EU Project No:601043 (Integrated Project (IP))

DIACHRON

Managing the Evolution and Preservation of the Data Web DIACHRON

Dissemination level:	
Type of Document:	
Contractual date of delivery:	
Actual Date of Delivery:	
Deliverable Number:	
Deliverable Name:	
Deliverable Leader:	
Work package(s):	
Status & version:	
Number of pages:	
WP contributing to the deliverable:	
WP / Task responsible:	
Coordinator (name / contact):	
Other Contributors:	
EC Project Officer:	Federico Milani
Keywords:	
Abstract:	
Some meaningful abstract here.	

Document History			
Ver.	Date	Contributor(s)	Description
0.1	24.06.2014	Jeremy Debattista	Created TOC and LaTeX Setup

Table of Contents

Table of Figures

List of Tables

1 Introduction

1.1 Scope and Objectives

1.2 Context of this Document

1.3 Document Structure

2 Data Quality Framework

2.1 High-Level Architecture

The purpose of the Quality Framework is to provide an integrated platform that:

1. assesses RDF datasets and triple stores in a scalable manner;
2. provides queryable quality metadata on the assessed datasets;
3. provides visualisations on the quality

Furthermore, we aim to create an infrastructure and a platform that (i) can be easily extensible by different third party by creating their custom and more specific pluggable metrics required to assess their particular dataset domain, and (ii) having the the necessary ontology framework to represent the metadata about the quality of the assessed linked datasets.

Currently, there is no uniform infrastructure to address the quality assessment problem, allowing the extension or redefinition of custom-specific metrics such as those required by the DIACHRON use cases. Tools such as Trellis **??**, WIQA **??** and Sieve **??** implement a number of metrics but lacked flexibility wrt the level of automation, and user friendliness **??**.

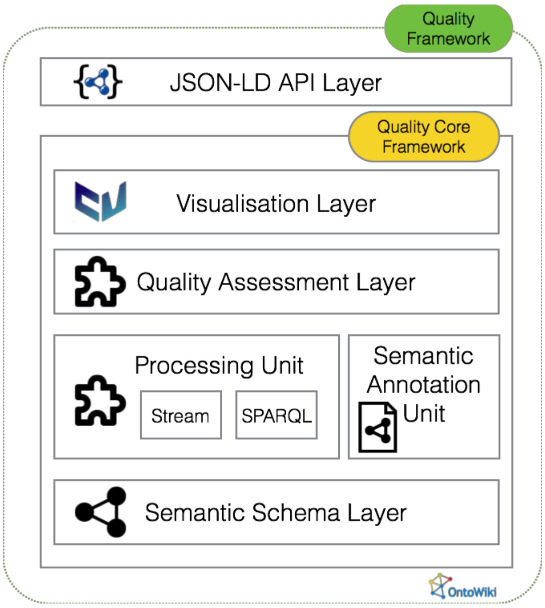

Figure 1: Quality Framework High Level Architecture Design

Figure 1 illustrates the high level architecture of the Quality Framework. The two main components are the API layer (cf. Section 2.4) and the Core framework. The core framework is made up of five modules: *Semantic Schema Layer*, *Processing Unit*, *Semantic Annotation Unit*, *Quality Assessment Layer*, and *Visualisation Layer*.

The quality framework is envisioned to run on top of OntoWiki[1], a wiki based on semantic technologies. The current technologies and extensions available in OntoWiki allow us to reach both the main objectives in this deliverable, and eventually to commence our initial prototypes of the quality framework (e.g. as an enterprise add-on tool to linked data publishers). Further investigation upon the usability of OntoWiki is still necessary in order to discover if the mentioned application can fulfil further the quality framework's ambitions, beyond the initial prototypes.

2.1.1 Semantic Schema Layer

The Quality Framework is based on semantic technologies and thus has an underlying semantic vocabulary layer which currently is made up of two ontologies: (i) the Dataset Quality Ontology (daQ)[2]; and (ii) the Quality Problem Report Ontology (qr)[3]. The former describes the quality metadata representation whilst the latter describes quality problems found in the dataset itself. The semantic schema layer is meant to be domain independent, where it could be reused in other similar frameworks. The daQ ontology (cf. Section 2.3) is the core vocabulary of this schema layer, and any other ontology part of this layer builds upon it.

The daQ ontology is a comprehensive generic vocabulary framework, based on three abstract concepts (Category, Dimension and Metric). Any newly implemented specific metric should have its representation in RDF, extending the daQ ontology. In DIACHRON, all metrics are defined in the Diachron Quality Metric vocabulary (dqm)[4]. Such vocabularies are easily integrated in the Quality Framework, since they adopt and extend the generic daQ vocabulary (by inheriting class and properties) as the way quality metadata is represented (cf. Section 2.3.3). The Quality Problem Report Ontology (qr) is made up of two classes a `qr:QualityReport` and `qr:QualityProblem`. The former represents a report on the problems detected during the assessment of quality on a dataset, whilst the latter represents a quality problem detected during the assessment of quality metrics on triples. Four properties are also defined in the ontology. The `qr:computedOn` represents the dataset URI on quality assessment has been made. This property is attached to a `qr:QualityReport`. `qr:hasProblem` links a `qr:QualityProblem` to a `qr:QualityReport`. The mentioned property identifies problem instances in a report. Each `qr:QualityProblem isDescribedBy` an instance of a `daq:Metric`[5]. The property `qr:problematicThing` represent the actual problematic instance from the dataset. This could be a list of resources (`rdf:Seq`) or a list of reified statements. Listing 1 represents an excerpt from a typical dataset showing the instance of `ex:JoeDoe` who is a `foaf:Researchers` working for `ex:UniBonn`. In these two instances there are three problematic triples:

(A) ⟨ `ex:JoeDoe a foaf:Researcher` ⟩ - The problem in this triple is caused by the usage of an undefined class, in this case `foaf:Researcher`;

(B) ⟨ `ex:JoeDoe rdfs:label "JoeDoe"` ⟩ - The literal (”JoeDoe”) in the triple causes the malformed capitalisation metric to point out a problem in this triple;

(C) ⟨ `ex:UniBonn rdfs:label "UniBonn"` ⟩ - The literal (”UniBonn”) in the triple causes the malformed capitalisation metric to point out a problem in this triple.

Listing 2 represent these three problems using the Quality Problem Report ontology.

```
ex:JoeDoe a foaf:Researcher ;
 rdfs:label "JoeDoe" ;
 ex:worksFor ex:UniBonn .

ex:UniBonn rdfs:label "UniBonn" ;
 foaf:name "University Bonn" .
```

Listing 1: An excerpt of a typical Dataset

[1] http://ontowiki.eu/Welcome

[2] `http://purl.org/eis/vocab/daq`

[3] `http://purl.org/eis/vocab/qr`

[4] `http://purl.org/eis/vocab/dqm`

[5] refer to Section 2.3

```
ex:QualityReport a qr:QualityReport ;
 qr:computedOn <uri:datasetResearchers> ;
 qr:hasProblem <#prob1>,<#prob2>,<#prob3> .

<#prob1> a qr:QualityProblem ;
 qr:isDescribedBy <urn:metric/UndefinedClasses123> ;
 qr:problematicThing [
    diachron:hasSubject ex:JoeDoe ;
  diachron:hasPredicate rdf:type ;
  diachron:hasObject foaf:Researcher ;
 ] .

 <#prob2> a qr:QualityProblem ;
  qr:isDescribedBy <urn:metric/Capitalisation789> ;
  qr:problematicThing [
    diachron:hasSubject ex:JoeDoe ;
    diachron:hasPredicate rdfs:label ;
    diachron:hasObject "JoeDoe" ;
  ] .

  <#prob3> a qr:QualityProblem ;
   qr:isDescribedBy <urn:metric/Capitalisation789> ;
   qr:problematicThing [
    diachron:hasSubject ex:UniBonn ;
    diachron:hasPredicate rdfs:label ;
    diachron:hasObject "UniBonn" ;
   ] .
```

Listing 2: An corresponding Quality Report for Listing 1

2.1.2 Processing Unit

The Processing Unit is an integral part of the Quality Framework. In this framework, we provide two main scalable processing units: a sequential stream processor (cf. Section 2.2) and SPARQL processor[6]. The former streams triples from RDF date dumps one by one in a sequential fashion. The latter allows the framework to assess quality on data that is available only in SPARQL endpoints. This unit is one of the two extensible modules (the other being Quality Assessment Layer) in the Quality Framework. For DIACHRON, the plan is to extend the sequential stream processor, enabling the de-reification of RDF statements into RDF triples.

Typically, an initialised processor has 2 inputs: the dataset URI (for the sequential stream processor) or the dataset SPARQL endpoint (in the case of the SPARQL processor), and a metric configuration file. Listing 3 shows an example of a typical metric configuration file.

```
@prefix diachron:     <http://www.diachron-fp7.eu/diachron#> .
@prefix qf:     <http://www.diachron-fp7.eu/qualityFramework#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfg:  <http://www.w3.org/2004/03/trix/rdfg-1> .
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .

_:metricAssessment a qf:metricConfiguration ;
  diachron:metric "accessibility.availability.SPARQLAccessibility" ;
  diachron:metric "accessibility.availability.RDFAccessibility" ;
  diachron:metric "accessibility.availability.Dereferencibility" ;
  diachron:metric "accessibility.performance.DataSourceScalability" ;
  diachron:metric "accessibility.performance.HighThroughput" ;
  diachron:metric "accessibility.performance.LowLatency" ;
  diachron:metric "intrinsic.accuracy.DefinedOntologyAuthor" ;
  diachron:metric "intrinsic.accuracy.SynonymUsage" ;
  diachron:metric "intrinsic.accuracy.POBODefinitionUsage" ;
  diachron:metric "intrinsic.consistency.EntitiesAsMembersOfDisjointClasses" ;
  diachron:metric "intrinsic.consistency.HomogeneousDatatypes" ;
  diachron:metric "intrinsic.consistency.MisplacedClassesOrProperties" ;
  diachron:metric "intrinsic.consistency.ObsoleteConceptsInOntology" ;
  diachron:metric "intrinsic.conciseness.OntologyVersioningConciseness" ;
  diachron:metric "dynamicity.timeliness.TimelinessOfResource" ;
  diachron:metric "dynamicity.currency.CurrencyDocumentStatements" ;
  diachron:metric "dynamicity.currency.TimeSinceModification" ;
```

[6] This processor is still being investigated and will not be ready by the deliverable deadline.

```
  diachron:metric "representational.understandability.HumanReadableLabelling" ;
  diachron:metric "representational.understandability.LowBlankNodeUsage" .
```

Listing 3: An typical metric configuration file

Each data processor in the Quality Framework has a defined 3-stage procedure (Listing 4): (i) processor initialisation; (ii) processing; and (iii) memory clean up. In the first process (processor initialisation), the processor create the necessary objects in memory to process data and load the required metrics that are instructed in the configuration file. Once the initialisation is ready, then processing is done by passing the streamed triples into the metrics. Finally, memory clean up ensures that no unused objects are using unnecessary computational power.

```
public interface IOProcessor {
  // Initialise the io processor with the necessary in-memory objects and metrics
  void setUpProcess();

  // Process the dataset for quality assessment
  void startProcessing() throws ProcessorNotInitialised;

  // Cleans up memory from unused objects after processing is finished
  void cleanUp() throws ProcessorNotInitialised;
}
```

Listing 4: IO Processor Interface

2.1.3 Quality Assessment Layer

The Quality Assessment Layer is unarguably the most important layer in this Quality Framework. The framework can be extended by any third party providing their own custom specific metric. This is already done in the DIACHRON project, where a number of metrics (cf. Section 4.2) required to assess the various use cases specified in Deliverable ??? are implemented. The Quality Assessment Layer provides two interfaces and an abstract class (cf. Figure 2) which facilitate the quality framework to be a pluggable and extensible platform. The interface `QualityMetric` is the core interface class which describes the metric classes. Each metric implementing this interface, must implement the following classes:

compute - This method assess the quad/triple which is passed by the stream processor by the defined metric;

metricValue - This method returns the value computed by the quality metric;

toDAQTriples - This method will return a list of daQ triples, containing quality metadata about the assessed metric, which will be stored in the dataset as a new named graph (quality graph);

getMetricURI - This method returns the URI of the Quality Metric from the ontology description (e.g. `http://purl.org/eis/vocab/dqm#DereferenceablityMetric`);

getQualityProblems - This method returns a typed (List⟨Resource⟩ or List⟨Quad⟩) ProblemList which will be used to create a quality report of the metric;

Furthermore, a metric might require some pre-processing or post-processing. Therefore, an interface (`ComplexQualityMetric`) extending `QualityMetric` was developed. This interface allow metric developers to perform such processing using the `void before()` and `void after()` methods.

In order to facilitate further such development of pluggable metrics, the `AbstractQualityMetric` class was developed, implementing the `QualityMetric` interface. In this abstract class, the method `List⟨Statement⟩ toDAQTriples()` is implemented, generating daQ observation instances (cf. Section 2.3) for the metric being assessed.

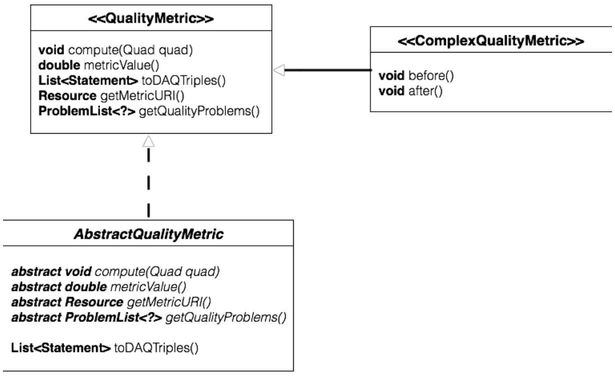
<<QualityMetric>>
void compute(Quad quad)
double metricValue()
List<Statement> toDAQTriples()
Resource getMetricURI()
ProblemList<?> getQualityProblems()
<<ComplexQualityMetric>>
void before()
void after()
AbstractQualityMetric
abstract void compute(Quad quad)
abstract double metricValue()
abstract Resource getMetricURI()
abstract ProblemList<?> getQualityProblems()
List<Statement> toDAQTriples()

Figure 2: Quality Assessment Layer Class Diagram - A Quality Framework as a Pluggable Platform

2.1.4 Semantic Annotation Unit

The Semantic Annotation Unit takes the generated triples (from the `toDAQTriples()` method) in order to create the quality metadata in a dataset. The unit provides a number of helper classes that provide inferencing queries on vocabularies that describe metrics (such as DQM) based on the core ontology daQ. Therefore, RDF descriptions of metrics extending the daQ (cf. Section 2.3.3) ontology is absolutely required. These inferencing queries enable the framework to create a complete metadata description (cf. Section 2.3) of an assessed quality metric.

2.1.5 Visualisation Layer

CubeViz is an OntoWiki extension for visualising data cubes (observation instances). Figures 3, 4, 5, and 6 depicts four different CubeViz chart visualisations from computed quality metadata[7].

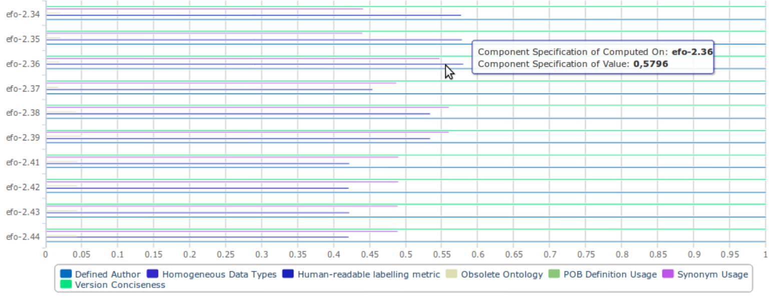

Figure 3: Horizontal Bar Chart

A *horizontal bar* represents each metric (Figure 3) and shows its value (x-axis) with respect to the dataset (y-axis). Here, the different datasets analysed are actually successive revisions of one dataset. This chart provides a clear view of how the value associated to each one of the measured metrics changes as the dataset evolves. The horizontal layout is appropriate when the range of metric values is wide, and the number of different datasets is relatively small.

Similar to the horizontal bars chart, the *vertical bar chart* (Figure 4) allows the user to compare the values computed for each of the metrics (y-axis), with respect to the dataset (x-axis). In contrast with its horizontal counterpart, this chart is more appropriate when there are many datasets analysed but the range of metric values is not so wide.

[7] The quality metadata used can be found in `https://raw.githubusercontent.com/diachron/quality/master/src/test/resources/cube_qg.trig`

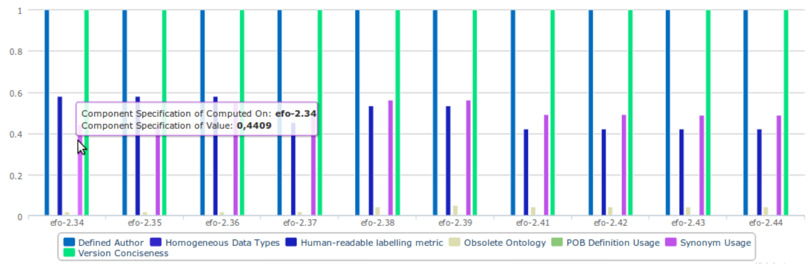

Figure 4: Vertical Bar Chart

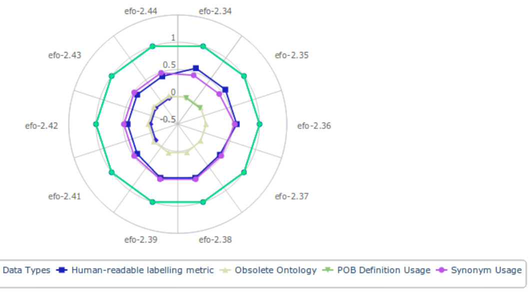

Figure 5: Radar Chart

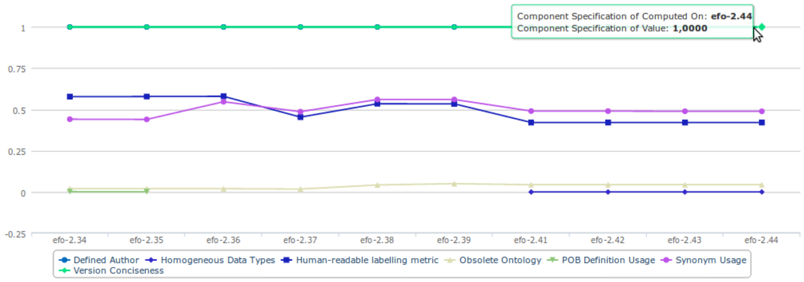

Figure 6: Lines Plot

In the *radar chart* (Figure 5), the datasets are represented as slices of a circle and the values corresponding to the metrics are depicted as points and lines of a particular color. This chart provides a clear view of how the values of the metric differ from each other for each particular dataset. Furthermore, it allows one to assess the overall quality of a dataset, by showing whether the values of the metrics are concentrated around sections of the circle regarded as good or bad.

The lines plot (Figure 6), lists the different datasets against the values of the metrics. Here, where different datasets are actually different revisions in the evolution of one dataset, this plot provides a comparison of the evolution of the quality of the dataset, with respect to each metric. The lines emphasise the points where the values of the metrics changed noticeably from one version to the next.

2.2 Sequential Stream Processor

In order to accurately assess linked dataset for quality measures, the assessment should on all triples in the assessed datasets. One must keep in mind that the computation of metrics on large datasets might be computationally expensive; thus, such stream processors computing dataset's quality must be scalable. In Figure 7, a closer look towards the quality assessment process is illustrated. A user first choose a dataset and the metrics which are required for the assessment of quality. The submitted information is passed to the Quality Framework via its API and initialise the processing unit (stream processor) in the core framework. The stream processor is then initialised by: (1) creating the necessary objects in memory, and (2) initialise the chosen metrics. In Figure 7, "Metric 1" is shadowed out - to illustrate that it was not chosen by the user for this particular use case. Once the objects created, the stream processor fetches the dataset and in a sequential fashion it starts streaming quads one by one to all initialised metrics in parallel. After all statements are assessed, the semantic annotation unit requests the metric value for each metric and creates (or updates - in case the dataset already has one) the Quality Graph. This named graph represents the quality metadata of a dataset using the representation defined in Section 2.3, and it is stored in the dataset itself. Having this metadata, it will allow us to rank and crawl datasets based on different quality attributes.

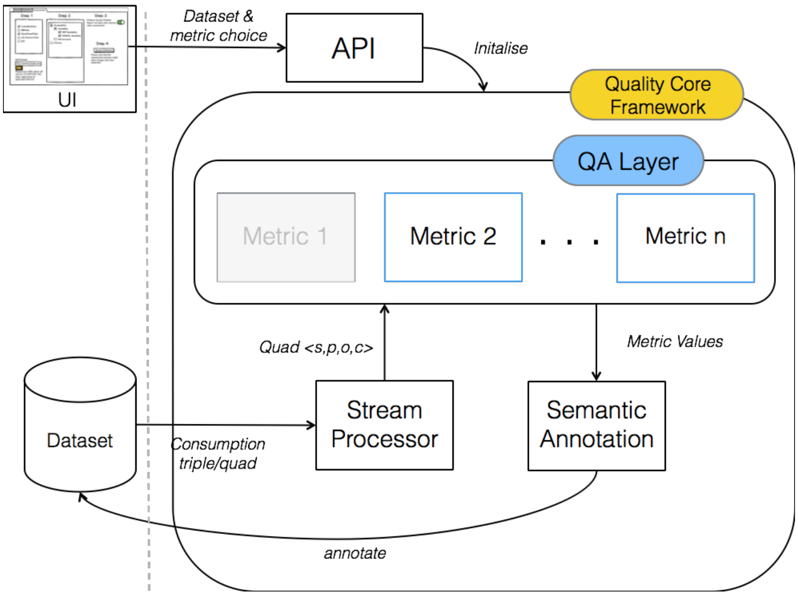

Figure 7: Closer look at the Quality Assessment process

Apache Jena[8] (cf. Section 3.3) provides a dedicated module for the reading and writing of RDF Data (RDF

[8] http://jena.apache.org

I/O Technology - RIOT[9]). The RIOT API functionality provides a number of classes. Typically the `RDFDataMgr` is used, which contains the main set of functions to read and load models and datasets. For the sequential stream processor, the Jena RIOT API was used.

2.2.1 The Initialisation Process - `setUpProcess()`

The first operation on the initialisation is the execution of the *setUpProcess()* method. Listing 1 is the pseudocode of the the process. The sequential stream processor starts its initialisation by first trying to identify the serialisation used by the available RDF data dump. The method `guessRDFSerialisation` analyses the file serialisation by mapping the file name to one of the Jena's in-built RDF languages (e.g. RDF/XML, NTriples, Turtle, NQuads, etc...) According to the file's serialisation, the process then assign different types of `PipedRDFIterator`[10] and either a`PipedQuadsStream`[11] or `PipedTriplesStream`[12] These two objects are required for the scalable execution of the sequential stream processor as together they act as a the "producer"[13] of sequential RDF triples from the RDF data dump. Once these are initialised, a flag is set to true to signal that the processor unit is in progress. Finally, the chosen metrics are loaded into memory. The loading of metrics is done dynamically during runtime, using the Java specific `newInstance()`[14] method.

Algorithm 1 The Initialisation of the Sequential Stream Process

```
procedure SETUPPROCESS
    rdfSerialisation = guessRDFSerialisation(datasetURI) ;
    if rdfSerialisation is Quads then
        iterator = new PipedRDFIterator⟨Quad⟩() ;
        rdfStream = new PipedQuadsStream((PipedRDFIterator⟨Quad⟩) iterator) ;
    if rdfSerialisation is Triple then
        iterator = new PipedRDFIterator⟨Triple⟩() ;
        rdfStream = new PipedTriplesStream((PipedRDFIterator⟨Triple⟩) iterator) ;
    set initialised boolean to true ;
    loadMetrics() ;
```

2.2.2 The Processing of Triples - `startProcessing()`

After the initialisation process, the method `startProcessing()` is invoked. The RDFStream `rdfStream` object starts parsing the RDF dump and producing triple or quad statements in the `iterator`. On a different thread, the "consumer" - the sequential stream processor - consumes these statements from the `iterator`, converts them into quads of ⟨s,p,o,c⟩, and passes them to all initialised metrics. The consumption process is repeated until all statements are exhausted from the `iterator`. The semantic annotation unit is then signalled to start its annotation. Listing 2 describes this process in pseudocode.

2.2.3 Clean Up - `cleanUp()`

The final process is to clean up the objects from memory. The processor follows a simple approach by assigning `null` to all objects and shutting down all running threads.

[9] http://jena.apache.org/documentation/io/rdf-input.html

[10] https://jena.apache.org/documentation/javadoc/arq/org/apache/jena/riot/lang/PipedRDFIterator.html

[11] https://jena.apache.org/documentation/javadoc/arq/org/apache/jena/riot/lang/PipedQuadsStream.html

[12] https://jena.apache.org/documentation/javadoc/arq/org/apache/jena/riot/lang/PipedTriplesStream.html

[13] As in the producer in the "Producer-Consumer problem"http://en.wikipedia.org/wiki/Producer?consumer_problem. The consumer is the on a separate thread, feeding the metrics.

[14] http://docs.oracle.com/javase/8/docs/api/java/lang/Class.html#newInstance--

Algorithm 2 Processing Triple/Quad Statements

```
procedure STARTPROCESSING
    if initialised == false then
        throw exception ;
    create new producer thread for rdfStream ;
    while (iterator has another statement) do
        quad = Object2Quad(iterator.next()) ;
        pass quad to all metrics and compute metric ;
    invoke semantic annotation unit ;
```

2.3 The Dataset Quality Ontology

The idea behind the Dataset Quality Ontology [?][15] (daQ) is to provide a comprehensive generic vocabulary framework, allowing a uniform definition of specific data quality metrics and thus suggest how quality metadata should be represented in datasets. This metric definition would then allow publishers to attach data quality metadata with quality benchmarking results to their linked dataset. Figure 8 depicts the current state of the daQ vocabulary.

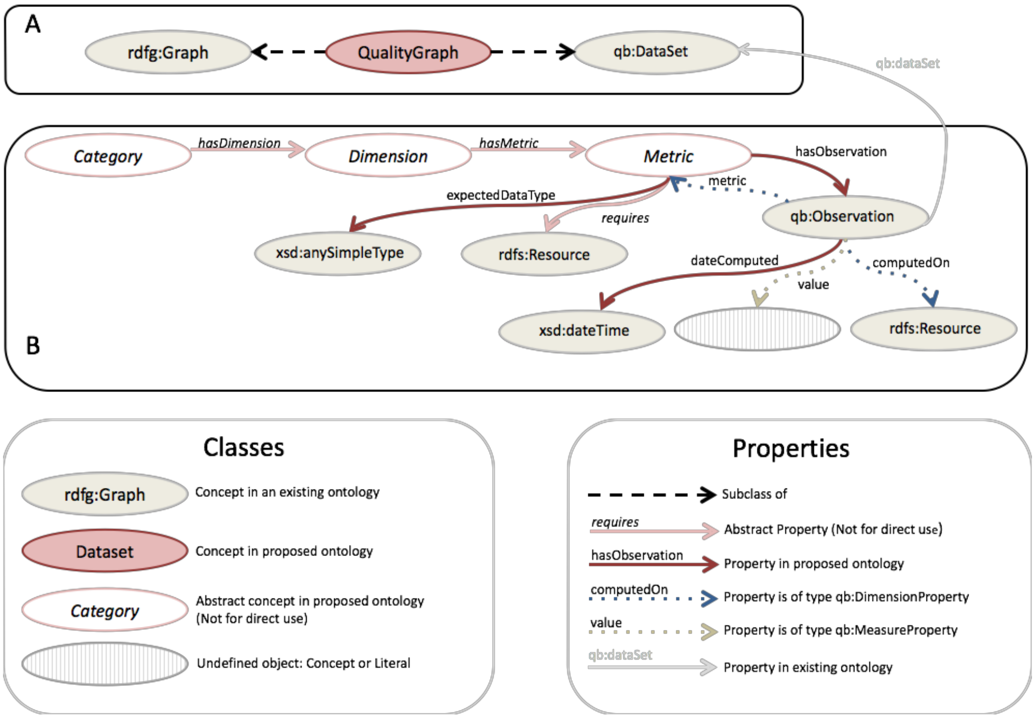

Figure 8: The extended Dataset Quality Ontology (daQ)

Using daQ, the quality metadata is intended to be stored in what we defined to be the *Quality Graph*. The latter concept is a subclass of `rdfg:Graph` [?]. This means that the quality metadata is stored and managed in a separate named graph from the assessed dataset. Named graphs are favoured due to

- the capability of separating the aggregated metadata with regard to computed quality metrics of a dataset from the dataset itself;

[15] `http://purl.org/eis/vocab/daq`

- their use in the Semantic Web Publishing vocabulary [?] to allow named graphs to be digitally signed, thus ensuring trust in the computed metrics and defined named graph instance. Therefore, in principle each `daq:QualityGraph` can have the following triple `:myQualityGraph swp:assertedBy :myWarrant .`

The daQ ontology distinguishes between three layers of abstraction, based on the survey work by Zaveri et al. [?]. As shown in Figure 8 Box B, a quality graph comprises of a number of different *Categories*, which in turn possess a number of quality *Dimensions*[16]. A quality dimension groups one or more computed quality *Metrics*. To formalise this, let G represent the named Quality Graph (`daq:QualityGraph`), $C = \{c_1, c_2, \ldots, c_x\}$ is the set of all possible quality categories (`daq:Category`), $D = \{d_1, d_2, \ldots, d_y\}$ is the set of all possible quality dimensions (`daq:Dimension`) and $M = \{m_1, m_2, \ldots, m_z\}$ is the set of all possible quality metrics (`daq:Metric`); where $x, z, y \in \mathbb{N}$, then:

Definition 1

$$G \subseteq C,$$
$$C \subset D,$$
$$D \subset M;$$

Figure 9 shows this formalisation in a pictorial manner using Venn diagrams.

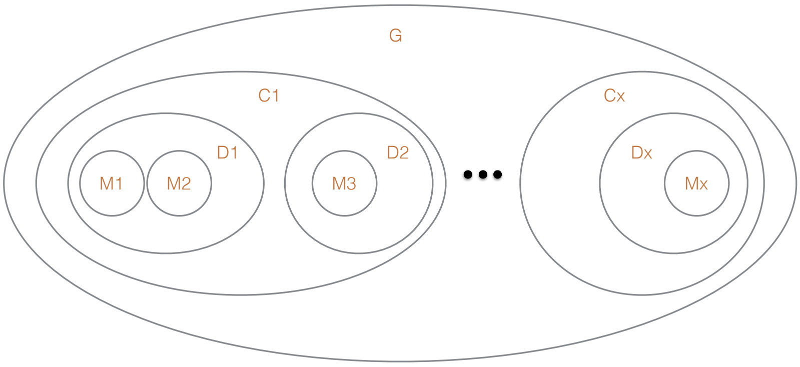

Figure 9: Venn Diagram depicting Definition 1

2.3.1 Extending daQ for Multi-Dimension Representation and Statistical Evaluation

The Data Cube Vocabulary [?] allows the representation of statistics about observations in a multidimensional attribute spaces. Multidimensional analysis of these observations, e.g. across the revision history of a dataset, would thus have required complex querying. Extending daQ with the standardised Data Cube Vocabulary allows us to represent quality metadata of a dataset as a collection of *Observations*, dimensions being the different quality metrics computed, the resources whose quality is assessed, revisions of these resources, and arbitrary further dimensions, such as the intended application scenario. It also permits applying the wide range of tools that support data cubes to quality graphs, including the CubeViz visualisation tool[17].

A *Quality Graph* is a special case of `qb:DataSet`, which allows us to represent a collection of quality observations complying to a defined dimensional structure. Each observation represents a quality metric measured out against a particular resource (e.g. a specific revision of a dataset). daQ defines the structure of such observations by the `qb:DataStructureDefinition` shown in Listing 5.

[16] In this deliverable we will refer to these as quality dimensions, in order to distinguish between the data cube dimensions

[17] `http://cubeviz.aksw.org`

```
daq:dsd a qb:DataStructureDefinition ;
  # Dimensions: metrics and what they were computed on
  qb:component [
    qb:dimension daq:metric ;
    qb:order 1 ; ] ;
  qb:component [
    qb:dimension daq:computedOn ;
    qb:order 2 ; ] ;
  # Measures (here: metric values)
  qb:component [ qb:measure daq:value ; ] ;
  # Attribute (here: the unit of measurement)
  qb:component [
    qb:attribute sdmx-attribute:unitMeasure ;
    qb:componentRequired false ;
    qb:componentAttachment qb:DataSet ; ] .
```

Listing 5: The Data Structure Definition (Turtle Syntax)

The `daq:QualityGraph` also defines one restriction that controls the property `qb:structure` and its value to the mentioned definition, thus ensuring that all *Quality Graph* instances make use of the standard definition. Having a standard definition ensures that all *Quality Graph*s conform to a common data structure definition, thus datasets with attached quality metadata can be compared. Listing 6 describes the definition of `daq:QualityGraph`.

```
daq:QualityGraph
  a rdfs:Class, owl:Class  ;
  rdfs:subClassOf rdfg:Graph , qb:DataSet ,
    [ rdf:type owl:Restriction ;
      owl:onProperty qb:structure ;
      owl:hasValue daq:dsd ];
  rdfs:comment "Defines a quality graph which will contain all metadata about quality metrics on the
      dataset." ;
  rdfs:label "Quality Graph Statistics" .
```

Listing 6: The Quality Graph Definition (Turtle Syntax)

2.3.2 Abstract Classes and Properties

This ontology framework (Figure **??**) has three abstract classes/concepts (`daq:Category`, `daq:Dimension`, `daq:Metric`) and three abstract properties (`daq:hasDimension`, `daq:hasMetric`, `daq:requires`) which should not be used directly in a quality instance. Instead these should be inherited as parent classes and properties for more specific quality metrics. The abstract concepts (and their related properties) are described as follows:

daq:Category represents the highest level of quality assessment. A category groups a number of dimensions.

daq:Dimension – In each dimension there is a number of metrics.

daq:Metric is smallest unit of measuring a quality dimension. Each metric instance is linked to one or more observations. Each observation has a value (`daq:value`), representing a score for the assessment of a quality attribute. This attribute is defined as a `qb:MeasureProperty`. Since this value is multi-typed (for example one metric might return true/false whilst another might require a floating point number), the value's `daq:hasValue` range is inherited by the actual metric's attribute defined by the property `daq:expectedDataType`. An observation must have the Dimension Properties (`qb:DimensionProperty`) `daq:computedOn` and `daq:metric`, which defines the assessed resource and the metric the mentioned resource was assessed by respectively. A metric might also require additional information (e.g. a gold standard dataset to compare with). Therefore, a concrete metric representation can also define such properties using subproperties of the `daq:requires` abstract property. Another important attribute for any observation is the `daq:dateComputed`, where it records the date of the observation's creation.

2.3.3 Extending daQ for Custom/Specific Quality Metrics

The classes of the core daQ vocabulary can be extended by more specific and custom quality metrics. In order to use the daQ, one should define the quality metrics that characterise the “fitness for use” [?] in a particular domain. We are currently in the process of defining the quality dimensions and metrics described in Deliverable 5.1. **Extending** the daQ vocabulary means adding new quality protocols that inherit the abstract concepts (Category-Dimension-Metric). Custom quality metrics do not need to be included in the daQ namespace itself; in fact, in accordance with LOD best practices, we recommend extenders to make them in their own namespaces. In Figure 10 we show an illustrative example of extending the daQ ontology (TBox) with a more specific quality attribute, i.e. the RDF Availability Metric as defined in [?], and an illustrative instance (ABox) of how it would be represented in a dataset.

The `Accessibility` concept is defined as an `rdfs:subClassOf` the abstract `daq:Category`. This category has five quality dimensions, one of which is the *Availability* dimension. This is defined as an `rdfs:subClassOf daq:Dimension`. Similarly, *RDFAvailabilityMetric* is defined as an `rdfs:subClassOf daq:Metric`. The specific properties *hasAvailabilityDimension* and *hasRDFAccessibilityMetric* (sub-properties of `daq:hasDimension` and `daq:hasMetric` respectively) are also defined (Figure 10).

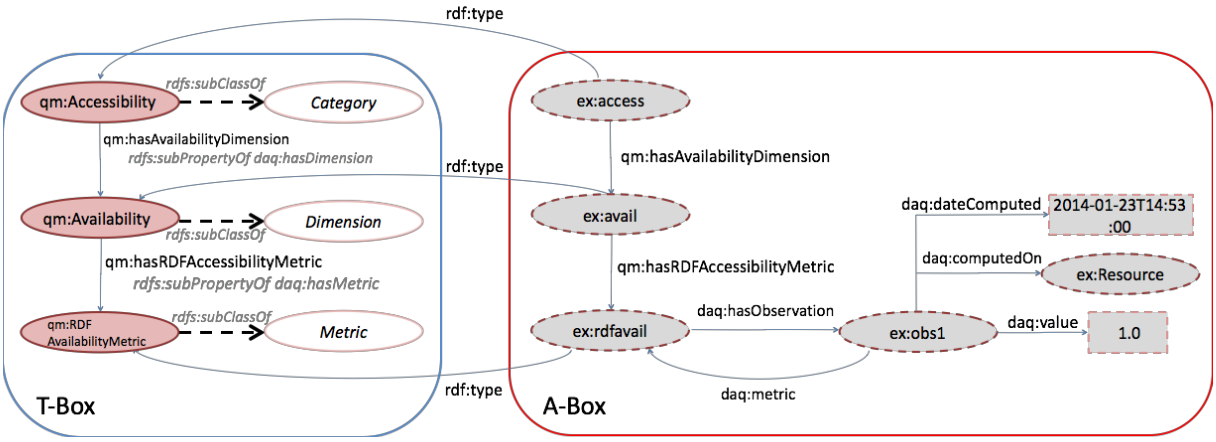

Figure 10: Extending the daQ Ontology TBox and ABox

2.3.4 A typical Quality Metadata Graph

The excerpt listing in 7 show a typical quality graph metadata in a dataset.

```
# ... prefixes

# ... dataset triples

ex:qualityGraph1 a daq:QualityGraph ;
  qb:structure daq:dsd .

ex:qualityGraph1 {

  # ... quality triples
  ex:accessibilityCategory a dqm:Accessibility ;
    dqm:hasAvailabilityDimension ex:availabilityDimension .

  ex:availabilityDimension a dqm:Availability ;
    dqm:hasEndPointAvailabilityMetric ex:endPointMetric ;
    dqm:hasRDFAvailabilityMetric ex:rdfAvailMetric .

  ex:endPointMetric a dqm:EndPointAvailabilityMetric ;
    daq:hasObservation ex:obs1, ex:obs2 .

  ex:obs1 a qb:Observation ;
```

```
    daq:computedOn <efo-2.43> ;
    daq:dateComputed "2014-01-23T14:53:00"^^xsd:dateTime ;
    daq:value "1.0"^^xsd:double ;
    daq:metric ex:endPointMetric ;
    qb:dataSet ex:qualityGraph1 .

  ex:obs2 a qb:Observation ;
    daq:computedOn <efo-2.44> ;
    daq:dateComputed "2014-01-25T14:53:00"^^xsd:dateTime ;
    daq:value "1.0"^^xsd:double ;
    daq:metric ex:endPointMetric ;
    qb:dataSet ex:qualityGraph1 .

  ex:rdfAvailMetric a dqm:RDFAvailabilityMetric ;
      daq:hasObservation ex:obs3, ex:obs4 .

  ex:obs3 a qb:Observation ;
    daq:computedOn <efo-2.43> ;
    daq:dateComputed "2014-01-23T14:53:01"^^xsd:dateTime ;
    daq:value "1.0"^^xsd:double ;
    daq:metric ex:rdfAvailMetric ;
    qb:dataSet ex:qualityGraph1 .

  ex:obs4 a qb:Observation ;
    daq:computedOn <efo-2.44> ;
    daq:dateComputed "2014-01-25T14:53:01"^^xsd:dateTime ;
    daq:value "0.0"^^xsd:double ;
    daq:metric ex:rdfAvailMetric ;
    qb:dataSet ex:qualityGraph1 .

  # ... more quality triples
}
```

Listing 7: A Quality Graph Excerpt (Turtle Syntax)

The instance *ex:qualityGraph1* is a named `daq:QualityGraph`. The defined graph is automatically a `qb:DataSet`, and due to the restriction placed on the `daq:QualityGraph` (see Listing 6), the value for the `qb:structure` property is defined as `daq:dsd` (see Listing 5). In the named graph, instances for the `daq:Accessibility`, `daq:Availability`, `daq:EndPointAvailabilityMetric` and `daq:RDFAvailabilityMetric` are shown. A metric instance has a number of observations. Each of these observations specifies the metric value (`daq:value`), the resource the metric was computed on (`daq:computedOn` here: different datasets, which are actually different revisions of one dataset), when it was computed (`daq:dateComputed`), the metric instance (`daq:metric`) and finally to what dataset the observation is defined in (`qb:dataSet`).

2.4 Quality RESTful API Design

The RESTful API design and activity diagrams are explained in Deliverable 6.1, Section 6.1.7. The only minor change is in the input parameters for the `/diachron/compute_quality` API call. In Deliverable 6.1 we define the following two parameters:

Dataset - An instance of a DIACHRON dataset URI;

QualityReportRequired - A boolean indicating whether a quality report is required.

The input parameter we introduce in this Deliverable is **MetricsConfiguration**. This parameter is an object with a list of metrics (cf. Listing 3) in JSON-LD format, identifying the metrics required to be used for the dataset quality assessment. Listing 8 shows a sample input message format with the newly added parameter **MetricsConfiguration**.

```
"Dataset": "http://exampleuri.com/rdfdump",
"QualityReportRequired": true,
"MetricsConfiguration" : [
    {
        "@id": "_:f4212571792b1",
        "@type": [
            "http://www.diachron-fp7.eu/qualityFramework#metricConfiguration"
        ],
        "http://www.diachron-fp7.eu/diachron#metric": [
            {
                "@value": "intrinsic.accuracy.DefinedOntologyAuthor"
            },
            {
                "@value": "accessibility.availability.RDFAccessibility"
            },
            {
                "@value": "representational.understandability.HumanReadableLabelling"
            },
            {
                "@value": "intrinsic.consistency.ObsoleteConceptsInOntology"
            },
            {
                "@value": "accessibility.availability.SPARQLAccessibility"
            },
            {
                "@value": "accessibility.performance.HighThroughput"
            },
            {
                "@value": "intrinsic.consistency.EntitiesAsMembersOfDisjointClasses"
            },
            {
                "@value": "intrinsic.accuracy.SynonymUsage"
            },
            {
                "@value": "dynamicity.currency.CurrencyDocumentStatements"
            },
            {
                "@value": "intrinsic.accuracy.POBODefinitionUsage"
            },
            {
                "@value": "accessibility.availability.Dereferencibility"
            },
            {
                "@value": "intrinsic.conciseness.OntologyVersioningConciseness"
            },
            {
                "@value": "dynamicity.currency.TimeSinceModification"
            },
            {
                "@value": "representational.understandability.LowBlankNodeUsage"
            },
            {
                "@value": "accessibility.performance.DataSourceScalability"
            },
            {
                "@value": "intrinsic.consistency.HomogeneousDatatypes"
            },
            {
                "@value": "intrinsic.consistency.MisplacedClassesOrProperties"
            },
            {
                "@value": "accessibility.performance.LowLatency"
            },
            {
                "@value": "dynamicity.timeliness.TimelinessOfResource"
            }
        ]
    }
]
```

Listing 8: API Call Input Message Format

3 Tools and Libraries Used

In this section we discuss the main tools and libraries used in our solution to help us achieve our goal.

3.1 OntoWiki

OntoWiki[18] is a tool providing support for agile, distributed knowledge engineering scenarios. Based on semantic technologies, OntoWiki provides an easy to use control management system (CMS) that allows users to manage the knowledge base (RDF data) underlying the application. OntoWiki is part of the LOD2 Stack, licensed under GPL and is open source. The Quality Framework is currently based on top of OntoWiki.

3.2 CubeViz

CubeViz[19] is an RDF DataCube browser and also an extension to OntoWiki. This extension allows users to visually represent statistical data represented in RDF, specifically data which is modelled by the RDF DataCube vocabulary. The Dataset Quality Vocabulary (daQ) is defined to use the mentioned RDF statistical vocabulary, thus CubeViz was a suitable extension to use to visualise statistical results about quality metadata.

3.3 Apache Jena

Apache Jena[20] is an open source Java framework for building Semantic Web and Linked Data applications. The framework is composed of different APIs interacting together to process RDF data. If you are new here, you might want to get started by following one of the tutorials. Apache Jena is licensed under the Apache License, Version 2.0. This framework is the underlying technology used for the “Quality Core Framework”.

4 Ranking Service

4.1 Data Quality Assessment Process

4.2 Data Quality Metrics

- Metric input is a quad ¡?s, ?p, ?o, ?g¿ -

4.2.1 Accessibility Category

Availability Dimension

Dereferenceability Metric
HTTP URIs should be dereferencable, i.e. HTTP clients can retrieve the resources identified by the URI. A typical web URI resource would return a `200 OK` code indicating that a request is successful and `4xx` or `5xx` if the request is unsuccessful. In Linked Data, a successful request should return a document (RDF) containing the description (triples) of the requested resource. In Linked Data, there are two possible ways which allow publishers make URIs dereferencable. These are the `303` URIs and the `hash` URIs[21]. Yang et. al [**?**] describes a mechanism to identify the dereferenceability process of linked data resource.

Calculates the number of valid redirects (303) or hashed links according to LOD Principles.

This metric (listing **??**) will count the number of valid dereferenceable URI resources found in the subject (?s) and object (?o) position of a triple. The `isDereferenceable(resource)` method uses the rules defined in [**?**]. The metric will return a ratio of the number of dereferenced URIs (deref) against the total number of triples in a dataset (totalTriples). The expected range is [0..1], where 0 is the worst rating and 1 is the best rating.

[18] http://ontowiki.net/
[19] http://cubeviz.aksw.org
[20] https://jena.apache.org
[21] http://www.w3.org/TR/cooluris/

Algorithm 3 Dereferenceablity Algorithm

```
procedure INIT
    totalTriples = 0 ;
    deref = 0 ;
procedure DEREFERENCE($\langle ?s, ?p, ?o, ?g \rangle$)
    if (isURI(?s)) && (isDereferenceable(?s)) then deref++ ;
    if (isURI(?o)) && (isDereferenceable(?o)) then deref++ ;
    totalTriples++;
```

4.2.2 Intrinsic Category

The intrinsic category metrics are independent of the user's context. They reflect whether information presented in data correctly represent the real world and whether information is logically consistent itself.

4.2.3 Accuracy

Malformed Datatype Literals

Literals are nodes in an RDF graph, used to identify values such as numbers and dates. A plain literal is a string combined with an optional language tag. A typed literal comprises a string (the lexical form of the literal) and a datatype (identified by a URI) which is supposed to denote a mapping from lexical forms to some space of values. The RDF specifies two types of literals: plain and typed. In the Turtle syntax typed literals are notated with syntax such as: *”13”8sd:int*. Malformed datatype literals are quit common and can be fixed by simple changes to the literals. This Malformed Datatype Literals metric intends to check if the value of a typed literal is valid with regards to the given xsd datatype.

Calculates the ratio of typed literals not valid regarding its given xsd datatype to all literals

Algorithm 4 Malformed Datatype Literals Algorithm

```
procedure INIT
    totalLiterals = 0 ;
    malformedLiterals = 0 ;
procedure COMPUTEMALFORMEDDATATYPELITERALS($\langle ?s, ?p, ?o, ?g \rangle$)
    if (isLiteral(?o)) then totalLiterals++ ;
        if (isTypedLiteral(?o)) && (!hasValidDatatype(?o)) then malformedLiterals++ ;
    metric=malformedLiterals/totalLiterals;
```

Literals Incompartible with Datatype Range

Defined Ontology Author

POBO Definition Usage

Synonym Usage

4.2.4 Consistency

Entities As Members of Disjoint Classes

Homogeneous Datatypes

Misplaced Classes or Properties

Misused Owl Datatype or Object Properties

Obsolete Concepts in Ontology

Ontology Hijacking

Undefined Classes Metric

Oftentimes a terms which is used in the object position of a triple and is not a literal is not formaly defined as being a class. 'Being defined' means that the term is defined either in some external ontology or at an earlier position in the given dataset. Regarding to Hogar [**?**] to the most used undefined classes belong `foaf:UserGroup`, `rss:item`, `linkedct:link`, `politico:Term` . The probability for undefined class in the subject position is very low, because the subject of a quad never references classes or properties in external vocabularies. Therefore they is no need to analyse the subject for this metric. For the most LOD data sets is sufficient to check object by the predicate `rdf:type`. In the case when LOD data set defines its own vocabulary the following predicates indicate that the object must be a defined class: `rdfs:domain`, `rdfs:range`, `rdfs:subClassOf`, `owl:allValuesFrom`, `owl:someValuesFrom`, `owl:equivalentClass`, `owl:complementOf`, `owl:onClass`, `owl:disjointWith`. The undefined classes problem occurs due to spelling or syntactic mistakes resolvable through minor fixes to the respective ontologies. The missing classes should be define in corresponding ontology or in a separate namespace.

Calculates the ratio of undefined classes to all classes in object position in a dataset

Algorithm 5 Undefined Classes Metric Algorithm

```
procedure INIT
    totalClasses = 0 ;
    undefinedClasses = 0 ;
procedure CLASSNOTDEFINED($\langle ?s, ?p, ?o, ?g \rangle$)
    if (isClassProperty(?p) && (isURI(?o)) then ;
        if (!isDefined(?o)) undefinedClasses++ then;
    totalClasses++ ++;
```

Undefined Properties

Similar to the Undefined Classes metric 4.2.4 the Undefined Properties metric identifies terms in the predicate position that are used without any formal definition. Hogan [**?**] identified the following properties that are often used whithout being defined: *foaf:image*, *cycann:label* , *foaf:tagLine*. The following list of predicates indicate that the object of the quad must be a defined property:*rdfs:subPropertyOf*, *owl:onProperty*, *owl:assertionProperty*, *owl:equivalentProperty*, *owl:propertyDisjointWith*.

Calculates the ratio of undefined properties to all properties in the given data set

Algorithm 6 Undefined Properties Algorithm

```
procedure INIT
    totalProperties = 0 ;
    undefinedProperties = 0 ;
procedure PROPERTYNOTDEFINED($\langle ?s, ?p, ?o, ?g \rangle$)
    if (isURI(?p) totalProperties++ then ;
        if (!isDefined(?p)) udefinedProperties++ then;
        if (isFromList(?p)) && !isDefined(?o) udefinedProperties++ then;
        totalProperties ++;
```

4.2.5 Conciseness

Duplicate Instance

Extensional Conciseness

Ontology Versioning Conciseness

4.2.6 Representational Category

4.2.7 Representational Conciseness

Metrics in the representational dimensions capture aspects related to information representation.

Short URIs

4.2.8 Understandability

Empty Annotation Value Some languages, e.g. OWL distinguish annotation properties. Annotation properties are predicates that provide informal documentation annotations about ontologies, statements, or IRIs. A simple example for annotation property is *rdfs:comment* which is used to provide a comment. Unfortunately annotation properties are often used with empty literal values that couse inconsistences in data. The problem can be solved by the corresponding triples or by replacing empty literals by annotation strings. The following annotation properties were used in this metric:

- *skos:altLabel*
- *skos:hiddenLabel*
- *skos:prefLabel*

- *skos:changeNote*
- *skos:definition*
- *skos:editorialNote*
- *skos:example*
- *skos:historyNote*
- *skos:note*
- *skos:scopeNote*
- *dcterms:description*
- *dc:description*
- *rdf:label*
- *rdf:comment*

The metric Empty Annotation Value identifies triples whose property is an annotation property and whose object is an empty string.

> *Calculates the ratio of annotations with empty values to all annotations in the data set.*

Algorithm 7 Empty Annotation Value Algorithm

```
procedure INIT
    totalAnnotations = 0 ;
    emptyAnnotations = 0 ;
procedure COUNTEMPTYANNOTATIONS($\langle ?s, ?p, ?o, ?g \rangle$)
    if (isAnnotation(?p)) then
        if (isEmpty(?o)) then emptyAnnotations++ ;
        totalAnnotations++;
```

Human Readable Labelling

Labels Using Capitals

Low Blank Node Usage

Whitespace in Annotation

4.2.9 Dynamicity Category

Volatility Dimension

In Linked Data evolution appeared almost at each new published version of data. Following the idea of [**?**] curators could define a list of changes that occur frequently and correspond to one or more low-level changes (added or deleted triples). These changes termed as Simple Changes also in the context of DIACHRON and comprise an upper abstract level of changes which is pilot-specific to describe group of changes that appear a special interest for each pilot. The detection of Simple Changes achieved accordingly to the methodology presented in [**?**] and followed in change detection service of DIACHRON ([**?**]. The following three volatility metrics take into account these assumptions and background information.

Versions Volatility Metric

The comparison of two sequential (or not) versions of datasets could contain a number of simple changes for each pilot. In other cases, it makes sense to compare an old version of a dataset with the newest one.

> *Calculates the number of simple changes happened accross two specified versions.*

The Versions Volatility Metric can be applied to a pair of defined versions to count the detected number of Simple Changes. This achieved by querying the corresponding named graph where the the total number of Simple Changes have been stored which are returned as result.

Algorithm 8 Versions Volatility Algorithm

```
procedure INIT
    numberOfChanges = 0
procedure COMPUTE
    numberOfChanges = countSimpleChanges(v1,v2)
return numberOfChanges
```

The metric will return the total number of Simple Changes between two versions [integer number].

Average Volatility Metric

The number of detected simple changes could be varied accross different published versions for each curator/pilot. According to different scenarios some versions are similar while others appear many deltas. Thus, it is meaningful to examine all available published versions of datasets in order to find the average detected Simple Changes across each pair of versions.

> *Calculates the average number of simple changes detected across the published versions.*

The Average Volatility Metric firstly calculates the total number of published versions through a SPARQL query. Afterwards, it calculates the detected Simple Changes per versions pair and aggregate the sum of changes. Finally, it calculates and returns the ratio between aggregated sum and the number of examined pairs.

Algorithm 9 Average Volatility Metric Algorithm

```
procedure INIT
    changesTotal = 0
    versionsNo = 0
    retValue = 0
procedure COMPUTE
    $Versions[] = countVersions(SPARQL)$
    for all $v[i], v[i+1] \in Versions$ do
        $changesTotal = changesTotal + countSimpleChanges(v[i], v[i+1])$
retValue = changesTotal / versionsNo -1
return retValue
```

The metric will return the ratio [0..1] of average detected simple changes across the published dataset versions.

Weighted Volatility Metric

In some applications pilots are interested more in evolution of specified versions. By applying a weighted sum model [?] for each sequential pair of versions, we could adapt this preference for each pilot.

Calculates the average weighted sum of simple changes that has been detected across the published versions.

The Weighted Volatility Metric after finding the total number of published versions it loads the weights from the curator preference table. Afterwards, it calculates the simple changes per pair and multiply with the corresponding weight. Finally it calculates the ratio of weighted sum of changes to the examined pairs of versions.

Algorithm 10 Weighted Volatility Metric Algorithm

```
procedure INIT
    aggregSChanges = 0
    versionsNo = 0
    retValue = 0
procedure LOADWEIGHTS
    $weights[] = fetchWeights()$
procedure COMPUTE
    $versions[] = countVersions(SPARQL)$
    for all $v[i], v[i+1] \in versions$ do
        for all $w[j] \in weights$ do
            $changesTotal = changesTotal + w[j] * countSimpleChanges(v[i], v[i+1])$
retValue = aggregSChanges / versionsNo -1
return retValue
```

The metric will return the ratio of [0..1] aggregated weighted sum detected simple changes across the published dataset versions.

4.3 The User Interface

4.4 Ranking of Quality-Computed Datasets

5 Crawling Service

6 Conclusions