ARCHERY, WATERSKI, SURFING, DIVING
MONEY	BANK, EXCHANGE
BARRIER	BLOCKS
WATER	WEIR, TOWER, DAM

Table 9: Classification scheme applied against POIs extracted from OpenStreetMap

The result of such extraction may refer not only to POIs, but also to other OSM features (e.g., linestrings representing road segments). What is lacking from OGR/GDAL is the ability to identify POIs and classify them

into specific *categories*. In order to extract POIs from OSM, we have configured a filter for OGR/GDAL that can extract OSM features into five layers stored into respective tables in a PostgreSQL/PostGIS database:

- *points* : OSM node elements that have significant tags attached;
- *lines* : OSM way elements that are recognized as non-area;
- *multilinestrings*: OSM relation elements that form a multilinestring (i.e., `type = 'multilinestring' OR type = 'route'`).
- *multipolygons* : OSM relation elements that form a multipolygon (i.e., `type = 'multipolygon' OR type = 'boundary'`), and "way" features that are recognized as area; and
- *other_relations* : OSM relation elements that do not belong to the above two layers.

Then, based on the tags available for each OSM element, we categorized the resulting records according to the two-tier classification scheme depicted in Table 9. This classification was obtained from OsmPoisPbf [OsmPoisPbf] and defines a user-specified filter file that maps a specific combination of OSM tags into a single class. This scheme consists of a *primary category* (one of the 15 categories listed in Table 9; each feature has exactly one category from this list) and a *secondary type* selected among 167 distinct values identified in OSM tags (each feature has exactly one type selected from the list). By parsing the tags available in each element (listed in the types column in Table 9), we determined their corresponding category (the first column in Table 9).

Finally, we compiled into a single table in the PostgreSQL/PostGIS database all OSM features classified as POIs with the aforementioned process. This table contains 7,447,693 POI records over Europe as illustrated in Figure 12; in the sequel, we refer to this real dataset ***D*** as *OSM 7.4M POIs*. In addition to the basic attributes extracted with OGR/GDAL, we specified filters with SQL queries over the set of OSM tags in order to isolate particular attributes concerning POIs. As listed in Table 10, these extra attributes refer to *address* and *contact* information, as well as other complementary attributes. Note that much more information from original OSM tags can also refer to POIs (e.g., payment methods, access to services, etc.), which can be potentially utilized in the POI data integration lifecycle.

Note that geometric information for these POIs concerns not only *Points* (in longitude/latitude coordinates) according to the OGC geometry types, but also *LineStrings*, *MultiLineStrings*, *MultiPolygons*, as well as Geometry Collections, as listed in Table 11. Typically for OSM data, all geometries in this dataset are georeferenced in WGS84 (EPSG:4326).

Basic attributes	Address-related attributes	Contact-related attributes	Other attributes
geometry	street name	phone number	international name
OSM identifier	house number	fax number	category
name	postal code	email address	country
type	city	webpage	opening hours

Table 10: Attributes extracted from OpenStreetMap concerning POIs

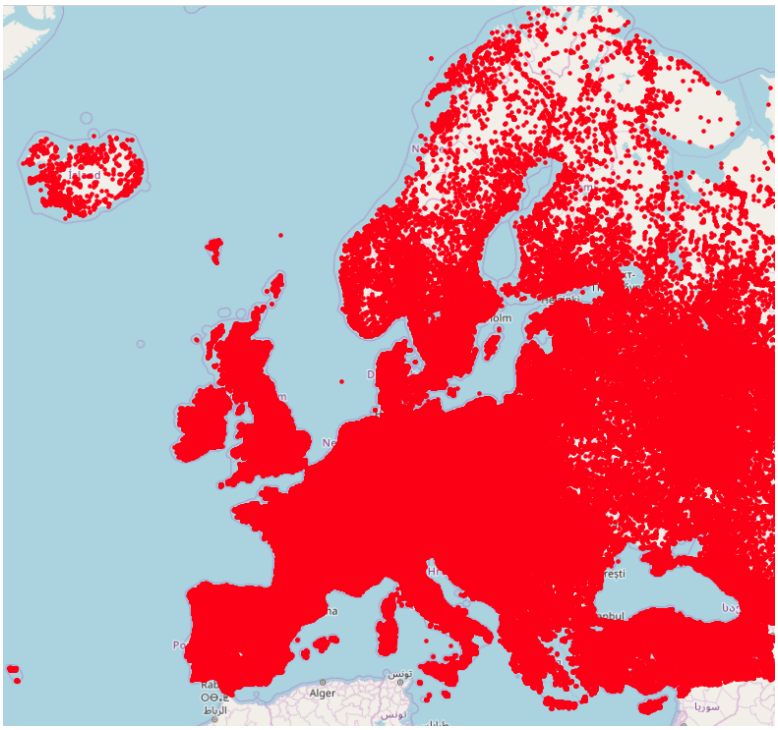


Figure 12: Points of Interest (POI) extracted from OpenStreetMap across Europe

OGC Geometry type	*D*: OSM 7.4M POIs	*S*: OSM 1M POIs
Point	5420386	738877
LineString	16361	2718
MultiLineString	3	0
MultiPolygon	2005337	257711
GeometryCollection	5606	694
TOTAL	7447693	1000000

Table 11: Number of POIs available per geometry type in the two *real* OSM datasets

In order to conduct tests that respect the capabilities of earlier versions of TripleGeo, we have also taken a subset of this real OSM dataset ***D*** by randomly selecting one million records. This sample dataset ***S***, called *OSM 1M POIs*, is used to compare performance in transformation amongst subsequent versions of TripleGeo, in handling an increasing number of attributes, as well as in testing the reverse transformation functionality.

In either of those two POI collections ***D*** and ***S***, OSM data was used "as is" without modifying original geometric or thematic information. Since these datasets reside in a *PostgreSQL/PostGIS* database, we have also extracted them in three other formats:

- *CSV* (comma separated values), where attribute values are actually separated by '|';
- *ESRI shapefile*, which is a de facto format for geographical vector information; and
- *Oracle Spatial*, which handles geometries in its own custom representation (`SDO_GEOMETRY`).

These alternative formats where employed in order to verify the correct functionality of TripleGeo when data has to be retrieved from different spatial repositories.

5.1.2. Generation of synthetic POI data

For scalability tests over Spark, we also synthetically generated some extra synthetic datasets. This generation has been carried out simply by replicating base dataset ***D*** several times (×2, ×4, ×8, ×16), as listed in Table 12. Note that for these synthetic datasets we have not applied the *synthetic generator* used for the benchmark tests in [SLIPO-D1.4]. Since our major concern was scalability and not their utilization in data integration tasks, no changes have been done to any attributes (spatial or thematic); instead, their contents were copied "as is" to each generated file in order to create very large POI collections of varying size.

Synthetic dataset	Size (#POIs)
$D \times 2$	4,895,394
$D \times 4$	29,790,788
$D \times 8$	59,581,576
$D \times 16$	119,163,152

Table 12: Number of POIs generated for the various *synthetic* POI datasets

5.2. Experimental Setup

We have repeatedly applied TripleGeo against various sample and third-party datasets in order to verify its correct functionality and smooth operation.

In terms of validating *performance* and *efficiency* of TripleGeo against POI data, we have specifically carried out tests against the two *real* OSM datasets presented in Section 5.1. These experiments were conducted on a Virtual Machine running Debian (Linux 3.16.0) on an Intel Core i7-3820 CPU with 4MB cache at 2.2 GHz. This VM was given 8GB RAM, 1GB swap, 4 (virtual) CPU cores and 300GB disk of storage space. All datasets were locally available in the same VM where TripleGeo is running, so no network delays were involved.

Scalability tests with TripleGeo over Spark were conducted on top of a YARN/Hadoop cluster consisting of seven (7) processing nodes using an HDFS cluster for storage (with a disk capacity of 1.2TB); by default, the HDFS block size is 64MBs. Each node has Intel Xeon CPU E5-2650 v3 @ 2.30GHz and can allocate 7 vcores and 15GB RAM for running YARN tasks. All nodes in the cluster are connected on a rack-local network with a Gigabit Ethernet switch and run Linux Ubuntu 16.04 LTS with Hadoop 2.9, Spark 2.2.3, and GeoSpark 1.1.2.

Since all datasets are available in either structured geographical files or geospatially-aware DBMSs, the XSLT mode in TripleGeo is not involved, as it can be only applied on semi-structured data (GML, KML, XML). Hence, we only present results concerning the *GRAPH*, *STREAM*, and *RML* modes.

Each experiment was executed in *cold runs*, i.e., invoking TripleGeo immediately after all caches of the operating system are cleared, the DBMS is re-started (if used), and no data is loaded into the system's main memory.

In the experiments, we primarily measure the *clock time* (in seconds) required to transform a given dataset. This refers to the *end-to-end* time elapsed since the beginning of the process, i.e., including accessing the

repository (i.e., the file or DBMS holding the input data), applying mappings and classifications, transforming all input features, and also writing the resulting RDF triples into files on disk. Note that computations in TripleGeo are both I/O and CPU intensive. In particular, heavy I/O is involved when reading large input datasets and finally writing the output triples; note that, in general, RDF triples are inherently verbose when compared with original records. Besides, most CPU activity is dedicated into actual transformation, when mappings are applied to each input feature with possible coordinate reprojections. Execution tests on top of Spark also indicate the *data partitioning* time.

In the two transformation modes of TripleGeo that work in a *streaming* fashion (i.e., *STREAM* and *RML*), we also provide results regarding the *average throughput*, i.e., the rate in triples/sec at which TripleGeo generates RDF triples as it progressively consumes the input dataset.

5.3. Results

Next, we provide indicative results for a set of experiments validating performance and scalability of TripleGeo both in *transformation* from geospatial formats to RDF and in *reverse transformation* from RDF back to geospatial files.

5.3.1. Transformation to RDF

5.3.1.1. Performance across Successive Software Releases

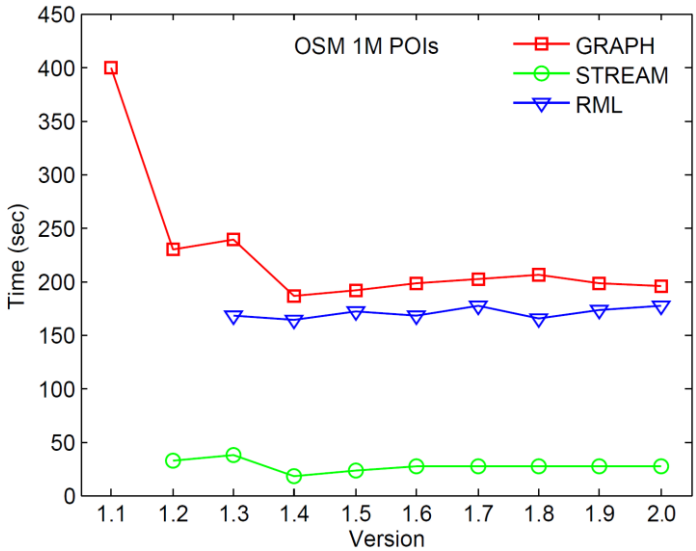


Figure 13: Performance of the various transformation modes as supported by each successive release of TripleGeo when dealing with *four attributes* per record

The first experiment compares performance of TripleGeo across its successive software releases, starting from ver.1.1 available in the beginning of the project until the latest ver.2.0. The plot in Figure 13 illustrates the time it takes in each release of TripleGeo to transform the sample dataset ***S*** (*OSM 1M POIs*), i.e., one million POIs in Europe available in a PostgreSQL/PostGIS repository. In order to provide a fair comparison regarding performance amongst versions, we only examine *four basic attributes* in each POI (first column

in Table 10). This restriction is imposed by the initial version 1.1, because this release (inherited from the GeoKnow project and used as a starting point in our development) could only support up to four attributes; by default, one of these attributes is the geometry and the rest are thematic ones. From the plot, it is obvious that the STREAM mode is extremely fast, almost an order of magnitude faster than the GRAPH mode originally available in ver.1.1. Note that RML is slower than STREAM because it must apply each one of the specified performers in the mapping against each input record, even in case of null values in the respective attributes. However, RML is faster than GRAPH, as it also works in a streaming fashion. Indeed, in either STREAM or RML each input record is transformed on-the-fly and readily propagated to the output, without the cost of building and updating a disk-based RDF model as in the GRAPH mode. However, even for the GRAPH mode, after ver.1.4 we have achieved to drop the cost by almost 50% by streamlining intermediate computations per record while also avoiding creation of a temporary set of triples before writing the results to the output file. Finally, notice the small increase in the cost for ver.1.3 compared with its predecessor ver.1.2. Indeed, ver.1.3 was the first release that included (partial) support for classification schemes, and practically emitted extra triples with string literals for the identified category. Since ver.1.4, TripleGeo maintains the user-specified classification scheme in memory; when transforming a record, it actually creates a link to the URI of its respective category. Not only is this conforming to the SLIPO ontology for POIs, but evidently it is more efficient in practice across all transformation modes. Also note that practically there is little change in performance amongst the three transformation modes after ver.1.4. Indeed, the overall architecture and functionality of the core modules had been established by then (M15 of the project) to facilitate transformation in standalone executions as fast as possible. Subsequent releases up to ver.2.0 have not essentially modified the core internal processing of the software, but only added extra capabilities and features with only marginal impact on performance.

5.3.1.2. Performance with Increasing Number of Attributes

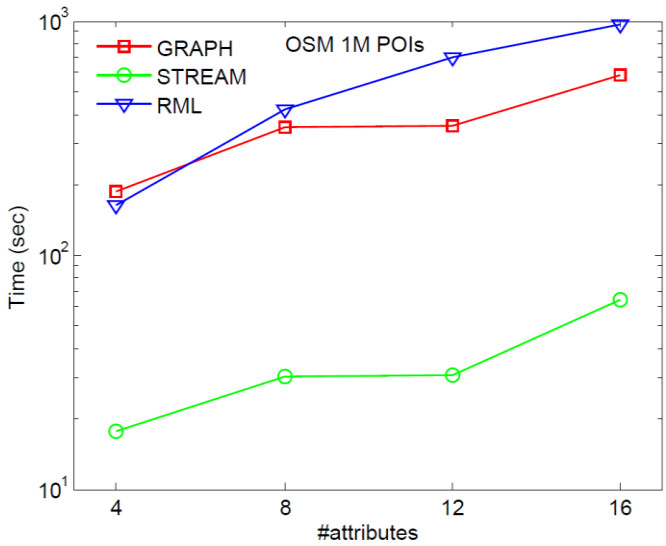


Figure 14: Performance of the various transformation modes in TripleGeo when handling an increasing number of attributes per input record

This experiment examines performance of TripleGeo ver.2.0 when dealing with increasing number of attributes per input record. Figure 14 depicts the overall execution time of transforming sample dataset *S* of one million POIs (*OSM 1M POIs*) to RDF, when each POI includes a varying number of attributes as listed in

Table 10. Observe that the scale along time measurements is logarithmic. Clearly, the more the attributes available per input record, the more the resulting RDF triples; hence, the cost should increase linearly with the number of attributes. This is exactly the case with the RML mode, since the RML performers employed in transformation must examine each attribute value irrespective of NULL values. But notice that cost for STREAM and GRAPH modes seem practically unaffected when dealing with 12 attributes instead of 8, i.e., when examined contact information per POI as well. This is because the majority of POIs in that dataset actually lacks most of this information and those extra four attributes have NULL values. Hence, YAML mappings are not applied at all in such cases and thus save processing cost. Of course, STREAM mode is at least an order of magnitude faster from GRAPH, and even more faster than RML for any number of attributes per input record. Even when all 16 attributes per record need be transformed, the process in STREAM mode takes less than 65 seconds to conclude, as opposed to 587 seconds in GRAPH and 964 seconds in RML mode. The only case where RML fares better than GRAPH is when only four (basic) attributes are examined. In that case, the cost for creating the disk-based model and then extracting the RDF triples is slightly greater (22.5 seconds higher) than applying the series of RML performers and collecting the triples for all input records. Overall, it is evident that TripleGeo can efficiently handle a varying number of thematic attributes per record with linear (worst-case) or sublinear (amortized) increase in transformation cost.

5.3.1.3. Performance with Increasing Data Volumes

The next set of experiments examine the performance of TripleGeo against increasing volumes of input records, up to one million POIs available across Europe (the sample dataset ***S***: *OSM 1M POIs*). In these tests, we employ TripleGeo in its STREAM mode, as this is manifestly the most efficient.

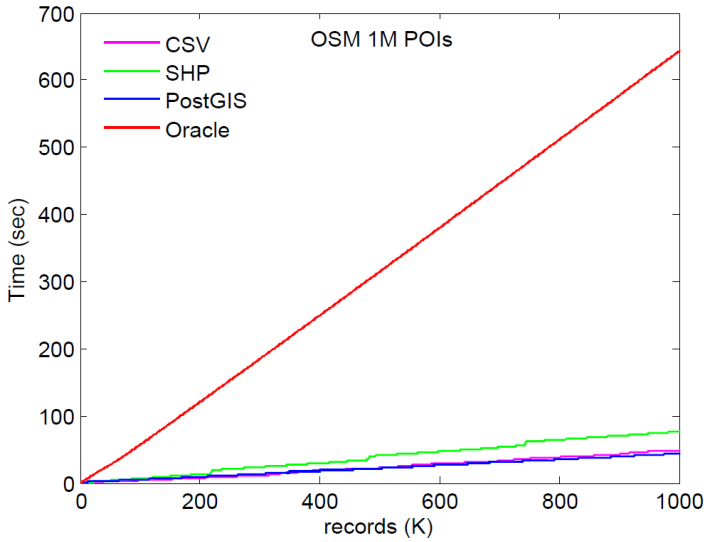


Figure 15: Performance of TripleGeo (in STREAM transformation mode) when handling an increasing number of input records from different data repositories

First, we measure execution times when accessing the same input data (all 16 attributes in Table 10) available from different repositories: either de facto file formats (CSV, ESRI shapefile) or a geospatially-aware DBMS (PostGIS, Oracle Spatial). From the plot in Figure 15, it is evident that performance grows linearly with the amount of input records retrieved from each repository. This is expectable, since the STREAM mode handles each record in isolation and promptly emits the resulting triples before proceeding to handle the next record

in the data. However, there is some divergence in performance amongst repositories. Indeed, when input data comes from CSV or PostGIS, transformation to RDF proceeds rapidly. This should be expected for a CSV file, as this is a plain text format and lines in the file can be consumed swiftly and then turned into records. But, the JDBC driver for PostGIS seems to be almost equally fast in providing records from the result set obtained after the SQL query is applied against the DBMS. In contrast, the respective JDBC for Oracle provides records at a much slower pace, hence the significant slowdown in performance. When input is available in ESRI shapefile format (SHP), then performance is also slower than CSV, because each record must first be parsed by the GeoTools library. This parsing almost doubles the cost w.r.t. CSV, before applying any transformation to RDF, hence the increase in the total execution time. Overall, this experiment indicates that the type of data repository plays an important role in the rate at which input data is accessed and has a strong impact in transformation cost.

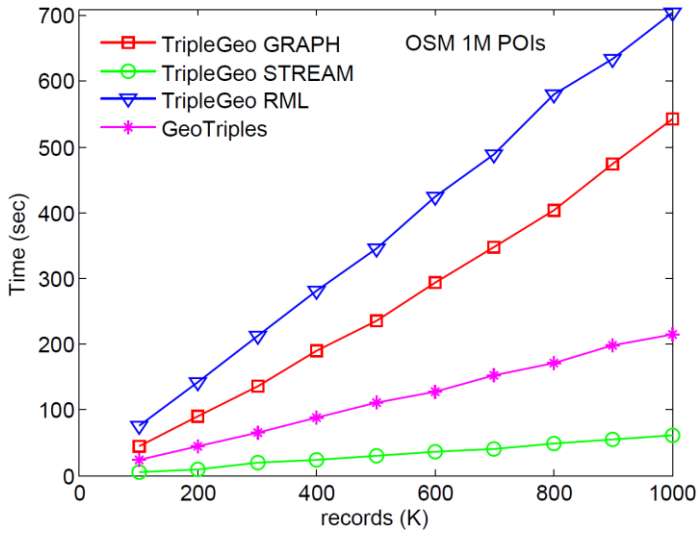


Figure 16: Performance of TripleGeo (in its various transformation modes) with GeoTriples when handling an increasing number of input records

The next test compares performance of TripleGeo in its various transformation modes and also against GeoTriples [KSV+18]. This concerns increasing volumes of input POIs in CSV format from the sample dataset *S*, i.e., up to one million OSM POIs involving all 16 attributes listed in Table 10. As Figure 16 indicates, the cost grows linearly with the input size in all three transformation modes. Although GRAPH involves a disk-based RDF repository that collects all triples before issuing any results, it is still better than the RML mode of TripleGeo for such moderate data sizes. This is due to cross-referencing among properties in the POI ontology (as illustrated in Figure 2), hence dependent RML mappings (e.g., for names or contacts) get triggered by their parent ones (i.e., for POIs). In effect, each input POI is examined against every defined RML mapping, which is unnecessary in case of missing attribute values. Instead, GeoTriples [KSV+18] optimizes the RML processor and extends it with inherent support for geometries, hence its better performance. Nevertheless, under its STREAM mode TripleGeo is always superior by far, as it handles each record in isolation and, thanks to its lightweight YAML mappings, it can promptly emit the resulting triples.

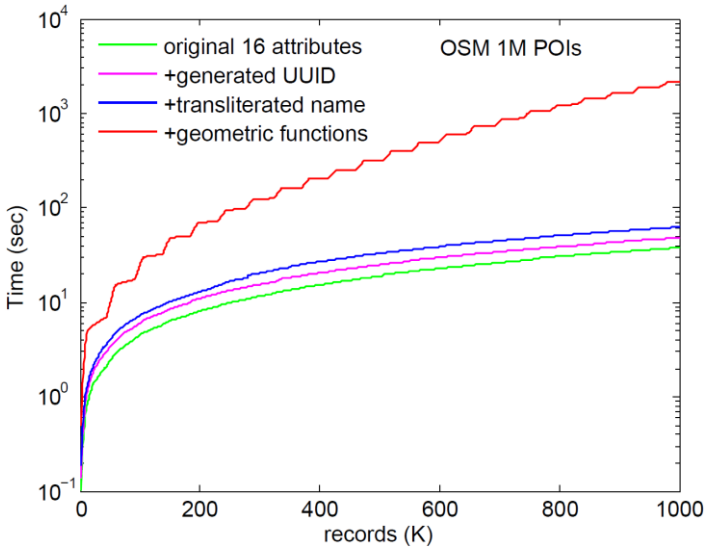


Figure 17: Performance of TripleGeo (in STREAM transformation mode) when required to dynamically generate extra properties on-the-fly against an increasing number of input records

The last test in this set of experiments concerns performance of TripleGeo when it has to generate extra properties on-the-fly according to *built-in functions* specified in the attribute mappings (in YAML format). As described in Section 4.4.3.2.2, such values may concern a wide range of features, such as transliteration of string literals from various alphabets into Latin, concatenation os string values, assignment of embedded categories used internally in SLIPO, etc. Another suite of these functions generates values related to geometry characteristics, e.g., calculation of area, perimeter, GeoHash encoding, etc. Finally, there is another set of functions employed for generation of UUIDs, as explained in Section 4.4.3.1. In this test, we measure the cumulative execution cost of transforming increasing amounts of POIs from the real dataset ***S*** (up to one million POIs in Europe) when TripleGeo has to handle the original 16 attributes listed in Table 10, and also generate several extra properties: (i) first, only *UUID* values to be used in the URIs; (ii) the properties in (i) and also *transliterate* POI names; and (iii) the extra properties in (i) and (ii), as well as *geomety-related values* concerning area and perimeter of polygons, length of linear features, lon/lat of their centroid, anda GeoHash string. The plot in Figure 17 illustrates (in logarithmic scale) the accumulated execution time for each test, compared against the cost of transforming original attributes only without use of any built-in functions. Clearly, the overhead of generating UUIDs and transliterating strings is very small. However, the impact of geometric functions is considerable and escalates when more and more POIs need to be handled. This step-wise increase should be attributed to certain large shapes (especially polygons, multipolygons, and complex geometries with hundreds or even thousands of vertices) included in this OSM dataset, as listed in Table 11. These detailed geometries incur significant overhead when they need be reprojected to calculate their centroid in lon/lat coordinates in WGS84, and particularly when reprojected to a flat CRS for calculating their length (perimeter) and area in square meters. However, including such extra properties in the transformed output is optional and depends on the use case at hand, since in most practical situations no such geometric calculations are involved, as our experience from the SLIPO use cases confirms [SLIPO-D5.2].

5.3.1.4. Scalability with Multiple Execution Threads

The next set of experiments are applied against the much larger real dataset ***D*** of OSM 7.4M POIs with the objective to examine performance of TripleGeo when multiple execution threads are employed in transformation. More specifically, the input dataset is split into several (up to 16) equal parts in CSV format, and a separate thread applies transformation to RDF under the same mode (GRAPH, STREAM, or RML).

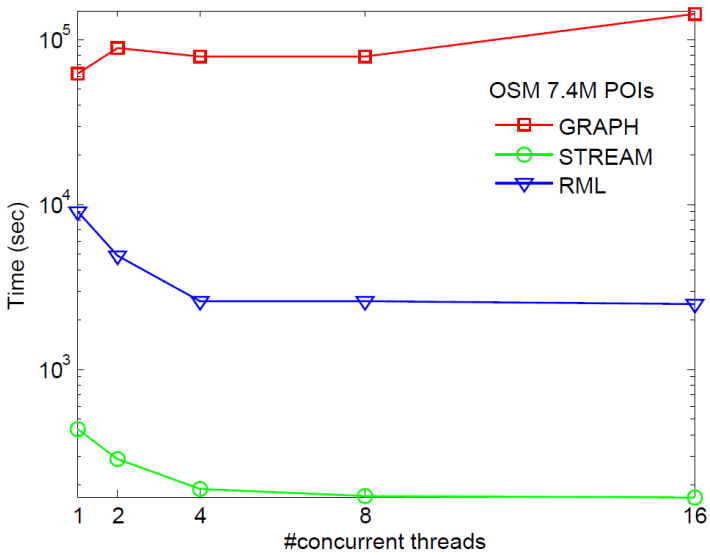


Figure 18: Performance of the various transformation modes in TripleGeo when employing a varying number of concurrent threads

Figure 18 illustrates the total execution time (shown in logarithmic scale) for transforming the entire dataset ***D*** with a varying number of *concurrent* threads as discussed in Section 4.4.4.1. Obviously, when a single thread is used, each mode deals with the entire dataset. In this case, we can observe that STREAM is an order of magnitude faster than RML (which also works in streaming fashion, but has to apply a series of RML performers against each record), and almost two orders of magnitude faster than GRAPH. The same pattern persists when multiple concurrent threads are used in each mode. It is no wonder that transformation cost drops with extra threads for STREAM and RML modes, as each thread handles separately a smaller chunk of the data and takes out the most of available system resources (memory, CPU). Of course, this performance gain gets less pronounced when invoking more than 8 threads, as the system cannot resourcefully sustain all of them concurrently and context switching inevitably ensues.

However, the cost for the GRAPH mode increases when multiple *concurrent* threads are involved as the plot in Figure 18 testifies. This is primarily due to the high I/O interaction of this mode, as each thread has to maintain its own disk-resident RDF model. In addition, RDF indices have to be constantly updated upon additions of new statements to a model, incurring extra CPU and I/O cost. The available memory is exhausted once these models grow bigger and the system cannot efficiently maintain all of them concurrently.

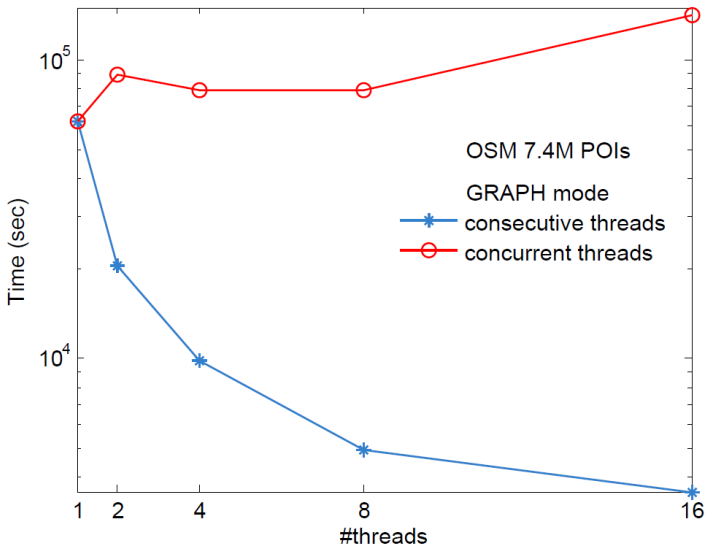


Figure 19: Performance of the GRAPH transformation mode in TripleGeo when using multiple, either consecutive or concurrent execution threads to process input

To verify this effect, we repeated the same test specifically in GRAPH mode when each thread is not invoked concurrently with the rest, but *sequentially*. More specifically, once a thread concludes transformation of its own part of the data, another one is invoked to handle another part. So, the same number of *consecutive* threads is applied, each one running alone without any impact from other concurrent threads of TripleGeo. The plot in Figure 19 confirms that using more threads against smaller chunks of data can boost performance even in GRAPH mode, provided that these threads are invoked in a sequence and not concurrently. Indicatively, transformation of the entire dataset with 16 *consecutive* threads concludes in less than 60 minutes, but when the same number of threads are invoked *concurrently* execution time soars to almost 24 hours (mind the log scale across time measurements)! Given sufficient system resources (CPU, memory), the smaller the input data chunk, the more efficient its loading to the disk-based model and the sooner its transformation to RDF. In contrast, executing those threads in parallel is counter-productive as it soon exhausts available resources and imposes frequent context switching. Even when executed with multiple consecutive threads, the GRAPH mode still trails behind the other two transformation modes (STREAM, RML) that run in a streaming fashion. This is inevitable, as these latter methods work entirely in main memory and only touch the disk for storing the resulting RDF triples; instead the GRAPH mode entails heavy interaction with the disk and this is the cause of delays. Of course, Jena allows memory-resident models to be used, but this cannot be applied even against input datasets of moderate size (less than a million records with our VM capabilities).

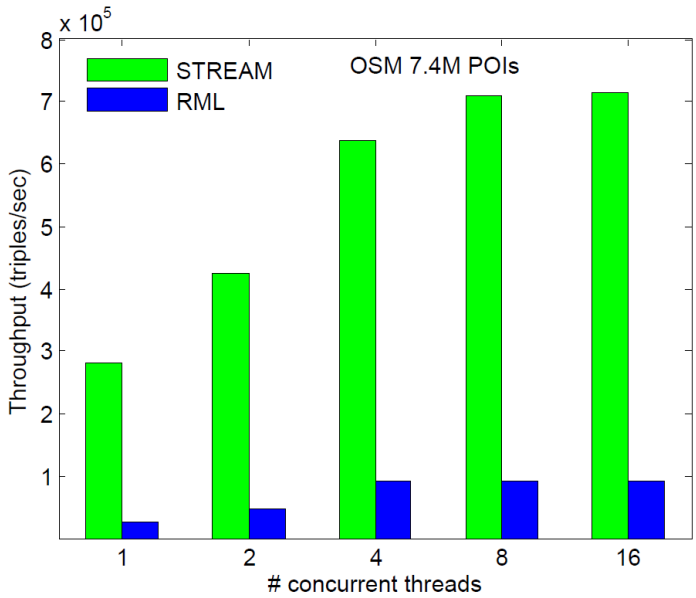


Figure 20: Throughput of the streaming transformation modes in TripleGeo when using multiple concurrent threads to process the input

Regarding especially the two streaming modes (STREAM, RML), Figure 20 depicts the average throughput, i.e., the rate (in triples/sec) at which RDF triples are generated with a varying number of concurrent threads. As expected, in both transformation modes throughput is increasing with extra threads, but the effect diminishes when reaching the ceiling of available system resources (i.e., when more than 8 threads are used). Again, the STREAM mode can generate up to 715,000 triples/sec and is much more efficient than the RML mode, as the former readily applies YAML mappings tailored for each attribute, whereas the latter has to iterate over all RML performers in order to identify the one with the RML mapping suitable for the given attribute value. It should be also noted that RML mode generates extra triples for RDF properties specified in the RML mappings (e.g., characterization of the name of a POI as official or brand name) even in presence of NULL values in the respective attributes. Although this has no implications to the correctness of the RDF output, it inflates its size with superfluous triples. In the future, we will attempt to alter the way RML performers handle such cases in order to avoid generation of such triples.

5.3.1.5. Scalability with Distributed Execution over Spark

This set of experiments examines the scalability of TripleGeo when running in STREAM mode on top of a Spark/GeoSpark cluster infrastructure, as discussed in Section 4.4.4.2.

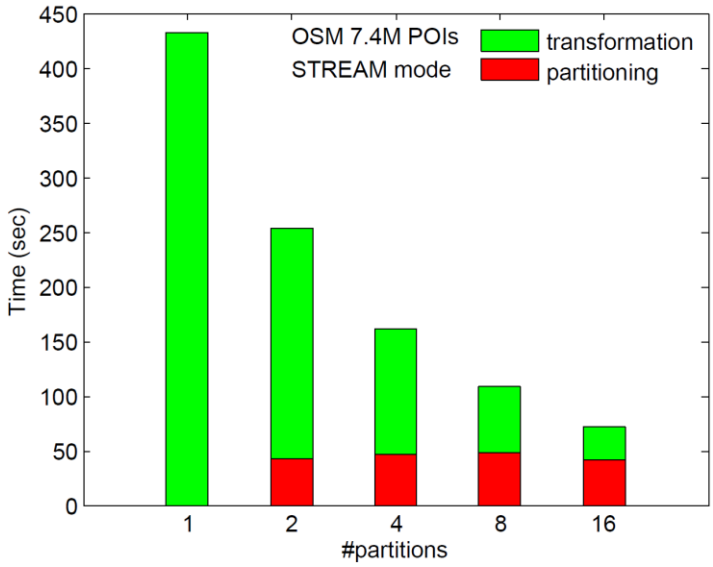


Figure 21: "Scale-up "of TripleGeo (in STREAM transformation mode) via data partitioning with a varying number of partitions

Figure 21 depicts execution time by fixing the data size ***D*** (7.4 million POIs in CSV format) and varying the number of its partitions. Each partition is assigned to a processing node for transformation. Clearly, the worst case is when no partitioning is involved, so a single node has to consume and transform the entire dataset ***D***. By doubling the number of partitions, the transformation cost drops by half, as each data chunk is processed separately and no shuffling is involved. Of course, partitioning incurs some overhead, but this depends on the input size and not on the number of partitions. This test clearly shows the "*scale-up*" efficiency of TripleGeo and its ability to handle large datasets in a distributed setting making the best utilization of the available system resources (HDFS, CPU cores).

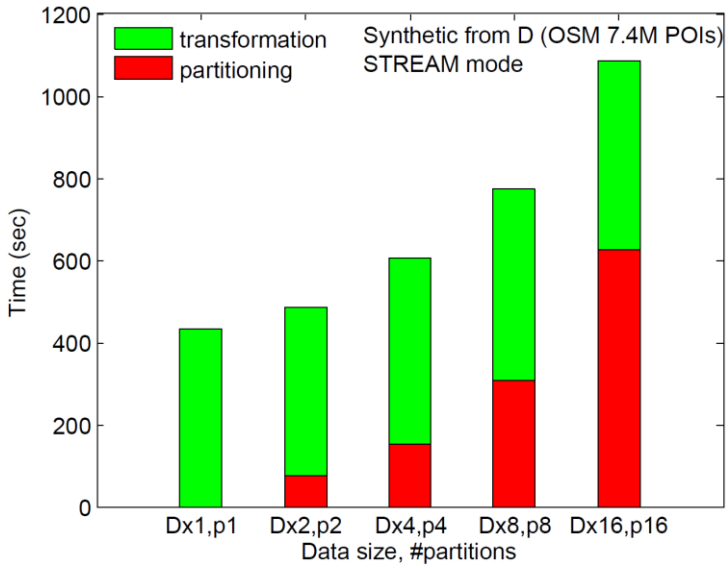


Figure 22: "Scale-out" of TripleGeo (in STREAM mode) against increasing data volumes and varying number of partitions

In the next experiment, we measure execution time by varying both the data size and the number of its partitions. More specifically, we synthetically increase the data size (multiples of original dataset ***D*** in CSV

format) and equally increase the number of partitions, so that their ratio is fixed in each test. As Figure 22 testifies, the partitioning cost escalates for growing data sizes, since it takes more time to split and distribute larger datasets. But afterwards, transformation tasks can run in parallel for each partition at their assigned worker nodes. As each data partition has the same size, their transformation ends up almost at the same time (we plot the maximum duration among transformation tasks in all nodes). Obviously, the impact of partitioning gets more pronounced for larger datasets, while transformation cost is practically stable for equi-sized data chunks. Note that the overall cost increases sublinearly with the data size, provided that more system resources are allocated. Indeed, processing dataset ***D*** × 16 (issuing over 2 billion RDF triples) can be carried out with only a ×2.5 increase in execution time compared to ***D***, underscoring the "*scale-out*" capabilities of TripleGeo.

5.3.2. Reverse Transformation

Remember that the reverse transformation (a.k.a. *export*) functionality of TripleGeo aims to reconstruct geospatial entities as records with thematic attributes from RDF datasets. Although performance of this module is not critical in the context of the SLIPO project because reverse transformation is not involved in the POI lifecycle, it is still important when the results of POI data integration (i.e., after interlinking, fusion, and enrichment) should be returned back to users in a de facto geospatial format. Of course, there exists an impedance mismatch in this reverse direction, given that the SLIPO ontology is semantically more expressive than the conventional POI schemata, thus POI attributes, relations and metadata will be richer than what can be supported by conventional formats. In this test, we applied reverse transformation to CSV or ESRI shapefile over the results received after transformation to RDF of the sample dataset ***S*** (OSM 1M POIs). As mentioned in Section 4.5.1, reverse transformation requires a SPARQL query against the RDF dataset, so we specify this query in order to reconstruct exactly the *same attributes* available in the original data. Apart from testing the capabilities of the reverse transformation module, this experiment can also verify that no information is lost when the original dataset is transformed into RDF by TripleGeo.

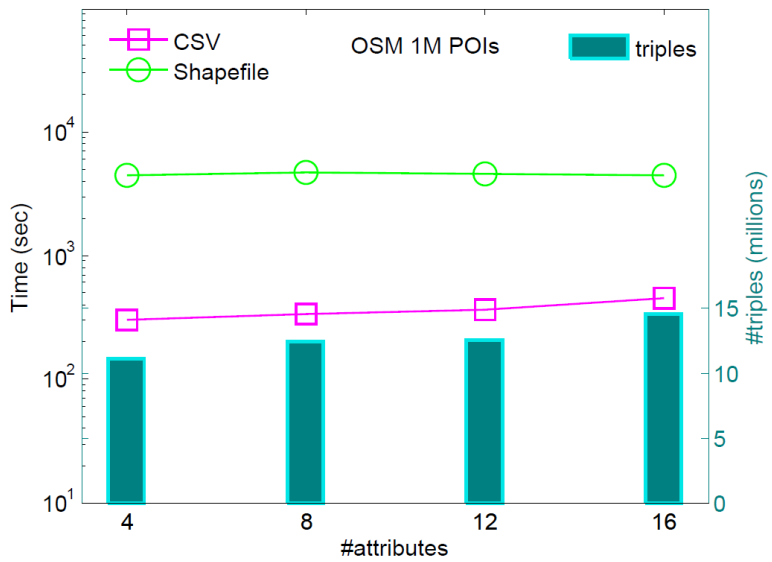


Figure 23: Performance of *reverse transformation* in TripleGeo for varying number of attributes per record

Figure 23 shows the time (in logarithmic scale along the left *y*-axis) taken by reverse transformation to reconstruct the data in CSV and ESRI shapefile formats when a varying number of original attributes had been extracted into RDF (as listed in Table 10). Note that, with an increasing number of extra attributes more RDF triples had been generated, as indicated with the bar plots. So, in reverse transformation, it takes more time to restore these triples into a disk-based RDF model and subsequently extract records by linking all available properties for each entity in the model. This is more evident when the result of reverse transformation is written into plain CSV format, which concludes very quickly. Even for 16 attributes, the restored CSV dataset is ready in almost 7.5 minutes, even though it has to get this information from more than 14.5 million triples. However, reverse transformation to shapefiles incur an order to magnitude more cost, up to 74 minutes for 16 attributes in this test. This is because the GeoTools library must first create a feature record and assign the respective attribute values (including the geometry) before storing this record into the shapefile. In addition, GeoTools performs tests regarding validity of geometries and their conformance to the geometry type, since shapefiles support only a specific type of geometries (i.e., either points or polylines or polygons, but not a mixture of them). Although such tests seem to incur a certain delay in the construction of shapefiles, this sometimes may be preferable as it guarantees consistency in the geospatial information that should not always taken for granted with WKT representations in CSV files. Finally, it should be noted that we have compared the reconstructed CSV files and shapefiles with the original data, confirming that we received the same number (one million) of records with no extra NULL values in any attribute. Overall, this experiment confirms that the reverse transformation functionality of TripleGeo can reconstruct original data with no information loss.

6. References

[AllegroGraph] Franz Inc. AllegroGraph Semantic Graph Database. https://franz.com/agraph/allegrograph/

[ADL+09] S. Auer, S. Dietzold, J. Lehmann, S. Hellmann, and D. Aumueller. Triplify: Light-weight linked data publication from relational databases. In Proceedings of the 18th International Conference on World Wide Web (WWW 2009), pp. 621-630, Madrid, Spain, April 2009.

[BK16] K. Bereta and M. Koubarakis. Ontop of Geospatial Databases. In Proceedings of the 15th International Semantic Web Conference (ISWC), pp. 37-52, Kobe, Japan, October 2016.

[BXK+16] K. Bereta, G. Xiao, M. Koubarakis, M. Hodrius, C. Bielski, and G. Zeug. Ontop-spatial: Geospatial Data Integration using GeoSPARQL-to-SQL Translation (*demo paper*). In Proceedings of the 15th International Semantic Web Conference (ISWC), Kobe, Japan, October 2016.

[ConverterToRdf] W3C. ConverterToRdf. https://www.w3.org/wiki/ConverterToRdf

[D2RServer] D2R Server: Accessing databases with SPARQL and as Linked Data. http://d2rq.org/d2r-server

[D2RQ] The D2RQ Mapping Language. http://d2rq.org/d2rq-language

[DKF+15] A. Dimou, D. Kontokostas, M. Freudenberg, R. Verborgh, J. Lehmann, E. Mannens, S. Hellmann, and R. Van de Walle. Assessing and Refining Mappings to RDF to Improve Dataset Quality. In Proceedings of the 14th International Semantic Web Conference (ISWC), pp. 133-149, Bethlehem, Pennsylvania, USA, 2015.

[DSC+14] A. Dimou, M. van der Sande, P. Colpaert, R. Verborgh, E. Mannens, and R. van de Walle. RML: A generic language for integrated RDF mappings of heterogeneous data. In Proceedings of the 7th Workshop on Linked Data on the Web (LDOW), Seoul, Korea, April 2014.

[CKAN] The CKAN data portal platform. Available online at: https://ckan.org/

[DBpedia] DBpedia. http://dbpedia.org

[DFMZ18] R. Domingues, M. Filippone, P. Michiardi, J. Zouaoui. A comparative evaluation of outlier detection algorithms: Experiments and analyses. Pattern Recognition 74: 406-421 (2018)

[DLS14] T. Dasu, J. M. Loh, D. Srivastava. Empirical glitch explanations. KDD 2014: 572-581

[EPSG] European Petroleum Survey Group Geodetic Parameter Registry. http://www.epsg-registry.org/

[ESRIGeoDB] ESRI Inc. Personal GeoDatabases. http://desktop.arcgis.com/en/arcmap/latest/manage-data/administer-file-gdbs/personal-geodatabases.htm

[ESRIshp] ESRI Inc. Shapefile Technical Description. http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf

[FMEworkbench] ESRI Inc. FME Workbench for ArcGIS Data Interoperability extension. http://www.esri.com/software/arcgis/extensions/datainteroperability/key-features/spatial-etl

[GDAL] GDAL/OGR - Geospatial Data Abstraction Library. http://www.gdal.org/

[GeoJSON] GeoJSON specification. http://geojson.org/

[GeoKettle] Spatialytics.ordg. GeoKettle Spatial ETL tool. http://www.spatialytics.org/projects/geokettle/

[GeoKnow] GeoKnow EU/FP7 project. http://geoknow.eu/

[GeoKnow-D2.2] GeoKnow EU/FP7 project. D2.2.1. Integration of Geospatial Databases. http://svn.aksw.org/projects/GeoKnow/Public/D2.2.1_Integration_of_Geospatial_Databases.pdf

[GeoNames] GeoNames geographical database. http://www.geonames.org/

[GeoRDF] GeoRDF Profile. http://www.w3.org/wiki/GeoRDF

[GeoSpark] GeoSpark. http://geospark.datasyslab.org/

[GeoTools] GeoTools library. Available from http://www.geotools.org/

[geo2rdf] geometry2rdf Utility. http://mayor2.dia.fi.upm.es/oeg-upm/index.php/en/technologies/151-geometry2rdf. Source code: https://github.com/boricles/geometry2rdf/tree/master/Geometry2RDF

[GeoPos84] Basic Geo (WGS84 lat/long) Vocabulary. http://www.w3.org/2003/01/geo/

[GPL3] GNU General Public License, version 3.0. http://www.gnu.org/licenses/gpl-3.0.html

[HABF14] F. Hamdi, N. Abadie, B. Bucher, and A. Feliachi. GeomRDF: A Geodata Converter with a Fine-Grained Structured Representation of Geometry in the Web. In Proceedings of the 2nd International Workshop on Geospatial Lined Data (GeoLD), 2014.

[HSTORE] PostgreSQL hstore. https://www.postgresql.org/docs/current/static/hstore.html

[IBM-DB2] IBM DB2 Spatial Extender. http://www.ibm.com/software/products/us/en/db2spaext/

[INSPIRE] INSPIRE Knowledge Base. https://inspire.ec.europa.eu/

[INSPIRE-Themes] INSPIRE Knowledge Base. Data Specifications - Themes. http://inspire.ec.europa.eu/Themes/Data-Specifications/2892

[Jena] Apache Jena project. https://jena.apache.org/

[JenaDoc] Jena RDF Model documentation. https://jena.apache.org/documentation/javadoc/jena/org/apache/jena/rdf/model/Model.html

[JenaStream] RDF Streams in Apache Jena. https://jena.apache.org/documentation/io/streaming-io.html

[JTS] JTS Topology Suite. https://github.com/locationtech/jts

[KVS+14] K. Kyzirakos, I. Vlachopoulos, D. Savva, S. Manegold and M. Koubarakis. GeoTriples: a Tool for Publishing Geospatial Data as RDF Graphs Using R2RML Mappings. In Proceedings of the 6th International Workshop on the Foundations, Technologies and Applications of the Geospatial Web (*Terra Cognita*), Riva del Garda, Trentino, Italy, October 2014.

[KSV+18] K. Kyzirakos, D. Savva, I. Vlachopoulos, A. Vasileiou, N. Karalis, M. Koubarakis, and S. Manegold. GeoTriples: Transforming Geospatial Data into RDF Graphs Using R2RML and RML Mappings. Journal of Web Semantics, 52-53: 16–32, 2018.

[LGD] LinkedGeoData project: Adding a spatial dimension to the Web of Data. http://linkedgeodata.org

[JSON] JavaScript Object Notation. http://www.json.org/

[MAVEN] Apache Maven. http://maven.apache.org/

[MSAccess] Microsoft Inc. Access DBMS. https://products.office.com/access

[MSSQLServer] Microsoft Inc. SQL Server. https://www.microsoft.com/sql-server

[mySQL] MySQL Database. Available from http://www.mysql.com/

[NeoGeo] NeoGeo Vocabulary Specification. http://geovocab.org/doc/neogeo

[ODM] OpenDataMonitor. Available online at: https://opendatamonitor.eu/

[OGC07] Open Geospatial Consortium Inc. OpenGIS Geography Markup Language (GML) Encoding Standard. Version 3.2.1, 27/08/2007. http://portal.opengeospatial.org/files/?artifact_id=20509

[OGC10] Open Geospatial Consortium Inc. OpenGIS Implementation Specification for Geographic information - Simple feature access- Part 2: SQL option. OpenGIS Implementation Standard. Version 1.2.1, 04/08/2010. http://portal.opengeospatial.org/files/?artifact_id=25354

[OGC11] Open Geospatial Consortium Inc. OpenGIS Implementation Standard for Geographic information - Simple feature access - Part 1: Common architecture. Version: 1.2.1, 28/05/2011. http://portal.opengeospatial.org/files/?artifact_id=25355

[OGC12] Open Geospatial Consortium Inc. OGC GeoSPARQL standard - A geographic query language for RDF data. Version 1.0, 27/04/2012. https://portal.opengeospatial.org/files/?artifact_id=47664

[OGC-POI] OGC Points of Interest SWG. http://www.opengeospatial.org/projects/groups/poiswg

[Oracle] Oracle Inc. Oracle Spatial and Graph. http://www.oracle.com/technetwork/database-options/spatialandgraph/overview/

[OracleRDFViews] Oracle Inc. RDF Views: Relational Data as RDF. https://docs.oracle.com/en/database/oracle/oracle-database/12.2/rdfrm/rdf-views.html

[OSM] OpenStreetMap project. http://www.openstreetmap.org/

[OSMOnto] OpenStreetMap Map Features. https://wiki.openstreetmap.org/wiki/Map_Features

[OsmPoisPbf] OsmPoisPbf tool. https://github.com/MorbZ/OsmPoisPbf

[Parliament] Parliament High-Performance Triple Store. http://parliament.semwebcentral.org/

[PSM+19] K. Patroumpas, D. Skoutas, G. Mandilaras, G. Giannopoulos, and S. Athanasiou. Exposing Points of Interest as Linked Geospatial Data. In Proceedings of the 16th International Symposium on Spatial and Temporal Databases (SSTD), pp. 21-30, Vienna, Austria, August 2019.

[Place] Place ontology. https://schema.org/Place

[PostGIS] PostGIS - Spatial and Geographic objects for PostgreSQL. http://postgis.net/

[PostgreSQL] PostgreSQL DBMS. http://www.postgresql.org/

[PROV-O] PROV-O: The PROV Ontology. https://www.w3.org/TR/2013/REC-prov-o-20130430/

[RR15] M. Rodriguez-Muro and M. Rezk. Efficient SPARQL-to-SQL with R2RML mappings. Journal of Web Semantics, 33: 141-169, 2015.

[SANSA] SANSA Semantic Analytics Stack. https://github.com/SANSA-Stack

[Spark] Apache Spark. https://spark.apache.org/

[TripleGeo] Athena R.C. TripleGeo. Java source code: https://github.com/SLIPO-EU/TripleGeo

[R2RML] W3C. R2RML: RDB to RDF Mapping Language. https://www.w3.org/TR/r2rml/

[RDFDirect] W3C. A Direct Mapping of Relational Data to RDF. https://www.w3.org/TR/rdb-direct-mapping/

[SDI4Apps] SDI4Apps project. http://sdi4apps.eu/

[SLIPO-D1.1] SLIPO Project. Deliverable D1.1: Use Cases and Requirements.

[SLIPO-D1.4] SLIPO Project. Deliverable D1.4: Final SLIPO System.

[SLIPO-D2.1] SLIPO Project. Deliverable D2.1: POI Models and Identifiers Registry.

[SLIPO-D2.2] SLIPO Project. Deliverable D2.2: Mapping specification and POI transformation service.

[SLIPO-D3.7] SLIPO Project. Deliverable D3.7: Final release of the POI Integration Software.

[SLIPO-D5.1] SLIPO Project. Deliverable D5.1: Specification of Pilots.

[SLIPO-D5.2] SLIPO Project. Deliverable D5.2: Pilot Report.

[Sponger] OpenLink Software. Virtuoso Sponger. http://vos.openlinksw.com/owiki/wiki/VOS/VirtSponger

[SPRQLF] Sparqlify. https://github.com/AKSW/Sparqlify

[SpatiaLite] SpatiaLite. https://www.gaia-gis.it/fossil/libspatialite/index

[UHAS12] J. Unbehauen, S. Hellmann, S. Auer, and C. Stadler. Knowledge Extraction from Structured Sources. Search Computing, pp. 34-52, Springer LNCS vol. 7538, 2012.

[Virtuoso] OpenLink Software. Virtuoso RDF Triple Store. https://virtuoso.openlinksw.com/rdf/

[VirtGeoRDF] Virtuoso RDF and Geometry. http://docs.openlinksw.com/virtuoso/rdfsparqlgeospat/

[VVS+10] L.M. Vilches-Blázquez, B. Villazón-Terrazas, V. Saquicela, A. de León, O. Corcho, and A. Gómez-Pérez. GeoLinked Data and INSPIRE through an Application Case. In Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM SIGSPATIAL GIS 2010), pp. 446-449, San Jose, California, USA, November 2010.

[XPATH] XML Path Language. Available online at: https://www.w3.org/TR/1999/REC-xpath-19991116/

[YZS19] J. Yu, Z. Zhang, and M. Sarwat. Spatial Data Management in Apache Spark: the GeoSpark Perspective and Beyond. Geoinformatica, 23(1):37-78, January 2019.

7. Annex I

7.1. Cardinality restrictions in POI ontology

The following cardinality restrictions have been defined in the various classes of the SLIPO ontology for representing POI assets. Those restrictions highlighted in grey concern the extra properties defined through the course of the project.

Class: **AccuracyInfo**

- accuracyType exactly 1 xsd:string
- accuracyValue exactly 1 rdf:PlainLiteral

Class: **Address**

- addressType max 1 xsd:string
- concatenatedAddress max 1 rdf:PlainLiteral
- country max 1 rdf:PlainLiteral
- crossStreet max 1 rdf:PlainLiteral
- crossStreetNumber max 1 rdf:PlainLiteral
- linearReference max 1 xsd:positiveInteger
- locality max 1 rdf:PlainLiteral
- number max 1 rdf:PlainLiteral
- postCode max 1 xsd:string
- region max 1 rdf:PlainLiteral
- street max 1 rdf:PlainLiteral

Class: **Classification**

- classificationTitle exactly 1 rdf:PlainLiteral
- classificationURL max 1 xsd:anyURI

Class: **Contact**

- contactType max 1 xsd:string
- contactValue exactly 1 rdf:PlainLiteral

Class: **KeyValAttribute**

- attrKey exactly 1 xsd:string
- attrValue exactly 1 rdf:PlainLiteral

Class: **LicenseInfo**

- lcnAttribution max 1 rdf:PlainLiteral
- lcnTitle max 1 rdf:PlainLiteral
- lcnURL max 1 xsd:anyURI

Class: **Media**

- mediaCreationTime max 1 xsd:dateTime
- mediaMimeType max 1 xsd:string
- mediaURL exactly 1 xsd:anyURI

Class: **Name**

- nameAbbreviation max 1 xsd:string
- nameAcronym max 1 xsd:string
- namePhonetic max 1 xsd:string
- nameTransliteration max 1 xsd:string
- nameType max 1 xsd:string
- nameValue exactly 1 rdf:PlainLiteral

Class: **PaymentMethod**

- paymentMethodAvailable max 1 xsd:boolean
- paymentType exactly 1 xsd:string

Class: **POI**

- alt max 1 xsd:double
- area max 1 xsd:double
- assignedCategory max 1 xsd:string
- category min 1 Term
- homepage max 1 xsd:anyURI
- lastUpdated max 1 xsd:dateTime
- lat max 1 xsd:double
- lon max 1 xsd:double
- length max 1 xsd:double
- geoHash max 1 rdf:PlainLiteral
- operator max 1 rdf:PlainLiteral
- poiID exactly 1 xsd:string

- priceTier max 1 Rating
- rating max 1 Rating
- slipoURI max 1 xsd:anyURI
- source min 1 SourceInfo
- validFrom max 1 xsd:dateTime
- validTo max 1 xsd:dateTime
- geo:hasGeometry min 1 geo:Geometry

Class: **POISet**

- member min 1 POI
- psCreationTime max 1 xsd:dateTime
- psSource max 1 POISource
- psTitle max 1 xsd:string

Class: **POISource**

- srcDescription max 1 rdf:PlainLiteral
- srcHomepage max 1 xsd:anyURI
- srcLicense max 1 LicenseInfo
- srcLogo max 1 xsd:anyURI
- srcRating max 1 Rating
- srcTitle exactly 1 rdf:PlainLiteral
- srcURI exactly 1 xsd:anyURI

Class: **Rating**

- rtCount max 1 xsd:int
- rtLevel exactly 1 xsd:double
- rtMaxLevel exactly 1 xsd:double
- rtMinLevel exactly 1 xsd:double
- rtSource exactly 1 xsd:string

Class: **Service**

- serviceAvailable exactly 1 xsd:boolean
- serviceType exactly 1 xsd:string

Class: **SourceInfo**

- sourceRef exactly 1 xsd:anyURI

- poiRef exactly 1 xsd:anyURI
- dateRetrieved max 1 xsd:dateTime

Class: **Term**

- termClassification exactly 1 Classification
- termValue exactly 1 rdf:PlainLiteral

Class: **TimeSlot**

- endDayOfMonth max 1 xsd:gDay
- endDayOfWeek max 1 DayOfWeekEnum
- endMonth max 1 xsd:gMonth
- endTime max 1 xsd:time
- startDayOfMonth max 1 xsd:gDay
- startDayOfWeek max 1 DayOfWeekEnum
- startMonth max 1 xsd:gMonth
- startTime max 1 xsd:time
- tsConcatenated max 1 rdf:PlainLiteral
- tsIsOpen max 1 xsd:boolean

7.2. Triple Geo Configuration Settings

7.2.1. TripleGeo Configuration for Transformation

The following listing is an indicative configuration (`shp_options.conf`) for TripleGeo ver.2.0 in order to transform data from an ESRI shapefile into RDF triples. It can be applied with the following command, assuming that binaries are bundled together in `/target/triplegeo-2.0-SNAPSHOT.jar`:

```
java -cp ./target/triplegeo-2.0-SNAPSHOT.jar eu.slipo.athenarc.triplegeo.
Extractor ./test/conf/shp_options.conf
```

This configuration file contains the following properties (explanatory comments are given in green colour):

```
## OPTIONAL parameters regarding execution platform and on-the-fly data partitioning.

## Possible values: JVM (default) for single- or multi-threaded execution; or SPARK for distributed execution

runtime = JVM

partitions = 3

##Possible input formats: SHAPEFILE, DBMS, CSV, GPX, GEOJSON, XML, OSM_XML, OSM_PBF, JSON

inputFormat = SHAPEFILE
```

##Transformation mode: specify either 'GRAPH' (on disk) or 'STREAM' (in-memory) or 'RML' (for applying user-specified RML mappings)

```
mode = STREAM
```

##Paths to directories and files used by the application

CURRENTLY SUPPORTED: You can specify MULTIPLE input files (of exactly the same format and attributes) separating them by ';' in order to activate multiple concurrent threads for their transformation.

```
tmpDir = ./tmp
```

```
inputFiles = ./test/data/points.shp
```

```
outputDir = ./test/output
```

OPTIONAL parameter regarding the encoding (character set) for strings in the input data. If not specified, UTF-8 encoding is assumed.

```
encoding = UTF-8
```

Possible serialization formats for output: RDF/XML, RDF/XML-ABBREV, N-TRIPLES, TURTLE (or TTL), N3

```
serialization = N-TRIPLES
```

Specify the spatial ontology for geometries in the exported data.

Possible values: 1) GeoSPARQL, 2) Virtuoso (legacy RDF ontology for points only), 3) wgs84_pos (for WGS84 Geoposition RDF vocabulary)

```
targetGeoOntology = GeoSPARQL
```

##File (in TTL or YAML format) specifying mappings of the input attribute schema to RDF properties; i.e., prescribing how input features will be transformed into RDF triples (typically according to an ontology).

```
mappingSpec = ./test/conf/points_mappings.yml
```

File (either in CSV or YAML format) containing a classification hierarchy in categories assigned to input features. Classification is only applied if a suitable mapping (including a category attribute) has been specified above. Leave blank if non applicable.

```
classificationSpec = ./test/ classification/points_sample_classification.csv
```

Boolean specifying whether the data features specify their category based on its identifier in the classification scheme (false) or the actual name of the category (true).

```
classifyByName = true
```

Attribute parameters (CASE-sensitive for shapefiles!!!)

```
attrKey = osm_id
```

```
attrGeometry = the_geom
```

```
attrName = name
```

```
attrCategory = type
```

```
## OPTIONAL parameter for spatial filter to select input geometries contained within the specified polygon:
spatialExtent=POLYGON((-5.71 50.04, -5.48 50.04, -5.48 50.19,-5.71 50.19, -5.71 50.04))
## OPTIONAL parameter for thematic filter (logical expression) to select input features based on their (case-sensitive) values on specific attributes
## CAUTION! Numeric values must be quoted in expressions! Comparison operators: =, <>, <, <=, >, >=.
filterSQLCondition = type='hotel' OR type LIKE '%pub%'
## Specify whether a .CSV file will be also extracted for registering features in the SLIPO Registry.
registerFeatures = true
## MANDATORY parameter that specifies the data source provider of the input features
featureSource = OSM_sample_points
## OPTIONAL parameters regarding namespaces of generated URIs:
nsOntology = http://slipo.eu/def#
nsGeometry = http://www.opengis.net/ont/geosparql#
nsFeatureURI = http://slipo.eu/id/poi/
nsClassURI = http://slipo.eu/id/term/
nsClassificationURI = http://slipo.eu/id/classification/
nsDataSourceURI = http://slipo.eu/id/poisource/
## Specify two lists (of comma separated values) with the correspondence between a prefix
## and its respective namespace (mainly used in attribute mappings)
prefixes = slipo, geo, xsd, rdfs, wgs84_pos
namespaces = http://slipo.eu/def#, http://www.opengis.net/ont/geosparql#, http://www.w3.org/2001/XMLSchema#, http://www.w3.org/1999/02/22-rdf-syntax-ns#, http://www.w3.org/2003/01/geo/wgs84_pos#
## Spatial Reference parameters
## If not specified, geometries are assumed in WGS84 reference system (EPSG:4326).
sourceCRS = EPSG:2100
targetCRS = EPSG:4326
## OPTIONAL parameter. Default language tag for string literals created in the output RDF:
defaultLang = en
```

7.2.2. TripleGeo Configuration for Reverse Transformation

The following listing is an indicative configuration (`shp_reverse.conf`) for TripleGeo ver.2.0 in order to transform data from RDF triples back to records in an ESRI shapefile. It can be applied with the following command, assuming that binaries are bundled together in `/target/triplegeo-2.0-SNAPSHOT.jar`:

```
java -cp ./target/triplegeo-2.0-SNAPSHOT.jar eu.slipo.athenarc.triplegeo.
ReverseExtractor ./test/conf/shp_reverse.conf
```

This configuration file contains the following properties (explanatory comments are given in green colour):

```
##Possible output formats: SHAPEFILE, GEOJSON, CSV

outputFormat = SHAPEFILE

##Paths to directories and files used by the application

## You can specify MULTIPLE input RDF files (of the same serialization) separating them by ';'.

inputFiles = ./test/output/poi_classification.nt;./test/output/points.nt

outputFile= ./test/output/points_reconstructed.shp

sparqlFile= ./test/conf/points_query.sparql

tmpDir = ./tmp

## OPTIONAL parameter regarding the encoding (character set) for strings in the output data. If not specified, UTF-8 encoding is assumed.

encoding = UTF-8

## Possible serialization formats for input: RDF/XML, RDF/XML-ABBREV, N-TRIPLES, TURTLE (or TTL), N3

serialization = N-TRIPLES

## Spatial Reference parameters

## If not specified, geometries are assumed in WGS84 reference system (EPSG:4326).

sourceCRS = EPSG:4326

targetCRS = EPSG:2100

## OPTIONAL property. Default language for the string literals used in the input RDF

defaultLang = en
```

7.3. TripleGeo Sample Mappings

7.3.1. Sample RML Mapping

The following is a listing concerning an indicative RML Mapping (Section 4.4.3.2.1) for a POI dataset to the SLIPO ontology. For clarity, attribute names are shown in bold and constant values in italics, while

explanatory comments are given in green colour. Note that the URIs regarding classification to categories (CATEGORY_URI) are assigned on-the-fly according to a suitable classification scheme specified by the user.

```
# Prefixes
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix rml: <http://semweb.mmlab.be/ns/rml#>.
@prefix ql: <http://semweb.mmlab.be/ns/ql#>.
@prefix slipo: <http://slipo.eu/def#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix rdfs: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix geo: <http://www.opengis.net/ont/geosparql#>.
@prefix sf: <http://www.opengis.net/ont/sf#>.
@prefix wgs84_pos: <http://www.w3.org/2003/01/geo/wgs84_pos#>.
# Mapping for Classification Hierarchy
# Employing a custom hierarchy specified by the OsmPoisPbf utility.
<#POIClassification>
  rml:logicalSource [
    rml:source "" ;
    rml:referenceFormulation ql:CSV
  ];
  rr:subjectMap [
    rr:constant <http://slipo.eu/id/classification/OsmPoisPbf>;
    rr:class slipo:Classification;
  ];
  rr:predicateObjectMap [
    rr:predicate slipo:classificationURL;
    rr:objectMap [
      rr:constant
<https://github.com/MorbZ/OsmPoisPbf/blob/master/doc/poi_types.csv>;
      ]
    ];
    rr:predicateObjectMap [
    rr:predicate slipo:classificationTitle;
```

```
    rr:objectMap [
      rr:constant "Custom Classification for OpenStreetMap Points Of Interest";
        rr:language "en";
        rr:datatype xsd:string
      ]
    ].
<#POITerm>
  rml:logicalSource [
    rml:source "" ;
    rml:referenceFormulation ql:CSV
  ];
  rr:subjectMap [
    rr:template "http://slipo.eu/id/term/{CATEGORY_URI}";
    rr:class slipo:Term;
  ];
    rr:predicateObjectMap [
    rr:predicate slipo:termClassification;
    rr:objectMap [
      rr:constant <http://slipo.eu/id/classification/OsmPoisPbf>;
      ]
    ];
  rr:predicateObjectMap [
    rr:predicate slipo:value;
    rr:objectMap [
      rml:reference "CATEGORY_NAME";
        rr:language "en";
        rr:datatype xsd:string
      ]
    ];
  rr:predicateObjectMap [
    rr:predicate slipo:parent;
```

```
    rr:objectMap [
      rr:template "http://slipo.eu/id/term/{CATEGORY_PARENT}";
      rr:class slipo:Term;
      ]
    ].
# Main mapping for Points of Interest (POI)
# By convention, attribute names referring to input data must be specified in UPPER-CASE letters.
<#POIMapping>
  rml:logicalSource [
    rml:source "" ;
    rml:referenceFormulation ql:CSV
  ];
  rr:subjectMap [
    rr:template "http://slipo.eu/id/poi/{UUID}";
    rr:class slipo:POI;
    rr:class geo:Feature
  ];
    rr:predicateObjectMap [
    rr:predicate slipo:source;
    rr:objectMap [
      rr:parentTriplesMap <#POISource>
        ]
    ];
    rr:predicateObjectMap [
    rr:predicate slipo:name;
    rr:objectMap [
      rr:parentTriplesMap <#POIName>
        ]
    ];
   rr:predicateObjectMap [
    rr:predicate slipo:category;
```

```
  rr:objectMap [
    rr:template "http://slipo.eu/id/term/{CATEGORY_URI}";
    rr:class slipo:Term;
      ]
  ];
rr:predicateObjectMap [
  rr:predicate slipo:assignedCategory;
  rr:objectMap [
    rml:reference "ASSIGNED_CATEGORY";
    ]
  ];
rr:predicateObjectMap [
  rr:predicate slipo:lastUpdated;
  rr:objectMap [
    rml:reference "TIMESTAMP";
    rr:datatype xsd:dateTime
    ]
  ];
rr:predicateObjectMap [
  rr:predicate wgs84_pos:long;
  rr:objectMap [
    rml:reference "LON";
    rr:datatype xsd:float
    ]
  ];
rr:predicateObjectMap [
  rr:predicate wgs84_pos:lat;
  rr:objectMap [
    rml:reference "LAT";
    rr:datatype xsd:float
    ]
```

```
    ];
  rr:predicateObjectMap [
    rr:predicate geo:hasGeometry;
    rr:objectMap [
      rr:parentTriplesMap <#POIGeometry>
        ]
      ].
# Mapping of properties concerning POI data sources
<#POISource>
  rml:logicalSource [
    rml:source "" ;
    rml:referenceFormulation ql:CSV
  ];
  rr:subjectMap [
    rr:template "http://slipo.eu/id/poi/{UUID}/sourceInfo";
    rr:class slipo:sourceInfo;
  ];
  rr:predicateObjectMap [
    rr:predicate slipo:sourceRef;
    rr:objectMap [
      rr:constant "OpenStreetMap"
      ]
    ];
    rr:predicateObjectMap [
    rr:predicate slipo:poiRef;
    rr:objectMap [
      rml:reference "OSM_ID";
      rr:termType rr:Literal;
      ]
    ].
# Mapping of properties concerning POI names
```

```
<#POIName>
  rml:logicalSource [
    rml:source "" ;
    rml:referenceFormulation ql:CSV
  ];
  rr:subjectMap [
    rr:template "http://slipo.eu/id/poi/{UUID}/name";
    rr:class slipo:Name
  ];
  rr:predicateObjectMap [
    rr:predicate slipo:nameLang;
    rr:objectMap [
      rr:constant "en"
      ]
    ];
  rr:predicateObjectMap [
    rr:predicate slipo:nameValue;
    rr:objectMap [
      rml:reference "NAME";
      rr:language "en";
      rr:termType rr:Literal
      ]
    ].
# Mapping of properties concerning POI geometries
<#POIGeometry>
  rml:logicalSource [
    rml:source "" ;
    rml:referenceFormulation ql:CSV
  ];
  rr:subjectMap [
```

```
    rr:template "http://slipo.eu/id/poi/{UUID}/geometry";
    rr:class sf:Geometry
  ];
  rr:predicateObjectMap [
    rr:predicate geo:asWKT;
    rr:objectMap [
      rml:reference "WKT";
      rr:datatype geo:wktLiteral
      ]
    ].
```

7.3.2. Sample YAML Mapping

The following is a listing concerning an indicative YAML Mapping (Section 4.4.3.2.2) for a POI dataset to the SLIPO ontology. Note that attributes concerning geometries need not be specified in this mapping, as they are either constructed or recognized on-the-fly when TripleGeo works in GRAPH or STREAM transformation modes. For clarity, attribute names are shown in bold. Also note that the URIs regarding classification to categories (`CATEGORY_URI`) and assignment to the SLIPO catagories (`ASSIGNED_CATEGORY`) are handled on-the-fly according to a suitable classification scheme specified by the user. Usage of various *built-in functions* to generate values on-the-fly is denoted in italics and the resulting (generated) attributes are written in uppercase characters. Comments are shown in green colour and begin with #.

```
URI:
  entity: uri
  generateWith: getUUID(DATA_SOURCE,osm_id)    ##Function call to generate a UUID
                                                ##by hashing the name of the data
                                                ##source and the identifier.
osm_id:
  partOf: sourceInfo
  entity: source
  predicate: slipo:poiRef
DATA_SOURCE:
  partOf: sourceInfo
  entity: source
```

```
  predicate: slipo:sourceRef
  generateWith: getDataSource        ##Function call to assign the data source
                                     ##provider of each POI.
CATEGORY_URI:                        ##Specifies the URI for the category
  entity: category                   ##originally describing each POI.
  predicate: slipo:category
  datatype: uri
ASSIGNED_CATEGORY:
  entity: assignedCategory
  predicate: slipo:assignedCategory
  generateWith: getEmbeddedCategory    ##Function call to assign a SLIPO category
                                       ##according to a default classification.
name:
  instanceOf: name
  entity: name
  predicate: slipo:name
  type: official
  language: en
timestamp:
  entity: lastUpdated
  predicate: slipo:lastUpdated
  datatype: timestamp
GEOHASH:
  entity: geohash
  predicate: slipo:geoHash
  datatype: string
  generateWith: geometry.getGeoHash(10)    ##Function call to generate this value
                                           ##on-the-fly using a 10-length string.
LONGITUDE:
  entity: lon
  predicate: wgs84_pos:long
```

```
  datatype: float
  generateWith: geometry.getLongitude      ##Function call to generate this value
                                           ##on-the-fly from the original geometry
LATITUDE:
  entity: lat
  predicate: wgs84_pos:lat
  datatype: float
  generateWith: geometry.getLatitude       ##Function call to generate this value
                                           ##on-the-fly from the original geometry
TRANSLIT:
  instanceOf: name
  entity: translit_name
  predicate: slipo:name
  type: transliterated
  language: en
  generateWith: getTransliteration(name)    ##Function call to generate names
                                            ##transliterated to Latin characters
```

7.4. TripleGeo Sample Classification Schemes

7.4.1. Classification Hierarchy in CSV format

The following listing is an indicative classification scheme in CSV format with two levels (*category, subcategory*) applicable against POI data extracted from OpenStreetMap (OSM). In such a CSV file, each line (record) represents a subcategory, also specifying its respective (parent) category. For each named category or subcategory, their corresponding identifier is also given. In this CSV file, comma is used as the delimiter character between attributes, whereas string values are enclosed in double quotes (e.g., `"HOTEL"`). The header of the CSV file (listed below in bold) contains the names of the corresponding attributes.

```
"category_id","category","subcategory_id","subcategory"
"A","ACCOMMODATION",1,"ALPINEHUT"
"A","ACCOMMODATION",2,"CAMPING"
"A","ACCOMMODATION",3,"CARAVAN"
```

```
"A","ACCOMMODATION",4,"CHALET"
"A","ACCOMMODATION",5,"GUESTHOUSE"
"A","ACCOMMODATION",6,"HOSTEL"
"A","ACCOMMODATION",7,"HOTEL"
"A","ACCOMMODATION",8,"MOTEL"
"B","BUSINESS",9,"ADVERTISING"
"B","BUSINESS",10,"ATM"
"B","BUSINESS",11,"AUDIOVISUAL"
"B","BUSINESS",12,"BANK"
"B","BUSINESS",13,"CONSTRUCTION"
"B","BUSINESS",14,"CURRENCY"
"B","BUSINESS",15,"EMPLOYMENT"
"B","BUSINESS",16,"ENERGY"
"B","BUSINESS",17,"FINANCIAL"
"B","BUSINESS",18,"INSURANCE"
"B","BUSINESS",19,"IT"
"B","BUSINESS",20,"LEGAL"
"B","BUSINESS",21,"PRESS"
"B","BUSINESS",22,"REALESTATE"
"B","BUSINESS",23,"TELECOM"
"B","BUSINESS",24,"TRAVELAGENCY"
"C","AUTOMOTIVE",25,"CARWASH"
"C","AUTOMOTIVE",26,"CHARGINGSTATION"
"C","AUTOMOTIVE",27,"FUEL"
"C","AUTOMOTIVE",28,"PARKING"
"C","AUTOMOTIVE",29,"RENTALCAR"
"C","AUTOMOTIVE",30,"RESTAREA"
"C","AUTOMOTIVE",31,"TOLLGATE"
"D","EAT/DRINK",32,"BAR"
"D","EAT/DRINK",33,"BIERGARTEN"
"D","EAT/DRINK",34,"CAFE"
```

```
"D","EAT/DRINK",35,"FASTFOOD"
"D","EAT/DRINK",36,"ICECREAM"
"D","EAT/DRINK",37,"INTERNETCAFE"
"D","EAT/DRINK",38,"PUB"
"D","EAT/DRINK",39,"RESTAURANT"
"E","EDUCATION",40,"COLLEGE"
"E","EDUCATION",41,"NURSERY"
"E","EDUCATION",42,"SCHOOL"
"E","EDUCATION",43,"UNIVERSITY"
"F","SPORT",44,"ARCHERY"
"F","SPORT",45,"BASEBALL"
"F","SPORT",46,"BASKETBALL"
"F","SPORT",47,"BOWLING"
"F","SPORT",48,"CANOE"
"F","SPORT",49,"CLIMBING"
"F","SPORT",50,"CRICKET"
"F","SPORT",51,"DIVING"
"F","SPORT",52,"FOOTBALL"
"F","SPORT",53,"GOLF"
"F","SPORT",54,"GYM"
"F","SPORT",55,"HANDBALL"
"F","SPORT",56,"HOCKEY"
"F","SPORT",57,"HORSE"
"F","SPORT",58,"ICESKATING"
"F","SPORT",59,"LEISURECENTER"
"F","SPORT",60,"MARINA"
"F","SPORT",61,"MOTORRACING"
"F","SPORT",62,"PLAYGROUND"
"F","SPORT",63,"RUGBY"
"F","SPORT",64,"SHOOTING"
"F","SPORT",65,"SKATING"
```

```
"F","SPORT",66,"SKIING"
"F","SPORT",67,"SNOOKER"
"F","SPORT",68,"SOCCER"
"F","SPORT",69,"STADIUM"
"F","SPORT",70,"SURFING"
"F","SPORT",71,"SWIMMING"
"F","SPORT",72,"TENNIS"
"F","SPORT",73,"VOLLEYBALL"
"F","SPORT",74,"WATERSKI"
"G","PUBLICSERVICE",75,"ASSOCIATION"
"G","PUBLICSERVICE",76,"COURT"
"G","PUBLICSERVICE",77,"EDUCATIONAL"
"G","PUBLICSERVICE",78,"EMBASSY"
"G","PUBLICSERVICE",79,"FIRESTATION"
"G","PUBLICSERVICE",80,"FOUNDATION"
"G","PUBLICSERVICE",81,"GOVERNMENT"
"G","PUBLICSERVICE",82,"LIBRARY"
"G","PUBLICSERVICE",83,"NGO"
"G","PUBLICSERVICE",84,"POLICE"
"G","PUBLICSERVICE",85,"POSTOFFICE"
"G","PUBLICSERVICE",86,"PRISON"
"G","PUBLICSERVICE",87,"PUBLICBUILDING"
"G","PUBLICSERVICE",88,"SOCIALCARE"
"G","PUBLICSERVICE",89,"TOILETS"
"G","PUBLICSERVICE",90,"TOWNHALL"
"H","HEALTH",91,"CLINIC"
"H","HEALTH",92,"DENTIST"
"H","HEALTH",93,"DOCTORS"
"H","HEALTH",94,"EMERGENCYSTATION"
"H","HEALTH",95,"HOSPITAL"
"H","HEALTH",96,"PHARMACY"
```

```
"H","HEALTH",97,"VETERINARY"
"I","RELIGIOUS",98,"BAHAI"
"I","RELIGIOUS",99,"BUDDHIST"
"I","RELIGIOUS",100,"CHRISTIAN"
"I","RELIGIOUS",101,"HINDU"
"I","RELIGIOUS",102,"ISLAMIC"
"I","RELIGIOUS",103,"JAIN"
"I","RELIGIOUS",104,"JEWISH"
"I","RELIGIOUS",105,"PAGAN"
"I","RELIGIOUS",106,"SHINTO"
"I","RELIGIOUS",107,"SIKH"
"I","RELIGIOUS",108,"TAOIST"
"I","RELIGIOUS",109,"UNKNOWN"
"I","RELIGIOUS",110,"ZOROASTRIAN"
"J","SETTLEMENTS",111,"CITY"
"J","SETTLEMENTS",112,"HAMLET"
"J","SETTLEMENTS",113,"SUBURB"
"J","SETTLEMENTS",114,"TOWN"
"J","SETTLEMENTS",115,"VILLAGE"
"K","SHOP",116,"ALCOHOL"
"K","SHOP",117,"APPLIANCE"
"K","SHOP",118,"ART"
"K","SHOP",119,"BABYGOODS"
"K","SHOP",120,"BAKERY"
"K","SHOP",121,"BICYCLE"
"K","SHOP",122,"BOOK"
"K","SHOP",123,"BUTCHER"
"K","SHOP",124,"CAR"
"K","SHOP",125,"CARREPAIR"
"K","SHOP",126,"CLOTHES"
"K","SHOP",127,"COMPUTER"
```

```
"K","SHOP",128,"CONFECTIONERY"
"K","SHOP",129,"CONVENIENCE"
"K","SHOP",130,"COPYSHOP"
"K","SHOP",131,"COSMETICS"
"K","SHOP",132,"DEPARTMENTSTORE"
"K","SHOP",133,"DIY"
"K","SHOP",134,"ELECTRONICS"
"K","SHOP",135,"FISH"
"K","SHOP",136,"FLORIST"
"K","SHOP",137,"FURNITURE"
"K","SHOP",138,"GAMPLING"
"K","SHOP",139,"GARDENCENTRE"
"K","SHOP",140,"GIFT"
"K","SHOP",141,"GREENGROCER"
"K","SHOP",142,"GUN"
"K","SHOP",143,"HAIRDRESSER"
"K","SHOP",144,"HEARINGAIDS"
"K","SHOP",145,"JEWELRY"
"K","SHOP",146,"KIOSK"
"K","SHOP",147,"KITCHEN"
"K","SHOP",148,"LAUNDRETTE"
"K","SHOP",149,"MARKETPLACE"
"K","SHOP",150,"MASSAGE"
"K","SHOP",151,"MOTORCYCLE"
"K","SHOP",152,"MUSIC"
"K","SHOP",153,"NEWSPAPER"
"K","SHOP",154,"OPTICIAN"
"K","SHOP",155,"PAWNBROKER"
"K","SHOP",156,"PET"
"K","SHOP",157,"PHONE"
"K","SHOP",158,"PHOTO"
```

"K","SHOP",159,"RETAIL"

"K","SHOP",160,"SHOES"

"K","SHOP",161,"SPORTS"

"K","SHOP",162,"SUPERMARKET"

"K","SHOP",163,"TAILOR"

"K","SHOP",164,"TATTOO"

"K","SHOP",165,"TOBACCO"

"K","SHOP",166,"TOYS"

"K","SHOP",167,"VIDEORENTAL"

"K","SHOP",168,"WATCHES"

"L","LANDUSE",169,"CEMETERY"

"L","LANDUSE",170,"CONIFEROUSFOREST"

"L","LANDUSE",171,"DAM"

"L","LANDUSE",172,"DECIDUOUSFOREST"

"L","LANDUSE",173,"DUMP"

"L","LANDUSE",174,"FARM"

"L","LANDUSE",175,"FOREST"

"L","LANDUSE",176,"GRASS"

"L","LANDUSE",177,"INDUSTRIAL"

"L","LANDUSE",178,"MILITARY"

"L","LANDUSE",179,"MINESHAFT"

"L","LANDUSE",180,"NATURERESERVE"

"L","LANDUSE",181,"PARK"

"L","LANDUSE",182,"QUARRY"

"L","LANDUSE",183,"RIVERBANK"

"L","LANDUSE",184,"ROCK"

"L","LANDUSE",185,"SCRUB"

"L","LANDUSE",186,"SWAMP"

"L","LANDUSE",187,"WATER"

"M","TOURISM",188,"ARCHAELOGICAL"

"M","TOURISM",189,"ART"

"M","TOURISM",190,"ATTRACTION"
"M","TOURISM",191,"BATTLEFIELD"
"M","TOURISM",192,"BEACH"
"M","TOURISM",193,"BUNKER"
"M","TOURISM",194,"CASINO"
"M","TOURISM",195,"CASTLE"
"M","TOURISM",196,"CAVE"
"M","TOURISM",197,"CINEMA"
"M","TOURISM",198,"CLOCK"
"M","TOURISM",199,"EVENTVENUE"
"M","TOURISM",200,"FOUNTAIN"
"M","TOURISM",201,"GATE"
"M","TOURISM",202,"GLACIER"
"M","TOURISM",203,"INFORMATION"
"M","TOURISM",204,"LIGHTHOUSE"
"M","TOURISM",205,"MEMORIAL"
"M","TOURISM",206,"MONUMENT"
"M","TOURISM",207,"MUSEUM"
"M","TOURISM",208,"NIGHTCLUB"
"M","TOURISM",209,"PEAK"
"M","TOURISM",210,"PEDESTRIAN"
"M","TOURISM",211,"RUINS"
"M","TOURISM",212,"SHELTER"
"M","TOURISM",213,"THEATRE"
"M","TOURISM",214,"THEMEPARK"
"M","TOURISM",215,"TOWER"
"M","TOURISM",216,"TOWERCOMMUNICATION"
"M","TOURISM",217,"VOLCANO"
"M","TOURISM",218,"WATERMILL"
"M","TOURISM",219,"WATERTOWER"
"M","TOURISM",220,"WINDMILL"

```
"M","TOURISM",221,"WRECK"
"M","TOURISM",222,"ZOO"
"N","TRANSPORT",223,"AIRPORT"
"N","TRANSPORT",224,"AIRTERMINAL"
"N","TRANSPORT",225,"BUSSTOP"
"N","TRANSPORT",226,"FERRYTERMINAL"
"N","TRANSPORT",227,"STATION"
"N","TRANSPORT",228,"SUBWAY"
"N","TRANSPORT",229,"TRAMSTOP"
```

7.4.2. Classification Hierarchy in YAML Format

The aforementioned classification scheme for OSM POI data can be alternatively specified in YAML format. Such a user-prepared YAML file has indentations to denote breakdown of a given *category* (shown in bold) into *subcategories* (i.e., two blank characters in the beginning of a line at each extra level in the hierarchy). The identifier of each category (at any level) is specified after its name and it is preceded with a '#' character.

```
ACCOMMODATION #A
  ALPINEHUT #1
  CAMPING #2
  CARAVAN #3
  CHALET #4
  GUESTHOUSE #5
  HOSTEL #6
  HOTEL #7
  MOTEL #8
BUSINESS #B
  ADVERTISING #9
  ATM #10
  AUDIOVISUAL #11
  BANK #12
  CONSTRUCTION #13
  CURRENCY #14
  EMPLOYMENT #15
```

ENERGY #16

FINANCIAL #17

INSURANCE #18

IT #19

LEGAL #20

PRESS #21

REALESTATE #22

TELECOM #23

TRAVELAGENCY #24

AUTOMOTIVE #C

CARWASH #25

CHARGINGSTATION #26

FUEL #27

PARKING #28

RENTALCAR #29

RESTAREA #30

TOLLGATE #31

EAT/DRINK #D

BAR #32

BIERGARTEN #33

CAFE #34

FASTFOOD #35

ICECREAM #36

INTERNETCAFE #37

PUB #38

RESTAURANT #39

EDUCATION #E

COLLEGE #40

NURSERY #41

SCHOOL #42

UNIVERSITY #43

SPORT #F

ARCHERY #44

BASEBALL #45

BASKETBALL #46

BOWLING #47

CANOE #48

CLIMBING #49

CRICKET #50

DIVING #51

FOOTBALL #52

GOLF #53

GYM #54

HANDBALL #55

HOCKEY #56

HORSE #57

ICESKATING #58

LEISURECENTER #59

MARINA #60

MOTORRACING #61

PLAYGROUND #62

RUGBY #63

SHOOTING #64

SKATING #65

SKIING #66

SNOOKER #67

SOCCER #68

STADIUM #69

SURFING #70

SWIMMING #71

TENNIS #72

VOLLEYBALL #73

WATERSKI #74

PUBLICSERVICE #G

ASSOCIATION #75

COURT #76

EDUCATIONAL #77

EMBASSY #78

FIRESTATION #79

FOUNDATION #80

GOVERNMENT #81

LIBRARY #82

NGO #83

POLICE #84

POSTOFFICE #85

PRISON #86

PUBLICBUILDING #87

SOCIALCARE #88

TOILETS #89

TOWNHALL #90

HEALTH #H

CLINIC #91

DENTIST #92

DOCTORS #93

EMERGENCYSTATION #94

HOSPITAL #95

PHARMACY #96

VETERINARY #97

RELIGIOUS #I

BAHAI #98

BUDDHIST #99

CHRISTIAN #100

HINDU #101

ISLAMIC #102

JAIN #103

JEWISH #104

PAGAN #105

SHINTO #106

SIKH #107

TAOIST #108

UNKNOWN #109

ZOROASTRIAN #110

SETTLEMENTS #J

CITY #111

HAMLET #112

SUBURB #113

TOWN #114

VILLAGE #115

SHOP #K

ALCOHOL #116

APPLIANCE #117

ART #118

BABYGOODS #119

BAKERY #120

BICYCLE #121

BOOK #122

BUTCHER #123

CAR #124

CARREPAIR #125

CLOTHES #126

COMPUTER #127

CONFECTIONERY #128

CONVENIENCE #129

COPYSHOP #130

COSMETICS #131

DEPARTMENTSTORE #132

DIY #133

ELECTRONICS #134

FISH #135

FLORIST #136

FURNITURE #137

GAMPLING #138

GARDENCENTRE #139

GIFT #140

GREENGROCER #141

GUN #142

HAIRDRESSER #143

HEARINGAIDS #144

JEWELRY #145

KIOSK #146

KITCHEN #147

LAUNDRETTE #148

MARKETPLACE #149

MASSAGE #150

MOTORCYCLE #151

MUSIC #152

NEWSPAPER #153

OPTICIAN #154

PAWNBROKER #155

PET #156

PHONE #157

PHOTO #158

RETAIL #159

SHOES #160

SPORTS #161

SUPERMARKET #162

TAILOR #163

TATTOO #164

TOBACCO #165

TOYS #166

VIDEORENTAL #167

WATCHES #168

LANDUSE #L

CEMETERY #169

CONIFEROUSFOREST #170

DAM #171

DECIDUOUSFOREST #172

DUMP #173

FARM #174

FOREST #175

GRASS #176

INDUSTRIAL #177

MILITARY #178

MINESHAFT #179

NATURERESERVE #180

PARK #181

QUARRY #182

RIVERBANK #183

ROCK #184

SCRUB #185

SWAMP #186

WATER #187

TOURISM #M

ARCHAELOGICAL #188

ART #189

ATTRACTION #190

BATTLEFIELD #191

BEACH #192

BUNKER #193

CASINO #194

CASTLE #195

CAVE #196

CINEMA #197

CLOCK #198

EVENTVENUE #199

FOUNTAIN #200

GATE #201

GLACIER #202

INFORMATION #203

LIGHTHOUSE #204

MEMORIAL #205

MONUMENT #206

MUSEUM #207

NIGHTCLUB #208

PEAK #209

PEDESTRIAN #210

RUINS #211

SHELTER #212

THEATRE #213

THEMEPARK #214

TOWER #215

TOWERCOMMUNICATION #216

VOLCANO #217

WATERMILL #218

WATERTOWER #219

WINDMILL #220

WRECK #221

```
  ZOO #222
TRANSPORT #N
  AIRPORT #223
  AIRTERMINAL #224
  BUSSTOP #225
  FERRYTERMINAL #226
  STATION #227
  SUBWAY #228
  TRAMSTOP #229
```

7.5. Reverse Transformation

The following SPARQL query can be applied on an RDF graph regarding POI data extracted according to the SLIPO ontology. Depending on the properties contained in the RDF graph, extra conditions can be specified in the query in order to return more attributes. This example query shows how to reconstruct POI records with specific attributes (shown in bold) as declared in the SELECT clause:

```
PREFIX slipo: <http://slipo.eu/def#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX sf: <http://www.opengis.net/ont/sf#>
PREFIX wgs84: <http://www.w3.org/2003/01/geo/wgs84_pos#>
SELECT ?osm_id ?shape ?type ?name ?lon ?lat ?timestamp
WHERE {
            OPTIONAL {
                  ?uri slipo:source ?poiSourceInfo .
                  ?poiSourceInfo slipo:poiRef ?osm_id .
            }
            OPTIONAL {
                  ?uri slipo:category ?category_uri .
                  ?category_uri slipo:value ?type .
            }
            OPTIONAL {
                  ?uri geo:hasGeometry ?geometry .
```

```
        ?geometry geo:asWKT ?shape .
    }
    OPTIONAL {?uri wgs84:long ?lon . }
    OPTIONAL {?uri wgs84:lat ?lat . }
    OPTIONAL {
        ?uri slipo:name ?fName .
        ?fName slipo:nameType "official" .
        ?fName slipo:nameValue ?name .
    }
    OPTIONAL {
        ?uri slipo:lastUpdated ?timestamp .
    }
}
```

8. Annex II: PDH and PDE User Manual

In this annex, we provide a usage manual for the user of the PDH and PDE toolkits. First, we give the details about the minimum setup required for running the tools. Next, we examine the configuration of the tools before execution. Although data exploration has rather straightforward configuration, harvesting is a procedure that requires non-trivial input by the user. Finally, we take a look at the execution of both PDH and PDE, presenting results of specific examples.

8.1. Initial Setup

Both tools can be downloaded from the GitHub repository of the SLIPO project, along with a configuration file for the harvester to present its functionality. Although there is no installation procedure, some requirements must be met. These are mainly the programming languages used since the scripts are not compiled into executables. Therefore, one must install PHP in order to run PDH and python for PDE. The installation procedure varies depending on the operating system. Version 7 of PHP is recommended and python version at least 3.5.

PDH does not depend on any additional libraries, with the exception of PHP tidy. Tidy comes as an extension of PHP (the corresponding library for Windows is php_tidy.dll). All other libraries come as default with PHP installation. PDE, on the other hand, depends on some third-party tools. An easy way to install packages for python is through the package management system called pip. pip can be installed directly or through the anaconda software on Windows. With pip installed, the following command:

```
pip3 install <package_name>
```

can be issued from the command line (where <package_name> is meant to be replaced with the actual name of the package to be installed) in order to install a package for python 3. The packages required for data exploration are the following: pandas, numpy, re, collections, unicodedata and json.

8.2. Configuration

Both PDH and PDE come with a main executable script. In case of PDH, the corresponding script harvest takes just one argument, the path of the configuration file in JSON format. Details about this file are given in the next subsection. For the case of PDE, configuration is given directly during the call of the corresponding script explore, as it will be described in detail in the last subsection.

In all cases of harvesting, the configuration file should be in JSON format. In this file, one should determine the type of harvester to use, with possible values being one of web, list, api or ckan, the output file name and, optionally, the maximum number of simultaneous connections.

8.2.1. HTML Harvester Configuration

Below we give an example of the structure of the configuration file for PDH-HTML, and subsequently we examine each parameter individually. Before we proceed, it has to be clear that this configuration file is intended for HTML harvesting, as the first argument (type) implies.

Code snippet: Sample configuration for PDH-HTML

```
// Settings for web harvester
{
   "type": "web",
   // The requested url or path to local file to obtain the POI list (json)
   "url": "https://www.example.com/stores/action-get?type=restaurants",
   // The sequence of fields, separated by semicolon, under which the list is available
   "fields": "collection;stores",
   // The title of the link identifier
   "link_id": "slug",
   // The base url for the individual links for each POI
   "base_url": "https://www.example.com/restaurant",
   // The variable name of the pagination
   "page_id": "page",
   // The identifier, in case it exists, for the coordinates
   "coordinates_id": "coordinates",
   // The identifier for the name of the POI
   "name_id": "store_title",
   // The tags which will be searched inside the html
   "tags": {
            "style": "results-list-cuisine",
            "city": "::::Τηλέφωνο",
            "address": "::Τηλέφωνο",
            "phone_number": "Τηλέφωνο",
            "fax": "Fax:",
            "site": "Site",
            "facebook": "facebook",
```

```
            "price_range": "Τιμή",
            "opening_days": "Ανοιχτό",
            "facilities": "facilities",
            "rating": "RestaurantRating",
    },
    // The maximum number of simultaneous connections (default: 5)
    "max_connections": 5
}
```

List creation

First and most importantly, one has to provide the harvester with the URL of the Web site to be harvested. This is not necessarily the base URL of the site but rather the URL which provides the list with the hyperlinks of Web pages containing the desired data. In our example, this is the following:

```
https://www.example.com/stores/action-get?type=restaurants
```

This is supposed to return either an HTML page which contains the hyperlinks under discussion or a JSON response. For the sake of our discussion, we suppose that the hyperlinks correspond to pages about restaurants.

First, we take the case that under the above URL an API listens responding with a JSON upon our request. To be specific, suppose the following structure for the response:

Example: Sample response with list of restaurants

```
collection: {
        pagination: {...},
        stores: [
                0: {
                        slug: restaurant0,
                        store_title: Restaurant 0,
                        ...
                },
                1: {
                        slug: restaurant1,
                        store_title: Restaurant 1,
                        ...
```

```
                },
                ...
        ],
        ...
}
```

In this example, we also suppose that the URL of each restaurant is constructed if the value corresponding to the slug key is concatenated with a base URL. Therefore, what we would like to collect is exactly the values of these slugs. First, we have to define where the slug is located, that is under collection and subsequently under stores. This is the reason in configuration file given above, the value "collection;stores" have been given to "fields". The "link_id" is the actual link identifier key, which above has been defined as "slug". Each value of slug will be appended to the value of "base_url".

Let's suppose, now, that the HTML of the web page under this URL has, among other, the following content:

Example: Sample snippet of the HTML code of a Web page

```
<ul class="main-restaurants-list">
        <li class="main-restaurants-list-item">
                <a class="restaurants-item-link" href="/restaurant1"> Restaurant1 </a>
        </li>
        <li class="main-restaurants-list-item">
                <a class="restaurants-item-link" href="/restaurant2"> Restaurant2 </a>
        </li>
</ul>
<a itemprop="secondary-restaurants-item-link" href="/restaurant3"> Restaurant3 </a>
```

In this example, what we wish to collect is the href attribute of all <a> HTML tags that are referring to restaurants. Therefore, we have two identifiers for these tags, which in this case are represented by the class attribute, although in general they could be any characteristic item. These identifiers are provided with the fields key of the JSON configuration file:

```
"fields": "restaurants-item-link;secondary-restaurants-item-link"
```

However, it is not uncommon that several links have to be followed before we arrive on a page containing the actual URL links. In this case the "fields" parameter should be an array of values. For example, it could be:

Example: Configuration snippet

```
"fields": [
```

```
            "areas-list-item-link;areas-city-link",
            "url"
],
"link_id": "url"
```

With this configuration, the harvester first creates a list of links contained under the areas-list-item-link and areas-city-link attributes. Subsequently, it visits these links and creates another one with links contained under the url attribute. This last list is the final one, which is used for harvesting.

Another case is that a JSON file listing all the URLs is in our disposal. Then, there is no need to fetch the list. Instead, the JSON file can be directly supplied to the configuration file. One can provide the path to a JSON file instead of a URL in the url entry of harvest.conf. The structure of the JSON file can be described as usual defining the appropriate values for the fields link_id and base_url.

In all cases, it is quite common that the results are paginated. One should provide the page identifier page_id to cover this possibility. Once such a value is provided to the harvester, the results will be iterated over the values found in this attribute.

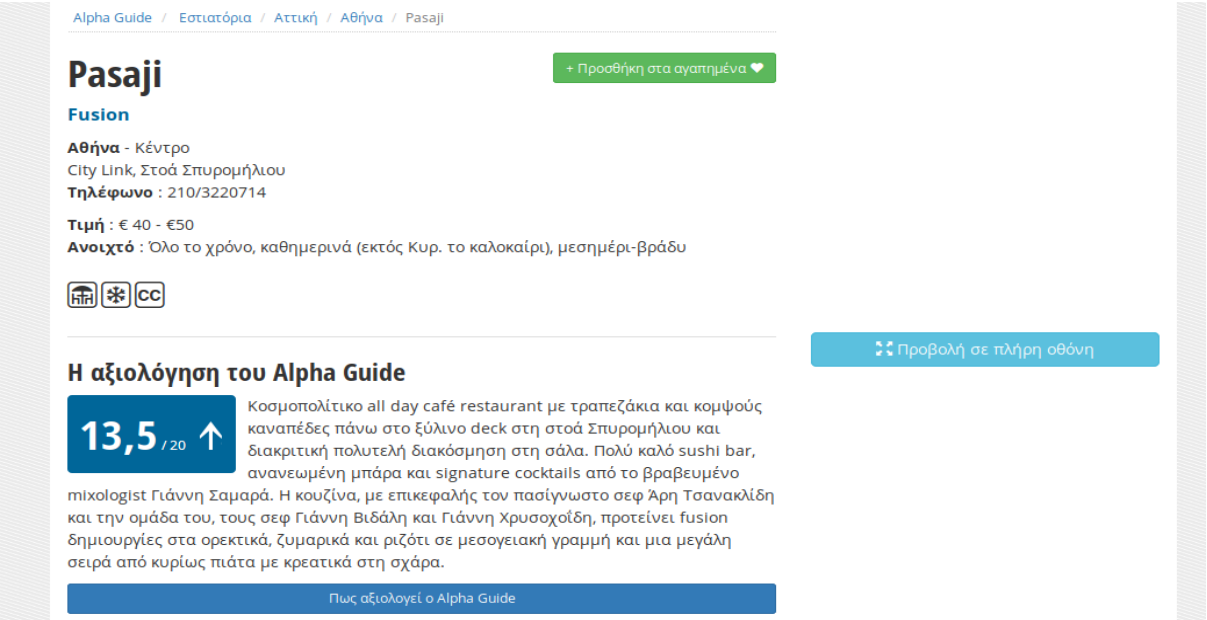
Finally, before we proceed to the configuration of the actual harvesting, it is important to note about two essential pieces of information that can be supplied inside the list. Namely, the title (name) of the POI and its coordinates. To cover this case, one can specify either the corresponding keys or the characteristic HTML attributes under the name_id and coordinates_id, accordingly, inside the configuration file.

Harvesting options

Once the final list with URLs has been constructed, the harvester starts searching for tags to populate the corresponding keys with values, either in a CSV file or in a JSON object in memory. The tags, as shown in the configuration example before, are defined in a key/value pair format. The key represents the name that the user gives to each field, i.e., the title of the column in a CSV file. The value is the element the harvester searches for inside the HTML structure.

As may have been noticed in the configuration example, some values are preceded with a semicolon (:). This semicolon has a special meaning. Each semicolon before a value is interpreted by the harvester as one place (one element) back. As will become clear in a while, during a presented example, this functionality is useful when there is no fixed point associated with the element which needs to be harvested. In this case, one has to find a fixed point and then it is possible to indicate its value and place by specifying the number of semicolons in front of it to be equal to the number of places before this point in the element under discussion.

We demonstrate this functionality by means of the following example. We assume that we would like to harvest a Web page about a restaurant with the following appearance:



The details in this Web page that we are interested in are the following:

1. Restaurant type (style): Fusion
2. City: Αθήνα
3. Address: City Link, Στοά Σπυρομήλιου
4. Phone number: 210/3220714
5. Price range: € 40 - €50
6. Opening days: Όλο το χρόνο, καθημερινά (εκτός Κυρ. το καλοκαίρι), μεσημέρι-βράδυ
7. Facilities: *the information contained in the three icons just after the opening days*
8. Rating: 13,5/20

For some of these details it is straightforward to find the appropriate rule. For example, for the phone number we could define a tag as

```
"phone_number": "Τηλέφωνο"
```

since beside the phone number there is the title "Τηλέφωνο". However, for most of the other pieces we have to dive into the HTML structure. Therefore, it is time to present the HTML code of this example page, at least the part involved in the details above:

```
686 <div class="details">
687 <p>
688 <span class="results-list-cuisine">Fusion</span>
689 </p>
690 <p>
691 <b>Αθήνα </b> - Κέντρο <br>City Link, Στοά Σπυρομήλιου <br>
692 <b>
693 Τηλέφωνο
694 </b>: 210/3220714<br />
695 <span style="display:none;">
696 <b>Fax: </b><br>
697 </span>
698 <span style="display:none;">
699 <b>Site</b>: <a href="http://" target="_blank"></a><br>
700 </span>
701 <span style="display:none;">
702 <span class="facebook-icon"></span><a href="" target="_blank">facebook</a>
703 </span>
704 </p>
705 <div class="clearfix"></div>
706 <p>
707 <span><b>
708 Τιμή
709 </b>: &euro; 40 - €50<br>
710 </span>
711 <span><b>
712 Ανοιχτό
713 </b>: Όλο το χρόνο, καθημερινά (εκτός Κυρ. το καλοκαίρι), μεσημέρι-βράδυ</span>
714 </p>
720 <ul class="facilities">
721 <li>
722 <div data-icon="B" class="icon" data-toggle="tooltip" data-placement="top" title="Σερβίρει και έξω"><span>Σερβίρει και έξω </span></div>
723 </li>
724 <li>
725 <div data-icon="b" class="icon" data-toggle="tooltip" data-placement="top" title="Κλιματισμός"><span>Κλιματισμός </span></div>
726 </li>
727 <li>
728 <div data-icon="q" class="icon" data-toggle="tooltip" data-placement="top" title="Δέχεται πιστωτικές κάρτες"><span>Δέχεται πιστωτικές κάρτες </span></div>
729 </li>
730 </ul>
731 <div class="clearfix"></div>
732 </div>
733 </div>
734 <ul class="list-inline specs" style="display:none;">
735 <li id="showReviews"><a href="#" id="show-reviews-detail">0
736 Αξιολογήσεις
737 </a></li>
748 <h3>
749 Η αξιολόγηση του Alpha Guide
750 </h3>
751 <span class="RestaurantRating">13,5<small> / 20</small>
752 <span class='restaurant-arrowup arrow-big'></span>
753 </span>
```

We can now observe the following:

1. Fusion is surrounded by a span tag with a unique class, therefore the first tag definition would be: "style": "results-list-cuisine".

2. The city name Αθήνα is not so trivial to harvest. It is not inside a unique tag neither has an identifier in front. Thus, we have to find a fixed point and define the position of Αθήνα relative to this. The fixed point we will pick in this example will be the element Τηλέφωνο. Αθήνα is then the fourth element before Τηλέφωνο. Therefore, the appropriate setting would be: "city": "::::Τηλέφωνο".

3. Using the same rationale, the setting for the address would be: "address": "::Τηλέφωνο".

In this manner, we can continue constructing the rest of the settings, resulting in the tags settings presented in the configuration example in the beginning of this section.

8.2.2. API Harvester Configuration

For PDH-API, the configuration file has a much simpler structure. First of all, the type attribute should be provided, and, in this case, it should be equal to api. The second essential option is the base_url parameter, which should correspond to the URL where the API is listening. As in every case, the output filename should also be provided. An optional attribute is max_connections used directly by cURL, which defaults to 5.

There are five additional configuration parameters:

- base_url : corresponds to the URL providing the list of the keys (ids).
- identifier : the name that the specific API uses for this key.
- query : this is optional and in case it is not provided all datasets are returned.
- other_params : also optional, this specifies additional parameters that could be attached to the request to the API.
- requests_limit : the maximum number of requests per minute.

8.2.3. CKAN Harvester Configuration

As in the previous cases, the configuration is provided in a JSON file. There are five parameters involved, with only two of them being mandatory. These are the parameters type and url. The former is set to ckan, whereas the latter is the URL of the data catalogue to be harvested. Additionally, one could determine the API version number, which by default is set to 3.

If no additional parameters are set, besides the two aforementioned ones, the CKAN harvester will attempt to fetch the entire contents of the catalogue. It will first retrieve the list with the total packages ids and subsequently it will retrieve the content of each package, merging all information in one file.

Alternatively, the user can choose to fetch only those datasets satisfying a specific query. In this case, it is possible to use one or both of the options query and filter_query. The former corresponds to a free text search, while with the latter can specify filters for searching in certain attributes. Boolean operators can also be used inside queries, as well as parentheses to define logical priorities. In case of filter queries, the values should be given in one of the two following formats:

```
"filter_query": "tags:<tag_value_to_search_for>"
```

or

```
"filter_query": "<vocabulary_name>:<tag_value_to_search_for>"
```

Concluding, if we need to harvest the entire data catalogue of, e.g., ckan.example.com, we should provide the harvester with the following configuration file:

Code snippet: Sample configuration for PDH-CKAN (without filtering)

```
// Settings for ckan harvester
{
    // The harvester type
    "type": "ckan",
    // The URL of the catalogue
    "url": "https://ckan.example.com",
    // Optionally, the CKAN API version
    "api_version": "3"
```

```
}
```

On the other hand, if we need to fetch only specific content, we could provide a configuration of the following type:

Code snippet: Sample configuration for PDH-CKAN (with filtering)

```
{
    // The harvester type
    "type": "ckan",
    // The URL of the catalogue
    "url": "https://ckan.example.com",
    // Optionally, the CKAN API version
    "api_version": "3",
    // Free text query
    "query": "This text should be included somewhere inside the dataset",
    // Filters
    "filter_query": "tags:(example1 OR (example2 AND example3))"
}
```

8.2.4. Configuration for POI Data Exploration

The PDE tool has a simpler configuration compared to PDH. In this case, we provide the necessary settings directly with the tool execution in the command line rather than defining them in a separate file. Each parameter has the following form:

```
parameter_name=parameter_value
```

There are four parameters, of which only one is mandatory. This is the filename, which specifies the full path of the file containing the dataset to be assessed. If only filename is supplied, then the tool returns a JSON file with the general statistics of the dataset.

In order to compute more detailed statistics for a specific column, the name of the column has to be supplied in the parameter column. In that case, the CSV file of the dataset has to include a first line with the column names. The value passed to column has to coincide with one of these names. If no other option is passed to the script, then a generic analysis is performed to the data contained inside this column. The results are returned in a JSON structure suitable for generating, by default, a pie chart. This can be altered by specifying the value of the parameter chart_type which can be either pie or bar.

The last parameter is category. This parameter can take any of the values generic, categorical, schedule, name, cost, address, phone or rating, with generic being the default. The corresponding functionality associated with each value of this parameter has been explained in Section 3.3.2.2.

Concluding, the syntax of the command issued to run the script with the available parameters is:

```
python explore.py filename=<filename>.csv [column=<column_name> [category=generic |
categorical | schedule | name | cost |address | phone | rating] [chart_type=pie|bar] ]
```

8.3. Execution

In the following, we provide some examples demonstrating the execution of PDH and PDE. Through these examples, we explore the capabilities of the tools, and provide an explanation of the results.

8.3.1. Executing PDH

We have tested the harvesting toolkit in a variety of cases. To demonstrate its functionality, we present the following concrete examples:

- an execution of PDH-HTML on a Web site about restaurants located in Greece
- an execution of PDH-API on the Web site https://openchargemap.org
- an execution of PDH-CKAN on the European open data portal http://data.europa.eu.

For the first case, we provide an indicative example for harvesting POI data from the entertainment portal http://www.alpha-guide.gr. In this case, we have already created the list of URLs and saved it in JSON format. Thus, we use the following configuration file:

Example configuration file for the execution of PDH-HTML

```
{
  // Using web harvester
  "type": "web",
  // Output path and filename
  "filename": "data/alpha.csv",
  // The requested url or path to local file to obtain the POI list (json)
  "url": "./alpha.json",
  // The sequence of fields, separated by semicolon, under which the list is available
  "fields": "jsrestaurants",
  // The base url for the individual links for each POI
  "base_url": "http://www.alpha-guide.gr/el/restaurants",
  // The title of the link identifier
```

```
    "link_id": 1,
    // The identifier for the POI's name
    "name_id": 0,
    // The tags which will be searched inside the html
    "tags": {
            "Type": "results-list-cuisine",
            "RestaurantRating": "RestaurantRating",
            "City": ":::Τηλέφωνο",
            "Address": "::Τηλέφωνο",
            "Phone_number": "Τηλέφωνο",
            "Fax": "Fax:",
            "Site": "Site",
            "Facebook": "facebook",
            "Price": "Τιμή",
            "OpeningHours": "Ανοιχτό",
            "Facilities": "facilities"
    },
    // The maximum number of simultaneous connections (default: 5)
    "max_connections": 5
}
```

We proceed to the second example of retrieving the information about the charging stations worldwide. The information about this kind of POIs is provided by openchargemap.org via an API. It is possible to retrieve all the available information if we iterate over the total set of country codes. Although the API does not explicitly provide a list with the available codes, it uses the 2-digit ISO code format which can be obtained from various sources. Below is the configuration file we use with which we obtained a dataset with over 70,000 entries (around 500Mb).

Example configuration file for the execution of PDH-API

```
// Settings for Api Harvester
{
    "type": "api",
    "base_url": "https://api.openchargemap.io/v2/poi/",
```

```
    "identifier": "countrycode",
    "list_url": "http://country.io/names.json",
    "filename": "data/ocm.json",
    "other_params": "maxresults=100000",
    "requests_limit": 2
}
```

Note that using a similar configuration, we have also obtained information about weather conditions from openweathermap.org.

Finally, we present an example execution of PDH-CKAN, demonstrating also the benefits offered by the SOLR search engine to filter which datasets to retrieve. Particularly, in this example we are going to harvest the European Union Open Data portal, using the configuration shown below.

Example configuration file for the execution of PDH-CKAN

```
{
    "type": "ckan",
    "filename": "data/ckan.json",
    "url": "https://data.europa.eu/euodp/data",
    "filter_query": "tags:('Science and technology' AND 'Environment')",
    "query": "title:'Agricultural'",
    "results": 10000,
    "max_connections": 5
}
```

In this specific example, we are interested only on those datasets tagged with both labels "Science and technology" and "Environment", and additionally only those containing the term "Agricultural" inside the title of the dataset. Using this configuration, PDH-CKAN fetches only the respective datasets. It turns out that there are only two datasets that satisfy these filters, which were retrieved in less than a second.

8.3.2. Executing PDE

Next, we present an example of the execution of PDE. We use a dataset obtained by harvesting a Web site providing information about restaurants in Greece. The dataset is provided in a CSV file (dataset.csv).

First, we obtain general information for this dataset by running:

```
python explore.py filename=dataset.csv
```

The output is written in a JSON file containing the general information about the dataset. A visualization of the results is shown in Figure 24. The table on the top left presents some basic information, including the name of the file, the number of rows and columns, as well as any possible errors during the execution of the script. The main information is provided by means of a dynamic tree diagram. The central node in this diagram represents the entire dataset. The children of this node are the fields the dataset contains (the titles of the columns in the corresponding CSV file). If a field contains distinct values, the number of which does not exceed a given threshold, then the field is considered categorical. Then, these distinct values of this field are presented in the diagram as child nodes of the respective node. In this example, we can notice that this is for instance the case for attributes style and payment. In this way, the user can quickly obtain a high-level overview of the contents of the dataset.

Next, to drill down to more detailed information about a specific column, the following command can be issued:

```
python explore.py column=best_for
```

In this case, a generic analysis is performed for the column entitled best_for. The resulting JSON response is included in the table below, including, for instance, the total number of values, the number of distinct values, the most frequent value, etc.

A more elaborate analysis can be performed in this particular case since this field is found to contain categorical data. Instead of the previous command, we can now run:

```
python explore.py filename=dataset.csv column=best_for category=categorical
```

Figure 25 shows a pie chart visualization of the more detailed results produced now.

Alternatively, we can choose to present these results in the form of a bar chart. In that case, we execute the following commnad:

```
python explore.py filename=dataset.csv column=best_for category=categorical chart_type=bar
```

which produces the bar chart depicted in Figure 26.

filename	dataset.csv
rows	1043
columns	11
errors	

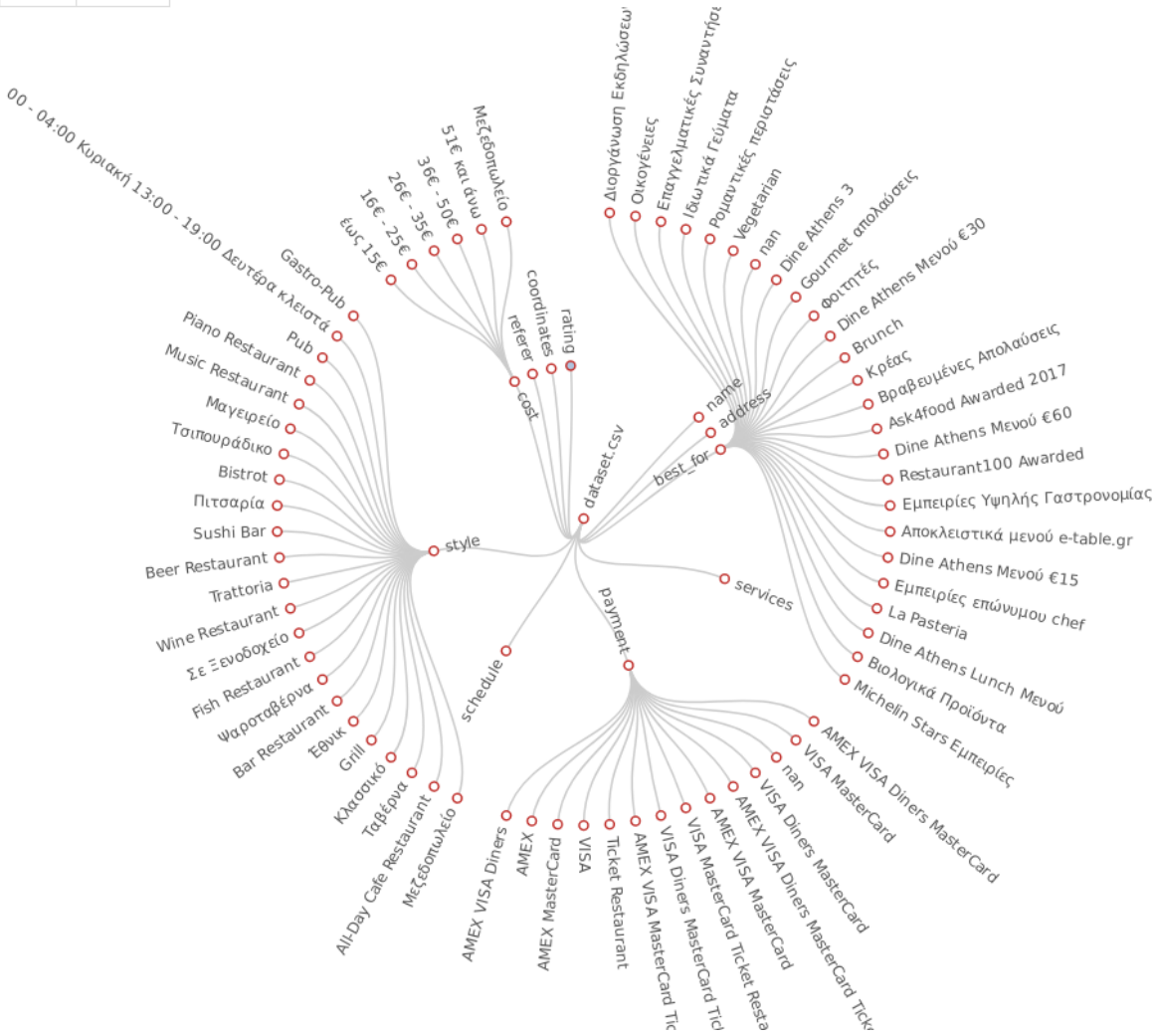


Figure 24: Example visualization of the general analysis of a CSV POI dataset with PDE.

Sample result from the analysis of a given attribue.
{
"description": {
"count": 2611,
"unique": 24,
"top": "Διοργάνωση Εκδηλώσεων",
"freq": 571,

```
            "null": 0,
            "minimum length": 5,
            "maximum length": 29
  },
  "generic": {
            "type": "pie",
            "data": [
            {
            "value": 2093,
            "name": "greek word(s) + \nE.g.:Ρομαντικές περιστάσεις"
            },
            {
            "value": 518,
            "name": "english word(s) + \nE.g.:Vegetarian"
            }
            ]
  }
}
```

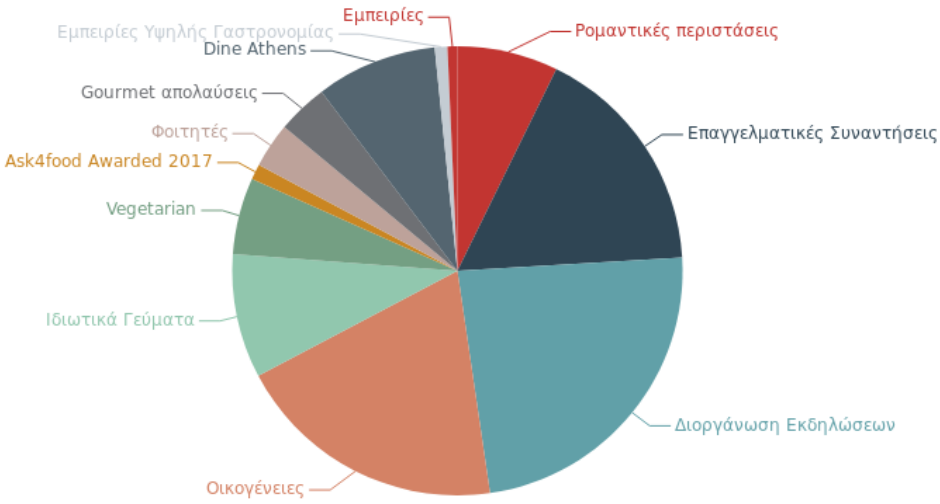


Figure 25: Example of a pie chart visualizing the analysis results of a given attribute.

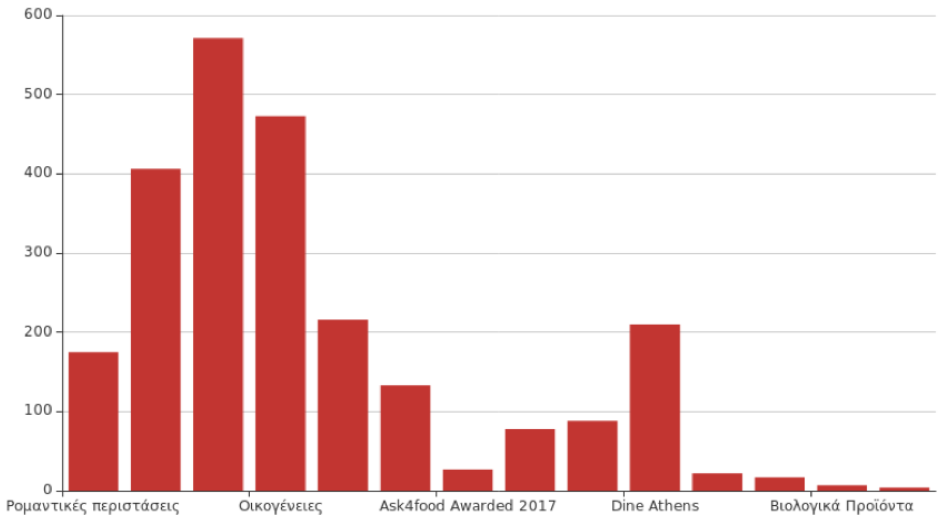


Figure 26: Example of a bar chart visualizing the analysis results of a given attribute.

Finally, we present two more cases. The first refers to a pie chart obtained by analysing the column corresponding to the name of the POIs (see Figure 27). In this case, information about the number of words and the language is given. The second involves the attribute address and is presented in Figure 28. In this case, one could retrieve information about the format of the address. In this specific example, we observe

that none of the addresses contain a postal code. Most of them are in the form "words + number + words", which likely indicates street name, street number and location. There exist also a few cases with two street numbers, which likely indicates addresses located in street corners.

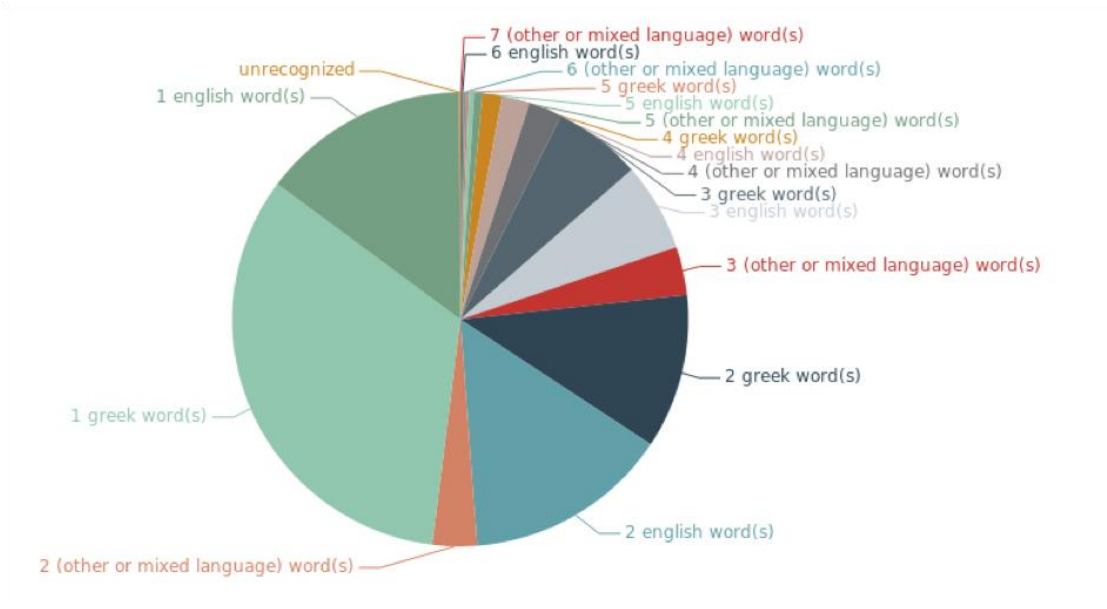


Figure 27: Example pie chart showing the result of the analysis on POI names.

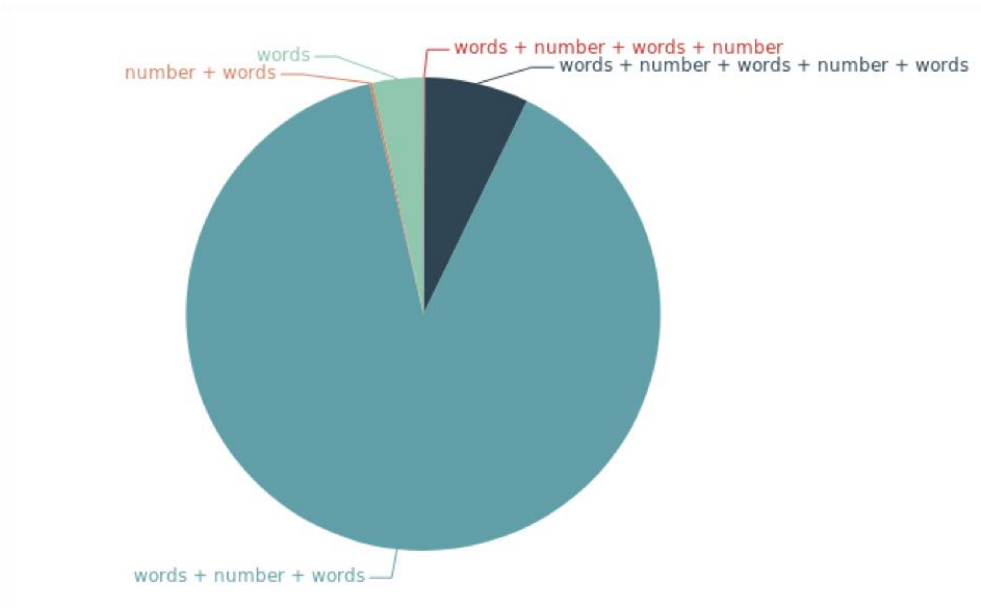


Figure 28: Example pie chart showing the result of the analysis on POI addresses.

9. Annex III: TripleGeo User Manual

In this Annex, we provide a complete user manual for TripleGeo ver2.0. First, we give details on building the application from the Java source code, along with particular details on certain of its dependencies. Next, we describe configuration settings both for transformation to RDF and for reverse transformation. Finally, we provide execution examples that demonstrate its operation and indicate certain limitations of the software mostly related to specific platforms.

9.1. Building Installation

Version 2.0 of TripleGeo, as well as all its previous releases, are publicly available, offering the entire Java source code as well as indicative configurations [TripleGeo]. TripleGeo is a command-line utility and has several dependencies on open-source and third-party, freely redistributable libraries.

Java SDK 1.8 (or later) as well as Maven 3.5.0 (or later) [MAVEN] must be installed and properly configured in order to compile and execute TripleGeo. The `pom.xml` file contains the project's configuration in Maven and has been successfully tested in both MS Windows and Linux environments.

Special note on JDBC drivers for database connections: In case that data will be extracted from a geospatially-enabled DBMS (e.g., PostGIS), either the user must include the respective .jar (e.g., `postgresql-9.4-1206-jdbc4.jar`) in the classpath at runtime or to specify the respective dependency in `pom.xml` and then rebuild the application.

Special note on manual installation of a JDBC driver for Oracle DBMS: Due to Oracle license restrictions, there are no public repositories that provide `ojdbc7.jar` (or any other Oracle JDBC driver) for enabling JDBC connections to an Oracle database. You need to download it and install in your local repository. First, this jar must be downloaded from Oracle and then installed it in a local maven repository as follows:

```
mvn install:install-file -Dfile=/<*YOUR_LOCAL_DIR*>/ojdbc7.jar
-DgroupId=com.oracle -DartifactId=ojdbc7 -Dversion=12.1.0.1 -Dpackaging=jar
```

Starting from version 1.3, TripleGeo includes support for custom transformation of thematic attributes according to the RDF Mapping language [RML]. In order to enable RML transformation mode, the respective library `RML-Mapper.jar` specially prepared for TripleGeo execution must be installed in a local Maven repository as follows:

```
mvn install:install-file -Dfile=/<*YOUR_LOCAL_DIR*>/RML-Mapper.jar
-DgroupId=be.ugent.mmlab.rml -DartifactId=rml-mapper -Dversion=0.3
-Dpackaging=jar
```

Building the application with Maven can be done as follows:

```
mvn clean package
```

and results into a `triplegeo-2.0-SNAPSHOT.jar` under directory target according to what has been specified in the `pom.xml` file.

The current distribution (ver. 2.0) comes with dummy configuration templates `file_options.conf` for geographical files (ESRI shapefiles, CSV, GPX, KML, etc.) and `dbms_options.conf` for database contents (from PostGIS, Oracle Spatial, etc.). These files contain indicative values for the most important properties when accessing data from geographical files or a spatial DBMS. Self-contained brief instructions guide the user into the extraction and transformation process.

Configuration files for several cases and file formats are available in the SLIPO GitHub [TripleGeo] to guide the users into building their own. Indicative configurations are also given in the Annex (Section 7.2).

9.2. Configuration Settings

Execution platform
runtime

Number of partitions
partitions

Spark logging
spark_logger_level

Transformation mode
mode

Export format
serialization

Spatial Ontology
targetGeoOntology

Input/Output
inputFormat
encoding
batchSize
inputFiles
tmpDir
outputDir

CSV specifications
delimiter
quote

Mapping specification
mappingSpec

Spatial Reference Systems
sourceCRS
targetCRS

Classification scheme
classificationSpec
classifyByName

Data Source and Namespaces
featureSource
nsOntology
nsGeometry
nsFeatureURI
nsClassificationURI
nsClassURI
nsDataSourceURI
prefixes
namespaces

DBMS connection and data details
dbType
dbName
dbUserName
dbPassword
dbHost
dbPort
tableName

Filtering
filterSQLCondition
spatialExtent

Basic attributes
attrKey
attrGeometry
attrX
attrY
attrName
attrCategory
registerFeatures

Other properties
defaultLang

Table 13: Properties specified in configuration files for transformation to RDF using TripleGeo ver.2.0

9.2.1. Configuration for Transformation to RDF

Before attempting any transformation using TripleGeo, a configuration file must be prepared. This file lists several *mandatory* properties that define how input data will be accessed, where they will be exported and into which format, as well as *optional* features (e.g., reprojection into another spatial reference system, classification scheme for assigning categories to features). The list of all properties that can be specified for transformation into RDF is shown in Table 13. An indicative configuration file that has been applied to transform data from an ESRI shapefile into RDF is provided in the Annex (Section 7.2.1). Some of the basic properties in this configuration are listed in Figure 29.

```
inputFormat = SHAPEFILE
inputFiles = ./test/data/points.shp
encoding = UTF-8

tmpDir = ./tmp
outputDir = ./test/output
serialization = N-TRIPLES

targetGeoOntology = GeoSPARQL

mappingSpec = ./test/poi_mappings.yml

classificationSpec = ./test/poi_classification.csv

mode = STREAM

attrKey = osm_id
attrGeometry = the_geom
attrName = name
attrCategory = type

nsOntology = http://slipo.eu/def#
nsGeometry = http://www.opengis.net/ont/geosparql#
nsFeatureURI = http://slipo.eu/id/poi/

sourceCRS = EPSG:2100
targetCRS = EPSG:4326
```

Figure 29: Example of a configuration file as used in TripleGeo

Next, we provide full details about all properties that can be specified in this *configuration file*:

1. ***Execution environment*** parameters specify how the transformation process will be executed. The following *optional* parameters regarding execution platform and on-the-fly data partitioning are applicable for transformation of ESRI shapefiles and CSV data only:

- *runtime* specifies the execution platform. Possible values are either *JVM* (*default*) for single- or concurrent multi-threaded execution using the Java Virtual Machine or *SPARK* for parallelized execution on top of Apache Spark.
- *partitions* specifies an integer value for the number of data partitions. The input data file will be split into that number of partitions; each part will be transformed separately either using multi-threaded execution over the JVM or using Spark, depending on the specification on the runtime parameter.
- *spark_logger_level* specifies the level of logging when running over SPARK. Default value is *WARN*.

2. ***Transformation mode***. The user must specify *mandatory* property

- *mode*: Controls the execution mode for the transformation and can take one of the following values: *GRAPH* (using a disk-based Jena model), *STREAM* (in-memory conversion with prompt creation of triples per input feature), *RML* (for applying user-specified RML schema mappings), or *XSLT* (for handling XML/GML/KML/INSPIRE-aligned input with XSLT transformation).

3. ***Input and output properties***.

- *inputFormat*: This *mandatory* property specifies the format of the input data. In case of *geographical file(s)*, supported vector data formats include: *SHAPEFILE*, *CSV*, *GPX*, *GEOJSON*, *XML* (for handling XML/GML/KML/INSPIRE-aligned input), *OSM_XML* (for OpenStreetMap XML format), *OSM_PBF* (for OpenStreetMap compressed PBF format), and *JSON*. In case that input features reside in a *database*, value *DBMS* must be specified for this parameter.
- *encoding*: This optional property specifies the encoding (character set) for strings in the *input* data, such as UTF-8, ISO-8859-1, ISO-8859-7, WINDOWS-1253, etc. If not specified, UTF-8 encoding is the considered as the default. Note that string literals in *output* RDF triples are always in UTF-8.
- *batchSize*: This *optional* parameter indicates the number of input features to transform in each batch before storing output to file. This is done for performance, since too frequent writes to disk may slow down the speed of transformation. By default, the current version spills output triples to file once a batch of 10 input records is processed.

The user must always specify (relative or absolute) *paths* to directories and files used during processing:

- *inputFiles*: The user may specify *multiple* input files (of exactly the same format and attribute schema) separating them by ';' in order to activate multiple concurrent threads for their transformation. Specification of input files should be omitted in case that data resides in a geospatial DBMS (there is another parameter concerning the *tableName* that should be specified instead).
- *tmpDir*: The working directory that may be used for storing intermediate files temporarily created during transformation (e.g., the Jena model created on disk with the GRAPH transformation mode).
- *outputDir*: Directory where the output RDF file(s) will be stored. By default, the output file name(s) are automatically composed from the original input file name(s) with the extension of the respective RDF serialization, e.g., *points.nt*. Files with the same name previously created in the output directory will be overwritten.

In case of CSV input, two additional parameters must be specified (omitted for any other input formats):

- *delimiter*: designates the character delimiting attribute values for each input line (i.e., record). This single character must not appear in any attribute value.
- *quote*: designates the quote character for enclosing string values in attributes.

4 ***Export format*** for the output file(s) can be specified with mandatory property:

- *serialization*, which can be set to one of the following values: *RDF/XML* (used as default for the XSLT transformation mode), *RDF/XML-ABBREV*, *N-TRIPLES* (used as default for the STREAM and RML transformation modes), TURTLE (or TTL), and N3.

5. ***Spatial ontology*** for geometries in the exported RDF data. This depends on the triple store where the exported data will be imported (e.g., Virtuoso, Oracle), since geometric representation and geospatial support varies widely amongst them. This must be defined in mandatory property

- *targetGeoOntology*, which can currently support three possible options: *GeoSPARQL* (default) for subsequent import to compliant triple stores (e.g., Oracle, Parliament), *Virtuoso* for extracting point features using the legacy RDF ontology only for points in Virtuoso RDF (namespace *virtrdf:*) [VirtRDF] or *wgs84_pos* for point features under the WGS84 Geoposition RDF vocabulary [GeoPos84].

6. ***Mapping specification***. The current version of TripleGeo supports mappings from the input attribute schema into an ontology for RDF features that guides the transformation (i.e., creating RDF properties, constructing URIs, defining links between entities, etc.). These mappings will be utilized in transformation once configuration property

- *mappingSpec* specifies the (absolute or relative) path to a file (in TTL, YAML, or XSL format) that contains these mappings.

Such mappings can be defined in three alternative file formats and employed in diverse transformation modes:

i. In *RML* transformation mode, a TTL file (in TURTLE format) contains RML mappings from input schema to RDF. In RML mode, specifying mappings with this file is *mandatory*, otherwise no RDF triples will be produced.

ii. In *GRAPH* or *STREAM* transformation modes, a file (in YAML format) contains mappings from input schema to RDF according to a custom ontology (such as the SLIPO ontology for POIs [SLIPO-D2.1]). In GRAPH/STREAM modes, this parameter is *optional*; if left blank or omitted, then an RDF property will be created for each thematic attribute in the original schema, by borrowing its attribute name.

iii. In *XSLT* transformation mode, a XSL style sheet file determines the XSL schema mapping for attributes to be converted. Specifying such a file is *mandatory* for XSLT transformation, otherwise no RDF triples will be produced.

7. ***Classification scheme***. Optionally, classification of input features into categories can be also performed during transformation, provided that the user specifies a (possibly hierarchical, multi-tier) classification scheme (e.g., possible amenities for Points of Interest, a list of road types for a Road Network). Classification is only applied if a suitable *mappingSpec* (including a category attribute) has been also specified. This classification scheme can be prescribed with these configuration properties:

- *classificationSpec*: File (either in CSV or YAML format) containing a classification hierarchy in categories assigned to input features. This property should be left blank in case of no applicable classification scheme.

- *classifyByName*: Boolean parameter indicating whether the data features specify their category based on its identifier in the classification scheme (*false*) or the actual name of the category (*true*). By default, transformation uses identifiers of categories in the classification scheme. This parameter is ignored if no classification hierarchy has been properly defined (i.e., missing or wrong path in parameter *classificationSpec*).

8. ***DBMS connection and data details***. The following properties are *mandatory* when connecting to a DBMS and extracting features from a spatial table. In case that any other value has been specified in parameter *inputFormat*, these parameters should be omitted altogether.

- *dbType*: Specify the DBMS backend where spatial data is stored. Possible values: *MSAccess*; *MySQL*; *Oracle*; *PostGIS*; *DB2*; *SQLServer*; *SpatiaLite*.
- *dbName*: Name of the database to connect. For MS Access databases, specify absolute or relative path to the .mdb database file.
- *dbUserName*: Username for JDBC connection. For MSAccess databases, credentials are optional; must be specified only if required to access the .mdb database file.
- *dbPassword*: Password for JDBC connection. For MSAccess databases, credentials are optional; must be specified only if required to access the .mdb database file.
- *dbHost*: The host name or IP address on which DBMS server listens for TCP/IP connections from client applications. Value *localhost* should be set if working with a local database server. Omit for MSAccess database connections.
- *dbPort*: Specify the TCP/IP port on which the DBMS server listens for connections from client applications. Omit for MSAccess database connections.
- *tableName*: *mandatory* property that indicates the database table or (predefined) view which contains the spatial features to be extracted. This property may be optionally coupled with a valid *filterSQLCondition* (described next) to select any subset of the table's records according to the SQL dialect pertinent to the DBMS.

9. ***Filtering*** parameters enable selection of input data that qualify to spatial or thematic criteria; if both criteria are specified in the configuration settings, they are considered as conjunctive filters. Such conditions can be *optionally* specified with the following properties:

- *spatialExtent* specifies a region of interest *for geograhical files only*. This region, defined as a valid WKT closed geometry (polygon or multipolygon), will be used as a filter to select for transformation only input geometries *contained within* the specified region. Containment is considered as the default topological operator to be applied against the input spatial features. In case that input data is accessed from a DBMS, this property should be omitted; any spatial filter can be included in the following property (*filterSQLCondition*) according to the spatial semantics and topological operators of the underlying DBMS.
- *filterSQLCondition*: this *optional* property specifies a filter for selecting qualifying records with syntax equivalent to a `WHERE` clause in SQL, e.g., `town_type = 'TOWN' OR town_type = 'VILLAGE' OR town_type = 'CITY'`. When accessing a table or view in a *DBMS*, this filter can be any valid condition (including topological ones), as if it were specified in the `WHERE` clause of an SQL statement (i.e., allowing use of `AND`, `OR`, `LIKE`, `BETWEEN` etc.) against the table. In case of *geographical data files*, this SQL-like logical expression also allows selection of input features based on their (case-sensitive) values on the specific *thematic attributes* included in the logical expression with the following restrictions: (i) Numeric values must also be quoted in these expressions; (ii) expressions may include `AND`, `OR` logical operators, but `NOT` is not supported; and (iii) Comparison

operators =, <>, <, <=, >, >=, `LIKE` are only allowed. In case this property is left blank, no filter is applied and all features in the dataset will be extracted.

10. ***Basic attributes*** that characterize each input feature and may be optionally used for registering it into the SLIPO Registry. Those basic attributes can be specified with the following properties:

- *attrKey* : mandatory column name containing unique identifier for each feature (i.e., each record). Until TripleGeo ver.1.3, this was required in order to guarantee suitable, unique URIs. Starting from TripleGeo ver. 1.4, a unique key for each input record is no longer required, since URIs are based on UUIDs, which are generated on-the-fly. However, it should be recommended that input features have a unique key in order to avoid duplicate triples with the same input contents.
- *attrGeometry*: specifies the name of the geometry column in the input table or file. Omit this parameter if geometry representation is available with columns specifying pairs of *X,Y* coordinates for points; otherwise, this parameter is mandatory.
- *attrX*: Specifies the attribute holding *X*-coordinates (or longitude) of point locations. Mandatory if a geometry attribute has not been specified above; otherwise, this property may be omitted.
- *attrY*: Specifies the attribute holding *Y*-coordinates (or latitude) of point locations. Mandatory if a geometry attribute has not been specified above; otherwise, this property may be omitted.
- *attrName*: optional property that specifies the column name containing name literals (i.e., strings). By default, NULL values in this attribute are suppressed and are not exported in order to avoid blank nodes. Applicable when name values are used for registering features in the SLIPO Registry. Leave blank if non applicable.
- *attrCategory*: optional property that specifies the column name(s) containing literals regarding classification into categories (e.g., type of points, road classes etc.) for each feature. This list of attribute names (specified with a *comma* delimiter) in the original dataset refers to classification items listed from finest to coarser (e.g., subcategory, category, etc.). during transformation, TripleGeo iterates through all attributes representing classification items (from finest to coarser) and the first *NOT NULL* classification value will be used to assign a category to this feature. Otherwise, a single attribute name should be specified, indicating that this provides the finest classification available for each input spatial feature. The appropriate classification value will be joined with the classification scheme specified in order to generate links of output entities with category URIs. By default, NULL values regarding are suppressed and are not exported in order to avoid blank nodes. This property is applicable when category values are used for registering features in the SLIPO Registry; leave blank if non applicable.

Such basic attribute values per feature along with its assigned URI (generated on-the-fly during transformation) can be exported from TripleGeo into a separate CSV file in order to be used in the *SLIPO Registry*. This is controlled by property

- *registerFeatures*: an *optional* Boolean parameter that denotes whether a .CSV file will be also extracted (*true*) or not (*false*) specifically for registering features in the SLIPO Registry. This property should be omitted in all other cases.

11. ***Data Source and Namespace properties***.

- *featureSource* is a mandatory string value (e.g., OpenStreetMap) that specifies the data source provider of the input features in order to include this information in the resulting triples for each entity. Unlike previous versions of TripleGeo, in ver.1.4 this value is no longer used in the assignment of URIs to features.

The namespaces and prefixes for the utilized ontology and the resources that will be generated are set with the following optional properties:

- *nsOntology*: the namespace of the underlying ontology. Used in creating properties for the RDF triples., e.g., http://slipo.eu/def# for the SLIPO ontology (*default*).
- *nsGeometry*: the namespace for the underlying geospatial ontology, e.g., http://www.opengis.net/ont/geosparql# for GeoSPARQL-compliant geometries (*default*).
- *nsFeatureURI* : the common URI namespace for all generated resources, e.g., http://slipo.eu/id/poi/ in the SLIPO ontology (*default*).
- *nsClassificationURI*: the common URI namespace for the classification scheme, e.g., http://slipo.eu/id/classification/ in the SLIPO ontology (*default*).
- *nsClassURI*: the common URI namespace for categories used in the classification scheme, e.g., http://slipo.eu/id/term/ in the SLIPO ontology (*default*).
- *nsDataSourceURI*: the common URI namespace for the data source provider, e.g., http://slipo.eu/id/poisource/ in the SLIPO ontology (*default*).

In addition, the user may also define two lists (of comma separated values) with the correspondence between a prefix and its respective namespace (mainly used in attribute mappings):

- *prefixes*: A list of prefixes employed in the (RML or YAML) mapping files, e.g., *slipo, geo, rdfs* .
- *namespaces*: the namespaces corresponding to the aforementioned prefixes, e.g., http://slipo.eu/def#, http://www.opengis.net/ont/geosparql#, http://www.w3.org/1999/02/22-rdf-syntax-ns# .

11. ***Spatial Reference Systems***. If geographic reprojection is required for geometries in the output triples, then the following properties must be filled in the configuration:

- *sourceCRS*: the EPSG identifier for the coordinate reference system (CRS) of the input geometries;
- *targetCRS*: the EPSG identifier for the coordinate reference system (CRS) of the output geometries.

In case that *either* of these properties is missing, the respective geometries are assumed to be in the WGS84 reference system (EPSG:4326). TripleGeo works for any valid EPSG reference systems [EPSG] and transforms all geometries in the dataset, for example from *sourceCRS*=EPSG:2100 (i.e., Greek Grid 1987) to *targetCRS*=EPSG:4326 (i.e., WGS84).

12. ***Other properties***.

- *defaultLang* : This optional property affects the default language tag to be assigned to any string literals created in the output RDF. Unless otherwise specified, the default value is English (i.e., *defaultLang=en*).

9.2.2. Configuration for Reverse Transformation

Reverse Transformation from RDF into geographical files is also controlled via a user-specified configuration file like the one listed in the Annex (Section 7.2.2). Such a file lists several *mandatory* properties that define how input data will be accessed, where they will be exported and into which format, as well as *optional* features (e.g., reprojection into another spatial reference), as listed in Table 14.

Input/Output	*CSV specifications*
serialization	delimiter
outputFormat	quote
encoding	
batchSize	*Spatial Reference Systems*
inputFiles	sourceCRS
tmpDir	targetCRS
outputFile	*Other properties*
sparqlFile	defaultLang

Table 14: Properties specified in configuration files for reverse transformation using TripleGeo ver.2.0

More specifically, this configuration file has the following structure and properties:

1. ***Input and output properties.***

- *serialization* is *mandatory* and concerns the format of the input RDF file(s) and can be set to one of the following values: *RDF/XML*, *RDF/XML-ABBREV*, *N-TRIPLES*, *TURTLE* (or TTL), and *N3*.
- *outputFormat*: This *mandatory* property specifies the format of the output data. The following three de facto formats of *geographical file(s)* are supported: *SHAPEFILE*, *GEOJSON*, or *CSV*.
- *encoding*: This *optional* parameter specifies the encoding (character set) for strings in the *output* data, such as UTF-8, ISO-8859-1, ISO-8859-7, WINDOWS-1253, etc. If not specified, UTF-8 encoding is the considered as default. String literals in *input* RDF triples are always in UTF-8.
- *batchSize*: This optional parameter indicates the number of output records to hold in each batch before storing them to file. This is done for performance, since too frequent writes to disk may slow down the speed of transformation. By default, the current version works with batches of 10 records.

The user must always specify (relative or absolute) *paths* to directories and files used during processing:

- *inputFiles*: The user may specify *multiple* input RDF files (of exactly the same serialization and ontology) separating them by ';' in order to create a single graph with all contents.
- *tmpDir*: The working directory that may be used for storing intermediate files temporarily created during reverse transformation (e.g., the Jena model created on disk from the RDF input).
- *outputFile*: File in which to store the output (reconstructed) records. A single output file will contain all transformed records, even if input comes from multiple files. Files with the same name previously created in the same directory will be overwritten.

- *sparqlFile*: File containing a user-specified `SELECT` query (in SPARQL) that will retrieve results (records) from the input RDF triples. This query should conform with the underlying ontology of the input RDF triples.

In case of CSV output, two additional parameters must be specified (omitted for any other output formats):

- *delimiter*: designates the character delimiting attribute values for each output line (i.e., record). This single character must not appear in any attribute value.
- *quote*: designates the quote character for enclosing string values in attributes.

2. ***Spatial Reference Systems***. If geographic reprojection is required for geometries in the output records, then the following properties must be filled in the configuration:

- *sourceCRS*: the EPSG identifier for the coordinate reference system (CRS) of the input geometries;
- *targetCRS*: the EPSG identifier for the coordinate reference system (CRS) of the output geometries.

In case that *either* of these properties is missing, the respective geometries are assumed to be in the WGS84 reference system (EPSG:4326). TripleGeo works for any valid EPSG reference systems [EPSG] and transforms all geometries in the dataset, for isntance from *sourceCRS*=EPSG:2100 (Greek Grid 1987) to *targetCRS*=EPSG:4326 (WGS84).

3. ***Other properties***.

- *defaultLang* : This optional property affects the default language tag assigned to string literals in the input RDF. Unless otherwise specified, the default value is English (i.e., *defaultLang=en*).

9.3. Execution

As already explained, TripleGeo supports *two-way* transformation of geospatial features:

- *Transformation* of geospatial datasets from various conventional formats into RDF data. This supports *attribute mappings* into an ontology for RDF features and also specification of a *classification scheme* for assigning categories into input features.
- *Reverse Transformation* of RDF data into de facto geospatial formats (CSV, GeoJSON, or ESRI shapefiles). TripleGeo retrieves data from a graph constructed on-the-fly from the RDF data and creates records with a geometry attribute and thematic attributes reflecting the underlying ontology of the input RDF data.

In either case, Java JRE (or SDK) 1.8 (or later) must have been installed. In addition, suitable values must have been set to all required properties in the configuration file, as explained in Section 9.2. If triples are to be extracted from a DBMS, then correct credentials must be given in the configuration.

Indicative explanation and usage tips for both transformation modules of TripleGeo ver.2.0, as well as regarding its auxiliary utilities are given next.

9.3.1. Executing Transformation to RDF

Next, we explain how to use TripleGeo ver.2.0 in order to transform geospatial data into RDF triples:

- In case that triples will be extracted from a geographical file (e.g., ESRI shapefiles) as specified in the user-defined configuration file in `./test/conf/shp_options.conf`, and assuming that binaries are bundled together in */target/triplegeo-2.0-SNAPSHOT.jar*, give a command like this:

```
java -cp ./target/triplegeo-2.0-SNAPSHOT.jar eu.slipo.athenarc.triplegeo.
Extractor ./test/conf/shp_options.conf
```

- If triples will be extracted from a geospatially-enabled DBMS (e.g., PostGIS), a suitable configuration file (e.g., located at `./test/conf/PostGIS_options.conf`) should include all information required to connect and extract data from the DBMS, and the Java command must also invoke a runtime linking to the JDBC driver for enabling connections to the DBMS (e.g., assuming that this JDBC driver is located at `./lib/postgresql-9.4-jdbc4.jar`):

```
Java -cp ./lib/postgresql-9.4-jdbc4.jar;./target/triplegeo-2.0-SNAPSHOT.jar
eu.slipo.athenarc.triplegeo.Extractor ./test/conf/PostGIS_options.conf
```

- TripleGeo supports data in *GML* (Geography Markup Language) and *KML* (Keyhole Markup Language). It can also handle *INSPIRE-aligned* GML data for seven Data Themes (Annex I), as well as INSPIRE-aligned XML geospatial metadata. Any such transformation is performed via XSLT, as specified in the respective configuration settings (e.g., `./test/conf/KML_options.conf`) as follows:

```
java -cp ./target/triplegeo-2.0-SNAPSHOT.jar eu.slipo.athenarc.triplegeo.
Extractor ./test/conf/KML_options.conf
```

- Starting from ver. 1.7, TripleGeo also enables distributed transformation of geographical files (currently, CSV, GeoJSON, and ESRI shapefiles) into RDF on top of Apache Spark and its geospatial extension GeoSpark. Configuration settings for such transformations are exactly as in the case of standalone execution over JVM, with extra specifications for the number of worker nodes (i.e., data partitions). Assuming a user-defined configuration file (also specifies the number of partitions over the input data) is available in `./test/conf/shp_spark_options.conf`, transformation can be executed by sumbitting a Spark job like this:

```
spark-submit --class eu.slipo.athenarc.triplegeo.Extractor --master local[*]
target/triplegeo-2.0-SNAPSHOT.jar ./test/conf/shp_spark_options.conf
```

In all cases, one the process gets finished, the resulting output files can be received in the output directory specified by the user in the configuration.

9.3.2. Executing Reverse Transformation from RDF

This is how to use TripleGeo ver.2.0 in order to transform RDF triples (a.k.a. *export*) into a geospatial data file:

- In the configuration file, the user must specify one or multiple files that contain the RDF triples that will be given as input to the reverse transformation process.
- The user must also specify a valid SPARQL `SELECT` query that will be applied against the RDF graph and will fetch the resulting records. The path to the file containing this SPARQL command must be

specified in the configuration. It is assumed that the user is aware of the underlying ontology of the RDF graph. If the SPARQL query is not valid, then no or partial results may be retrieved. By default, the names of the variables in the `SELECT` clause will be used as attribute names in the output file.

- In case of ESRI shapefile as output format, make sure that all input RDF geometries are of the same type (i.e., either points or lines or polygons), because shapefiles can only support a single geometry type in a given file.
- Once parameters have been specified in a suitable configuration file (e.g., stored in path `./test/conf/shp_reverse.conf`), the following command can be used to launch the reverse transformation process:

```
java -cp ./target/triplegeo-2.0-SNAPSHOT.jar eu.slipo.athenarc.triplegeo.
ReverseExtractor ./test/conf/shp_reverse.conf
```

Once processing is finished and all records are written into a file, the user is notified about the total amount of extracted triples and the overall execution time.

9.3.3. Auxiliary Utilities

Classification Scheme Validator can be used to verify the consistence and suitability of a classification hierarchy where the spatial entities refer to. TripleGeo supports multi-tier classification hierarchies (e.g., POI categories, subcategories, etc.) specified in YML or CSV files (take a look here for example classifications). This auxiliary utlity can be invoked as follows:

```
java -cp target/triplegeo-2.0-SNAPSHOT.jar eu.slipo.athenarc.triplegeo.extra.
ClassificationScheme Validator <path-to-CSV-or-YML-classification-file>
<boolean-flag> <output-CSV-or-YML-format>
```

where the three arguments have the following role:

- `<path-to-CSV-or-YML-classification-file>` specifies the file containing the classification hiererchy (in CSV or YML format);
- the `<boolean-flag>` specifies whether each category is referenced by its identifier in the classification scheme (*false*) or by the actual name of the category (*true*); and
- `<output-CSV-or-YML-format>` indicates the format (either *CSV* or *YML*) that will be used for printing out the reconstructed classification after its validation.

RDF Graph Sanity Tester is an auxiliary utility that can be used to verify whether the transformed triples are valid and queryable. First, it loads triples (in any typical serialization) from data file(s) into a disk-based RDF graph and then runs a simple sanity test with a user-specified SELECT query in SPARQL. If successful, it reports the number of triples stored in the graph. This utility can be executed as follows:

```
java -cp target/triplegeo-2.0-SNAPSHOT.jar eu.slipo.athenarc.triplegeo.extra.
RDFGraphSanityTester <path-to-triples-file(s)> <triple-serialization-format>
<path-to-temp-dir> <path-to-SPARQL-query-file>
```

where the four arguments respectively specify:

- <path-to-triples-file(s)> is the path to the RDF file(s) that constitute the graph;
- <triple-serialization-format> is the serialization of the RDF files (e.g., N-TRIPLES, TTL);
- <path-to-temp-dir> is the path to an existing directory on disk where the RDF graph model will be temporarily created; and
- <path-to-SPARQL-query-file> is the path to the file with the SPARQL SELECT command that will be used to query the RDF graph and extract results.

9.3.4. Known Limitations

Handling large datasets. Judging from our experience with extraction of triples from several geospatial repositories (cf. Section 5.3.1), it seems that this process may take several *minutes* to conclude in case of datasets that include thousands of records. Hence, in case of extremely large datasets (e.g., millions of records), it is advisable to split the input in several smaller parts and then extract triples from each one in multiple concurrent threads. When large datasets are handled, execution settings should also include suitable values for Java *heap size* in main memory (i.e., calling the executable with the `-Xms<size>` option) depending on the available system resources.

JDBC connection to geospatial DBMSs. Connection to a geospatial DBMS is performed through a JDBC bridge, so a suitable driver should be available. Until ver.1.1, TripleGeo was shipped with several such freely available drivers (e.g., for PostgreSQL), although certain software vendors restrain usage of such tools only to customers that have purchased their DBMS platform (e.g., IBM DB2). Starting from ver.1.2, TripleGeo *no longer includes any JDBC drivers*, so users may either specify them in the *pom.xml* and rebuild the application or directly invoke a particular .jar that contains the necessary drivers at the command line. So, before attempting to execute TripleGeo against data residing in any DBMS, the user should make sure that the necessary JDBC driver(s) for that version of the DBMS software are available in their system and accessible by the TripleGeo utility.

Interacting with Oracle databases on Linux platforms. When attempting to export triples from Oracle Server Enterprise Edition - Version: 11.1.0.6 to 11.2.0.2.0 [Release: 11.1 to 11.2] on Linux platforms, connection is established via the suitable JDBC driver. But as soon as records are to be retrieved, the following error may be issued from Oracle:

```
ORA-29516: Error in module Aurora: Assertion failure at joez.c:3311

Bulk load of method java/lang/Object.<init> failed; insufficient shm-object
space
```

It seems that this error relates to the just-in-time (JIT) compiler for Oracle JVM environment, which is intended for faster execution as invalidation, recompilation, and storage of code is done dynamically. JIT is controlled by parameter `java_jit_enabled`, and if it is set to `TRUE` then the Java methods are automatically compiled to native code by the JIT compiler and made available for use by all sessions.

But if error ORA-29516 appears on a Linux x64bit platform, the workaround to overcome that error is to turn off the JIT compiler by giving this SQL command to Oracle (administrative privileges are required):

```
ALTER SYSTEM SET java_jit_enabled=false;
```

Afterwards, exporting of triples is carried out without errors, but at the expense of a rather slow rate especially for larger datasets, as indicated from evaluation results in Section 5.3.1.

Besides, JDBC connections to Oracle DBMS use some "random numbers" to encrypt the connection information, and the lack of these numbers may cause failures. To solve this issue, it is necessary to define a different origin for random numbers than the default. The parameter to define the origin of random numbers is: `-Djava.security.egd=`, and one of the available choices inn Linux for this "random origin" is `/dev/urandom`. Hence, invoking TripleGeo for extracting and transformation geospatial data stored in an Oracle database should include the following directive in the Java command line so as to avoid serious delays in JDBC connections:

```
-Djava.security.egd=file:/dev/urandom
```

9.3.5. Demonstration

In this Section, we walkthrough indicative executions of TripleGeo by making use of the configuration settings (Section 7.2), attribute mappings (Section 7.3) and classification schemes (Section 7.4) listed in the Annex regarding a sample POI dataset. Note that TripleGeo is a command-line tool without a graphical interface of its own. However, it is integrated in the SLIPO Workbench, so it is possible to configure its parameters and launch execution (either transformation to RDF or reverse transformation from RDF) in a user-friendly fashion.

9.3.5.1. Example for Transformation to RDF

In this example regarding *transformation* to RDF of a shapefile dataset (containing a sample of one million POIs extracted from OSM) according to a configuration file (`shp_options.conf`), the user should give the following command to start the transformation process with TripleGeo ver.2.0:

```
java   -cp   ./target/triplegeo-2.0-SNAPSHOT.jar   eu.slipo.athenarc.triplegeo.
Extractor ./test/conf/shp_options.conf
```

If all paths are properly defined and the configuration is valid, the transformation process is launched successfully and the user is notified accordingly:

```
Encoding: UTF-8

Conversion mode: STREAM

Output serialization: N-TRIPLES

Transformation will take place from EPSG:2100 to EPSG:4326 reference system.
```

In case that a *classification* scheme has been specified (like the sample ones listed in Section 7.4), then TripleGeo proceeds to transform its contents into RDF and keeps in memory their corresponding URIs in order to create links to them from POIs in a subsequent step. The following notifications correspond to transformation of categories:

```
Classification hierarchy reconstructed from CSV file.
```

```
2019-11-12 14:08:16.424 GMT Thread main completed successfully in 223 ms. 181 records transformed into 709 triples and exported to N-TRIPLES file: ./test/output/poi_classification.nt.

2019-11-12 14:08:16.562 GMT Classification hierarchy read successfully!
```

Afterwards, TripleGeo starts consuming input POI data and creating RDF triples according to the chosen transformation mode. In case that multiple threads are employed, each one periodically (every 1000 input records) notifies the user on its progress:

```
2019-11-12 14:08:19.451 GMT pool-1-thread-1 Processed 24000 records...
```

Once a thread concludes transformation of its input data, it issues the following message:

```
2019-11-12 14:09:37.465 GMT Thread pool-1-thread-1 completed successfully in 76150 ms. 1000000 records transformed into 23510309 triples and exported to N-TRIPLES file: ./test/output/points.nt.
```

At the end of the process, TripleGeo notifies about the spatial extent of the transformed geometries (MBRS in WGS84), as well as the path to all output files containing RDF triples. Note that multiple such files may be obtained in case that multiple threads have been employed to handle disjoint pieces of the input POI data:

```
MBR of transformed geometries: X_min=-31.2638012, Y_min=29.9851487, X_max=46.6462048, Y_max=80.5156505

2019-11-12 14:09:37.472 GMT Transformation process concluded successfully in 79318 ms.

RDF results written into the following output files: [./test/output/poi_classification.nt, ./test/output/points.nt]
```

9.3.5.2. Example for Reverse Transformation

In this example, *reverse transformation* from RDF datasets to ESRI shapefile format is specified according to a configuration file (`shp_reverse.conf`) and it can be invoked as follows:

```
java -cp ./target/triplegeo-2.0-SNAPSHOT.jar eu.slipo.athenarc.triplegeo.ReverseExtractor ./test/conf/shp_reverse.conf
```

If all paths are properly defined and the configuration is valid, the reverse transformation process is launched successfully and the user is notified accordingly:

```
Encoding: UTF-8

Transformation will take place from EPSG:4326 to EPSG:2100 reference system.
```

Afterwards, TripleGeo starts creating a disk-based RDF model that will include all triples read from the RDF files specified as input. The tool provides a notification once each piece of input RDF data is loaded to the RDF graph:

```
2019-11-12 14:34:25.315 GMT Initializing RDF graph to hold input triples... Done!

2019-11-12 14:34:26.360 GMT Reading triples from file ./test/output/poi_classification.nt... Done!

2019-11-12 14:34:26.514 GMT Reading triples from file ./test/output/points.nt... Done!
```

Once all RDF input is loaded, TripleGeo issues notifications about the size of the created graph, as well as specific warnings in case that restrictions apply regarding geometry types (e.g., ESRI shapefiles can only support a single geometry type):

```
2019-11-12 14:40:12.895 GMT RDF graph loaded successfully and contains 14510626 statements in total.

All geometries are expected to be of type POINT in order to be included in this shapefile.
```

Afterwards, TripleGeo applies the user-specified SPARQL query against this RDF graph and starts emitting output records notifying the user about its progress:

```
2019-11-12 14:40:27.210 GMT Processed 42800 records...
```

Once all records are reconstructed, TripleGeo notifies the user and points to the file where the output is available:

```
2019-11-12 15:49:02.618 GMT 1000000 results retrieved from the RDF graph. 1000000 features created.

2019-11-12 15:49:02.812 GMT Reverse transformation process terminated successfully!

Records were written into output file:./test/output/points_reconstructed.shp
```