SANSA Tutorial

In this tutorial, we will introduce and describe different functionalities of SANSA. First, we will cover a brief introduction to the underlying processing engine of SANSA, Apache Spark. Later, we will dive into the details of SANSA's various building blocks organized as its four layers described below.

Introduction to RDF

A Resource Description Framework (RDF) is a W3C standard for describing resources. A resource is a fact or a thing which can be described and identified. A person, a home page, this tutorial is a resource. An RDF resource is identified by a URI reference.
Through this tutorial, we will consider people as resources. An RDF graph is a directed graph containing nodes and arrows. See Figure 1. below as one example of a triple.

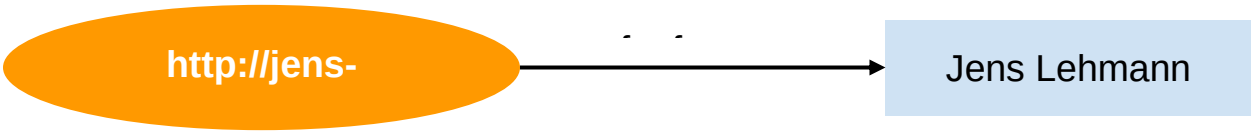


Figure 1 : An RDF graph excerpt.

The resource, “Jens Lehmann”, is identified by a Uniform Resource Identifier (URI). Each resource contains a property which represent the relation between the subject (the resource itself) and its value (object).

Triples

Each arc in an RDF model is known as a triple. Each triple describe a fact about a resource. It contains : the subject -- the resource being described, the predicate -- the property that labels the connection, and the object -- the resource or literal pointed to by the predicate.

SANSA Introduction

Scalable Semantic Analytic Stack (SANSA)[1] [1] framework is an open-source[2] distributed data flow engine which allows scalable analysis of large-scale RDF datasets to facilitate applications such as link prediction, knowledge graph completion and reasoning.

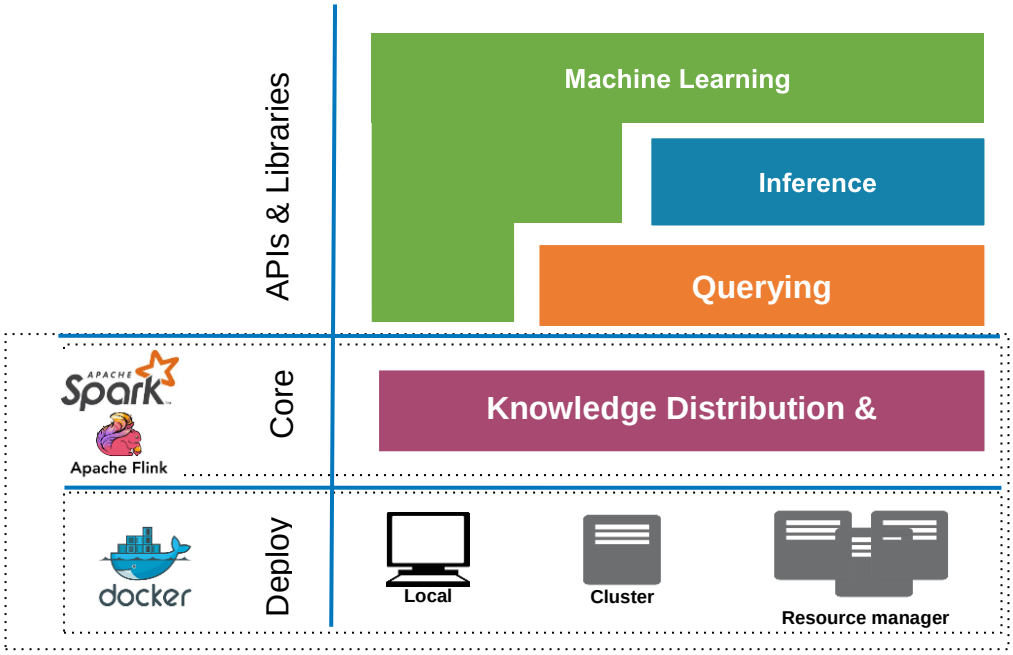


Figure 2 : SANSA Stack overview

[1] http://sansa-stack.net/
[2] https://github.com/SANSA-Stack

Setup

SANSA's interactive Spark Notebooks will be used to demonstrate the APIs of different layers. SANSA-Notebooks are easy to deploy using docker-compose. The deployment stack includes Hadoop for HDFS, Spark for running SANSA APIs, Hue for navigating and copying file to HDFS. All examples and code snippets will be shown and prepared using SANSA-Notebooks throughout this tutorial. We recommend setting up SANSA-Notebooks for this tutorial, however we provide a guideline on how to setup Scala-IDE as well in case someone prefers using the IDE.

Scala IDE

This tutorial will guide you to setup SANSA on your Scala-IDE[3]. At the time of writing, the latest version of Scala-IDE is 4.7.0. The version of Java used on this tutorial is 1.8. The operation system used is Ubuntu 16.04, but overall this is not a problem. The only environment requirement are Scala-IDE, Java, Maven, and git to check out the source code of the SANSA.

Scala-IDE comes with Maven integration by default. Follow the instructions on our "Getting started with SANSA"[4] to check out the code from the Git repository. In this tutorial we will use our Maven template[5] to generate a SANSA project using Apache Spark.

```
git clone https://github.com/SANSA-Stack/SANSA-Template-Maven-Spark.git
cd SANSA-Template-Maven-Spark

mvn -U install
```

After you are done with the configurations mentioned above, you will be able to import the project.

In order to import a Maven project, you should use the Import from the Scala-IDE. Right click somewhere on the left panel (Package Explorer) and then choose Import. The Import form will give you the option to choose the way to import project from. In our case, we will be using the Maven template, therefore, choose "Existing Maven Projects". Figure 3. list the options to import maven projects.

[3] http://scala-ide.org/
[4] http://sansa-stack.net/downloads-usage/#IDE_Setup
[5] https://github.com/SANSA-Stack/SANSA-Template-Maven-Spark

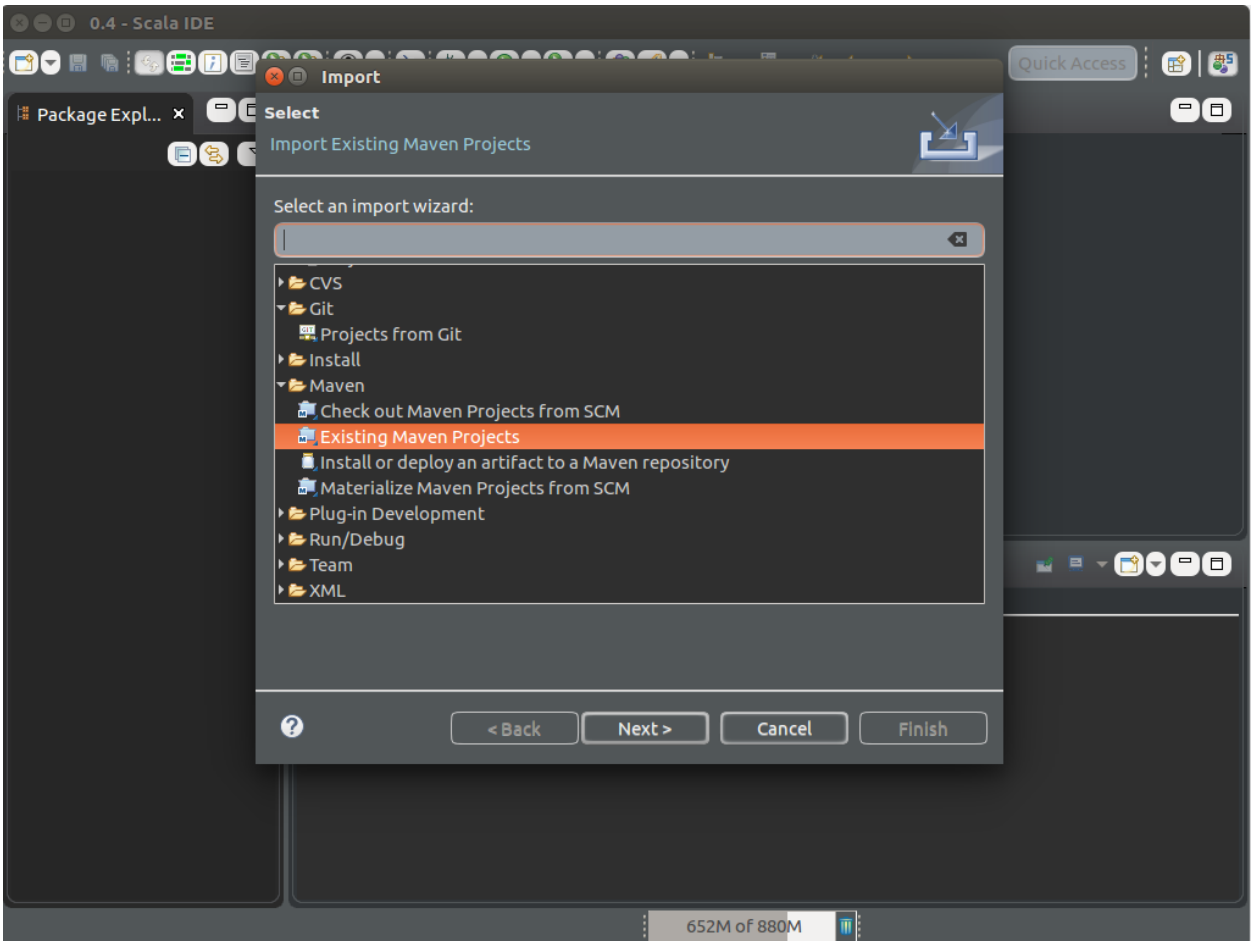


Figure 3 : Import Maven project into Scala-IDE.

After you choose the SANSA-Template-Maven-Spark into Scala-IDE, you could import the project.

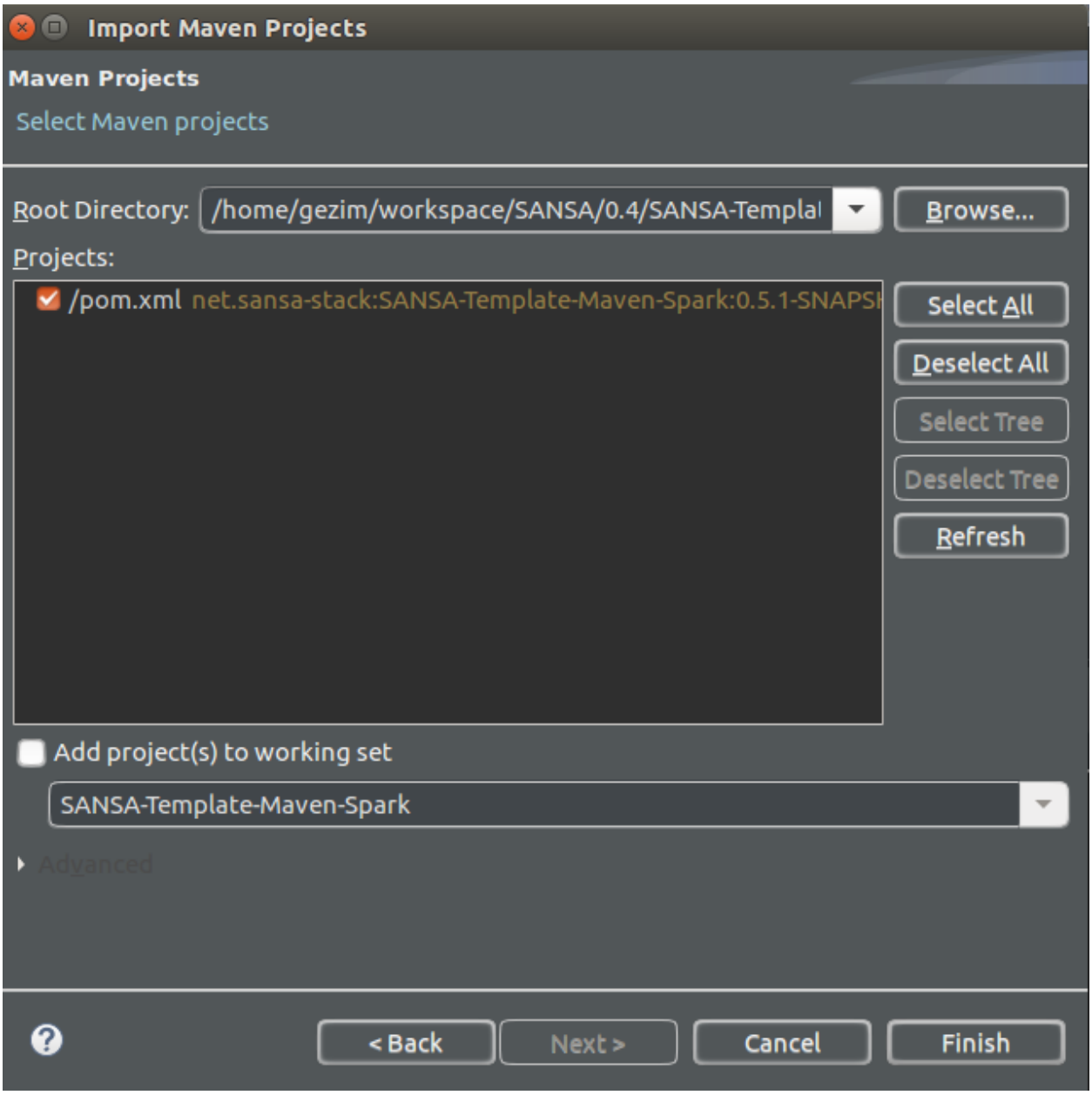


Figure 4 : SANSA Maven project into Scala-IDE.

When the import is done, the default Scala version will be 2.12.x (see Figure 5.). Please, change it to Scala 2.11.11 as that is the version SANSA supports.

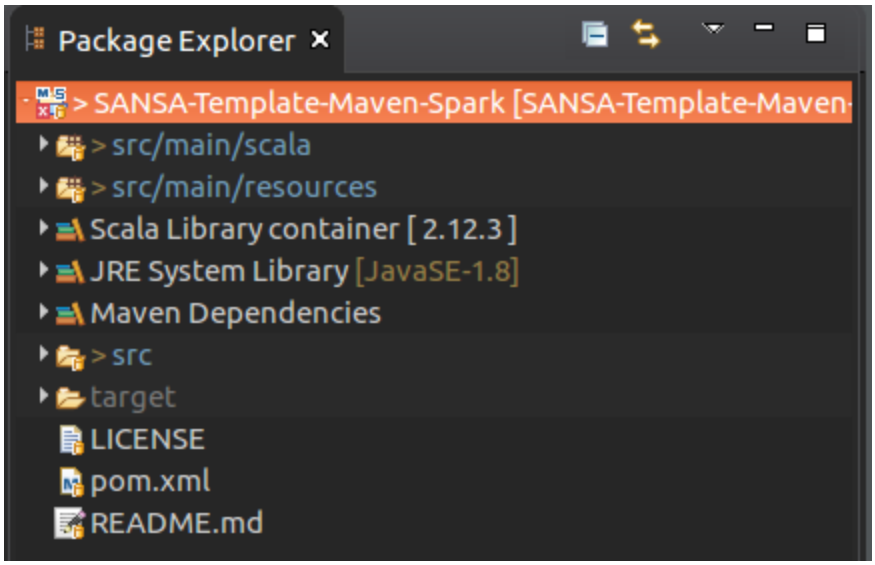


Figure 5 : SANSA Maven project structure.

For changing the Scala version you should do these steps:

1. Right click on the “Scala Library container [2.12.x]” → Properties,
2. Select Scala 2.11.11 from the Scala library container list (see Figure 6.)

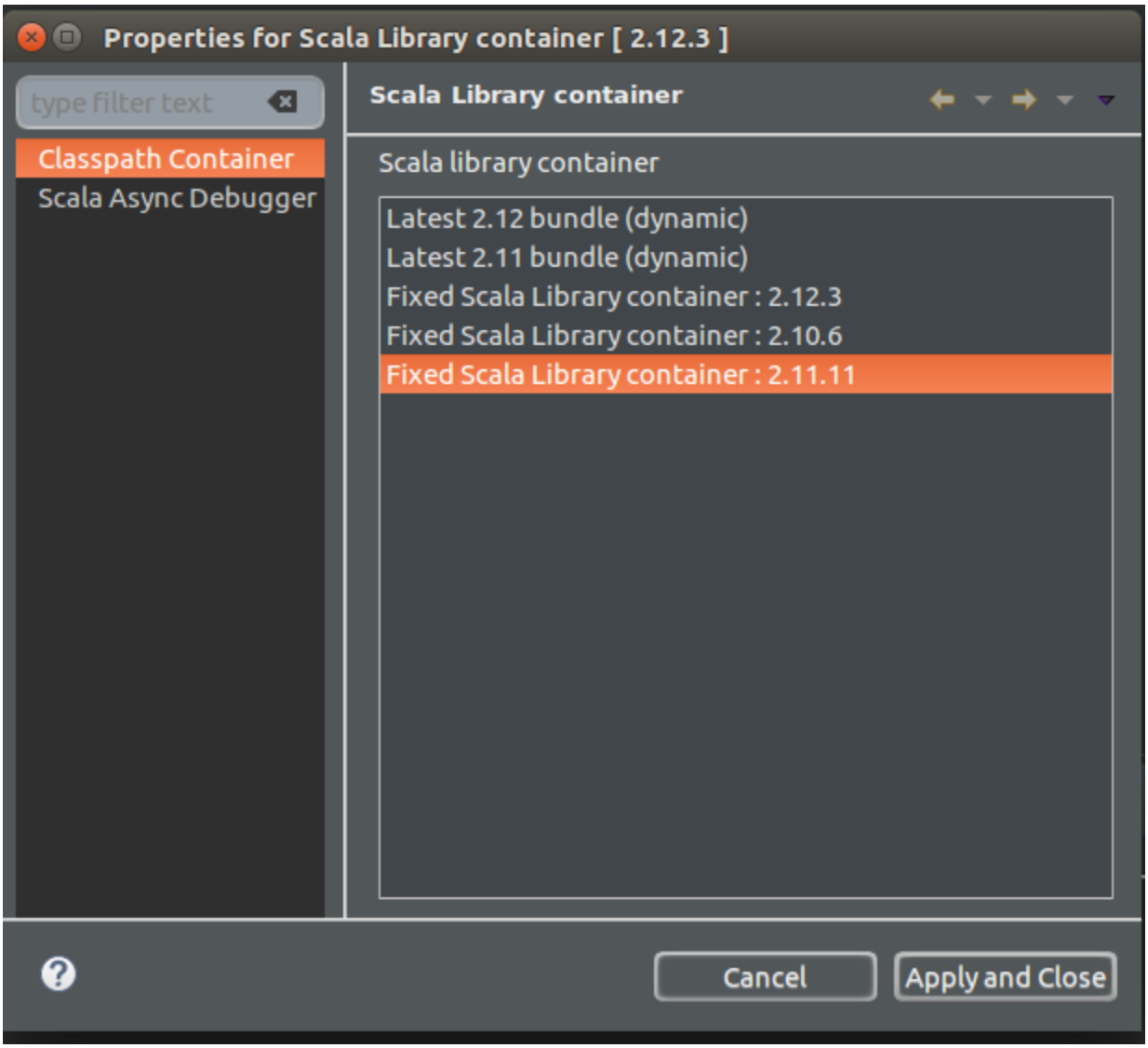


Figure 6 : Scala Library container.

In case of missing depencies from the SANSA stack on your maven (local) repository and you haven't done `mvn -U install` when cloning the repo. Do it manually by :

1. Right click on the SANSA-Template-Maven-Spark project on the left side of the IDE
2. Go to Maven → Update Project and then choose "Force Update of Snapshots/Releases" (see Figure 7.) in order to get the SNAPSHOTS releases of the stack.

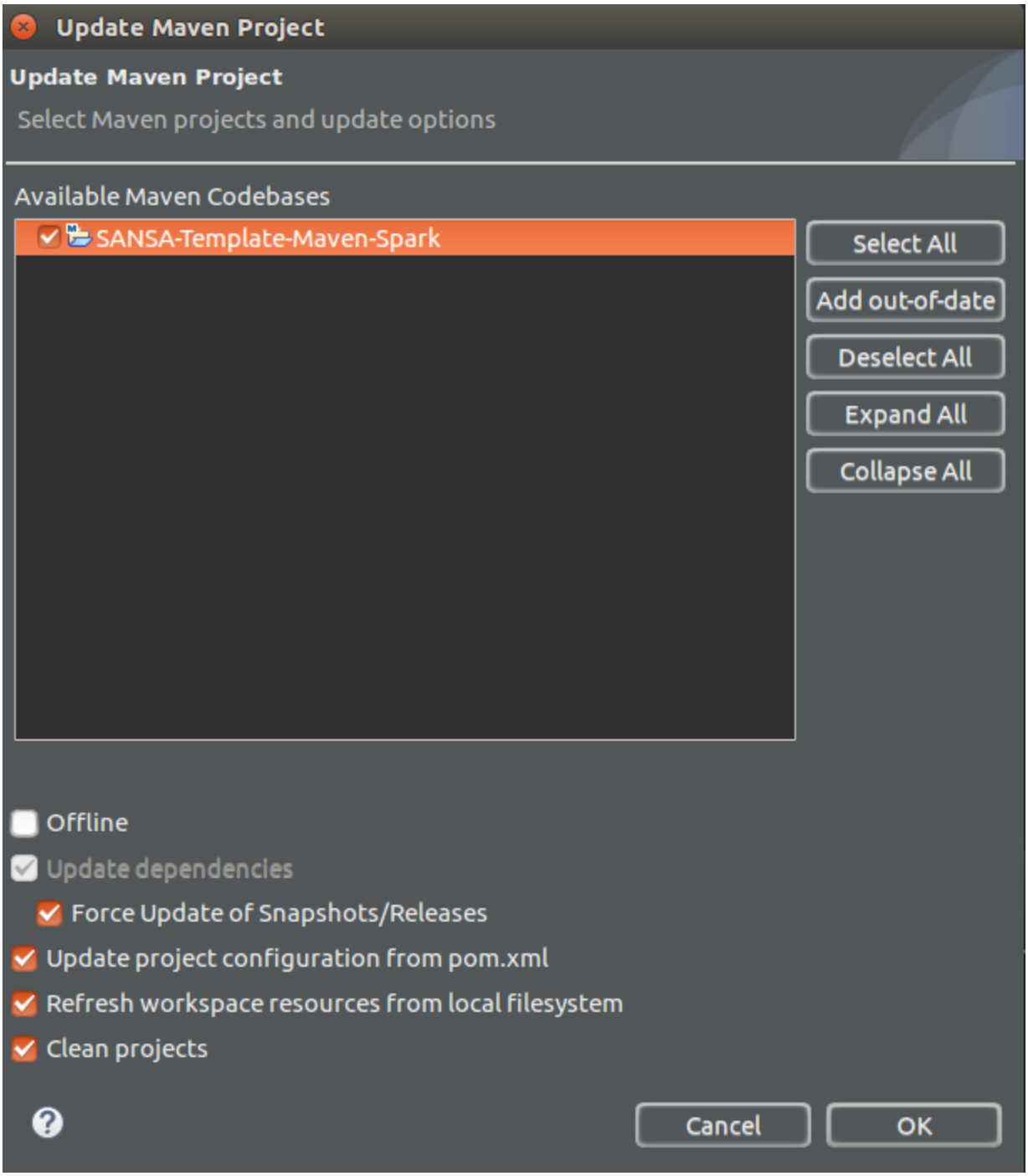


Figure 7 : Update SANSA maven project.

After you are done (you may need to wait a bit when run for the first time, due to dependency downloads) with the setup, you will be able to run SANSA code on your IDE using Apache Spark. Enjoy it :).

SANSA Notebooks

As another alternative to the IDE, we provide you a developer friendly access to SANSA — the SANSA-Notebooks[6]. SANSA's interactive Spark Notebooks will be used to demonstrate the APIs of different layers. SANSA-Notebooks are easy to deploy using docker-compose. The deployment stack includes Hadoop for HDFS, Spark for running SANSA APIs, Hue for navigating and copying file to HDFS. All examples and code snippets will be shown and prepared using SANSA-Notebooks throughout this tutorial.

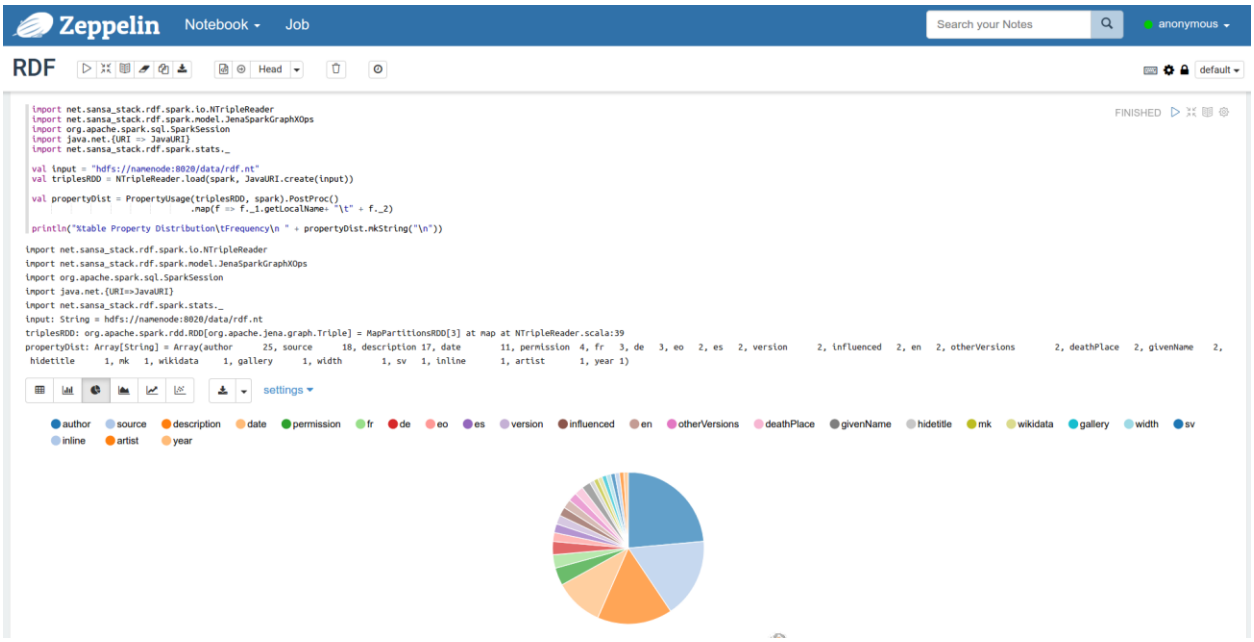


Figure 8 : SANSA Notebooks Running RDF Examples.

During this tutorial we will run SANSA-Notebooks on the cluster locally.

In order to run SANSA notebooks your system should fulfill these requirements:

- `docker` ≥ 1.13.0 [instructions]
- `docker-compose` ≥ 1.10.0 [instructions]
- around 5 GB of disk space for the Docker images.

After you have installed the docker, the notebooks can be **downloaded**, **deployed** and **run** by just doing:

```
git clone https://github.com/SANSA-Stack/SANSA-Notebooks

cd SANSA-Notebooks
```

[6] https://github.com/SANSA-Stack/SANSA-Notebooks

Since SANSA-Notebooks bundle the SANSA-Examples, you should get the SANSA Examples jar file. To do so, just run:

```
make
```

When the jar has been downloaded, start the cluster (this will lead to downloading BDE docker images, it may take a while :)):

```
make up
```

When start-up is done you will be able to access the following interfaces:

- http://localhost:8080/ (Spark Master)
- http://localhost:8088/home (Hue HDFS Filebrowser)
- http://localhost/ (Zeppelin)

To load the data to your cluster simply do:

```
make load-data
```

Go on and open Zeppelin (http://localhost/), choose any of the notebooks available and try to execute them.

Note: If the Notebook doesn't recognize the Spark interpreter, you could refresh the page by : Go to Spark interpreter, press Refresh and then "Save" it. After you are done, refresh the page or "F5" and you are ready to go.

After you are done with the notebooks, you can stop the whole stack by:

```
make down
```

RDF Processing Layer

SANSA provides APIs to load/store RDF data in several native formats, from HDFS or a local drive. SANSA has a rich set of functionalities that can perform distributed manipulations over given data. This includes filtering, computation of statistics [2], and quality assessment over large-scale RDF datasets, in a distributed manner ensuring resilience and horizontal scalability.

SANSA uses the RDF data model for representing graphs consisting of triples with subject, predicate and object. RDF datasets may contain multiple RDF graphs and record information about each graph, allowing any of the upper layers of SANSA (Querying and ML) to make queries that involve information from more than one graph. Instead of directly dealing with RDF datasets, the target RDF datasets need to be converted into an RDD of triples. We name such an RDD the main dataset. The main dataset is based on an RDD data structure, which is a basic building block of the Spark framework. RDDs are in-memory collections of records that can be operated on in parallel on large clusters.

Reading RDF

SANSA provide mechanism of reading RDF model in the format of RDD/DataFrame/Dataset of triples.

Listing 1 read RDF into NTRIPLES format and generate RDD representation of it (*Line 7*). SANSA support different RDF serialization formats[7] (e.g. NTRIPLES/N3, XML/RDF, TURTLE, QUAD).

Listing 1. Triple reader example.

```
1  import net.sansa_stack.rdf.spark.io._
2  import org.apache.jena.riot.Lang
3
4  val input = "hdfs://namenode:8020/data/rdf.nt"
5  val lang = Lang.NTRIPLES
6
7  val triples = spark.rdf(lang)(input)
8
9  triples.take(5).foreach(println(_))
```

Writing RDF

After reading RDF triples, one can save them back in the format of NTRIPLES. SANSA, provide such functionality out-of-the-box.

[7] https://github.com/SANSA-Stack/SANSA-RDF/tree/develop/sansa-rdf-spark/src/main/scala/net/sansa_stack/rdf/spark/io

Listing 2. Read an RDF graph and represent it as an RDD of triples (*Line7*) . Afterwords, it save it back to disk as an NTRIPLE file (*Line 9*) .

Listing 2. Triple writer example.

```
1  import net.sansa_stack.rdf.spark.io._
2  import org.apache.jena.riot.Lang
3
4  val input = "hdfs://namenode:8020/data/rdf.nt"
5  val lang = Lang.NTRIPLES
6
7  val triples = spark.rdf(lang)(input)
8
9  triples.saveAsNTriplesFile(output)
```

SANSA RDF Models

Within the SANSA you can represent triples in different formats (e.g. RDD, DataFrame, Dataset, or Graph). For such representation we provide a triple operations as a build in operation in the SANSA model. The transformation among these transformation is also possible. Below we will include such operations and their transformations.

Operations on Models

The very basic transformation in SANSA is so-called RDD of triples. Each triple which contains <subject, predicate, object> is represented in the format of immutable block of records using the RDD → RDD[Triple].

This model a set of triple operations in the format of RDDs. We will cover (see Listing 3. for more details) some of the most important functions implemented in SANSA such as: union (Line 21), difference (Line 23), add (Line 34), remove (Line 43), contains (Line 58), and more which work on the triple level.

Listing 3. RDD triple operations.

```
1  import net.sansa_stack.rdf.spark.io._
2  import net.sansa_stack.rdf.spark.model._
3  import org.apache.jena.graph.{Node, NodeFactory, Triple}
4  import org.apache.jena.riot.Lang
5
6  val input = "hdfs://namenode:8020/data/rdf.nt"
7  val lang = Lang.NTRIPLES
8
9  val triples = spark.rdf(lang)(input)
10
```

```scala
// getting all the subjects
val graph = triples.getSubjects()
// filtering subjects which are URI
val graph = triples.filterSubjects(_.isURI())
// filtering objects which are literals
val graph = triples.filterObjects(_.isLiteral())

// another RDD of triples
val other = spark.rdf(lang)(input)

// union of two RDF graph
val graph = triples.union(other)

// difference of two RDF graph
val graph = triples.difference(other)

// intersection of two RDF graph
val graph = triples.intersection(other)

// add a statement to the RDF graph
val triple = Triple.create(
      NodeFactory.createURI("http://dbpedia.org/resource/Guy_de_Maupassant"),
      NodeFactory.createURI("http://xmlns.com/foaf/0.1/givenName"),
      NodeFactory.createLiteral("Guy De"))

val graph = triples.add(triple)

// remove a statement from the RDF graph

val triple = Triple.create(
      NodeFactory.createURI("http://example.org/show/218"),
      NodeFactory.createURI("http://www.w3.org/2000/01/rdf-schema#label"),
      NodeFactory.createLiteral("That Seventies Show", "en"))

val graph = triples.remove(triple)

// finding a statement via S, P, O to the RDF graph
val subject = NodeFactory.createURI("http://example.org/show/218")
val predicate = NodeFactory.createURI("http://example.org/show/localName")
val `object` = NodeFactory.createLiteral("That Seventies Show", "en")

val graph = triples.find(Some(subject), Some(predicate), Some(`object`))

// checks if a triple is present in the RDF graph

val triple = Triple.create(
      NodeFactory.createURI("http://example.org/show/218"),
      NodeFactory.createURI("http://example.org/show/localName"),
      NodeFactory.createLiteral("That Seventies Show", "en"))

val contains = triples.contains(triple)

// converting RDD of triples into DataFrame
val graph = triples.toDF()
```

```
63
64 // converting RDD of triples into DataSet
65 val graph = triples.toDS()
66
67 //Triples filtered by subject ( "http://dbpedia.org/resource/Charles_Dickens" )
68 println("All triples related to Dickens:\n" +
69 triples.find(Some(NodeFactory.createURI("http://dbpedia.org/resource/Charles_Di
70 ckens")), None, None).collect().mkString("\n"))
71
72 //Triples filtered by predicate ( "http://dbpedia.org/ontology/influenced" )
73 println("All triples for predicate influenced:\n" + triples.find(None,
74 Some(NodeFactory.createURI("http://dbpedia.org/ontology/influenced")),
75 None).collect().mkString("\n"))
76
77 //Triples filtered by object ( <http://dbpedia.org/resource/Henry_James> )
78 println("All triples influenced by Henry_James:\n" + triples.find(None, None,
79 Some(NodeFactory.createURI("http://dbpedia.org/resource/Henry_James"))).collect
   ().mkString("\n"))
```

The same operations are also provided for the DataFrame and DataSet. The only differ is that while reading triples, you should define it using DataFrame or Dataset. Listing 4. gives one example on how to read triples in the format of DataFrame (Line 8).

Listing 4. Read triples using DataFrame.

```
1 import net.sansa_stack.rdf.spark.io._
2 import org.apache.jena.riot.Lang
3
4 val input = "hdfs://namenode:8020/data/rdf.nt"
5 val lang = Lang.NTRIPLES
6
7 val triples = spark.read.rdf(lang)(input)
8
```

The graph representation (using GraphX) of triples is given as a Property Graph Model which contains a VertexRDD of unique URIs (from subject and objects) and their relations represented via the EdgeRDD. Listing 5. gives an example on how to transform an RDD of triples as a graph (Line 10) and compute pagerank or resources (Line 11) from the graph.

Listing 5. Convert RDD of triples into the graph and compute the pagerank.

```
1 import org.apache.spark.graphx.Graph
2 import net.sansa_stack.rdf.spark.io._
3 import net.sansa_stack.rdf.spark.model.graph._
4 import org.apache.jena.riot.Lang
5
6 val input = "hdfs://namenode:8020/data/rdf.nt"
7 val lang = Lang.NTRIPLES
```

```
8  val triples = spark.rdf(lang)(input)
9
10 val graph = triples.asGraph()
11 val pagerank = graph.pageRank(0.00001).vertices
12
13 val report = pagerank.join(graph.vertices)
14   .map({ case (k, (r, v)) => (r, v, k) })
15   .sortBy(50 - _._1)
16 val rankedreport = report.map(f => f._2 + "\t" + f._1 )
17
18 println("%table resource\t rank\n " + rankedreport.take(50).mkString("\n"))
```

SANSA RDF layer provide a build in dataset statistics and quality assessment computation over large-scale RDF datasets. DistLODStats[8] compute 32 pre-computed statistics and report a Void description of the data. The statistics can be computed (voidifyed) as a whole or individually, based on the user preferences. Listing 6. gives one example on how to compute the property distribution (Line 10) over the graph.

Listing 6. Compute property distribution statistics.

```
1  import net.sansa_stack.rdf.spark.io._
2  import org.apache.jena.riot.Lang
3  import net.sansa_stack.rdf.spark.stats._
4
5  val input = "hdfs://namenode:8020/data/rdf.nt"
6  val lang = Lang.NTRIPLES
7
8  val triples = spark.rdf(lang)(input)
9
10 val propertyDist = triples.statsPropertyUsage()
11
12 val sortedValues = propertyDist.sortBy(_._2, false).take(100)
13                                 .map(f => f._1.getLocalName+ "\t" + f._2)
14
15 println("%table Property Distribution\tFrequency\n " +
16 sortedValues.mkString("\n"))
```

In the same way, the DistQualityAssessment compute a set of quality assessment and return the numerical value of the metric. Listing 7. Gives one example on how to compute some of the quality assessment metrics.

Listing 7. RDF quality assessment example.

```
1  import org.apache.jena.riot.Lang
2  import net.sansa_stack.rdf.spark.qualityassessment._
3  import net.sansa_stack.rdf.spark.io._
4
```

[8] http://sansa-stack.net/distlodstats/

```
val input = "hdfs://namenode:8020/data/rdf.nt"
val lang = Lang.NTRIPLES
val triples = spark.rdf(lang)(input)

// compute  quality assessment
val completeness_schema = triples.assessSchemaCompleteness()
val completeness_interlinking = triples.assessInterlinkingCompleteness()
val completeness_property = triples.assessPropertyCompleteness()

val syntacticvalidity_literalnumeric =
triples.assessLiteralNumericRangeChecker()
val syntacticvalidity_XSDDatatypeCompatibleLiterals =
triples.assessXSDDatatypeCompatibleLiterals()

val availability_DereferenceableUris = triples.assessDereferenceableUris()

val relevancy_CoverageDetail = triples.assessCoverageDetail()
val relevancy_CoverageScope = triples.assessCoverageScope()
val relevancy_AmountOfTriples = triples.assessAmountOfTriples()

val performance_NoHashURIs = triples.assessNoHashUris()
val understandability_LabeledResources = triples.assessLabeledResources()

val AssessQualityStr = s"""%table
metric\tvalue
completeness_schema\t$completeness_schema
completeness_interlinking\t$completeness_interlinking
completeness_property\t$completeness_property
syntacticvalidity_literalnumeric\t$syntacticvalidity_literalnumeric
syntacticvalidity_XSDDatatypeCompatibleLiterals\t$syntacticvalidity_XSDDatatype
CompatibleLiterals
availability_DereferenceableUris\t$availability_DereferenceableUris
relevancy_CoverageDetail\t$relevancy_CoverageDetail
relevancy_CoverageScope\t$relevancy_CoverageScope
relevancy_AmountOfTriples\t$relevancy_AmountOfTriples
performance_NoHashURIs\t$performance_NoHashURIs
understandability_LabeledResources\t$understandability_LabeledResources
"""

z.show(AssessQualityStr)
```

Querying Layer

Querying is the primary approach for searching, exploring and extracting insights from an underlying RDF graph. SPARQL is the de facto W3C standard for querying RDF graphs. SANSA provides cross-representational transformations and partitioning strategies for efficient query processing. It provides APIs for performing SPARQL queries directly in Spark programs. We will cover the following query engines of SANSA:

Triplify

The default approach for querying RDF data in SANSA is based on SPARQL-to-SQL. It uses a flexible triple-based partitioning strategy on top of RDF (such as predicate tables with sub-partitioning by data types). Currently, the Sparklify[9] implementation serves as a baseline.

The approach (see Listing 8.) gets as an input the RDD of triples (Line 8) and a SPARQL query (Line 8) and start evaluating it (Line 12).

Listing 8. Sparklify example.

```
1  import org.apache.jena.riot.Lang
2  import net.sansa_stack.rdf.spark.io._
3  import net.sansa_stack.query.spark.query._
4
5  val input = "hdfs://namenode:8020/data/rdf.nt"
6  val lang = Lang.NTRIPLES
7
8  val triples = spark.rdf(lang)(input)
9  val sparqlQuery = """SELECT ?s ?p ?o
10                              WHERE {?s ?p ?o }
11                        LIMIT 10"""
12 val result = triples.sparql(sparqlQuery)
13 z.show(result)
```

Data Lake

SANSA's DataLake [4] component allows to query Data Lakes, i.e., heterogeneous and large data sources directly on their original forms, without prior transformation. Data Lake sources are non-RDF by nature, but are queried uniformly using SPARQL. Sources can range from large files stored in HDFS to various NoSQL stores. SANSA DataLake currently supports CSV, Parquet files, Cassandra, MongoDB, and various JDBC sources e.g., MySQL, SQL Server.

Prior to querying user provides two input files: one is mappings and one is configurations. Mappings are links between elements from data schema (e.g., table and attributes) to elements

[9] http://sansa-stack.net/sparklify/

from ontologies (classes and properties). Configurations contain information needed to access the data source, e.g., user/pass, host/port, cluster name, etc.

User issues a SPARQL query, which uses terms from the ontology. The query is internally decomposed into subqueries, each triggering a relevant source selection process, which looks at the mappings and finds what data sources contain schema elements mapping to SPARQL ontology terms (properties and classes). Once relevant data is found, it is loaded on-the-fly into Spark DataFrames where data can undergo various (relational) operations like, projection, selection, ordering, etc.. Subresults are finally joined generating a final DataFrame, which is returned to the user in a tabular format.

Data is queried in the following way:

Listing 9. SANSA DataLake example.

```
1  import net.sansa_stack.query.spark.query._
2
3  val configFile = "/config"
4  val mappingsFile = "/mappings.ttl"
5  val query = "SPARQL query"
6
7  val resultsDF = spark.sparqlDL(query, mappingsFile, configFile)
```

Where configFile (Line 3) and mappingsFile (Line 4) are the paths to Configurations (example) and Mappings (example) files, and query (example) is a string variable holding the SPARQL query.

Inference Layer

RDFS and OWL are well known for containing schema information as an addition to assertions or facts. The forward chaining inference process applies inference rules on the existing facts in a knowledge base to infer new facts iteratively. Doing so, it enriches the knowledge base with new knowledge and allows inconsistencies-detection. This process is usually data-intensive and compute-intensive. SANSA provides an efficient rule engine for the well-known reasoning profiles RDFS (with different subsets) and OWL-Horst. By using SANSA, applications will be able to fine-tune the rules they require and - in case of scalability issues - adjust them accordingly.

Listing 10. run a SANSA inference over triples (Line 15) using the forward chaining approach.

Listing 10. Triple based forward chaining.

```
1  import net.sansa_stack.inference.spark.data.loader.RDFGraphLoader
2  import
3  net.sansa_stack.inference.spark.forwardchaining.triples.ForwardRuleReasonerRDFS
4
5  val input = "hdfs://namenode:8020/data/rdf.nt"
6
7  // load triples from disk
8  val graph = RDFGraphLoader.loadFromDisk(spark, URI.create(input), parallelism)
9  println(s"|G|=${graph.size()}")
10
11 // create reasoner
12 val reasoner = new ForwardRuleReasonerRDFS(spark.sparkContext, parallelism)
13
14 // compute inferred graph
15 val inferredGraph = reasoner.apply(graph)
16 println(s"|G_inferred|=${inferredGraph.size()}")
```

Besides running inference engine over triples, SANSA provide a functionality to infer new knowledge using OWL axioms as well. Listing 11. gives one example of inferring OWL axioms using SANSA.

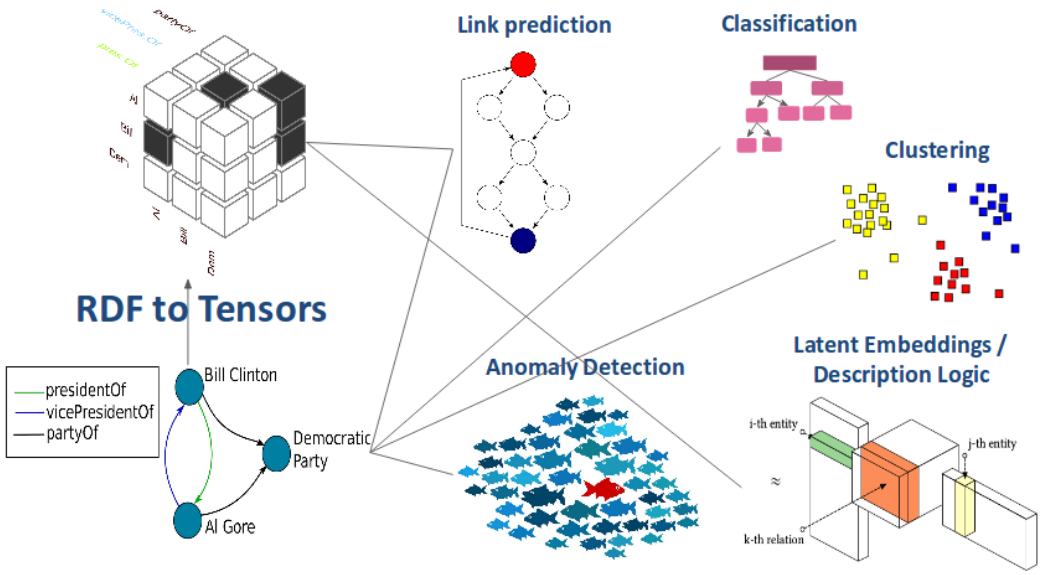
Listing 11. Axiom based forward chaining.

```
1  import
2  net.sansa_stack.inference.spark.forwardchaining.axioms.ForwardRuleReasonerRDFS
3
4  var owlAxioms = spark.owl(Syntax.FUNCTIONAL)(input)
5
6  val inferredGraph = new ForwardRuleReasonerRDFS(spark.sparkContext, parallelism
7  = 4)(owlAxioms)
8
9
```

```
println(s"|G_inf| = ${inferredGraph.count()}")
```

Analytics Layer

SANSA makes full use of, and derives benefit from the graph structure and semantics of the background knowledge specified using RDF and OWL standards. SANSA tries to exploit the expressivity of the knowledge structures in order to attain either better performance or more human-readable results. Currently, this layer provides algorithms for: Clustering, Anomaly detection [3], Rule Mining and Graph Kernels.



Clustering algorithms:-

So far, SANSA supports various clustering algorithms:

- Kmeans;
- Power Iteration Clustering;
- Graph-based clustering;

Each of the clustering algorithms takes RDF N-triples as input and output an RDD of *cluster id* and the *triples* i.e, `RDD[cluster_id, List(Triples)]`.

Listing 12. KMeans clustering example.

```
import org.apache.jena.riot.Lang
import net.sansa_stack.ml.spark.clustering.algorithms._

import net.sansa_stack.rdf.spark.io._

val lang = Lang.NTRIPLES
val triples = spark.rdf(lang)(input)

val conf = ConfigFactory.load()
val data = new DataProcessing(spark = spark, conf = conf, triples)
val pois = data.pois
val poiCategorySet = pois.map(poi => (poi.poi_id,
poi.categories.categories.toSet)).persist()
```

```
14 val (avgVectorDF, word2Vec) = new Encoder().wordVectorEncoder(poiCategorySet,
15 spark)
16 val avgVectorClusters = new Kmeans().kmClustering(numClusters = 8, maxIter=5,
17 df=avgVectorDF, spark=spark)
```

The idea would then be to benefit from the genericity of the stack (especially in terms of intermediate data structure) to *e.g.* pipe together various kinds of clustering algorithms.

Listing 13. DBSCAN clustering example.

```
1  import org.apache.jena.riot.Lang
2  import net.sansa_stack.ml.spark.clustering.algorithms._
3
4  import net.sansa_stack.rdf.spark.io._
5
6   val lang = Lang.NTRIPLES
7   val triples = spark.rdf(lang)(input)
8
9  val conf = ConfigFactory.load()
10 val data = new DataProcessing(spark = spark, conf = conf,triples)
11 val pois = data.pois
12 val geometryFactory = new GeometryFactory()
13 import spark.implicits._
14
15 val dbParam = pois.map { f =>
16       val id = f.poi_id.toString()
17       val name = ""
18       val lat = f.coordinate.latitude
19       val long = f.coordinate.longitude
20       val cat = f.categories.categories.mkString(";")
21       val point = geometryFactory.createPoint(new Coordinate(long, lat))
22
23       point.setUserData(id)
24       point
25     }
26 val dbscan = new DBSCAN()
27 val clusteredLatLong = dbscan.dbclusters(dbParam, epsilon, minPoints, sark)
28
```

RDF Graph Kernels

Given an RDF graph, The RDF kernel constructs a tree for each instance and counts the number of paths in it. Given that the literals in RDF can only occur as objects in triples and therefore, have no out-going edges in the RDF graph. Term Frequency-Inverse Document Frequency(TF-IDF) is applied to litzerals by creating a dictionary and vectorizing the string literal values.

Listing 13.RDF Graph Kernel example.

```
import net.sansa_stack.ml.spark.kernel._
import net.sansa_stack.rdf.spark.io._
import org.apache.jena.riot.Lang

val triples = spark.rdf(lang)(input).
             filter(_.getPredicate.getURI !=
"http://swrc.ontoware.org/ontology#employs")

val rdfFastGraphKernel = RDFFastGraphKernel(spark, triples,
"http://swrc.ontoware.org/ontology#affiliation")
val data = rdfFastGraphKernel.getMLLibLabeledPoints

RDFFastTreeGraphKernelUtil.predictLogisticRegressionMLLIB(data, 4, iteration)
```

References

[1]. Lehmann, Jens, et al. "Distributed semantic analytics using the sansa stack." International Semantic Web Conference. Springer, Cham, 2017.

[2]. Sejdiu, Gezim, et al. "DistLODStats: Distributed Computation of RDF Dataset Statistics." In Proceedings of 17th International Semantic Web Conference, 2018.

[3]. Jabeen, Hajira, et al. "Divided We Stand Out! Forging Cohorts fOr Numeric Outlier Detection in Large Scale Knowledge Graphs (CONOD)." European Knowledge Acquisition Workshop. Springer, Cham, 2018.

[4] Mami, Mohamed Nadjib, et al. "Querying Data Lakes using Spark and Presto.". The Web Conference Demonstrations, 2019.