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

DIACHRON

Managing the Evolution and Preservation of the Data Web DIACHRON

Dissemination level:	Public
Type of Document:	Report
Contractual date of delivery:	M16
Actual Date of Delivery:	
Deliverable Number:	D5.2
Deliverable Name:	Software prototype of the crawling, ranking and appraisal services
Deliverable Leader:	UBONN
Work package(s):	WP5
Status & version:	0.1 (DRAFT)
Number of pages:	
WP contributing to the deliverable:	WP5
WP / Task responsible:	T5.2, T5.3
Coordinator (name / contact):	Jeremy Debattista (University of Bonn)
Other Contributors:	
EC Project Officer:	Federico Milani

Keywords: data quality framework, quality metadata, quality metrics, dataset ranking, dataset crawling

Abstract:
The report describes the Data Quality Framework required to analyse and assess datasets for quality which would enable crawling and ranking.

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

Furthuremore, 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 **??**.

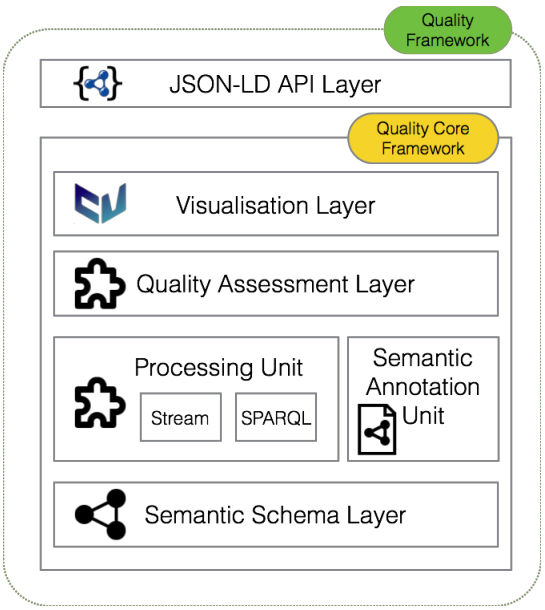

Figure 1: Quality Framework High Level Architecture Design

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*.

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)[1]; and (ii) the Quality Problem Report Ontology (qr)[2]. 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)[3]. 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`[4]. 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

```
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 ;
```

[1] http://purl.org/eis/vocab/daq
[2] http://purl.org/eis/vocab/qr
[3] http://purl.org/eis/vocab/dqm
[4] refer to Section 2.3

```
] .

<#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[5]. 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" ;
  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

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

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;

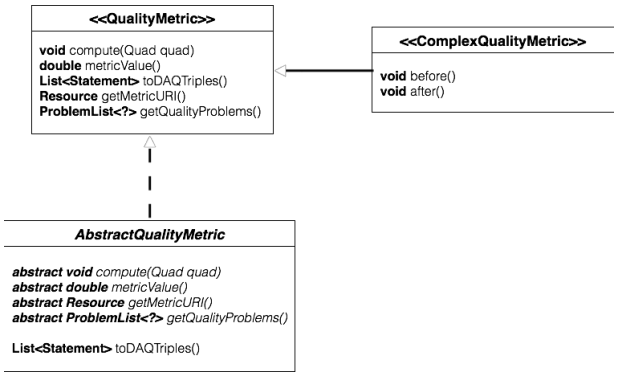

Figure 2: Quality Assessment Layer Class Diagram - A Quality Framework as a Pluggable Platform

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.

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

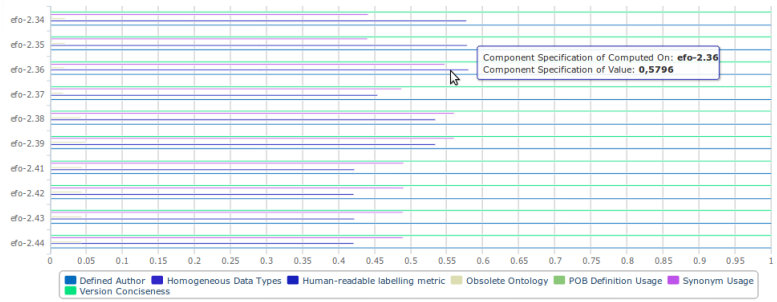

Figure 3: Horizontal Bar Chart

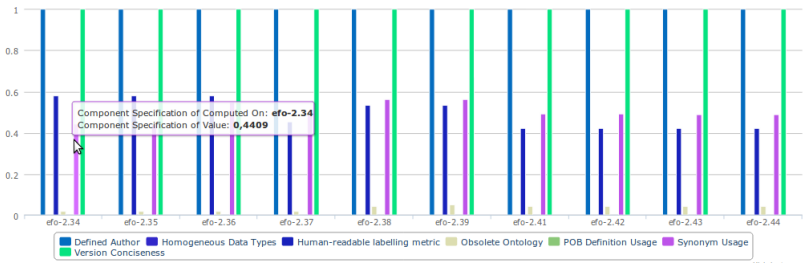

Figure 4: Vertical Bar Chart

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.

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

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

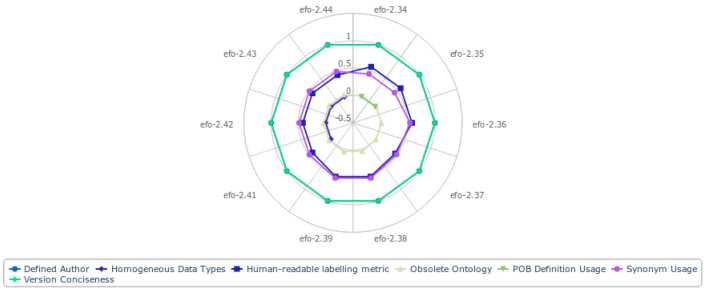

Figure 5: Radar Chart

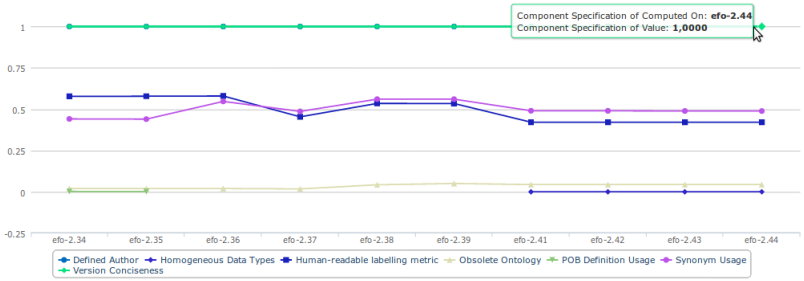

Figure 6: Lines Plot

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.

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.

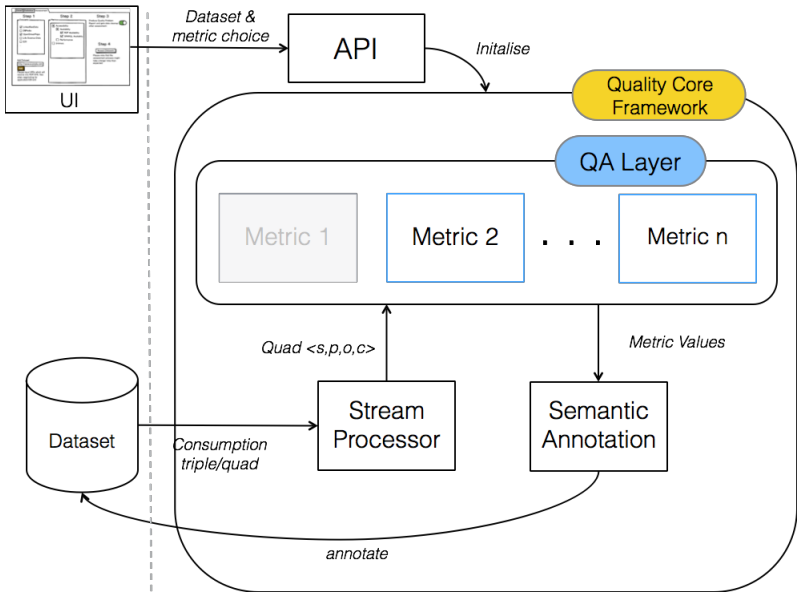

Figure 7: Closer look at the Quality Assessment process

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

[8] http://jena.apache.org
[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 1 The Initialisation of the Sequential Stream Process

```
1:  procedure SETUPPROCESS
2:      rdfSerialisation = guessRDFSerialisation(datasetURI) ;
3:      if rdfSerialisation is Quads then
4:          iterator = new PipedRDFIterator⟨Quad⟩() ;
5:          rdfStream = new PipedQuadsStream((PipedRDFIterator⟨Quad⟩) iterator) ;
6:      if rdfSerialisation is Triple then
7:          iterator = new PipedRDFIterator⟨Triple⟩() ;
8:          rdfStream = new PipedTriplesStream((PipedRDFIterator⟨Triple⟩) iterator) ;
9:      set initialised boolean to true ;
10:     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.

Algorithm 2 Processing Triple/Quad Statements

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

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.

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.

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`

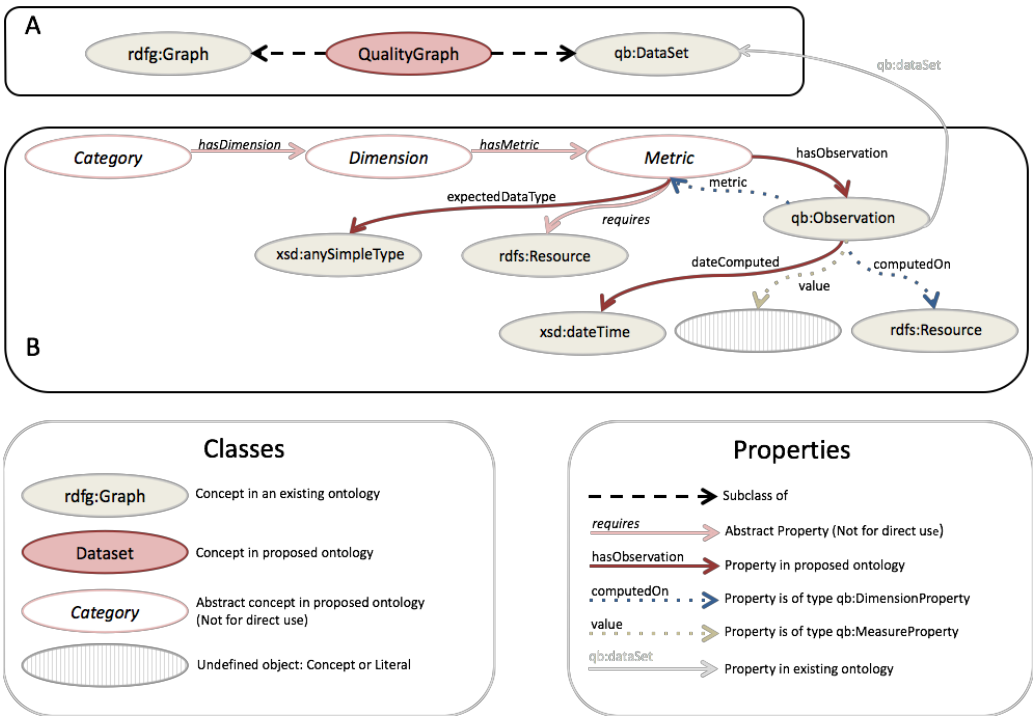

Figure 8: The extended Dataset Quality Ontology (daQ)

- 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.

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

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

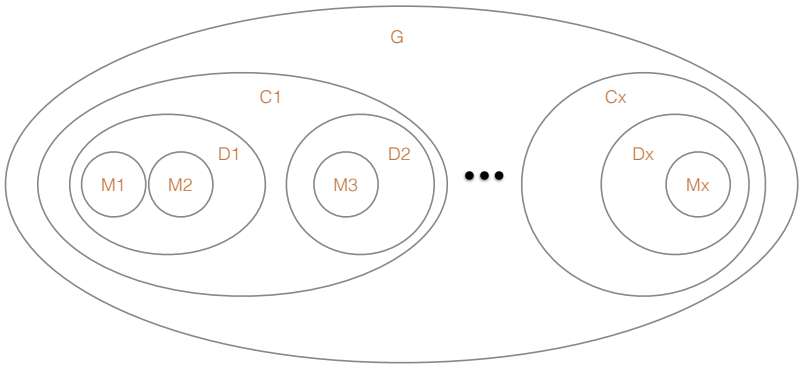

Figure 9: Venn Diagram depicting Definition 1

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.

```
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)

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

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).

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 .
```

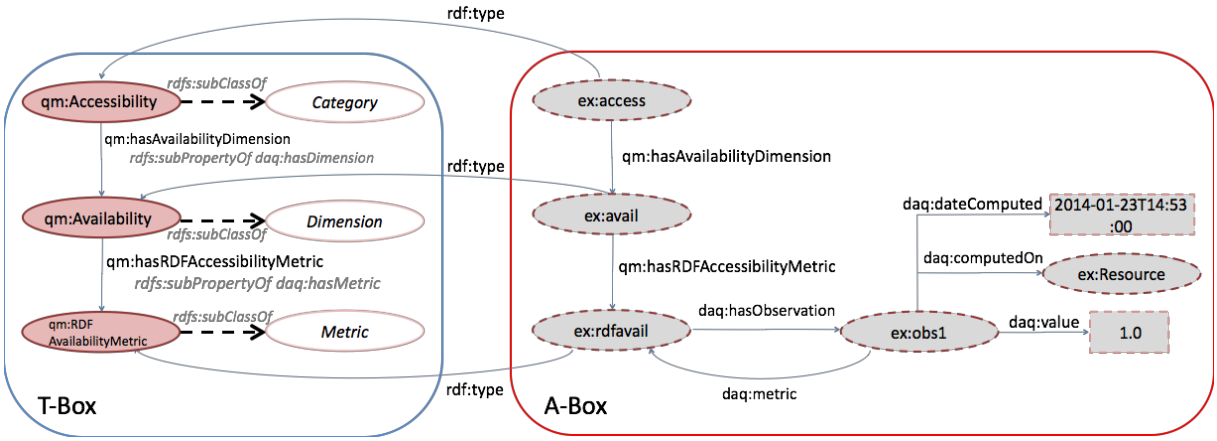

Figure 10: Extending the daQ Ontology TBox and ABox

```
  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".

3.4 OpenRDF Sesame

OpenRDF Sesame [21] is a de-facto standard framework for processing RDF data. This includes parsers, storage solutions (RDF databases a.ka. triplestores), reasoning and querying, using the SPARQL query language. It offers a flexible and easy to use Java API that can be connected to all leading RDF storage solutions. The framework is licesed under BSD.

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

3.5 Virtuoso

Virtuoso [22] is an innovative enterprise grade multi-model data server for agile enterprises and individuals. It delivers an unrivaled platform agnostic solution for data management, access, and integration. Virtuoso is licensed under GNU General Public License (GPL) Version 2.

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[23]. 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

Algorithm 3 Dereferenceablity Algorithm

```
1: procedure INIT
2:     totalTriples = 0 ;
3:     deref = 0 ;
4: procedure DEREFERENCE(⟨?s, ?p, ?o, ?g⟩)
5:     if (isURI(?s)) && (isDereferenceable(?s)) then deref++ ;
6:     if (isURI(?o)) && (isDereferenceable(?o)) then deref++ ;
7:     totalTriples++;
```

dataset (totalTriples). The expected range is [0..1], where 0 is the worst rating and 1 is the best rating.

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.

[22] http://virtuoso.openlinksw.com/
[23] http://www.w3.org/TR/cooluris/

4.2.3 Accuracy

Accuracy dimension metrics reflect the degree of correctness and precision with which the given dataset represent the real world facts.

Malformed Datatype Literals

Literals that are incorrect regarding their data type are a very common problem. Literals are nodes in an RDF graph, used to identify values such as numbers and dates. The RDF specifies two types of literals: plain and typed. 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. In the Turtle syntax typed literals are notated with syntax such as: *"13"8sd:int* This Malformed Datatype Literals metric intends to check if the value of a typed literal is valid with regards to the given *xsd* datatype. The algorithm 4 describes the metric computation in more details.

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

Algorithm 4 Malformed Datatype Literals Algorithm

```
1: procedure INIT
2:     totalLiterals = 0 ;
3:     malformedLiterals = 0 ;
4: procedure COMPUTE(⟨?s, ?p, ?o, ?g⟩)
5:     if (isLiteral(?o)) then totalLiterals++ ;
6:         if (isTypedLiteral(?o)) && (!hasValidDatatype(?o)) then malformedLiterals++ ;
       return malformedLiterals/totalLiterals
```

Only the typed literals are considered by the metric. The metric values vary in the interval [0, 1], where the 0 indicates the best quality.

Literals Incompatible with Datatype Range

Similar to the previous metric the 'literals incompatible with datatype range' metric verifies the correctness of literals regarding their datatype. Apart from typed literals described below a Literal datatype can also be defined through the predicate of a triple. The range of attribute property (property corresponding to recourse with literal value) may be constrained to be a certain datatype. Please see 5 the computational algorithm for more details.

Calculates the ratio of literals incompatible with datatype range to all literals

Algorithm 5 Literals Incompatible with datatype range

```
1: procedure INIT
2:     totalLiterals = 0
3:     incompartibleLiterals = 0
4: procedure COMPUTE(⟨?s, ?p, ?o, ?g⟩)
5:     if isLiteral(?o) then totalLiterals++ ;
6:         if hasRange(?p) && (literalDatatype(?o)!=rangeDatatype(?p))  then imcompartibleLiterals++;
       return incompartibleLiterals/totalLiterals
```

Only the literals referring by the property with the range characteristic are considered by the metric. The metric values vary in the interval [0, 1], where the 0 indicates the best quality.

Defined Ontology Author

POBO Definition Usage

Synonym Usage

4.2.4 Consistency

Consistency metrics intend to identify any kinds of contradictions in data.

Entities As Members of Disjoint Classes

Homogeneous Datatypes

This metric deal with literals that conflict regarding their datatype. Having restrictions to the literal datatype it's easy to validate the correctness of the data. However, even if no restriction regarding literals is defined, different datatypes for a literals corresponding to the same property could point to inconsistences in the data. In contrast to the previous metrics described in 4.2.3 and in 4.2.3, this metric deals only with literals which data type is not defined. The metric computation therefore contains the following steps:

- Count frequency of different datatypes occurring with a particular predicates.
- Identify properties corresponding to heterogeneous datatype literals.

Calculates the ratio of properties containing heterogeneous datatype literals to all properties

Algorithm 6 Homogeneous Datatypes

```
1: procedure INIT
2:     totalProperties = 0;
3:     heterogeneousDatatypeProperties = 0
4:     propertyMap = map(Property, List¡Datatypes¿)
5: procedure ANALYZEPROPERTIES(⟨?s, ?p, ?o, ?g⟩)
6:     if isLiteral(?o) then propertyMap.put(?p, Set.add.datatypeOf(?o))
       ;
7: procedure COMPUTE
8:     for all Properties in propertyMap do
9:         if size(Set¡Datatype¿) ¿ 1 then heterogeneousDatatypeProperties++;
           totalProperties ++;
       return eterogeneousDatatypeProperties/totalProperties
```

The metric values vary in the interval [0,1], when the 0 indicates no properties containing heterogeneous datatype literals

If there are just a few triples of one predicate having a datatype different from all other triples, they are considered as outliers and are all reported. In case if there is no obvious ratio of possibly wrong and possibly right triples the conflicting property will be reported to the user.

Misplaced Classes or Properties

In some cases a URI that occurs in the predicate position of a triple is defined in the corresponding vocabulary as a class, or a contrariwise a URI in the object position is a property. The common problem is e.g. the usage of property assigned to *rdf:type* predicate. These kind of inconsistences make machine interpretation of the data more complex. More details about the metric computation are presented in 7.

> *Number of properties in a class position + number of classes used as predicate to number of all classes and properties.*

Algorithm 7 Misplaced Classes or Properties Metric Algorithm

```
1: procedure INIT
2:     totalClassesUndProperties = 0 ;
3:     misplacedClasses = 0 ;
4:     misplacedProperties = 0 ;
5: procedure COMPUTE(⟨?s, ?p, ?o, ?g⟩)
6:     if isURI(?p) then totalClassesUndProperties++ ;
7:     if !isProperty(?p) then misplacedProperties++ ;
8:     if isURI(?o) then totalClassesUndProperties++;
9:     if !isClass(?o) then misplacedClasses++ ;
       return (misplacedClasses+misplacedProperties)/totalClassesUndProperties
```

The metric values vary in the interval [0, 1], where the 0 indicates the best quality.

Misused Owl Datatype or Object Properties

OWL language defines additional characteristics to some properties. If it's defined, a property can be either an instance of the *owl:ObjectProperty* or *owl:DatatypeProperty* class. A datatype property relates some resource to a literal value, while an object property describes the relation between two resources. Wrong usage of the datatype and object properties indicates inconsistences in the data. The following algorithm 8 presents more details about how the metric is computed.

> *Number of misused datatype and object properties to all properties*

Algorithm 8 Misused Owl Datatype or Object Properties Metric Algorithm

```
1: procedure INIT
2:     totalProperties = 0 ;
3:     misusedObjectProperties = 0 ;
4:     misusedDatatypeProperties = 0 ;
5: procedure COMPUTE(⟨?s, ?p, ?o, ?g⟩)
6:     if isURI(?p) then totalProperties++ ;
7:     if isDatatypeProperty(?p) && isURI(?o) then misusedDatatypeProperties++ ;
8:     if isObjectProperty(?p) && isLiteral(?o) then misusedObjectProperties++ ;
       return (misusedObjectProperties + misusedDatatypeProperties)/totalProperties
```

The metric values vary in the interval [0,1], where the 0 indicates the best quality.

Obsolete Concepts in Ontology

Ontology Hijacking

The 'ontology hijacking' term was first introduced in Hogan et. al [**?**] and is defined as 'the contribution of statements about classes and/or properties in a non-authoritative source such that reasoning on those classes and/or properties is affected'. In other words ontology hijacking refers to cases where external concepts are redefined in a local ontology. Defining new super classes or properties of third-party classes or properties is an example for this problem, e.g. declaring *rdfs:subPropertyOf* which is defined as a property, to be a *rdfs:Class*. The challenging question is how to define authoritative source. We assume the ontologies/vocabularies published according to the best-practices as authoritative, while all other vocabularies as third parties documents. More formal, a concept (class or property) is authoritative if it's not a blank node and if the corresponding vocabulary is retrievable. To identify hijacked terms we define a set of properties P = *rdf:type*, *rdf:domain*, *rdf:range*, *rdfs:subClassOf*, *rdfs:subPropertyOf owl:equivalentClass*, *owl:equivalentProperty*, *owl:inverseOf*, *owl:onProperty*, *owl:hasValue*, *owl:someValuesFrom*, *owl:allValuesFrom*, *owl:intersectionOf*, *owl:unionOf*, *owl:maxCardinality*, *owl:cardinality*, *owl:oneOf* and a set of classes C = *owl:FunctionalProperty*, *owl:InverseFunctionalProperty*, *owl:TransitiveProperty*, *owl:SymmetricProperty*. Metric check the two following cases:

- Classes in **C** appear in a position other than the object of a *rdf:type* triple. This is the a property redefinition.
- Properties in **P** appear in a position other than the predicate position.

Hogan et. al [**?**] present more detail about the metric computation.

Calculates the ratio of hijacked triples to all triples

Algorithm 9 Ontology Hijacking Algorithm

```
1: procedure INIT
2:     hijackedTriples = 0 ;
3:     totalTriples = 0 ;
4: procedure COMPUTE(⟨?s, ?p, ?o, ?g⟩)
5:     if ((?s ∈ C)——(?o ∈ C))&& (?p == rdf:type) then hijackedTriples++ ;
6:     if (?p ∈ P) && isAuthority(?s) && !isAuthority(?o) then hijackedTriples++;
       return hijackedTriples/totalTriples
```

Undefined Classes Metric

Oftentimes a terms which is used in the object position of a triple and is not a literal is not formally 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 analyze 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 defining in corresponding ontology or in a separate namespaces.

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

Algorithm 10 Undefined Classes Metric Algorithm

```
1: procedure INIT
2:     totalClasses = 0 ;
3:     undefinedClasses = 0 ;
4: procedure COMPUTE(⟨?s, ?p, ?o, ?g⟩)
5:     if (isClassProperty(?p) && (isURI(?o)) then totalClasses++ ;
6:     if (!isDefinedClass(?o)) undefinedClasses++ then;
       return undefinedClasses/totalClasses
```

The metric values vary in the interval [0, 1], where the 0 indicates the best quality.

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 without 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 11 Undefined Properties Algorithm

```
1: procedure INIT
2:     totalProperties = 0 ;
3:     undefinedProperties = 0 ;
4: procedure COMPUTE(⟨?s, ?p, ?o, ?g⟩)
5:     if (isURI(?p) then totalProperties++;
6:     if !isDefined(?p) then udefinedProperties++;
7:     if isFromList(?p) && !isDefined(?o) then udefinedProperties++ ;
       return undefinedProperties/totalProperties
```

The metric values vary in the interval [0,1], where the 0 indicates the best quality.

4.2.5 Conciseness

Duplicate Instance Metric

The information contained in Linked Data resources should not be redundant, which means that the instances contained in a dataset should, ideally, be unique. As stated by Yuangui Lei et al in [**?**], the mapping between the real-world objects described by the data sources and the instances contained in the semantic metadata should be one to one. That is, each statement about the existence of a real-world object should correspond to one and only one instance declaration. Resources are divided into groups called classes. The members of a class are known as instances of the class. A triple of the form: R rdf:type C, states that C is an instance of rdfs:Class and R is an instance of C, as defined in the RDF Schema specification [24].

[24] *http://www.w3.org/TR/rdf-schema/#ch_type*

> *Computes the Duplicate Instance metric as one minus the ratio of the number of instances violating the uniqueness rule to the total number of instances in the dataset.*

As shown in algorithm 12, this metric is implemented by computing the subtraction one minus the ratio of the number of non-unique instances to the total number of declared instances. An instance is regarded as non-unique, if there is another instance declaration (i.e. rdf:type annotation) with its same subject URI and object value. The

Algorithm 12 Duplicate Instance Algorithm

```
1: procedure INIT
2:     mapDeclaredInstances = new Map⟨URI, Instance⟩();
3:     countNonUniqueInstances = 0;
4:     countTotalInstances = 0;
5: procedure COMPUTE(⟨?s, ?p, ?o, ?g⟩)
6:     if containsKey(mapDeclaredInstances, ?s) then countNonUniqueInstances++;
7:     countTotalInstances++;
8: return 1.0 - (countNonUniqueInstances / countTotalInstances);
```

metric will return one minus the ratio of the number of unique instance declarations in the dataset, to the total number of instance declarations existing in the dataset. The expected range is [0..1], where 0 is the best rating (no duplicate instance declarations exist) and 1 is the worst rating (all instance declarations in the dataset are redundant).

Extensional Conciseness Metric

The conciseness of a dataset can be considered at the data level, as the redundancy of objects (i.e. instances) contained into the dataset. As defined by Mendes et al [**?**], a dataset is concise (on the extensional or instance level), if it does not contain redundant objects, that is, objects being equivalent in their contents, yet having different identifiers. Ideally, Linked Data resources should not contain redundant information, which implies that all the objects described by them should be unique. Uniqueness of objects is determined from their properties: one object is said to be unique if and only if there are no other objects with the same set of properties and corresponding values.

> *Calculates the ratio of the number of unique objects (i.e. instances) to the Total Number of objects. Two objects are equivalent if they have the same set of properties, all with the same values (but not necessarily the same ids).*

In the implementation of this metric (algorithm 13), objects are identified by their URI (the value of the subject attribute of the triples). The uniqueness of objects is determined from its properties: one object is said to be unique if and only if there is no other subject equivalent to it. Note that two equivalent objects may differ in their URI. The metric will return a ratio of the number of unique objects in the dataset (i.e. objects whose properties and their

Algorithm 13 Extensional Conciseness Algorithm

```
1: procedure INIT
2:     mapDescribedObjs = new Map⟨URI, Object⟩();
3: procedure COMPUTE(⟨?s, ?p, ?o, ?g⟩)
4:     curDescribedObj = getElementWithKey(mapDescribedObjs, ?s);
5:     setObjectProperty(curDescribedObj, ?p, ?o);
6: return countUniqueObjects(mapDescribedObjs)/countTotalObjects(mapDescribedObjs);
```

values are not duplicated in another object), to the total number of objects described in the dataset. The expected range is [0..1], where 0 is the worst rating (all objects are the same) and 1 is the best rating (all objects are unique).

Ontology Versioning Conciseness

4.2.6 Representational Category

Representational dimensions reflect the quality aspects like conciseness, consistency and interpretability of information.

4.2.7 Representational Conciseness

Short URIs

URIs play a key role in how information is represented in Linked Data resources, as they are used to name the entities being described. Therefore, having compact, well formatted URIs has a positive effect in the clearness and conciseness of data. As suggested by [**?**], data providers that locally mint (on average) shorter URIs are deemed as being more compliant with Linked Data best practices.

> *Detects whether, in average, short URIs are being used, which suggests that information is compactly represented and that readability is favored.*

Implementation details regarding these metric are provided in algorithm 14. All URIs identifying instances, that are defined locally, are considered by the metric. The calculation is performed as the average of the lengths of the URIs corresponding to the subjects of all instance declarations (i.e. statements using the *rdf:type* predicate). The metric will return the average length of all the URIs locally defined in the dataset. The expected range is a real

Algorithm 14 Short URIs Algorithm

```
1: procedure INIT
2:     accumulatedURIsLength = 0;
3:     countLocallyDefURIs = 0;
4: procedure COMPUTE(⟨?s, ?p, ?o, ?g⟩)
5:     if equalsURI(?p, rdf:type) && isURI(?s) then
6:         accumulatedURIsLength += lengthOfURI(?s);
7:         countLocallyDefURIs++;
8: return accumulatedURIsLength / countLocallyDefURIs;
```

number in the range [0, $+\infty$). Lower values represent better rankings.

4.2.8 Understandability

Data understandability is a very important prerequisite for an information consumer.

Empty Annotation Value In some languages, e.g. OWL annotation properties are distinguished. 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 cause 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 15 Empty Annotation Value Algorithm

```
1: procedure INIT
2:     totalAnnotations = 0 ;
3:     emptyAnnotations = 0 ;
4: procedure COMPUTE(⟨?s, ?p, ?o, ?g⟩)
5:     if isAnnotation(?p) then totalAnnotations++;
6:     if isEmpty(?o) then emptyAnnotations++ ;
       return emptyAnnotations/totalAnnotations
```

The metric values vary in the interval [0,1], where the 0 indicates the best quality.

Human Readable Labelling

Labels Using Capitals

Low Blank Node Usage

Whitespace in Annotation

4.2.9 Dynamicity Category

Currency Dimension

Virtually all application domains are interested in getting access to data that is as up-to-date as possible. Actually, it is generally the case that data sources decrease in value as they become outdated. Therefore, the currency dimension is a key feature of Liked Data resources. The following metrics, intended to assess the currency of datasets, are based in the definition of this dimension provided by Kahn et al. ([**?**]) as ”the degree to which information is up-to-date”.

Currency of Documents/Statements Metric

This metric is based in the definition of currency provided by Rula et al. ([**?**]) as "the age of a value, where the age of a value is computed as the difference between the current time (the observation time) and the time when the value was last modified". The age of a dataset can be computed at both, the resource level (by comparing its last time of modification with the observation time) and the triples level (by comparing the value of last-modified statements instead).

> *Measures the degree to which data is up to date, by comparing the time when the data was observed (approximately the current time), with the time when the data (the document and each triple) was last modified.*

As listed in algorithm 16, the metric is computed as the average of the comparisons of the observation time versus the last modification time of each triple, normalized by the time elapsed since the publication of the document and the observation time. The last modification time is extracted from the properties: *http://purl.org/dc/terms/modified* and *http://semantic-mediawiki.org/swivt/1.0#wikiPageModificationDate*. Likewise, the publication time of the document is extracted from *http://purl.org/dc/terms/created*, *http://purl.org/dc/terms/issued* and *http://semantic-mediawiki.org/swivt/1.0#creationDate*, as suggested by the study conducted by Rula et al. at [**?**]. The metric will return the average of the difference between one and

Algorithm 16 Currency of Documents/Statements Algorithm

```
1:  procedure INIT
2:      accumTimeDiffs = 0;
3:      countModifiedObjs = 0;
4:      observationTime = getCurrentTime();
5:      publishingTime = null;
6:  procedure COMPUTE(⟨?s, ?p, ?o, ?g⟩)
7:      if isLastModifiedTimeURI(?p) then
8:          accumTimeDiffs += (observationTime - parseAsTime(?o));
9:          countModifiedObjs++;
10:     if isPublishingTimeURI(?p) then
11:         publishingTime = parseAsTime(?o);
12: return (countModifiedObjs-(accumTimeDiffs/(observationTime-publishingTime)))/countModifiedObjs;
```

the ratio of the time elapsed since the last modification of each triple to the total time the dataset has been available. Since the former time span cannot be larger that the latter, the expected range is [0, 1]. The higher the value, the better, as it reflects how recently have the resources been updated. A value of 0 indicates that the resources have never been updated after their publication.

Time Since Modification Metric

As suggested above, the difference between the observation time (the time at which the resource is examined) and the time when the data was last modified, is a natural measure of how current is the information provided by a dataset. In contrast with the Currency of Documents/Statements metric, a simpler approach can be taken in order to measure the degree to which data is up to date. This metric does so by computing the plain difference between the observation time and the time when the data values were last modified (note that the normalization factor, based on the publishing time of the document is omitted here).

> *Provides a measure of the degree to which information is up to date, by taking the average difference between the observation time (i.e. the instant when the present calculation of the metric was initiated) and the time when the data (each triple in the dataset) were last modified.*

The computation performed by the metric is detailed in algorithm 17. The observation time corresponds to the instant when the calculation of the metric was initiated and only the triples providing a last time of modification are processed during the computation. The result is in the range [0, $+\infty$] and represents an amount of time in

Algorithm 17 Time Since Modification Algorithm

```
1: procedure INIT
2:     accumTimeDiffs = 0;
3:     countModifiedObjs = 0;
4:     observationTime = getCurrentTime();
5: procedure COMPUTE(⟨?s, ?p, ?o, ?g⟩)
6:     if isLastModifiedTimeURI(?p) then
7:         accumTimeDiffs += (observationTime - parseAsTime(?o));
8:         countModifiedObjs++;
9: return accumTimeDiffs/countModifiedObjs;
```

milliseconds. The lower the value, the better, as this metric measures the average time elapsed since the last modification of the information contained in the resource. That is, high values indicate that data provided by the resource was last updated long ago.

Exclusion of Outdated Data Metric

The currency of a Linked Data resource can also be measured relative to the amount of outdated entities it contains. However, in order to do so, it is necessary to somehow be able recognize when an entity is to be deemed as outdated, which in turn requires that temporal metadata is available and represented according to an appropriate model, such as those proposed by Rula et al. in [**?**]. In this particular case, we consider an entity to be outdated, if information about its period of validity is provided and according to it, data has already expired. The proportion of outdated data present in a dataset is an important quality factor, since it is usually the case that outdated data is no longer valid.

> *Determines the extent to which information provided in a dataset is outdated, by comparing the total number of entities described by the dataset, versus how many of those that are recognized to be outdated.*

The implementation of the metric, as presented in algorithm 18, makes use of the property *http://purl.org/dc/terms/valid* (when provided) to determine the expiration time of each examined triple and then, whether the described entity is outdated. After processing all triples of the dataset, the metric is computed as the ratio of the number of outdated entities to the total number of entities in the dataset. The metric values vary in the

Algorithm 18 Exclusion of Outdated Data Algorithm

```
1: procedure INIT
2:     countTotalOutdatedObjs = 0;
3:     countTotalDescribedObjs = 0;
4:     observationTime = getCurrentTime();
5: procedure COMPUTE(⟨?s, ?p, ?o, ?g⟩)
6:     if isValidTimeURI(?p) then
7:         if observationTime < parseAsTime(?o) then
8:             countTotalOutdatedObjs++;
9:     if equalsURI(?p, rdf:type) && isURI(?s) then
10:        countTotalDescribedObjs++;
11: return 1-(countTotalOutdatedObjs/countTotalDescribedObjs);
```

interval [0, 1], where a value of 1 indicates the best quality (no entities were found to be outdated).

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 19 Versions Volatility Algorithm

```
1: procedure INIT
2:     numberOfChanges = 0
3: procedure COMPUTE
4:     numberOfChanges = countSimpleChanges(v1,v2)
5: 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.

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.*

Algorithm 20 Average Volatility Metric Algorithm

```
1: procedure INIT
2:     changesTotal = 0
3:     versionsNo = 0
4:     retValue = 0
5: procedure COMPUTE
6:     Versions[] = countVersions(SPARQL)
7:     for all v[i], v[i+1] ∈ Versions do
8:         changesTotal = changesTotal + countSimpleChanges(v[i], v[i+1])
9: retValue = changesTotal / versionsNo -1
10: return retValue
```

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 21 Weighted Volatility Metric Algorithm

```
1: procedure INIT
2:     aggregSChanges = 0
3:     versionsNo = 0
4:     retValue = 0
5: procedure LOADWEIGHTS
6:     weights[] = fetchWeights()
7: procedure COMPUTE
8:     versions[] = countVersions(SPARQL)
9:     for all v[i], v[i+1] ∈ versions do
10:        for all w[j] ∈ weights do
11:            changesTotal = changesTotal + w[j] * countSimpleChanges(v[i], v[i+1])
12: retValue = aggregSChanges / versionsNo -1
13: return retValue
```

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

Time Validity Interval Metric

An alternative to measure how frequently the data varies with time, is to make use of information about the expiration of entities in order to assess how long does data remain valid. Consequently, such measurement would require two pieces of information about entities described in the dataset: the expiry time of entities (i.e. validity) and the time when data became available (i.e. publishing time). By combining these two components, the interval during which the information will remain valid can be computed.

> *Estimates the frequency with which data will be updated, by calculating the average length of the time interval during which entities remain valid.*

As shown in algorithm 22, information about the expiration time and issued time of entities is extracted from the triples, thereby accumulating the data necessary to compute the average length of the validity interval of all entities for which such information is provided in the dataset. The result is in the range [0, $+\infty$] and represents an amount of time in seconds. Lower values correspond to better quality rankings, as they indicate that data is expected to be updated more frequently and therefore would be fresher.

Algorithm 22 Time Validity Interval Algorithm

```
1: procedure INIT
2:     mapObjsWithValidityInfo = new Map⟨URI, Object⟩();
3: procedure COMPUTE(⟨?s, ?p, ?o, ?g⟩)
4:     if isValidityTimeURI(?p) then
5:         setValidityTimeOfObj(getElementWithKey(mapObjsWithValidityInfo, ?s), parseAsTime(?o));
6:     if isPublishingTimeURI(?p) then
7:         setIssueTimeOfObj(getElementWithKey(mapObjsWithValidityInfo, ?s), parseAsTime(?o));
8: return calcTotalValidityMinusIssueTime(mapObjsWithValidityInfo)/countTotalElements(mapObjsWithValidityInfo);
```

Timeliness Dimension

The degree to which Linked Data is considered to be up-to-date or outdated can heavily depend on the task at hand. Depending on the application, information updated one month ago can be considered highly current (e.g. the list of pet shops in a city) or unacceptably outdated (e.g. a pricing list of foreign currencies). Therefore, in order to align the quality measures related to the age of data with the application of interest, it makes sense to provide currency measures relative to a specific task. The metrics part of this dimension, are based on the concept of timeliness as defined by Gamble et al. ([**?**]): "Timeliness is a measure of utility, is a comparison of the date the annotation was updated with the consumer's requirement".

Timeliness of the Resource Metric

If the expiration time of the data comprised by a Linked Data resource is provided, it is reasonable to expect such data to be updated before or shortly after that time. Otherwise, it would suggest that the data might be outdated. This, added up to the fact that the expiration time of information is closely associated to the application domain, allows to define Timeliness of the Resource as an application-domain-related metric that measures the currency of data.

> *Indicates how up-to-date data is, relative to a specific task, by measuring the difference between the invalid time (expiry time of the data) and the observation time (current time).*

Algorithm 23 illustrates how this metric is computed. The procedure determines whether the triple contains temporal information stating when the described entity expires (i.e. its validity). This is done by evaluating the property regarded as source of the Expiration/Valid Time (namely, *http://purl.org/dc/terms/valid*, as suggested by Rula et al. at [**?**]), if such a property is found, its value is subtracted from the current observation time, and the result is accumulated. The total accumulated differences will be used afterwards, to calculate the final value of the metric. The result of the metric is a real number in the range $[-\infty, +\infty]$, as it represents the average length of the

Algorithm 23 Timeliness of the Resource Algorithm

```
1: procedure INIT
2:     accumValidTimeDiffs = 0;
3:     countTotalAccountedObjs = 0;
4:     observationTime = getCurrentTime();
5: procedure COMPUTE(⟨?s, ?p, ?o, ?g⟩)
6:     if isExpirationTimeURI(?p) then
7:         accumValidTimeDiffs += (observationTime - parseAsTime(?o));
8:         countTotalAccountedObjs++;
   return accumValidTimeDiffs/countTotalAccountedObjs;
```

gap (in milliseconds) between the expiration time of the data and the actual time. The lower the value, the better, since a higher, positive value indicates that the resources are possibly outdated.

4.3 The User Interface

In the following section, we will describe a set of mockups which will be implemented as a Web User Interface (UI) for the assessment and ranking of datasets software prototype. The User Interface will be made up of 3 parts: Details, Statistics and Assessment. The Quality Framework Web-UI will be a mix of PHP and JavaScript and is envisioned to run on top of OntoWiki, 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. The CubeViz extension is used to visualise statistical graphs about the quality metadata of the datasets. The UI will communicate with the core assessment framework (cf. Figure **??**) via the defined RESTful APIs (cf. Section 2.4).

4.3.1 Details Tab

The Details Tab (Figure 11) will contain the ranked datasets according to a chosen metric by the user from the facet section. This is very similar to the datahub.io[25] interface. The user will also be able to enable filters to the datasets, and also to search through the resultant datasets. This tab will only display those datasets which have a corresponding Quality Graph, in order to enable the ranking and retrieval of datasets based on quality criteria.

The Details Tab will have the following functionality:

1. **Facets -** When the datasets are retrieved, a SPARQL query is performed to get all available quality categories used. The result set is used to populate the Category box in the facet. Once a category is chosen, the dimension box is populated dynamically via a SPARQL query which fetches the dimensions related to the chosen category. Similar procedure is done to populate the metric box. When a metric is chosen, then the datasets are ranked in ascending order on the right hand side of the container.

2. **Filters -** Filters will dynamically change according to the chosen metric. That is, if the `daq:expectedDatatype` (cf. Section 2.3) of a metric is boolean then we have a filter which can be a true/false checkbox, whilst if we have a metric with a double type then we could have some slider regulating the minimum value for the metric.

3. **Buttons -** The buttons have the same functionality as in CKAN. The additional button *quality metadata/⟨file type⟩* returns the quality graph in a specific format. The *Visualise Quality* button takes the user to the Statistics tab in the web page.

4.3.2 Statistics Tab

The Quality Assessment Framework Web-UI will also give the opportunity to its users to visualise statistical information about a dataset quality. The Statistical Tab (Figure 12) will present the user with a number of graphs (cf. Section 2.1.5 for the description of the different visualisation graphs). Users can compare different datasets together or even how a dataset changed in its quality over time.

4.3.3 Assessment Tab

The Quality Framework Web-UI will also allow users to assess datasets for their quality. Figure 13 shows a mockup of how the Assessment Tab will look like. The user is guided step by step to assess a dataset:

1. The user choose the dataset to be assessed or give a dataset URI;

2. User decide what metrics to be assessed;

[25] `http://www.datahub.io`

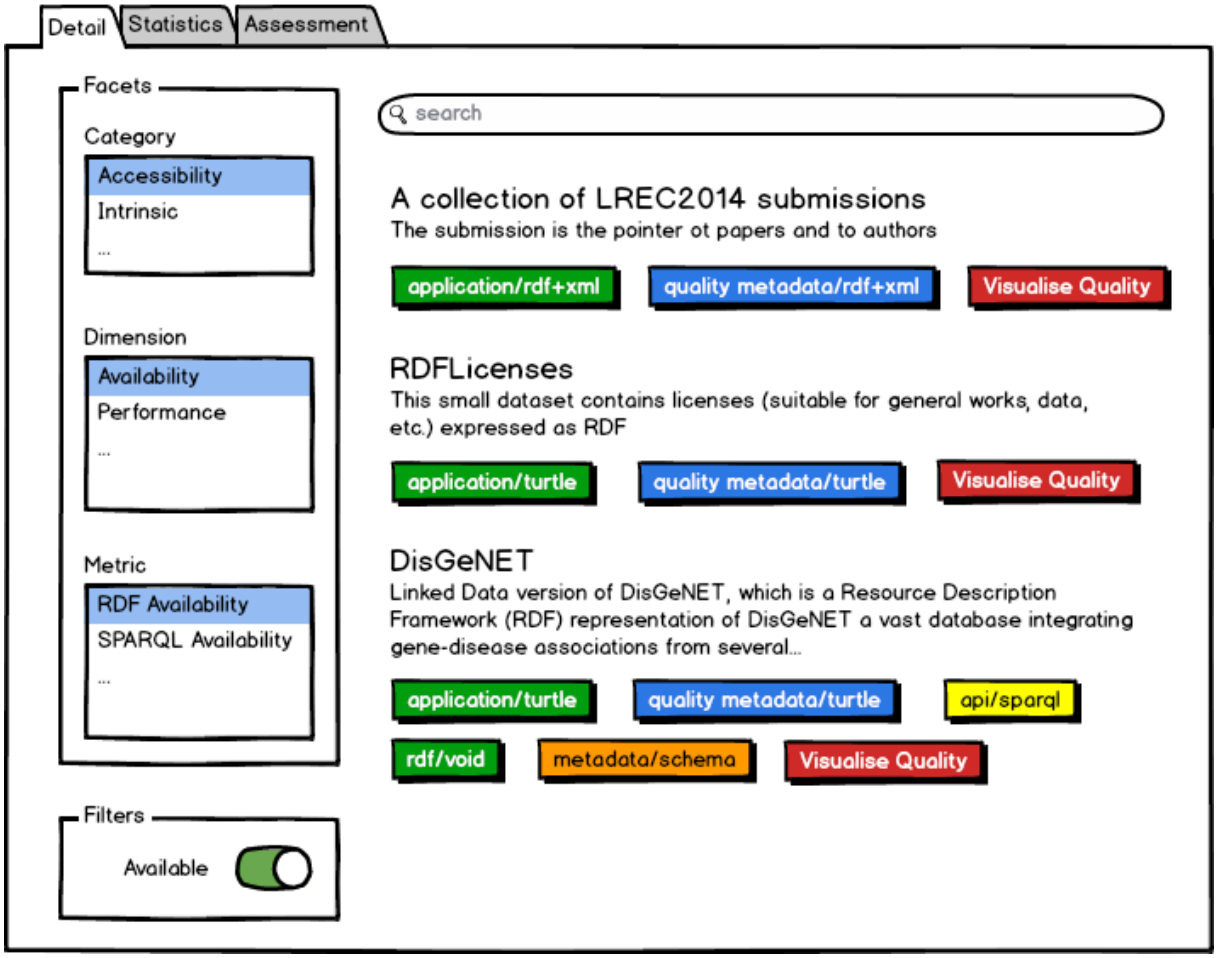

Figure 11: Mockup for the Details Tab

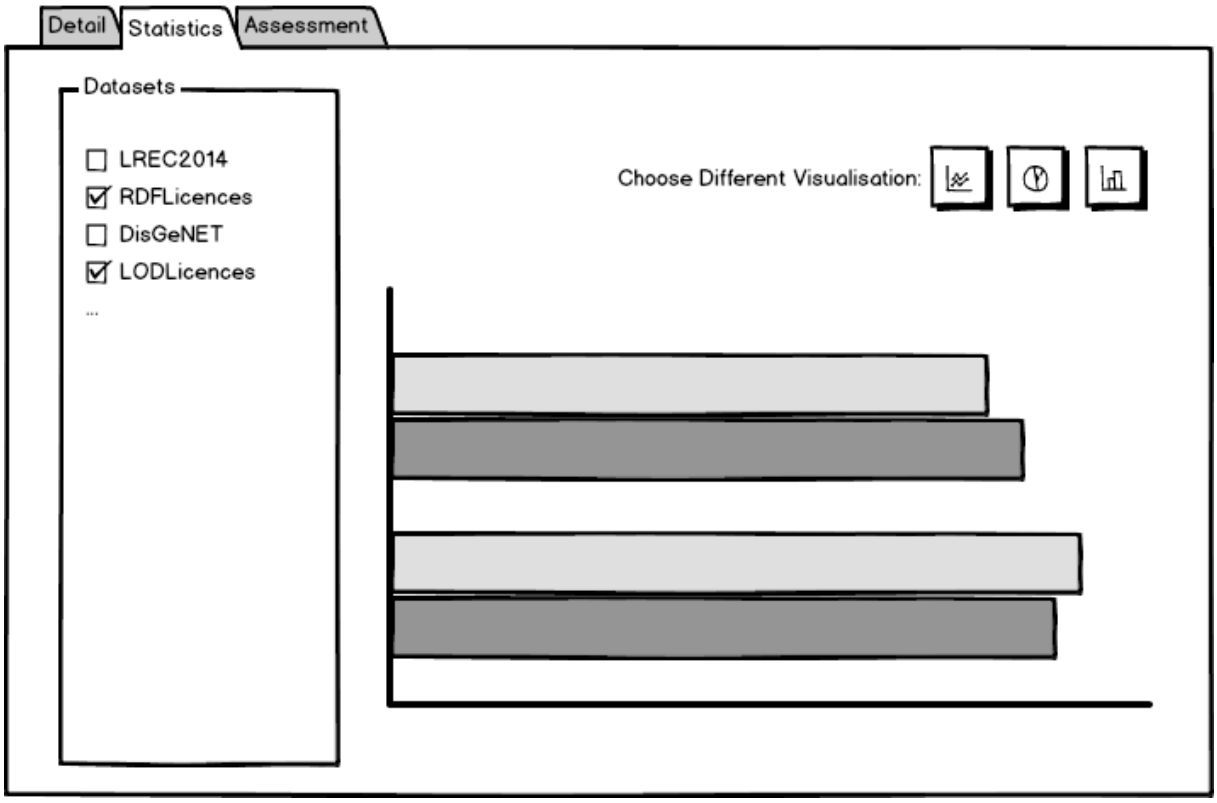

Figure 12: Mockup for the Statistics Tab

3. User decide if a quality problem report is required, which would allow the user to semi-automatically clean up the data (cf. Deliverable 3.1);

4. User clicks the *assess* button to start assessing the chosen datasets.

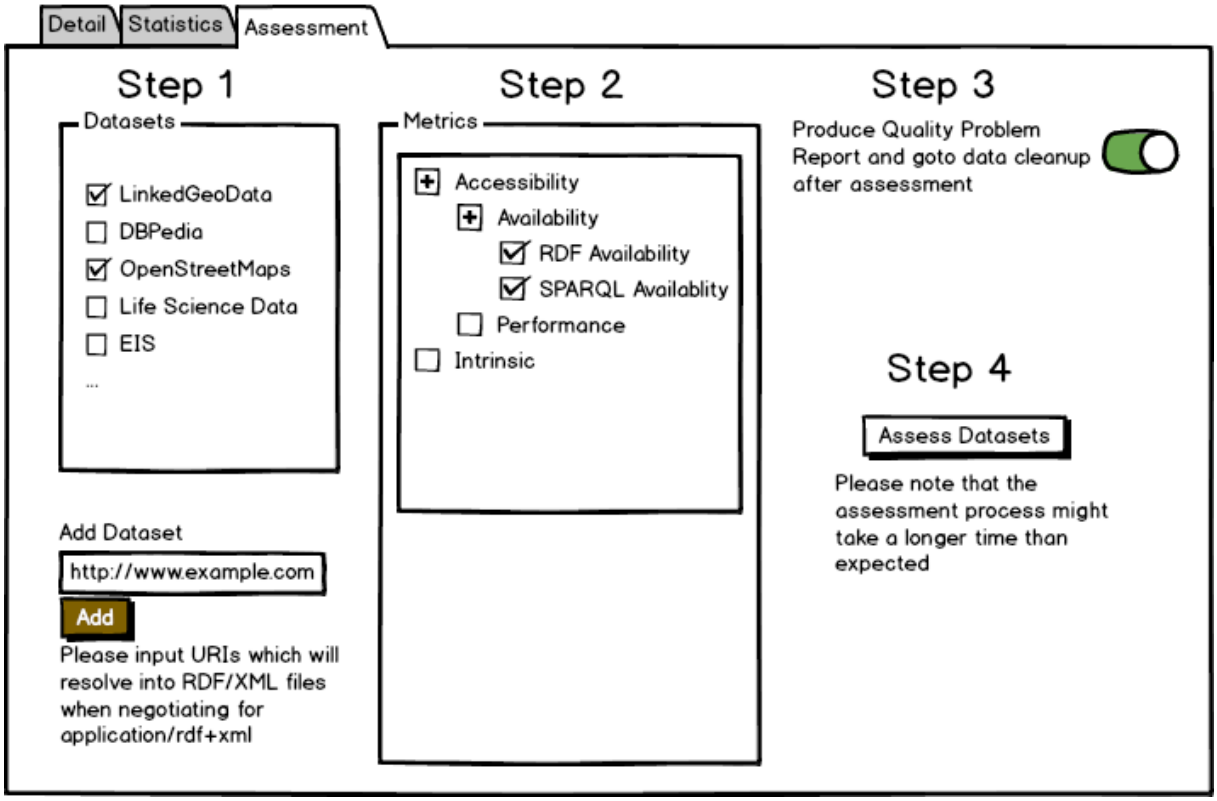

Figure 13: Mockup for the Assessment Tab

4.4 Ranking of Quality-Computed Datasets

Tools for data consumers, such as CKAN, usually provide features such as faceted browsing and sorting, in order to allow prospective dataset users to search within the large dataset archive. Using faceted browsing, datasets could be filtered according to tags or values of metadata properties. The datasets could also be ranked or sorted according to values of properties such as relevance, size or the date of last modification. With many datasets available, filtering or ranking by quality can become a challenge. Talking about "quality" as a whole might not make sense, as different aspects of quality matter for different applications. It does, however, make sense to restrict quality-based filtering or ranking to those quality categories and/or dimensions that are relevant in the given situation, or to assign custom weights to different dimensions, and compute the overall quality as a weighted sum. The daQ vocabulary provides flexible filtering and ranking possibilities in that it facilitates access to dataset quality metrics in these different dimensions and thus facilitates the (re)computation of custom aggregated metrics derived from base metrics. To keep quality metrics information easily accessible, each assessed dataset contains the relevant daQ metadata graph in the dataset itself.

We aim to achieve a quality-biased ranking of LOD datasets, promoting:

1. High quality datasets;

2. Datasets on which a larger number of quality metrics is calculated.

Therefore, datasets of poor quality but having more quality metadata might end up ranked higher than those with excellent quality on the only metric assessed on. Publishers might be doubtful about the data they publish and they will hide their doubt in the dataset itself. One possibility is that the publishers might hide this psychological nature by claiming that their dataset is of good quality in certain aspects. Quality assessment frameworks should not be only about positive or negative assertions of a the dataset, but also it should place the publishers doubts in the assertion spectrum. With our proposed ranking algorithm, such effects will also be given a small share of the *ranking-glory*.

The approach we take for ranking takes into consideration the total number of metrics assessed by the most complete dataset available in the datastore[26]. It also takes into consideration any facet filters chosen by the user, enabling the dynamic change of weights and thus the final ranking. The ranking algorithm is split into a number of steps:

1. Get the total number of metrics assessed (m);
2. Adjust weight for those metrics selected in the facet (Definition 2);
3. Adjust weight for the rest of the metrics (Definition 3);
4. Calculate the metric value by weight to find out ranking (Definition 4) and rank;

4.4.1 Weight Assignment

The weight assignment is the most crucial aspect of the ranking algorithm. It should be evenly distributed amongst the chosen (filtered) metrics, whilst also giving a share of the weight to the other metrics. In this way we ensure the quality-bias ranking and promote not just the high quality datasets, but also giving a smaller share to those that give more information about quality than others. In Definition 2 we calculate a weight value for the number of metrics chosen (f_m). Together with the number of metrics chosen, we add 1 to represent a small share which will be divided equally with the rest of the metrics ($\overline{f_m}$ represents the metrics not chosen). This is defined in Definition 3. To explain this in a simpler manner, if we have five metrics and only one was chosen, then the chosen metric will have a weight of 0.5 whilst the rest will have 0.5 shared equally (i.e. 0.125).

Definition 2.

$$\theta = \frac{1}{(f_m + 1)}$$

Definition 3.

$$\rho = \frac{1}{((f_m + 1) \times \overline{f_m})}$$

4.4.2 Ranking

After the weights have been distributed, the ranking algorithm retrieves all the metric values from the quality graph. The values of chosen metrics f_{m_i} (where i is a range from 0 to the number of possible metrics) are added together and multiplied by the weight θ. Similarly, those which are not chosen ($\overline{f_{m_i}}$) are multiplied by the weight ρ. These two are added together which gives us a value τ for the dataset (Definition 4). This is repeated for all possible datasets and then are ranked accordingly.

Definition 4.

$$\tau = (\sum_{f_m}^{m} f_{m_i} \times \theta) + (\sum_{\overline{f_m}}^{m} \overline{f_{m_i}} \times \rho)$$

[26] We are assuming that the metrics used are common to all datasets in the domain

Listing 9 shows a typical configuration of the retrieval of metric assessment values from the Quality Graph. In this query the metric value of the latest observation is taken into consideration.

```
SELECT ?metric, ?value WHERE {
  ?graph a daq:QualityGraph .
  GRAPH ?graph {
    ?metric a ?metricType .
    ?metric daq:hasObservation ?obs .
    ?obs daq:value ?value .
    ?obs daq:dateComputed ?dateComputed .
  }
  GRAPH <http://www.diachron-fp7.eu/dqm#> {
    ?metricType rdfs:subClassOf daq:Metric .
  }
} ORDER BY DESC(?dateComputed) LIMIT 1
```

Listing 9: Retrieving metric assessment value from the Quality Graph.

4.4.3 Ranking Example

In the following subsection we introduce an example to further help the reader to understand how ranking will work. Consider the following dataset scenario in Table 4.4.3. In our datastore we have four datasets each having a Quality Graph with a number of assessed metrics and their values. *Dataset D* has been assessed by all eight metrics whilst the others have been assessed with six, five and five respectively.

Metric	Dataset A	Dataset B	Dataset C	Dataset D
A	0.2	0.9	0.5	0.3
B	0.5	0.9	0.8	0.4
C	0.9	0.3	0.7	0.7
D	0.8	-	0.9	0.1
E	0.7	-	0.4	0.1
F	-	0.2	-	0.5
G	-	0.8	-	0.9
H	0.3	-	-	0.9

Table 1: Dataset example

In Table 4.4.3 we show how the ranking value τ would dynamically change for different scenarios. Initially, no filter is chosen by the user. A possible scenario for this is on page load. When no filter is chosen (in this example) all weights have a ρ value of 0.125. *Dataset D* will be ranked first, followed by *Dataset A*, *Dataset C* and *Dataset B*. This shows that although at first glance it seems that *Dataset B* is a high quality dataset (three out of five assess quality metrics are over 0.8), it got penalised by the fact that other datasets had more quality metadata about their dataset. The single filters **A** and **E** gave expected results, with *Dataset B* and *Dataset A* being top ranked respectively. The multiple filter **A,B** also gave expected results, with *Dataset D* being ranked third ahead of *Dataset A* due to the it having more quality metadata. More examples of different scenarios are in the Table 4.4.3, where the blue coloured cells shows the top-ranked dataset.

5 Crawling Service

6 Conclusions

Chosen Filter	Dataset A	Dataset B	Dataset C	Dataset D		θ	ρ
No Filter	0.425	0.3875	0.4125	0.4875		1	0.125
A	0.32857143	0.60714286	0.45	0.40714286		0.5	0.07142857
E	0.54285714	0.22142857	0.40714286	0.32142857		0.5	0.07142857
A,B	0.38333333	0.67222222	0.54444444	0.41111111		0.33333333	0.05555556
A,B,C	0.49	0.575	0.565	0.475		0.25	0.05
A,E	0.43888889	0.42222222	0.43333333	0.32777778		0.33333333	0.05555556
B,D,E,H	0.515	0.29	0.48	0.42		0.2	0.05

Table 2: Dataset Ranking for different Scenarios