Chapter 7
Scalable Knowledge Graph Processing using SANSA

Hajira Jabeen[1], Damien Graux[2], and Gezim Sejdiu[1,3]

[1] Smart Data Analytics, University of Bonn, Germany
[2] ADAPT SFI Research Centre, Trinity College Dublin, Ireland
[3] Deutsche Post DHL Group, Germany

Abstract. The size and number of knowledge graphs have increased tremendously in recent years. In the meantime, the distributed data processing technology has also advanced to deal with big data and large scale knowledge graphs. This chapter introduces Scalable Semantic Analytics Stack (SANSA), that addresses the challenge of dealing with large scale RDF data at scale and provides a unified framework for applications like link prediction, knowledge base completion, querying, and reasoning. We discuss the motivation, background and the architecture of SANSA. SANSA is built using general-purpose processing engines Apache Spark and Apache Flink. After reading this chapter, the reader should have an understanding of the different layers and corresponding APIs available to handle Knowledge Graphs at scale using SANSA.

1 Introduction

Over the past decade, vast amounts of machine-readable structured information have become available through the increasing popularity of semantic knowledge graphs using semantic technologies in a variety of application domains including life sciences, publishing, source code of open source projects [264], patents and the internet of things. These knowledge bases are becoming more prevalent and this trend can be expected to continue in future.

The size of knowledge graphs has reached the scale where centralised analytical approaches have become infeasible. Recent technological progress has enabled powerful distributed in-memory analytics that have been shown to work well on simple data structures. However, the application of such distributed analytics approaches on semantic knowledge graphs is lagging behind significantly. To advance both the scalability and accuracy of large-scale knowledge graph analytics to a new level, fundamental research on methods of leveraging distributed in-memory computing and semantic technologies in combination with advancements in analytics approaches is indispensable.

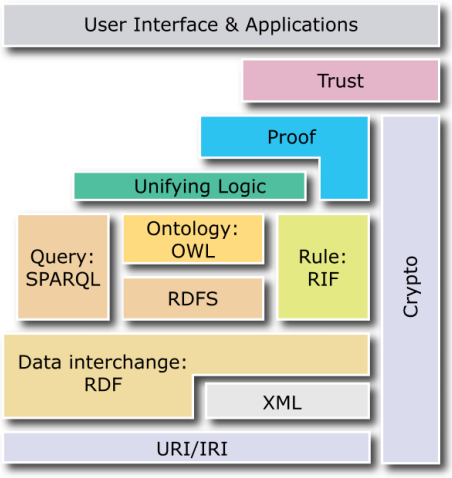

Fig. 1: W3C Semantic Web layer cake.

2 Semantic Layer Cake

As presented in the previous chapters, there are many different types of data source available that are collecting and providing information structured *via* different formats. In addition, most of them are available on the Web and often share some information about the same concepts or entities; as a consequence, the need to allow alignments between sources has increased. This motivation fuelled the Semantic Web initiative where the main idea is to enable linkage between remote data entities so that several facets of information become available at once. The Semantic Web mainly relies on the dereferencing concept where identifiers (IRIs - Internationalised Resource Identifier) are used to represent entities and are therefore to navigate from one piece of information to another.

The Semantic Web has been mainly pushed by the World Wide Web Consortium (W3C), which proposed a set of standards to technically back up this movement. Practically, these standards are built following a "layer cake" structure where standards are constructed on top of other ones (see Figure 1). In particular, the stack is completely built on top of the identifier concept, which serves as a basis then to represent data using the following RDF structure.

The Semantic Web does not limit its scope to only linking and representing data on the web; it also provides a range of specifications to help users enrich their knowledge. First of all, RDF comes with an associated query language (SPARQL) in order to extract data from sources. Moreover, several standards specify how to structure the data:

1. The RDF Schema (RDFS) lists a set of classes with certain properties using the RDF representation data model and provides basic elements for the description of ontologies.

2. The Web Ontology Language (OWL) is a family of knowledge representation languages for authoring ontologies which are a formal ways to describe taxonomies and classification networks, essentially defining the structure of knowledge for various domains.
3. The Shapes Constraint Language (SHACL) allows to design validations over graph-based data considering a set of conditions. Among others, it includes features to express conditions that constrain the number of values that a property may have, the type of such values, numeric ranges etc. ...

These specifications then allow users to specify several properties about Semantic Web data and therefore one can use them to extend one's own knowledge. Indeed, ontologies are the cornerstone of all the studies made around inferring data from a set of triples *e.g.* using the structure of the graph, it is possible to "materialize" additional statements and thereby to extend the general knowledge.

As a consequence, the W3C – *via* the diverse standards and recommendations it set up – allows users to structure pieces of information. However, the large majority of existing tools are focusing on one or two standards at once, meaning that they are usually not encompassing the full scope of what the Semantic Web is supposed to provide and enable. Indeed, designing such a "wide-scope" Semantic Web tool is challenging. Recently, such an initiative was created: SANSA [409]; in addition, SANSA also pays attention to the Big Data context of the Semantic Web and adopts a fully distributed strategy.

3 Processing Big Knowledge Graphs with SANSA

In a nutshell, SANSA[4] presents:

1. efficient data distribution techniques and semantics-aware computation of latent resource embeddings for knowledge graphs;
2. adaptive distributed querying;
3. efficient self-optimising inference execution plans; and
4. efficient distributed machine learning on semantic knowledge graphs of extremely large scale.

3.1 Knowledge Representation & Distribution

SANSA follows the modular architecture where each layer represents a unique component of functionality, which could be used by other layers of the SANSA framework. The Knowledge Representation & Distribution is the lowest layer on top of the existing distributed computing framework (either Apache Spark[5] or Apache Flink[6]). Within this layer, SANSA provides the functionality to read and write native RDF or OWL data from HDFS or a local dive and represents

[4] http://sansa-stack.net/
[5] http://spark.apache.org/
[6] https://flink.apache.org/

it in native distributed data structures of the framework. Currently, it supports different RDF and OWL serializations / syntax formats. Furthermore, it provides a dedicated serialization mechanism for faster I/O. The layer also supports Jena and OWL API interfaces for processing RDF and OWL data, respectively. This particularly targets usability, as many users are already familiar with the corresponding libraries.

This layer also gives access to a mechanism for RDF data compression in order to lower the space and processing time when querying RDF data (c.f Section 3.2). It also provides different partitioning strategies in order to facilitate better maintenance and faster access to this scale of data. Partitioning the RDF data is the process of dividing datasets in a specific logical and/or physical representation in order to ease faster access and better maintenance. Often, this process is performed to improve the system availability, load balancing and query processing time. There are many different data partitioning techniques proposed in the literature. Within SANSA, we provide 1) semantic-based partitioning [390], 2) vertical-based partitioning [409], and 3) graph-based partitioning.

Semantic-based partitioning – A semantically partitioned fact is a tuple (S, R) containing pieces of information $R \in (P, O)$ about the same S where S is a unique subject on the RDF graph and R represents all its associated facts i.e predicates P and objects O. This partitioned technique was proposed in the SHARD [374] system. We have implemented this technique using the in-memory processing engine, Apache Spark, for better performance.

Vertical partitioning – The vertical partitioning approach in SANSA is designed to support extensible partitioning of RDF data. Instead of dealing with a single three-column table (s, p, o), data is partitioned into multiple tables based on the used RDF predicates, RDF term types and literal datatypes. The first column of these tables is always a string representing the subject. The second column always represents the literal value as a Scala/Java datatype. Tables for storing literals with language tags have an additional third string column for the language tag.

In addition, this layer of SANSA allows users to compute RDF statistics [389] and to apply quality assessment [391] in a distributed manner. More specifically, it provides a possibility to compute different RDF dataset statistics in a distributed manner via the so-called DistLODStats [390] software component. It describes the first distributed in-memory approach for computing 32 different statistical criteria for RDF datasets using Apache Spark. The computation of statistical criteria consists of three steps: (1) saving RDF data in scalable storage, (2) parsing and mapping the RDF data into the *main dataset* – an RDD data structure composed of three elements: *Subject*, *Property* and *Object*, and (3) performing statistical criteria evaluation on the *main dataset* and generating results.

Fetching the RDF data (Step 1): RDF data needs first to be loaded into a large-scale storage that Spark can efficiently read from. For this purpose, we use HDFS (Hadoop Distributed File-System). HDFS is able to accommodate any type of data in its raw format, horizontally scale to arbitrary number of

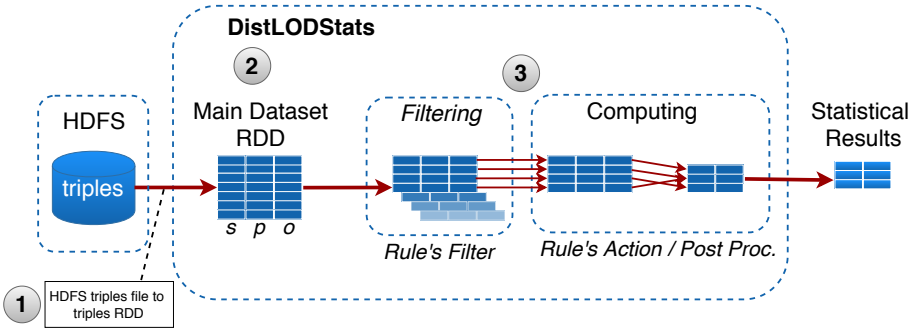

Fig. 2: Overview of DistLODStats's abstract architecture [390].

nodes, and replicate data among the cluster nodes for fault tolerance. In such a distributed environment, Spark adopts different data locality strategies to try to perform computations as close to the needed data as possible in HDFS and thus avoid data transfer overhead.

Parsing and mapping RDF into the main dataset (Step 2): In the course of Spark execution, data is parsed into triples and loaded into an RDD of the following format: *Triple<Subj,Pred,Obj>* (by using the Spark *map* transformation).

Statistical criteria evaluation (Step 3): For each criterion, Spark generates an execution plan, which is composed of one or more of the following Spark transformations: *map*, *filter*, *reduce* and *group-by*. *Filtering* operation apply the Rule's Filter and produce a new *filtered* RDD. The *filtered RDD* will serve as an input to the next step: *Computing* where the rule's action and/or post processing are effectively applied. The output of the *Computing* phase will be the statistical results represented in a human-readable format, e.g. VoID, or row data.

Often when designing and performing large-scale RDF processing tasks, the quality of the data is one of the key components to be considered. Existing solutions are not capable of dealing with such amounts of data, therefore a need for a distributed solution for a quality check arises. To address this, within SANSA we present DistQualityAssessment [391] – an open-source implementation of quality assessment of large RDF datasets that can scale out to a cluster of machines. This is the first distributed, in-memory approach for computing different quality metrics for large RDF datasets using Apache Spark. We also provide a quality assessment pattern that can be used to generate new scalable metrics that can be applied to big data. A more detailed overview of the approach is given below. The computation of the quality assessment using the Spark framework consists of four steps:

Defining quality metrics parameters The metric definitions are kept in a dedicated file, which contains most of the configurations needed for the system to evaluate quality metrics and gather result sets.

Retrieving the RDF data RDF data first needs to be loaded into a large-scale storage that Spark can efficiently read from. We use Hadoop Distributed File-

System (HDFS). HDFS is able to fit and store any type of data in its Hadoop-native format and parallelize them across a cluster while replicating them for fault tolerance. In such a distributed environment, Spark automatically adopts different data locality strategies to perform computations as close to the needed data as possible in HDFS and thus avoids data transfer overhead.

Parsing and mapping RDF into the main dataset We first create a distributed dataset called *main dataset* that represent the HDFS file as a collection of triples. In Spark, this dataset is parsed and loaded into an RDD of triples having the format *Triple<(s,p,o)>*.

Quality metric evaluation Considering the particular quality metric, Spark generates an execution plan, which is composed of one or more Spark transformations and actions. The numerical output of the final action is the quality of the input RDF corresponding to the given metric.

3.2 Query

As presented before, the Semantic Web designed several standards on top of RDF. Among them, one is to manipulate RDF data: SPARQL. In a nutshell, it constitutes the *de facto* querying language for RDF data and hereby provides a wide range of possibilities to either extract, create or display information.

The evaluation of SPARQL has been a deeply researched topic by the Semantic Web communities for approximately twenty years now; dozens of evaluators have been implemented, following as many different approaches to store and organise RDF data[7]. Recently, with the increase of cloud-based applications, a new range of evaluators have been proposed following the distributed paradigm which usually suits Big Data applications[8].

Distributed RDF data As part of the SANSA stack, a layer has been developed to handle SPARQL queries in a distributed manner and it offers several strategies in order to fit users' needs. Actually, following existing studies from the literature, the developers decided by default to rely on the Apache Spark SQL engine: in practice, the SPARQL queries asked by the users are automatically translated in SQL to retrieve information from the in-memory virtual tables (the Sparklify [409] approach) created from the RDF datasets. Such a method then allows SANSA to take advantage of the relational engine of Spark especially designed to deal with distributed Big Data. In parallel, other evaluation strategies are available to fit specific use-cases as they consist of different distribution strategies of the original RDF data in memory. While the default (vertical) partitioning scheme splits datasets into blocks based on common predicates, SANSA provides an implementation of the semantic partitioning [390] based on common subjects. It also has built-in features enabling compression on-the-fly, which allows it to handle bigger datasets.

[7] See [130] for a comprehensive survey of single-node RDF triplestores.
[8] See [234] or [169] for an extensive review of the cloud-based SPARQL evaluators.

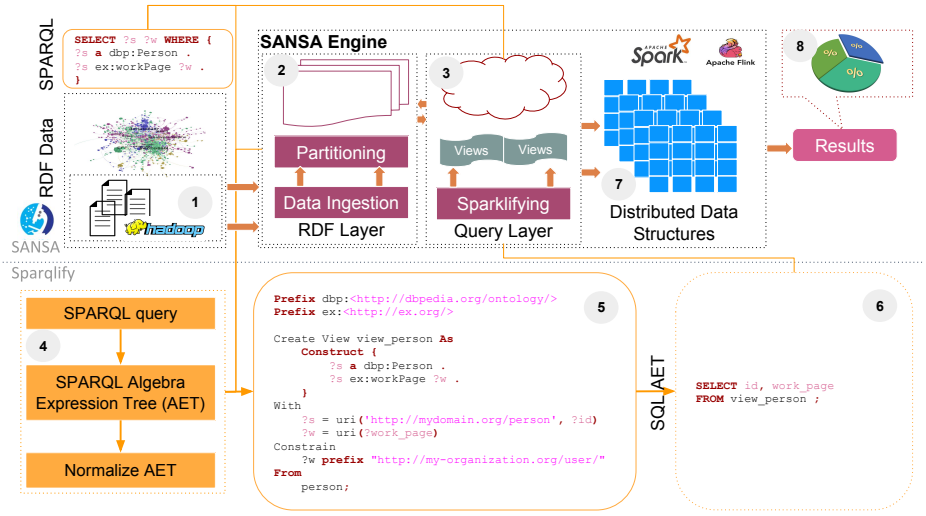

Fig. 3: SANSA's Query Layer Architecture Overview.

The overall system architecture is shown in Figure 3. It consists of four main components: Data Model, Mappings, Query Translator and Query Evaluator.

Data Ingestion (step 1) RDF data first needs to be loaded into large-scale storage that Spark can efficiently read from.

We use the Hadoop Distributed File-System (HDFS) [61]. Spark employs different data locality schemes in order to accomplish computations nearest to the desired data in HDFS, as a result avoiding i/o overhead.

Data Partition (step 2) The vertical partitioning approach in SANSA is designed to support extensible partitioning of RDF data. Instead of dealing with a single three-column table (s, p, o), data is partitioned into multiple tables based on the used RDF predicates, RDF term types and literal datatypes. The first column of these tables is always a string representing the subject. The second column always represents the literal value as a Scala/Java datatype. Tables for storing literals with language tags have an additional third string column for the language tag.

Mappings/Views After the RDF data has been partitioned using the extensible VP (as it has been described on *step 2*), the relational-to-RDF mapping is performed. Sparqlify supports both the W3C standard R2RML sparqlification [410].

The main entities defined with SML are *view definitions*. See *step 5* in the Figure 3 as an example. The actual view definition is declared by the *Create View ... As* in the first line. The remainder of the view contains these parts: (1) the *From* directive defines the logical table based on the partitioned table (see *step 2*). (2) an RDF template is defined in the *Construct* block containing,

URI, blank node or literals constants (e.g. *ex:worksAt*) and variables (e.g. *?emp*, *?institute*). The *With* block defines the variables used in the template by means of RDF term constructor expressions whose arguments refer to columns of the logical table.

Query Translation This process generates a SQL query from the SPARQL query using the bindings determined in the mapping/view construction phases. It walks through the SPARQL query (*step 4*) using Jena ARQ[9] and generates the SPARQL Algebra Expression Tree (AET). Essentially, rewriting SPARQL basic graph patterns and filters over views yields AETs that are UNIONS of JOINS. Further, these AETs are normalized and pruned in order to remove UNION members that are known to yield empty results, such as joins based on IRIs with disjointed sets of known namespaces, or joins between different RDF term types (e.g. literal and IRI). Finally, the SQL is generated (*step 6*) using the bindings corresponding to the views (*step 5*).

Query Evaluation Finally, the SQL query created as described in the previous section can now be evaluated directly into the Spark SQL engine. The result set of this SQL query is a distributed data structure of Spark (e.g. DataFrame)(*step 7*), which then is mapped into SPARQL bindings. The result set can be further used for analysis and visualization using the SANSA-Notebooks[10] (*step 8*).

Data Lake SANSA also has a DataLake component which allows it to query heterogeneous data sources ranging from different databases to large files stored in HDFS, to NoSQL stores, using SPARQL. SANSA DataLake currently supports CSV, Parquet files, Cassandra, MongoDB, Couchbase, ElasticSearch, and various JDBC sources e.g., MySQL, SQL Server. Technically, the given SPARQL queries are internally decomposed into subqueries, each extracting a subset of the results.

The DataLake layer consists of four main components (see numbered boxes in the Figure 4). For the sake of clarity, we use here the generic ParSets and DEE concepts instead of the underlying equivalent concrete terms, which differ from engine to engine. ParSet, from Parallel dataSet, is a data structure that can be distributed and operated in parallel. It follows certain data models, like tables in tabular databases, graphs in graph databases, or documents in a document database. DEE, from Distributed Execution Environment, is the shared physical space where ParSets can be transformed, aggregated and joined together.
The architecture accepts three user inputs:

– Mappings: it contains associations between data source entities[11] and attributes to ontology properties and classes.

[9] https://jena.apache.org/documentation/query/
[10] https://github.com/SANSA-Stack/SANSA-Notebooks
[11] These entities can be, for example, table and column in a tabular database or collection and document in a document database.

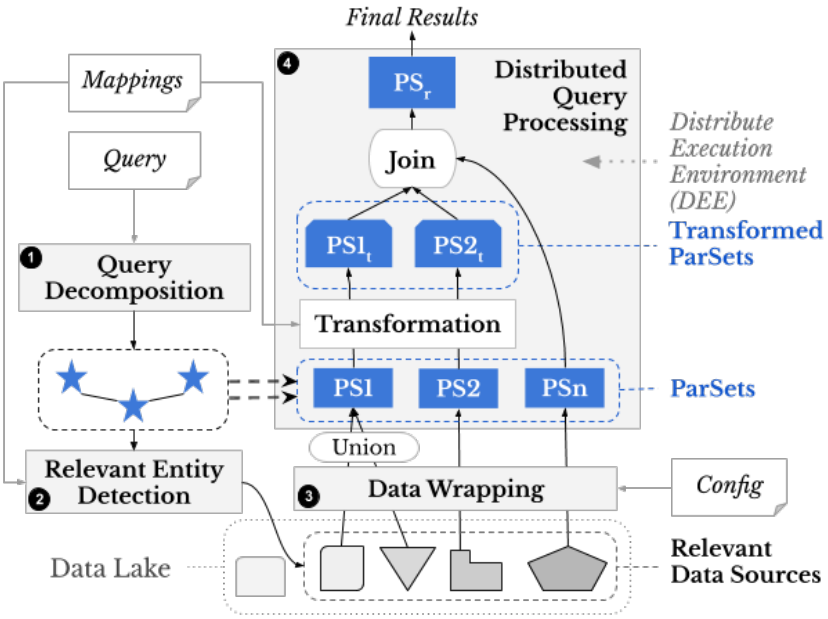

Fig. 4: SANSA's DataLake Layer Internal Architecture [293].

- Config: it contains the access information needed to connect to the heterogeneous data sources, *e.g.*, username, password, or cluster setting, *e.g.*, hosts, ports, cluster name, etc.
- Query: a query in the SPARQL query language.

The fours components of the architecture are described as follows:

Query Decomposor This component is commonly found in OBDA and query federation systems. It decomposes the query's Basic Graph Pattern (BGP, conjunctive set of triple patterns in the where clause) into a set of star-shaped sub-BGPs, where each sub-BGP contains all the triple patterns sharing the same subject variable. We refer to these sub-BGPs as stars for brevity (see below figure left; stars are shown in distinct colored boxes).

Relevant Entity Extractor For every extracted star, this component looks in the Mappings for entities that have attributes mapping to each of the properties of the star. Such entities are relevant to the star.

Data Wrapper In the classical OBDA, a SPARQL query has to be translated to the query language of the relevant data sources. This is, in practice, hard to achieve in the highly heterogeneous Data Lake settings. Therefore, numerous recent publications advocated for the use of an intermediate query language. In our case, the intermediate query language is DEE's query language, dictated by

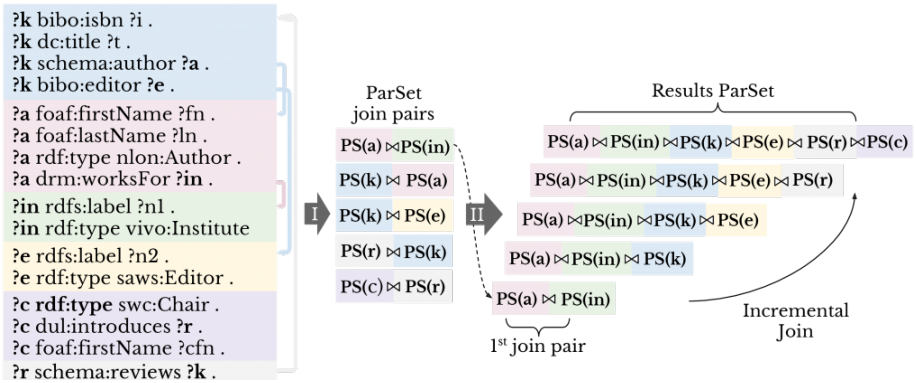

Fig. 5: From query to ParSets to joins between ParSets.

its internal data structure. The Data Wrapper generates data in POA's data structure at query-time, which allows for the parallel execution of expensive operations, *e.g.*, join. There must exist wrappers to convert data entities from the source to DEE's data structure, either fully or partially if parts of the data can be pushed down to the original source. Each identified star from step (1) will generate exactly one ParSet. If more than an entity is relevant, the ParSet is formed as a union. An auxiliary user input Config is used to guide the conversion process, *e.g.*, authentication, or deployment specifications.

Distributed Query Processor Finally, ParSets are joined together forming the final results. ParSets in the DEE can undergo any query operation, *e.g.*, selection, aggregation, ordering, etc. However, since our focus is on querying multiple data sources, the emphasis is on the join operation. Joins between stars translate into joins between ParSets (Figure 5 phase I). Next, ParSet pairs are all iteratively joined to form the Results ParSet (Figure 5 phase II). In short, extracted join pairs are initially stored in an array. After the first pair is joined, it iterates through each remaining pair to attempt further joins or, else, add to a queue. Next, the queue is similarly iterated; when a pair is joined, it is unqueued. The algorithm completes when the queue is empty. As the Results ParSet is a ParSet, it can also undergo query operations. The join capability of ParSets in the DEE replaces the lack of the join common in many NoSQL databases, *e.g.*, Cassandra, MongoDB. Sometimes ParSets cannot be readily joined due to a syntactic mismatch between attribute values; nevertheless, SANSA provides a method to correct these mismatches, thereby enabling the joins.

3.3 Inference

Both RDFS and OWL contain schema information in addition to links between different resources. This additional information and rules allows users to perform reasoning on the knowledge bases in order to infer new knowledge and expand

existing knowledge. The core of the inference process is to continuously apply schema-related rules on the input data to infer new facts. This process is helpful for deriving new knowledge and for detecting inconsistencies. SANSA provides an adaptive rule engine that can use a given set of arbitrary rules and derive an efficient execution plan from those. Later, that execution plan is evaluated and run against underlying engines, i.e. Spark SQL, for an efficient and scalable inference process.

3.4 Machine Learning

SANSA-ML is the Machine Learning (ML) library in SANSA. Algorithms in this repository perform various machine learning tasks directly on RDF/OWL input data. While most machine learning algorithms are based on processing simple features, the machine learning algorithms in SANSA-ML exploit the graph structure and semantics of the background knowledge specified using the RDF and OWL standards. In many cases, this allows users to obtain either more accurate or more human-understandable results. In contrast to most other algorithms supporting background knowledge, the algorithms in SANSA scale horizontally using Apache Spark. The ML layer currently supports numerous algorithms for Clustering, Similarity Assessment of entities, Entity Linking, Anomaly Detection and Classification using Graph Kernels. We will cover these algorithms in the context of knowledge graphs in the following section.

3.5 Semantic Similarity Measures

SANSA covers the semantic similarities used to estimate the similarity of concepts defined in ontologies and, hence, to assess the semantic proximity of the resources indexed by them. Most of the approaches covered in the SANSA similarity assessment module are feature-based. The feature model requires the semantic objects to be represented as sets of features. Tversky was the first to formulate the concept of semantic similarity using the feature model, from which a family of semantic measures has been derived. The similarity measure in this context is defined as a function (set-based or distance-based measure) on the common features of the objects under assessment.

Jaccard Similarity For any two nodes u and v of a data set, the Jaccard similarity is defined as:

$$\mathrm{Sim}_{Jaccard}(u, v) = \frac{|f(u) \cap f(v)|}{|f(u) \cup f(v)|} \tag{1}$$

Here, $f(u)$ is the subset of all neighbours of the node u and $|f(u)|$ the cardinality of $f(u)$ that counts the number of elements in $f(u)$.

Rodríguez and Egenhofer similarity Another example of feature-based measure implemented in SANSA is by Rodríguez and Egenhofer [181].

$$\mathrm{Sim}_{RE}(u,v) = \frac{|f(u) \cap f(v)|}{\gamma \cdot |f(u) \setminus f(v)| + (1-\gamma) \cdot |f(v) \setminus f(u)| + |f(u) \cap f(v)|} \quad (2)$$

where $\gamma \in [0,1]$ allows to adjust measure symmetry.

Ratio Model Tversky defined a parameterized semantic similarity measure which is called the ratio model (SimRM) [181]. It can be used to compare two semantic objects $(u; v)$ through its respective sets of features U and V :

$$\mathrm{Sim}_{RM}(u,v) = \frac{|f(u) \cap f(v))|}{\alpha|f(u) \setminus f(v)|) + \beta|f(v) \setminus f(u)| + \gamma|f(u) \cap f(v))|} \quad (3)$$

with α, β and $\gamma \geq 0$.

Here, $|f(u)|$ is the cardinality of the set $f(u)$ composed of all neighbours of u. Setting Sim_{RM} with $\alpha = \beta = 1$ leads to the Jaccard index, and setting $\alpha = \beta = 0.5$ leads to the Dice coefficient. In other words, set-based measures can be used to easily express abstract formulations of similarity measures. Here, we set $\alpha = \beta = 0.5$.

Batet Similarity Batet et al. represent the taxonomic distance as the ratio between distinct and shared features [30]. Batet similarity can be defined as follows:

$$\mathrm{Sim}_{Batet}(u,v) = \log_2\left(1 + \frac{|f(u) \setminus f(v)| + |f(v) \setminus f(u)|}{|f(u) \setminus f(v)| + |f(v) \setminus f(u)| + |f(u) \cap f(v)|}\right) \quad (4)$$

For any node u, the notation $f(u)$ stands for the set of all neighbours of u.

3.6 Clustering

Clustering is the class of unsupervised learning algorithms that can learn without the need for the training data. Clustering is aimed to search for common patterns and similar trends in the knowledge graphs. The similarity of patterns is mostly measured by a given similarity measure, e.g the measures covered in the previous section. Below, we cover the clustering algorithms implemented in SANSA for knowledge graphs.

PowerIteration Clustering PowerIteration (PIC) [282] is a fast spectral clustering technique. It is a simple (it only requires a matrix-vector multiplication process) and scalable algorithm in terms of time complexity, O(n). PIC requires pairwise vertices and their similarities as input and outputs the clusters of vertices by using a pseudo-eigenvector of the normalized affinity matrix of the graph.

Although the PowerIteration method approximates only one eigenvalue of a matrix, it remains useful for certain computational problems. For instance, Google uses it to calculate the PageRank of documents in its search engine, and Twitter uses it to show follow recommendations. Spark.mllib includes an implementation of PIC using GraphX. It takes an RDD of tuples, which are vertices of an edge, and the similarity among the two vertices and outputs a model with clustering assignments.

BorderFlow Clustering BorderFlow [323] is a local graph clustering which takes each node as the starting seed and iteratively builds clusters by merging the nodes using BorderFlow-ratio. The clusters must have a maximal intra-cluster density and inter-cluster sparseness. When considering a graph as the description of a flow system, this definition of a cluster implies that a cluster X is a set of nodes such that the flow within X is maximal while the flow from X to the outside is minimal. At each step, a pair of nodes is merged if the border flow ratio is maximised and this process is repeated until the termination criterion is met. BorderFlow is a parameter-free algorithm and it has been used successfully in diverse applications including clustering protein-protein interaction (PPI) data [322] and query clustering for benchmarking [311].

Linked-based Clustering Link information plays an important role in discovering knowledge from data. The link-based graph clustering [156] algorithm results in overlapping clusters. Initially, each link represents its own group; the algorithm recursively merges the links using similarity criteria to optimize the partition density until all links are merged into one, or until the termination condition is met. To optimize performance, instead of selecting arbitrary links, the algorithm only considers the pair of links that share a node for merging.

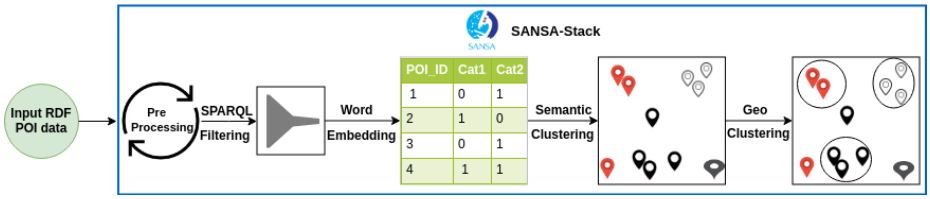

Fig. 6: A Semantic-Geo Clustering flow.

Building clustering processes [94] SANSA proposes a flexible architecture to design clustering pipelines. For example, having points of interest (POI) datasets, SANSA can aggregate them according to several dimensions in one pipeline: their labels on the first hand and their localisation on the other hand. Such an architecture is presented in Figure 6.

The approach contains up to five main components (which could be enabled/disabled if necessary), namely: data pre-processing, SPARQL filtering, word embedding, semantic clustering and geo-clustering. In semantic-based clustering algorithms (which do not consider POI locations but rather aim at grouping POIs according to shared labels), there is a need to transform the POIs categorical values to numerical vectors to find the distance between them. So far, any word-embedding technique can be selected among the three available ones, namely one-hot encoding, Word2Vec and Multi-Dimensional Scaling. All the abovementioned methods convert categorical variables into a form that could be provided to semantic clustering algorithms to form groups of non-location-based similarities. For example, all restaurants are in one cluster whereas all the ATMs are in another one. On the other hand, the geo-clustering methods help to group the spatially closed coordinates within each semantic cluster.

More generically, SANSA's architecture and implementation allow users to design any kind of clustering combinations they would like. Actually, the solution is flexible enough to pipe together more than two clustering "blocks" and even to add additional RDF datasets into the process after several clustering rounds.

3.7 Anomaly Detection

With the recent advances in data integration and the concept of data lakes, massive pools of heterogeneous data are being curated as Knowledge Graphs (KGs). In addition to data collection, it is of the utmost importance to gain meaningful insights from this composite data. However, given the graph-like representation, the multimodal nature, and large size of data, most of the traditional analytic approaches are no longer directly applicable. The traditional approaches collect all values of a particular attribute, e.g. height, and perform anomaly detection for this attribute. However, it is conceptually inaccurate to compare one attribute representing different entities, e.g. the height of buildings against the height of animals. Therefore, there is a strong need to develop fundamentally new approaches for outlier detection in KGs. SANSA presents a scalable approach that can deal with multimodal data and performs adaptive outlier detection against the cohorts of classes they represent, where a cohort is a set of classes that are similar based on a set of selected properties. An overview of the scalable anomaly detection [215] in SANSA can be seen in 7.

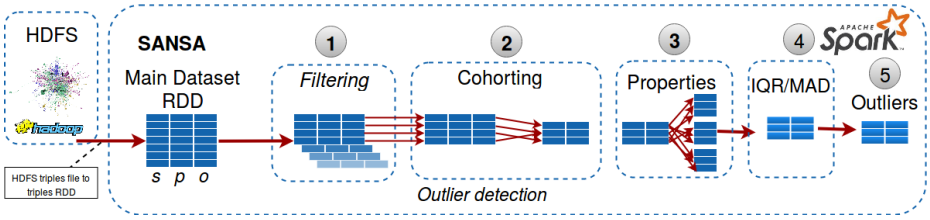

Fig. 7: Anomaly detection execution pipeline.

3.8 Entity Linking

Entity resolution is the crucial task of recognizing and linking entities that point to the same real-world object in various information spaces. Entity linking finds its application in numerous tasks like de-duplicating entities in federal datasets related to medicine, finance, transportation, business and law enforcement, etc. With the growth of the web in terms of volume and velocity, the task of linking records in heterogeneous data collections has become more complicated. It is difficult to find semantic relations between entities across different datasets containing noisy data and missing values with loose schema bindings. At the same time, pairwise comparison of entities over large datasets implies and exhibits quadratic complexity. Some recent approaches reduce this complexity by aggregating similar entities into blocks. In SANSA, we implement a more generic method for entity resolution that does not use blocking and significantly reduces the quadratic comparisons. In SANSA, we use scalable techniques like vectorization using hashingTF, count-vectorization and Locality Sensitive Hashing [189] to achieve almost linear performance for large-scale entity resolution. An overview of the approach used in SANSA can be seen in Figure 8.

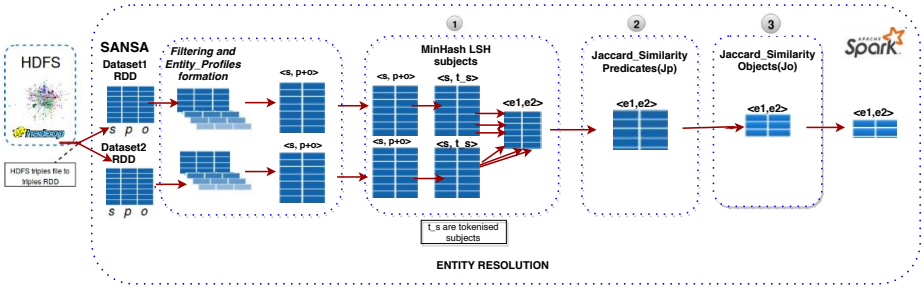

Fig. 8: Overview of Scalable Entity Linking.

3.9 Graph Kernels for RDF

Many machine learning algorithms strongly depend on the specific structure of the data, which forces users to fit their observations in a particular predefined setting or re-implement the algorithms to fit their requirements. For dynamic data models like Knowledge Graphs that can operate on schema-free structures, techniques like propositionalization or graph kernels are used. Inspired by [285], we developed graph kernels in SANSA. The walk kernel corresponds to a weighted sum of the cardinality of walks up to a given length. The number of paths can be calculated either by breadth-first search or by multiplication of the adjacency matrix. A path kernel is similar to walk kernel, but it counts the number of paths instead. Unlike walks, paths must consist of distinct vertices. SubtreekKernels attempt to limit the calculations of kernels by selecting subgraphs identified with

a central entity, and sharing a common structure. This enables a replacement of the intersection graph with other suitable structures. The full subtree kernels are based on the number of full subtrees contained in the intersection graph. The kernels, in general, return the set of feature vectors for the entities that can be further used in algorithms, like neural networks support vector machines or similar algorithms working on numerical data.

Apart from the analytics mentioned in this section, SANSA provides additional algorithms for rule mining, cluster evaluation, graph kernels as well. All of these algorithms are being continuously extended and improved. In addition, more algorithms are being added with time.

4 Grand Challenges and Conclusions

In this chapter, we provide an overview of SANSA's functionalities: an engine that attempts to fill the gap pointed in Chapter 3. SANSA is the only comprehensive system that addresses several challenges and provides libraries for the development of a knowledge graph value chain ranging from acquisition, distribution, and querying to complex analytics (see for instance [170, 413] where complex analyses were successfully computed on the Ethereum blockchain using SANSA).

The SANSA stack is a step in the direction of offering a seamless solution to help users dealing with big knowledge graphs. As a consequence, there are still **grand challenges** to face:

- Availability of data in RDF. This challenge is to be linked to the research directions on federated queries (Chapter 5) and to the design of mappings (Chapter 4) to pave the road for datalake-oriented solutions such as the one presented by Mami *et al.* [293]. While the representation of data as knowledge graphs has gained lots of traction and large-scale knowledge graphs are being created, a majority of data being created and stored is not-RDF and therefore challenges such as the necessary efforts for data cleaning, and/or data maintenance should be taken into account.
- RDF and Query layer. The distributed context requires smart partitioning methods (see [52] and [234] for detailed taxonomies) aligned with the querying strategies. One possibility would be to have dynamic partitioning paradigms which could be automatically selected based on data shape and/or query patterns, as envisioned in [14].
- In a distributed context, processes often share resources with concurrent processes, and therefore the definition itself of what is a "good" query answer time may vary, as reviewed in the context of distributed RDF solutions by Graux *et al.* in [169]. One could think of basing this performance evaluation on use-cases.
- Machine Learning and Partial access to data. Most machine learning algorithms generally require access to all the training data and work by iterating over the training data to fit the desired loss function. This is challenging

in the distributed setting where one might need to use multiple local learners or query processors (each working on a subset of the data) and optimize globally over (or collect) partial local results. For very large-scale distributed data, this working model may not be suitable [341]. Hence, there is a strong need to develop fundamentally new algorithms that can work with partial access to the data,

– Challenge on the Semantic Web itself. At the moment, using W3C standards, it is hard to be as expressive as with Property Graphs. This has led to the creation of RDF* [185, 184] in order to allow Semantic Web users to express statements of statements within an RDF extension. These new possibilities imply that the current landscape incorporates this extension while guaranteeing the same performances as before.