# A Rule-based Framework for Creating Instance Data from *OpenStreetMap*

Thomas Eiter[1], Jeff Z. Pan[3], Patrik Schneider[1,4], Mantas Šimkus[1], and Guohui Xiao[2]

[1] Institute of Information Systems, Vienna University of Technology, Austria
[2] Faculty of Computer Science, Free University of Bozen-Bolzano, Italy
[3] University of Aberdeen, UK
[4] Vienna University of Economics and Business, Austria

**Abstract.** Reasoning engines for ontological and rule-based knowledge bases are becoming increasingly important in areas like the Semantic Web or information integration. It has been acknowledged however that judging the performance of such reasoners and their underlying algorithms is difficult due to the lack of publicly available data sets with large amounts of *(real-life) instance data*. In this paper we describe a framework and a toolbox for creating such data sets, which is based on extracting instances from the publicly available *OpenStreetMap* (*OSM*) geospatial database. To this end, we give a formalization of OSM and present a rule-based language to specify the rules to extract instance data from OSM data. The declarative nature of the approach in combination with external functions and parameters allows one to create several variants of the data set via small modifications of the specification. We describe a highly flexible engine to extract instance data from a given OSM map and a given set of rules. We have employed our tools to create benchmarks that have already been fruitfully used in practice.

## 1 Introduction

Reasoning over ontological and rule-based knowledge bases (KBs) is receiving increasing attention. In particular *Description Logics* (DLs), which provide the logical foundations to OWL ontology languages [26], are a well-established family of decidable logics for knowledge representation and reasoning [3]. They offer a range of expressivity well-aligned with computational complexity. Furthermore, DLs have been enriched with rules in different approaches (see [9]). Moreover, several systems have been developed in the last decade to reason over DL KBs, which usually consist of a *TBox* that describes the domain in terms of *concepts* and *roles*, an *ABox* that stores information about known *instances* of concepts and their participation in roles. [5]

Naturally, classical reasoning tasks like TBox satisfiability and subsumption under a TBox have received most attention and many reasoners have been devoted to them (e.g. FaCT++ [31], HermiT [29], or ELK [17]). These and other mature systems geared towards TBox reasoning have been rigorously tested and compared (e.g., *JustBench* [4]) using several real-life ontologies like GALEN and SNOMED-CT.

[5] Further components might be present depending on the DL, e.g., *rules* as for OWL 2 RL

A different category are reasoners for *ontology-based query answering* (*OQA*), which are designed to answer queries over DL KBs in the presence of large data instances (see e.g. Ontop [20], Pellet [28],and OWL-BGP [19]). TBoxes in this setting are usually expressed in low complexity DLs, and are relatively small in size compared to the instance data. These features make reasoners for OQA different from classical (TBox) reasoners. The DL community is aware that judging the performance of OQA reasoners and their underlying algorithms is difficult due to the lack of publicly available benchmarks consisting of large amounts of *real-life instance data*. In particular, the popular Lehigh University Benchmark (LUBM) [15] only allows to generate random instance data, which provides only a limited insight into the performance of OQA systems (see Section 8 for further related work).

In this paper, we consider publicly available geographic datasets as a source of test data for OQA systems and other types of reasoners. For the benchmark creation, we need a framework and a toolbox for extracting and enhancing instance data from *OpenStreetMap* (*OSM*) geospatial data. [6] The OSM project aims to collaboratively create an open map of the world. It has proven hugely successful and the map is constantly updated and extended. OSM data describes maps in terms of (possibly *tagged*) points, geometries, and more complex aggregate objects called *relations*. We believe the following features make OSM a good source to obtain instance data for reasoners:

- Datasets of different sizes exist; e.g., OSM maps for all major cities, countries, and continents are directly available or can be easily generated.
- Depending on the location (e.g., urban versus rural), the density, separation, and compactness of object location varies strongly. [7]
- Spatial objects have an inherent structure of containment, bordering, and overlapping, which can be exploited to generate spatial relations (e.g., `contains`).
- Spatial objects are usually tagged with semantic information like the type of an object (e.g., hospitals, smoking/non-smoking area), or the cuisine of a restaurant. In the DL world this information can be naturally represented in terms of concepts and roles.

Motivated by this, we present a rule-based framework and a toolbox to create benchmark instances from OSM datasets. Briefly, our contributions are the following:

- We give a model-based formalization of OSM datasets which aims at abstracting from the currently employed but rather ad-hoc XML or object-relational representation. It allows one to view OSM maps as relational structures, possibly enriched with computable predicates like the spatial relations `contains` or `next`.
- Building on the above formalization, we present a rule-based language to extract information from OSM datasets (viewed as relational structures). In particular, a user can specify in a declarative way rules which prescribe how to transform points/-geometries/relations of an OSM map into ABox assertions. Different benchmark ABoxes can be created via small modifications of external functions, input parameter, and the rules of the specification.
- Our language is based on an extension of *Datalog*, which enjoys clear and well accepted semantics [1]. It has convenient features useful for benchmark generation.

[6] http://www.openstreetmap.org

[7] E.g., visible in https://www.mapbox.com/osm-data-report/

- We have implemented an engine to create ABoxes from given input sources (e.g. an OSM database) and a given set of rules, which may be recursive and use negation. The engine is highly configurable and can operate on various input and output sources, like text files, RDF datasets, or geospatial RDBMSs, and even integrate computation using external functions.
- By employing the above generation toolbox, we show on a proof-of-concept benchmark, how parameterizable and extensible the framework is. The toolbox has been already fruitfully been used for two benchmarks [8, 12].

In summary, our framework and toolbox provide an attractive means to develop tailored benchmarks for evaluating reasoning engines, to gain new insights about the underlying algorithms.

## 2 Formalization of OSM

In this section we formally describe our model for OSM data, which we later employ to describe our rule-based language to extract instance data from OSM data. Maps in OSM are represented using four basic constructs (a.k.a. *elements*): [8]

- *nodes*, which correspond to points with a geographic location;
- *geometries* (a.k.a. *ways*), which are given as sequences of nodes;
- *tuples* (a.k.a. *relations*), which are a sequences of nodes, geometries, and tuples;
- *tags*, which are used to describe metadata about nodes, geometries, and tuples.

Geometries are used in OSM to express polylines and polygons, in this way describing streets, rivers, parks, etc. OSM tuples are used to relate several elements, e.g. to indicate the turn priority in an intersection of two streets.

To formalize OSM maps, which in practice are encoded in XML, we assume infinite mutually disjoint sets $\mathsf{M}_{\mathsf{nid}}, \mathsf{M}_{\mathsf{gid}}, \mathsf{M}_{\mathsf{tid}}$ and $\mathsf{M}_{\mathsf{tags}}$ of *node identifiers*, *geometry identifiers*, *tuple identifiers* and *tags*, respectively. We let $\mathsf{M}_{\mathsf{id}} = \mathsf{M}_{\mathsf{nid}} \cup \mathsf{M}_{\mathsf{gid}} \cup \mathsf{M}_{\mathsf{tid}}$ and call it the set of *identifiers*. An *(OSM) map* is a triple $\mathcal{M} = (\mathcal{D}, \mathcal{E}, \mathcal{L})$ as follows.

1. $\mathcal{D} \subseteq \mathsf{M}_{\mathsf{id}}$ is a finite set of identifiers called the *domain* of $\mathcal{M}$.
2. $\mathcal{E}$ is a function from $\mathcal{D}$ such that:
   (a) if $e \in \mathsf{M}_{\mathsf{nid}}$, then $\mathcal{E}(e) \in \mathbf{R} \times \mathbf{R}$;
   (b) if $e \in \mathsf{M}_{\mathsf{gid}}$, then $\mathcal{E}(e) = (e_1, \ldots, e_m)$ with $\{e_1, \ldots, e_m\} \subseteq \mathcal{D} \cap \mathsf{M}_{\mathsf{nid}}$;
   (c) if $e \in \mathsf{M}_{\mathsf{tid}}$, then $\mathcal{E}(e) = (e_1, \ldots, e_m)$ with $\{e_1, \ldots, e_m\} \subseteq \mathcal{D}$.
3. $\mathcal{L}$ is a *labeling* function $\mathcal{L} : \mathcal{D} \to 2^{\mathsf{M}_{\mathsf{tags}}}$.

Intuitively, in a map $\mathcal{M} = (\mathcal{D}, \mathcal{E}, \mathcal{L})$ the function $\mathcal{E}$ assigns to each node identifier a coordinate, to each geometry identifier a sequence of nodes, and to each tuple identifier a sequence of arbitrary identifiers.

*Example 1.* Assume we want to represent a bus route that, for the sake of simplicity, goes in a straight line from the point with coordinate $(0, 0)$ to the point with coordinate $(2, 0)$. In addition, the bus stops are at 3 locations with coordinates $(0, 0)$, $(1, 0)$ and $(2, 0)$. The names of the 3 stops are *S0*, *S1* and *S2*, respectively. This can be represented via the following map $\mathcal{M} = (\mathcal{D}, \mathcal{E}, \mathcal{L})$, where

- $\mathcal{D} = \{n_0, n_1, n_2, g, t\}$ with $\{n_0, n_1, n_2\} \subseteq \mathsf{M}_{\mathsf{nid}}$, $g \in \mathsf{M}_{\mathsf{gid}}$ and $t \in \mathsf{M}_{\mathsf{tid}}$,

[8] For clarity, we rename the expressions used in OSM

- $\mathcal{E}(n_0) = (0, 0)$, $\mathcal{E}(n_1) = (1, 0)$, $\mathcal{E}(n_2) = (2, 0)$,
- $\mathcal{E}(g) = (n_0, n_2)$ and $\mathcal{E}(t) = (g, n_0, n_1, n_2)$,
- $\mathcal{L}(n_0) = \{S0\}$, $\mathcal{L}(n_1) = \{S1\}$ and $\mathcal{L}(n_2) = \{S2\}$.

The tuple $(g, n_0, n_1, n_2)$ encodes the 3 stops $n_0, n_1, n_2$ tied to the route given by $g$.

*Enriching Maps with Computable Relations* The above formalizes the raw representation of OSM data. To make it more useful, we support incorporation of information that need not be given explicitly but can be computed from a map. In particular, we allow to enrich maps with arbitrary computable relations over $\mathsf{M}_{\mathsf{id}}$. Let $\mathsf{M}_{\mathsf{rels}}$ be an infinite set of *map relation* symbols, each with an associated nonnegative integer, called the *arity*.

An *enriched map* is a tuple $\mathcal{M} = (\mathcal{D}, \mathcal{E}, \mathcal{L}, \cdot^{\mathcal{M}})$, where $(\mathcal{D}, \mathcal{E}, \mathcal{L})$ is a map and $\cdot^{\mathcal{M}}$ is a partial function that assigns to a map relation symbol $R \in \mathsf{M}_{\mathsf{rels}}$ a relation $R^{\mathcal{M}} \subseteq \mathcal{D}^n$, where $n$ is the arity of $R$. In this way, a map can be enriched with externally computed spatial relations like the binary relations "is closer than 100m", "inside a country", "reachable from", etc. For the examples below, we assume that an enriched map $\mathcal{M}$ as above always defines the unary relation $\mathsf{Tag}_\alpha$ for every tag $\alpha \in \mathsf{M}_{\mathsf{tags}}$. In particular, we let $e \in \mathsf{Tag}_\alpha^{\mathcal{M}}$ iff $\alpha \in \mathcal{L}(e)$, where $e \in \mathcal{D}$. We will also use the binary relation $\mathsf{Inside}(x, y)$, which captures the fact that a point $x$ is located inside a geometry $y$.

## 3 A Rule Language for Data Transformation

We define a rule-based language that can be used to describe how an ABox is created from an enriched map. Our language is based on *Datalog with stratified negation* [1].

Let $\mathsf{D}_{\mathsf{rels}}$ be an infinite set of *Datalog relation symbols*, each with an associated *arity*. For simplicity, and with a slight abuse of notation, we assume that DL concept and role names form a subset of Datalog relations. Formally, we take an infinite set $\mathsf{D}_{\mathsf{concepts}} \subseteq \mathsf{D}_{\mathsf{rels}}$ of unary relations called *concept names* and an infinite set $\mathsf{D}_{\mathsf{roles}} \subseteq \mathsf{D}_{\mathsf{rels}}$ of binary relations called *role names*. Let $\mathsf{D}_{\mathsf{vars}}$ be a countably infinite set of *variables*. Elements of $\mathsf{M}_{\mathsf{id}} \cup \mathsf{D}_{\mathsf{vars}}$ are called *terms*.

An *atom* is an expression of the form $R(\boldsymbol{t})$ or $not\ R(\boldsymbol{t})$, where $R$ is a map or a Datalog relation symbol of arity $n$, and $\boldsymbol{t}$ is an $n$-tuple of terms. We call $R(\boldsymbol{t})$ and $not\ R(\boldsymbol{t})$ a *positive atom* and a *negative atom*, respectively. A *rule* $r$ is an expression of the form $B_1, \ldots, B_n \rightarrow H$, where $B_1, \ldots, B_n$ are atoms (called *body atoms*) and $H$ is a positive atom with a Datalog relation symbol (called the *head atom*). We use $\mathsf{body}^+(r)$ and $\mathsf{body}^-(r)$ for the sets of positive and negative atoms in $\{B_1, \ldots, B_n\}$, respectively. We assume *(Datalog) safety*, i.e. each variable of a rule occurs in some positive body atom. A *program* $P$ is any finite set of rules. A rule or program is *ground* if it has no occurrences of variables. A rule $r$ is *positive* if $\mathsf{body}^-(r) = \emptyset$. A program $P$ is *positive* if all rules of $P$ are positive. A program $P$ is *stratified* if it can be partitioned into programs $P_1, \ldots, P_n$ such that:

(i) If $r \in P_i$ and $not\ R(\boldsymbol{t}) \in \mathsf{body}^-(r)$, then there is no $j \geq i$ such that $P_j$ has a rule with $R$ occurring in the head.

(ii) If $r \in P_i$ and $R(\boldsymbol{t}) \in \mathsf{body}^+(r)$, then there is no $j > i$ such that $P_j$ has a rule with $R$ occurring in the head.

The semantics of a program $P$ is given relative to an enriched map $\mathcal{M}$. The *grounding* of a program $P$ w.r.t. $\mathcal{M}$ is the (ground) program $\mathsf{ground}(P, \mathcal{M})$ that can be obtained from $P$ by replacing in all possible ways the variables in rules of $P$ with identifiers occurring in $\mathcal{M}$ or $P$. We use a variant of the Gelfond-Lifschitz reduct [14] to get rid of map atoms in a program. The *reduct* of $P$ w.r.t. $\mathcal{M}$ is the program $P^{\mathcal{M}}$ obtained from $\mathsf{ground}(P, \mathcal{M})$ as follows:

(a) Delete from the body of every rule $r$ every map atom $not\ R(\boldsymbol{t})$ with $\boldsymbol{t} \notin R^{\mathcal{M}}$.

(b) Delete every rule $r$ whose body contains a map atom $not\ R(\boldsymbol{t})$ with $\boldsymbol{t} \in R^{\mathcal{M}}$.

Observe that $P^{\mathcal{M}}$ is an ordinary stratified Datalog program with identifiers acting as constants. We let $PM(\mathcal{M}, P)$ denote the *perfect model* of the program $P^{\mathcal{M}}$. See [1] for the construction of $PM(\mathcal{M}, P)$ by fix-point computation along the stratification. We are now ready to extract an ABox. Given a map $\mathcal{M}$ and a program $P$, we denote by $\mathsf{ABox}(\mathcal{M}, P)$ the restriction of $PM(\mathcal{M}, P)$ to the atoms over concept and role names.

We next illustrate some features of our rule language. The basic available service is to extract instances of concepts or roles by posing a standard conjunctive query over an OSM map. F.i., the following rule collects in the role `hasCinema` the cinemas of a city (we use sans-serif and typewriter font for map and Datalog relations, respectively):

$$\mathsf{Point}(x), \mathsf{Tag}_{\mathsf{cinema}}(x), \mathsf{Geom}(y), \mathsf{Tag}_{\mathsf{city}}(y), \mathsf{Inside}(x, y) \rightarrow \mathtt{hasCinema}(y, x).$$

Negation in rule bodies can be used for default, closed-world conclusions. E.g., the rule states that recreational areas include all parks that are not known to be private:

$$\mathsf{Geom}(x), \mathsf{Tag}_{\mathsf{park}}(x), not\ \mathsf{Tag}_{\mathsf{private}}(x) \rightarrow \mathtt{RecreationalArea}(x)$$

Another use of negation is for data cleaning purposes. Indeed, one of the major problems in OSM is duplicate information about a single real-life object. For an example, assume we are extracting shops and thus collecting points tagged with *Store*. However, due to the possible duplication, we want to omit points that are close ($\leq$10m) to a point with the more informative tag *GroceryStore*. This can be easily expressed with the following rules, where $\mathsf{next}_{10m}$ is an external relation for pairs of points with 10m distance:

$$\begin{aligned} \mathsf{next}_{10m}(x, y), \mathsf{Point}(y), \mathsf{Tag}_{\mathsf{GroceryStore}}(y) &\rightarrow \mathtt{hasCloseStore}(x) \\ \mathsf{Point}(x), \mathsf{Tag}_{\mathsf{Store}}(x), not\ \mathtt{hasCloseStore}(x) &\rightarrow \mathtt{Shop}(x). \end{aligned}$$

Recursion is also useful and e.g., allows to deal with reachability, which appears naturally and in many forms in the context of geographic data. E.g. suppose we want to collect pairs $b_1, b_2$ of bus stops such that $b_2$ is reachable from $b_1$ using public buses. To this end, we can assume the availability of an external binary relation $\mathsf{hasStop}$ which relates bus routes and their stops, i.e. $\mathsf{hasStop}(x, y)$ is true in case $x$ is a geometry identifier corresponding to a bus route and $y$ is a point identifier corresponding to a bus stop in the route represented by $x$. Then the desired pairs of bus stops can be collected in the role `ReachByBus` using the following recursive rules:

$$\begin{aligned} \mathsf{hasStop}(x, y_1), \mathsf{hasStop}(x, y_2) &\rightarrow \mathtt{ReachByBus}(y_1, y_2) \\ \mathtt{ReachByBus}(y_1, y_2), \mathtt{ReachByBus}(y_2, y_3) &\rightarrow \mathtt{ReachByBus}(y_1, y_3). \end{aligned}$$

## 4 Extending the Rule Language with ETL Features

In this section we introduce the custom language for the benchmark generation, which *extends* the Datalog language of the previous section with *extract*, *transform*, and *load* (ETL) features. The combined language consequently consisting of *Data Source Declarations*, *Mapping Axioms*, and *Datalog Rules*.

We address simplicity and extensibility as our main goal on two levels. The first level concerns the extensibility of the mapping language and the related evaluation method by providing the following modes: A *Direct mode* which resembles a simple *extract*, *transform*, and *load* (ETL) process of classical data transformation tools. Based on the *Direct mode*, we offer a *Datalog mode* which includes the rule language from the previous section. The second level concerns data sources (e.g., OSM databases) and external evaluation (e.g., calculating spatial relations). In simple cases, we do not need to deal with implementation details, as the data is read from OSM tables stored in an (geospatial) RDBMS and the results are mapped directly to the output files.

**Data Source Declarations**. The first section of a benchmark generation definition contains general declarations like RDBMS connection strings. The following declarations are available at present, whereby every definition has an *identifier* denoted as `id` which is used for the referencing in the mapping axioms. We allow declarations for *PostgreSQL/PostGIS connections*, *Text files*, *Python scripts*, and *Constants*.

**Mapping Axioms**. A mapping axiom defines a single ETL step, where the syntax is an extension of the Ontop mapping language. [9] It is defined either as a pair of *source* and *target* or as a triple of *source*, *transform*, and *target*:

```
Source      source_id      source_parameter
Transform   transform_id   transform_parameter
Target      target_id      target_parameter
```

where the first column is constant, the second column refers to the data source declarations, and the third column has to be configured according to the source, target, or transformation, respectively. The `Transform` is intended for adding scripts to provide a mapping of values from the source to the target (see Section 5).

**Available Sources**. The following sources are currently available: [10]
*PostgreSQL/PostGIS Queries*: In this case, the `source_parameter` defines an SQL *Select-From-Where* statement which is executed on the referenced database. The result is processed as a set of tuples, which can be accessed by its index in the *target* step.

*Text Files*: The referenced file in `source_id` is read line-by-line and converted into tuples using a field delimiter (e.g., a semicolon). In case the `source_parameter` is defined, only the lines fulfilling its regular expression are returned.

*RDF Files*: A SPARQL query in `source_parameter` is computed over the referenced file. As with SQL queries, the result is converted into tuples, accessible by its index.

*Constants*: Constants are the simplest sources. They are defined by `Source constant id1 id2 etc.` and directly written to the target.

[9] https://github.com/ontop/ontop/wiki/ontopOBDAModel

[10] Text and RDF files are needed as sources since in a large generation process, intermediate results are kept as RDF triples

*Example 2.* We read all the parks from a OSM database defined in the data sources:

```
Source osmVienna Select osm_id, name, ST_AsEWKT(way) AS geo From
  planet_osm_polygon Where leisure = 'park'.
```

**Available Targets**. For producing the results the following targets are available:
*PostgreSQL/PostGIS Tables*: The definition is similar to the PostgreSQL sources, however the `target_parameter` is an SQL *Insert* statement. The later should be a template, where the tuples from the sources are referenced by its index (in curly brackets).
*Text Files*: Text files are linked similarly as in the sources. However, `target_parameter` has to be a textual template representing the result. Usually, the template contains multiple lines and represents triples that are defined according to the benchmark ontology.
*Stdout*: The target is similar to text files, but the result is written directly to the standard output stream (Stdout).

*Example 3.* For instance, we write three triples to a file which represent an instance of type `Playground` and `NamedIndividual` with a name and a polygon assigned to it:

```
Target file1 :{1} rdf:type tuwt:Playground, owl:NamedIndividual;
  gml:featurename "{2}"^^xsd:string;
  geo:polygon "{3}"^^xsd:string.
```

**Datalog Rules**. For the evaluation in the Datalog mode, one needs first to create the Extensional Database (EDB) atoms (facts) with a previous ETL step. Then, the rules, also called Intensional Database (IDB), have to be defined by external files written in (disjunctive) Datalog. In our case, the syntax of the Datalog rules is taken from DLV, since DLV is our main evaluation engine. [11] The EDB and IDB have to be defined into the mapping as a source by `Source evaluateDatalog fileEDB fileIDB`. The result is a tuple of the form $(predicate, value_1, ..., value_n)$. and should be be combined with `Transform` and `Target`.

*Example 4.* The following rules are the DLV encoding of the example from Section 2:

```
reachByBus(X,Z) :- hasStop(X,Y), hasStop(Z,Y).
reachByBus(X,Z) :- reachByBus(X,Y), reachByBus(Y,Z).
```

The above rules are defined in file_edb and are referenced as follows:

```
Source evaluateDatalog file_edb file_idb
Target file1 :{2} tuwt:reachedBy :{3}.
```

## 5 Benchmarking Framework

The extended rule language $\mathcal{L}$ of the previous section gives us solely the means to define the data transformations. We combine the language with an OSM database $\mathcal{S}$, a benchmark ontology $\mathcal{O}$, a set $\mathcal{Q}$ of conjunctive or SPARQL queries, a set $\mathcal{P}$ of generation parameters, and external functions $\mathcal{F}$. The combination leads to the benchmark framework denoted as $\mathcal{F} = \langle \mathcal{S}, \mathcal{O}, \mathcal{Q}, \mathcal{L}, \mathcal{P}, \mathcal{F} \rangle$ and produces a TBox $\mathcal{T}$ and a set of ABox instances denoted as $\mathcal{A} = (A_1, \ldots, A_n)$. Note that $\mathcal{T}$ is build from $\mathcal{O}$ and (normally

[11] http://www.dlvsystem.com/dlv/

small) modifications done by $\mathcal{F}$. Further, we need a clear *workflow* to clarify how to apply the framework for instance data generation.

**Workflow**. The workflow of creating a benchmark and evaluating the respective reasoners can be split into an *initial* and a *repeating* part. In case of solely TBox reasoning (e.g., subsumption with respect to a TBox), the repeating part can be ignored. The initial part consists of the following elements:

First, one has to choose the ontology for $\mathcal{O}$ and decide which ontology language should be investigated. The *ontology statistics* gives a first impression on the *expressivity* of the language such as DL-Lite$_R$ [7] or $\mathcal{EL}$ [2]. Then, $\mathcal{O}$ has to be customized (e.g., remove axioms which are not in the language) and loaded to the TBox $\mathcal{T}$ of the system. For $\mathcal{Q}$, either handcrafted queries (related to a practical domain) have to built or synthetic queries have to be generated. [12] After the initial part is finished, we are able to generate the instance data for the fixed $\mathcal{O}$ and $\mathcal{Q}$. This part of the workflow can be repeated until certain properties are reached. It has the following steps:

1. Creating an OSM database $\mathcal{S}$ with several instances, i.e., cities or countries; [13]
2. Applying *dataset statistics* to get a broad overview of the dataset, which leads to the selection of "interesting" datasets from $\mathcal{S}$;
3. Creating the rules of $\mathcal{L}$ to define the needed transformation for the instance generation from the datasets;
4. Defining the parameters $\mathcal{P}$ and choosing the needed external functions of $\mathcal{F}$;
5. Calling the generation toolbox (see Section 6) and create the instances of $\mathcal{A}$;
6. Using *ABox statistics* to evaluate $\mathcal{A}'s$ quality, if not satisfactory, repeat from 3.;
7. Load the instances of $\mathcal{A}$ to the KB of the systems and apply necessary conversions;
8. Finally, the benchmark can be evaluated on the tested reasoners, measuring the following attributes: *evaluation time, loading time, memory consumption, completeness,* and *evaluated query size*. These attributes are defined previously by the person doing the testing.

**Descriptive Statistics**. For the benchmark creation, descriptive statistics serves two purposes. First, we need a broad picture of the datasets, which is important to formulate the mapping rules. E.g., we could see what kind of supermarkets exist in a city. Second, we use the statistics to guide and fine tune the instance generation. I.e., for generating the `next` relation, different distances can be calculated leading to different sizes of $\mathcal{A}$.

Descriptive statistics can be applied on the following three levels. On the *ontology level*, ontology metrics regarding $\mathcal{O}$ can be produced using *owl-toolkit* [14] to calculate the number of concepts, roles, and the different types of axioms (e.g., sub-concept, sub-role, and, inverse roles). On the *dataset level*, we provide general information on the selected OSM DB instance including the main constructs *Points*, *Lines*, *Roads*, and *Polygons* and details about *frequent item sets* [6] of keys and tags (e.g. landuse=forest). [15] On the *ABox level* we provide the statistics of the generated instances in $\mathcal{A}$. For this, we count the assertion for every existing concept or role name of $\mathcal{T}$.

[12] We refer to Sygenia [16] as a possible generation tool

[13] See Prerequisites and Tools in https://github.com/ghxiao/city-bench

[14] https://github.com/ghxiao/owl-toolkit

[15] Statistics of all tags and keys is available on https://taginfo.openstreetmap.org/

Further, we provide the assertion statistics of $\mathcal{A}$ by stating which instances are asserted to the concept and role hierarchies of $\mathcal{T}$ *under the deductive closure*, denoted as $closure(\mathcal{T} \cup \mathcal{A})$. We extend the notion of a *subsumption graph* defined for DL-Lite$_R$ in [22]. The DL-Lite$_R$ notion cannot be used for $\mathcal{EL}$ [2] and more expressive DLs, since a *hyper-graph* is needed to capture axioms like $\exists R.A \sqsubseteq B$ (see [23]). Our notion of the *subsumption graph* is built from a *normalized* TBox $\mathcal{T}_\mathcal{N}$ (we refer to [3]). $\mathcal{T}_\mathcal{N}$ contains general concept inclusions (GCIs) of the form $A \sqsubseteq B$, $A_1 \sqcap A_2 \sqsubseteq B$, $A \sqsubseteq \exists R.B$, and $\exists R.A \sqsubseteq B$, where $A$, $A_1$, $A_2$, $B$ are atomic concepts and role inclusions (RI) of the form $R_1 \sqsubseteq R_2$.

A *subsumption graph* for $\mathcal{T}_\mathcal{N}$ is a directed graph $G_\mathcal{T} = (V, E, L)$, where $L$ is the set of labels $a_v \in L(v)$, where $a_v$ represent the number of instance assertions over $A_v$ resp. $R_v$ under $closure(\mathcal{T} \cup \mathcal{A})$. The vertices $V$ represent either concept or role names of $\mathcal{T}_\mathcal{N}$. The edges $E$ represent the inclusion axioms as follows: If there is a GCI or RI of the form $A \sqsubseteq B$, where $A$ and $B$ are concept resp. role names, we create an edge $e(A, B)$. In case of $A \sqcap B \sqsubseteq C$ resp. $\exists R.A \sqsubseteq B$, we create two edges of the form $e(A, C)$ and $e(B, C)$ resp. $e(R, B)$ and $e(A, B)$. Note that we split the hyper-edge as defined in [23] into two normal graph edges, this would lead to an over-count of instance assertions, which is overcome by (3) described in the next paragraph. In case of $A \sqsubseteq \exists R.B$, we create two edges of the form $e(A, R)$ and $e(A, B)$.

Then, we calculate the assertion statistics of $\mathcal{T}$ as follows. Normalize $\mathcal{T}$ to $\mathcal{T}_\mathcal{N}$ and translate $\mathcal{T}_\mathcal{N}$ into the *subsumption graph* $G_\mathcal{T}$. For each vertex $v \in V$ get either $A_v$ or $R_v$ and calculate $a_v \in L(v)$ as follows. We start with vertices $v \in V$ which have only outgoing edges (sources) and traverse $G_\mathcal{T}$ using breadth first search. For each vertex $v$ proceed as follows:

1. $v$ is a source: count the instance assertions of $A_v$ or $R_v$ in $\mathcal{A}$ and assign it to $a_v$;
2. $v$ has one incoming edge $e(A, B)$: add $a_A \in L(A)$ to $a_B \in L(B)$;
3. $v$ has multiple incoming edges: we use a "standard" OQA system for instance retrieval, such that $\mathcal{T} \cup \mathcal{A} \models C_v(a)$ resp. $\mathcal{T} \cup \mathcal{A} \models R_v(a, b)$ for every $a \in \mathcal{A}$ resp. $(a, b) \in \mathcal{A}$; then we count the number of entailments and assign it to $a_v$.

For future work, we aim to avoid in (3) calling an OQA system and we will build our own instance estimation based on sampling and the information about the selectivity of assertions between different concepts and roles.

**External Functions and Parameters**. External functions bridge the gap between $\mathcal{L}$ and external computations. They allow us to develop dataset-specific customization and functionalities, where the results (atoms) are associated with predicates from $\mathcal{L}$. In Table 1, we list the currently available external functions. In addition, we provide the functions *deleteRandom* and *deleteByFilter* which drop instances randomly or filter out instances from $\mathcal{A}$.

The parameters are the means to fine-tune the generation. They are often not directly observable, hence we need the statistics tool to get a better understanding of data sources. From recent literature [30, 24, 5], we identified the following parameters for the instance generation:

- *ABox Size*: choice of the OSM instance (e.g., major cities or countries), but also by applying *deleteRandom* and *deleteByFilter*;

Table 1: Available External Functions

| Name | Description | Associated Predicate |
|---|---|---|
| *transformOSM* | generates from OSM or other tags (e.g., landuse=park) atoms which represent concepts/roles of $\mathcal{O}$. It has to be customized to the signature of $\mathcal{O}$. | $\mathsf{Tag}_{\mathsf{Park}}(x)$ |
| *transformOSM-Random* | instead of generating directly from OSM tags, it generates atoms according to a probabilities $P$ assigned to a set $O$ of OSM tags, e.g., $P(PublicPark)$=0.8 and $P(PrivatePark)$=0.2. | $\mathsf{Tag}_{P,O}(x)$ |
| *generateRandom-Values* | simply generates random numeric values from a fixed domain. | $\mathsf{Random}(x)$ |
| *generateSpatial-Relation* | generates the spatial relations `contains` or `next`, where we need a threshold parameter for the distance between objects. | $\mathsf{next}_{10m}(x, y)$ |
| *generateStreet-Graph* | generates the road/transport graph by creating instances for *edges* and *vertices* based on streets and corners between them. | $\mathsf{connected}(x, y)$ $\mathsf{Tag}_{\mathsf{corner}}(x)$ |

- *ABox Completeness:* can be indirectly manipulated by the use of Datalog rules in $\mathcal{L}$ to generate instances which otherwise would be deduced. As shown in Example 4, we generate the transitive closure of the main roads;
- *Distribution/Density of Nominal Values:* input for *transformOSMRandom*;
- *Distribution/Density of Numeric Values:* input for *generateRandomValues*;
- *Selectivity of Concept/Role Assertions:* input for *transformOSM* and *transformOSM-Random* and choice of OSM instances;
- *Graph Structure:* choice of OSM instance and selected graph (e.g., road vs. public transport network) for *generateStreetGraph*.

## 6 Implementation

We have developed for the framework a generation toolbox in Python 2.7. The main script generate.py is called as follows: generate.py –mappingFile mapping.txt

**Modes**. As already mentioned, we provide two different modes with different evaluation strategies. The *Direct mode* is designed for simple bulk processing, where scalability and performance is crucial and complex calculations are moved to custom external scripts. We implemented the computation in a *data streaming*-based manner. Therefore, the data source components represent *iterators*, such that only a single tuple at a time is in-memory. External source scripts can be implemented as custom iterators, by returning result tuples with the command yield. The target components could also be extended to data writers for custom triple stores like *Jena TDB*. The mapping axioms are evaluated in sequential order, hence dependencies between sources and targets are not considered. The *Datalog mode* extends the Direct mode and is designed for Datalog programs using the DLV system for evaluation. The Datalog results are calculated in-memory and we follow a three-layered computation:

1. Previous ETL steps are evaluated to create the fact files for the EDB;
2. The defined Datalog programs (maintained in external files) are evaluated on the EDB files with the DLV module;

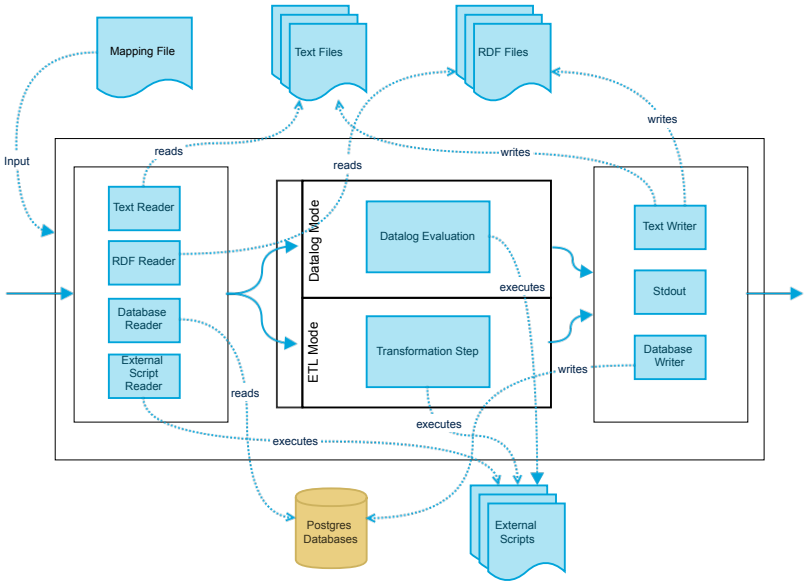


Fig. 1: System architecture, full lines are the control and dotted lines are the data flow

3. The (filtered) results (i.e., perfect models) are parsed and converted to tuples which then can be used by any target component. At the time of writing, we only have a single model due to our restriction to stratified Datalog.

**Architecture**. In Figure 1, we show the architecture of the framework. It naturally results from the two modes and the source and target components. The following source and target components are implemented. For *Text files*, we use the standard functions of python for reading, writing, and evaluating regular expressions. For *RDF files*, which are accessed by SPARQL queries, we leverage the functions of the *rdflib* library. At present for *RDBMSs*, we only include access to the spatial-extended RDBMS PostGIS 2.12 (for PostgreSQL), which is the most common database system for OSM. However, the database component is easily extendible to other RDBMSs.

**External Functions and Statistics**. Besides the main script, we provide the external functions from Section 5 developed as Python scripts for processing the (OSM) data.

GenerateSpatialRelation.py calculates the spatial relations reading from RDF files, which have to contain triples including *GeoRSS* (e.g., geo:point). GenerateSpatial RelationDatabase.py provides a more efficient way for large data sets using PostGIS directly to calculate the spatial relations between two sources.

GenerateStreetGraph.py processes the street and transport graph of OSM. From the street/transport network all nodes and edges are extracted and merged into a single graph. We exploit that in the model of OSM that the connected streets share the same *point* when they cross. Further, we connect other objects (e.g., shops) to the closest *points* on the street graph by computing the `next` relation with a distance of 50m.

StatsOSM.py and StatsABox.py are the statistical scripts for estimating the structure of the ABox and the main OSM elements *Points*, *Lines*, *Roads*, and *Polygons*. It

Table 2: TBox Metrics

| Metric | # | Metric | # |
|---|---|---|---|
| *Concepts* | 350 | *Inverse Role Axioms* | 4 |
| *Roles* | 38 | *Domain Role Axioms* | 22 |
| *Sub-Concept Axioms* | 354 | *Range Role Axioms* | 24 |
| *Disj. Concept Axioms* | 1 | *Sub Concept Depth* | 7 |
| *Sub-Role Axioms* | 22 | *Sub Property Depth* | 3 |

Table 3: Cities Dataset

| City | $\#_{Points}$ | $\#_{Lines}$ | $\#_{Polygons}$ |
|---|---|---|---|
| *Cork* | 6 068 | 14 378 | 4 934 |
| *Riga* | 19 172 | 43 042 | 67 708 |
| *Bern* | 68 831 | 83 351 | 151 195 |
| *Vienna* | 245 107 | 151 863 | 242 576 |
| *Berlin* | 236 114 | 218 664 | 430 652 |

Table 4: Road Network Instances

| City | $\#_{Road}$ | $\#_{Node}$ | $\#_{connect}$ | $\#_{Shop}$ | $\#_{Bank}$ | $\#_{hasBankOp}$ | $\#_{next_{50}}$ |
|---|---|---|---|---|---|---|---|
| *Cork* | 6 476 | 45 459 | 46 013 | 278 | 36 | 36 | 750 |
| *Riga* | 6 620 | 35 107 | 37 007 | 827 | 102 | 102 | 1 408 |
| *Bern* | 17 995 | 130 849 | 134 670 | 1 539 | 120 | 120 | 10 285 |
| *Vienna* | 40 915 | 191 220 | 207 429 | 5 259 | 506 | 506 | 23 151 |
| *Berlin* | 46 320 | 204 342 | 226 554 | 9 791 | 588 | 588 | 81 911 |

calculates the values for the most used field/tag combination, e.g. field *Amenity* has 50 "Restaurant" and 20 "Fuel" tags. Additionally, we find the most frequent item sets using the *FP-Growth* algorithm. [16] For the ABox, we use *rdflib* library to count and report all concept and role assertions (under closure).

## 7 Example Benchmark

In this section we introduce a proof-of-concept benchmark to show how extensible and parameterizable the framework is, in order to generate challenging benchmarks for OQA systems. All the mapping files, test datasets, and statistics are available online. [17]

**OSM Dataset, Benchmark Ontology, and Queries**. The whole OSM database (ca. 400GB) is clearly too large for a benchmark. However, it offers different subsets of different sizes and structures since OSM is a collaborative project. For the base dataset we chose the cities of *Cork*, *Riga*, *Bern*, *Vienna*, and *Berlin*. [18] Using the dataset statistics module for *Vienna*, we observe for the field *Shop* has 816 supermarkets, 453 hairdressers, and 380 bakeries related. For the field *Highway*, we have 29 392 residential, 4 087 secondary, and 3 973 primary streets with several edges.

The selected DL-Lite$_R$ [7] ontology for the benchmarks is taken from the MyITS project [11]. It is tailored to geospatial data sources and beyond to MyITS specific sources (e.g., a restaurant guide). The metrics of the ontology is shown in Table 2. The ontology is for OQA systems of average difficulty, because it has only a few existential quantification on the right-hand side of the inclusion axioms. However, due to its size

[16] https://github.com/enaeseth/python-fp-growth

[17] https://github.com/ghxiao/city-bench

[18] Downloaded on the 1.10.14 from http://download.bbbike.org/osm/bbbike/ and loaded with osm2pgsql into PostGIS

and concept and role hierarchy depth, it poses a challenge regarding the rewritten query size to some systems. We choose an exemplary query $q_1$ which is based on the extracted road graph. It queries the combination of the street graph, spatial relations and concept hierarchies and computes all the small banks which are connected by two edges to a shop. Only the residential street edges of the graph are queried.

$$\begin{aligned}\mathtt{q_1}(x,y) \leftarrow\ & \mathtt{Shop}(x), \mathtt{next}(x,u), \mathtt{Point}(u), \mathtt{path}(u,v),\\ & \mathtt{path}(v,w), \mathtt{Point}(w), \mathtt{next}(y,w), \mathtt{Bank}(y), \mathtt{hasOperator}(y,z),\\ & \mathtt{BankSmallOp}(z), \mathtt{isPartOf}(u,t), \mathtt{ResidentialRoute}(t)\end{aligned}$$

**Creation of the Street Network Benchmark**. In this benchmark, we extract the road network of the mentioned cities using the external function *generateStreetGraph*. Besides creating the concept assertions for banks and shops, we extract the entire street network of the city and encode the different roads into a single *road graph*. The road graph is represented by *nodes* which are asserted to the concept `Point`, and *edges* which are asserted to the role `connected`. By increasing the distances (e.g., from 50m to 100m) we could saturate the `next` relation and generate more instances. We also illustrate the generation of additional instances by using Datalog to calculate all paths (i.e. the transitive closure) of the street graph using:

$$\begin{aligned}\mathtt{connected}(x,y) &\rightarrow \mathtt{path}(x,y)\\ \mathtt{path}(x,y), \mathtt{path}(y,z) &\rightarrow \mathtt{path}(x,z).\end{aligned}$$

Note that this rule calculates the transitivity closure of selected parts of the graph, which saturates the ABox with instances which cannot be deduced by a DL-Lite$_R$ reasoner.

We calculated the ABox statistics shown in Table 3, the cities are of increasing size, starting with *Cork* containing 25 000 objects and ending with *Berlin* having 885 000 objects. All these cities are European major cities with an old city center; they have high data density of objects in the center and decreasing density towards the outskirts.

## 8 Related Work

In addition to the "de facto" standard benchmark LUBM [15] and extended LUBM [24] with randomly generated instance data with a fixed ontology, several other works deal with testing OQA systems which can be divided along conceptional reasoning, query generation, mere datasets, synthetic and real-life instance generation. The benchmarks provided by Perez-Urbina et al. [27] consist of a set of ontologies and handcrafted queries, tailored for testing query rewriting techniques. These benchmarks are a popular choice for comparing the sizes of generated queries. Recently Stoilos et al. [30] have provided tools to generate ABoxes for estimating the incompleteness of a given OQA system. In a similar spirit, Imprialou et al. [16] design tools to automatically generate conjunctive queries for testing correctness of OQA systems. The same authors also provide a collection of benchmarks for evaluating query rewriting systems. They did not offer any novel generation tool [25]. The NPD benchmark [21] for OBDA is designed based on real data from the Norwegian Petroleum Directorate FactPages. However, the focus of the NPD benchmark is solely on a fixed DL-Lite$_R$ ontology and queries. None of the above benchmarks provide large amounts of real-life instance data and an

extended framework including various parameters and external functions. Furthermore, most of the mentioned approaches do not consider an iterative generation process using statistics to guide the generation.

In the area of Spatial Semantic Web systems, a couple of benchmarks have been proposed to test geospatial extensions of SPARQL including the spatial extension of LUBM in [18] and the *Geographica* benchmark [13]. These benchmarks were precomputed by the authors and cannot be easily modified. They are geared towards testing spatial reasoning capabilities of systems, but not designed with OQA in mind. We note that the mapping language of this paper is a close relative of R2RML, a language proposed by W3C for mapping relational data to RDF triples. Our language lacks some features of R2RML, but is equipped with powerful means for benchmark generation, as calls to external functions and Datalog evaluation.

## 9 Conclusion and Outlook

In this paper, we have presented a flexible framework for generating instance data from a geospatial database for OQA systems. In particular, we have introduced a formalization of OSM and a Datalog-based mapping language as the formal underpinning of the framework. Datalog offers convenient features such as recursion and negation, which are useful for benchmark generation. We have implemented an instance generation tool supporting the main *Datalog* mode and a simple *Direct* (extract-transform-load) mode for several types of input sources. Finally, we have demonstrated our approach on a proof-of-concept benchmark.

Future research is naturally directed to variants and extensions of the presented framework. We aim to extend the implementation to capture more input and output sources, further parameters (e.g. various degrees of graph connectedness) and services. Furthermore, a tighter integration of the Datalog solver/engine and the source/target components using dlvhex [10] is desired, which leads to a more efficient evaluation and more advanced capabilities (e.g., creating different ABoxes using all calculated answer sets). An important functional extension would be to allow removing some or all assertions that are implied by the input ontology, in this way the information incompleteness could be better controlled. Then, we aim to apply our framework to generate benchmarks for an extensive study of different OQA reasoners with different underlying technologies. Further, we aim to extend the assertion statistics for instances regarding the concept and role hierarchies. In particular, we aim to replace the used OQA reasoner with our own estimation mechanism.

## References


1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. F. Baader, S. Brand, and C. Lutz. Pushing the $\mathcal{EL}$ envelope. In *Proc. of IJCAI 2005*, pages 364–369. Morgan-Kaufmann Publishers, 2005.
3. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. 2nd edition, 2007.
4. S. Bail, B. Parsia, and U. Sattler. Justbench: A framework for OWL benchmarking. In *Proc. of ISWC 2010*, pages 32–47. Springer, 2010.

5. C. Bizer and A. Schultz. The berlin SPARQL benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.
6. C. Borgelt. Frequent item set mining. *Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery*, 2(6):437–456, 2012.
7. D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The *dl-lite* family. *J. Autom. Reasoning*, 39(3):385–429, 2007.
8. T. Eiter, M. Fink, and D. Stepanova. Computing repairs for inconsistent dl-programs over $\mathcal{EL}$ ontologies. In *Proc. of JELIA 2014*, pages 426–441, 2014.
9. T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the Semantic Web. *Artificial Intelligence*, 172(12-13):1495–1539, 2008.
10. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. Effective integration of declarative rules with external evaluations for semantic-web reasoning. In *ESWC 2006*, volume 4011, pages 273–287. Springer, 2006.
11. T. Eiter, T. Krennwallner, and P. Schneider. Lightweight spatial conjunctive query answering using keywords. In *Proc. of ESWC 2013*, pages 243–258, 2013.
12. T. Eiter, P. Schneider, M. Simkus, and G. Xiao. Using openstreetmap data to create benchmarks for description logic reasoners. In *3rd International Workshop on OWL Reasoner Evaluation (ORE 2014)*, July 2014.
13. George Garbis, Kostis Kyzirakos, and Manolis Koubarakis. Geographica: A benchmark for geospatial rdf stores (long version). In *Proc. of ISWC 2013*, pages 343–359. Springer, 2013.
14. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080, 1988.
15. Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics*, 3(2-3):158 – 182, 2005.
16. M. Imprialou, G. Stoilos, and B. Cuenca Grau. Benchmarking ontology-based query rewriting systems. In *Proc. of AAAI 2012*, 2012.
17. Y. Kazakov, M. Krötzsch, and Simancík F. Concurrent classification of $\mathcal{EL}$ ontologies. In *Proc. ISWC 2011*, pages 305–320, Berlin, Heidelberg, 2011. Springer.
18. D. Kolas. A benchmark for spatial semantic web systems. In *4th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2008)*, October 2008.
19. I. Kollia and B. Glimm. Optimizing SPARQL query answering over OWL ontologies. *J. Artif. Intell. Res. (JAIR)*, 48:253–303, 2013.
20. R. Kontchakov, M. Rezk, M. Rodriguez-Muro, G. Xiao, and M. Zakharyaschev. Answering SPARQL queries over databases under OWL 2 QL entailment regime. In *Proc. of ISWC 2014*. Springer, 2014.
21. D. Lanti, M. Rezk, G. Xiao, and D. Calvanese. The NPD benchmark: Reality check for OBDA systems. In *Proc. of EDBT 2015*. ACM Press, 2015.
22. D. Lembo, V. Santarelli, and D. F. Savo. Graph-based ontology classification in OWL 2 QL. In *Proc. of ESWC 2013*, pages 320–334, 2013.
23. M. Ludwig and D. Walther. The logical difference for $\mathcal{ELH}$-terminologies using hypergraphs. In *Proc. of ECAI 2014*, pages 555–560, 2014.
24. L. Ma, Y. Yang, Z. Qiu, G. Xie, Y. Pan, and S. Liu. Towards a complete OWL ontology benchmark. In *Proc. of ESWC 2006*, pages 125–139. Springer.
25. J. Mora and O. Corcho. Towards a systematic benchmarking of ontology-based query rewriting systems. In *Proc. of ISWC 2013*, pages 376–391. Springer, 2013.
26. W3C OWL Working Group. *OWL 2 Web Ontology Language: Document Overview*. W3C Recommendation, 27 October 2009.
27. H. Pérez-Urbina, I. Horrocks, and B. Motik. Efficient query answering for OWL 2. In *Proc. of ISWC 2009*, pages 489–504, 2009.

28. E. Sirin, B. Parsia, B. Cuenca Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *J. Web Sem.*, 5(2):51–53, 2007.
29. G. Stoilos, B. Glimm, I. Horrocks, B. Motik, and R. Shearer. A novel approach to ontology classification. *Web Semantics: Science, Services and Agents on the WWW*, 14(0), 2012.
30. G. Stoilos, B. Cuenca Grau, and I. Horrocks. How incomplete is your semantic web reasoner? In *Proc. of AAAI 2010*. AAAI Press, 2010.
31. D. Tsarkov and I. Horrocks. FaCT++ description logic reasoner: System description. In *Proc. of IJCAR 2006*, pages 292–297, Berlin, Heidelberg, 2006. Springer.