OWLStats: Distributed Computation of OWL Dataset Statistics

Heba Mohamed*†, Said Fathalla*†, Jens Lehmann*‡, and Hajira Jabeen§

*Smart Data Analytics (SDA), University of Bonn, Bonn, Germany
Email: {hmohamed,fathalla,jens.lehmann@cs.uni-bonn.de}
†Faculty of Science, University of Alexandria, Alexandria, Egypt
Email: {heba.ibrahim, sm_fathalla@alexu.edu.eg}
‡Fraunhofer IAIS, Dresden, Germany
§Cluster of Excellence on Plant Sciences (CEPLAS), University of Cologne, Cologne, Germany
Email: hajira.jabeen@uni-koeln.de

***Abstract*—Nowadays, ontologies are used in various application areas, involving Artificial Intelligence, Natural Language Processing, Data Integration, and Knowledge Management. It is essential to know the internal structure, distribution, and coherence of the published datasets to make it easier for reuse, interlink, integrate, infer, or query. Therefore, there is a pressing need to obtain a clear view on OWL datasets became more prevalent. In this paper, we present OWLStats, a software component for computing statistical information about large scale OWL datasets in a distributed manner. We present the primary distributed in-memory approach for computing 50 different statistical criteria for OWL datasets utilizing Apache Spark, which can scale horizontally to a cluster of machines. OWLStats has been integrated into the SANSA framework and anther two uses cases are also presented. The preliminary results prove that OWLStats is linearly scalable in terms of both node and data scalability.**

***Index Terms*—Apache Spark, Distributed Processing, OWL Statistics, SANSA Framework**

I. Introduction

The ontologies are particularly widespread in the life sciences, where several large biomedical ontologies have been developed, including the Biological Pathways Exchange (BioPAX) ontology[1], the GALEN ontology[2], and the National Cancer Institute thesaurus[3]. The ontologies are being used in application areas like Artificial Intelligence [1], [2], Natural Language Processing [3], Data Integration [4] and Knowledge Management Systems [5]. It is of vital importance to collect comprehensive statistics on the datasets illustrating their internal structure and external consistency to assess the efficiency of the individual datasets as well as to monitor the progress of Web data publishing and integration. Obtaining detailed statistical analyzes of datasets facilitates a variety of instances of imperative use and offers key benefits. For example: 1) *Link target identification:* To build a web of data, the linking between different datasets is of crucial importance for many Linked data applications such as ontology merging and fusion. Getting legitimate insights almost the inner structure of a dataset (mainly about the used classes, properties, vocabularies, etc.), rapid the identification of appropriate target datasets for linking will be significantly simplified. 2) *Vocabulary reuse:* Evaluating the vocabulary reuse is of significance since built up vocabularies constitute a significant prerequisite for an interoperable Web of Data. Hence, calculating the commonly used vocabularies simplifies dataset creation and integration. 3) *Quality analysis:* Assessing and evaluating the quality expected, and determining whether it is sufficient for a particular application is highly important. It is crucial to analyze datasets concerning incoming and outgoing links, the used vocabularies, and properties values and ranges, in order to create similar measures on the Web of Data. 4) *Coverage analysis:* To ensure that the frequent dataset properties are used with similar entities. Furthermore, namespaces frequency is an indicator of a dataset domain, i.e., the more namespaces belonging to a domain, the more relevant the dataset to that domain. A variety of approaches offer such computational statistics about RDF datasets [6]–[8]. Despite this interest, the best of our knowledge, none of the previous work has introduced a statistical computation framework for OWL datasets. Most studies have only tended to focus on triple structure analysis, rather than the axiom structure of the datasets. OWLStats is the first distributed approach for collecting comprehensive statistics over large-scale OWL datasets. In order to overcome the memory limitation, distributed in-memory computing frameworks, e.g., Apache Spark[4] or Flink[5] can be used. Due to its efficiency in handling large-scale datasets and scalability, Apache Spark has recently gained considerable attention. The primary abstraction that Spark is providing is the *Resilient Distributed Dataset* (RDD). There are other advantages of using RDDs, including in-memory computation, fault tolerance, partitioning, and persistence.

In this paper, we introduce the first software component, i.e., *OWLStats*, for comprehensive statistical computations of large-scale OWL datasets. The main contributions of this work

[1]http://www.biopax.org/
[2]http://www.openclinical.org/prj_galen.html
[3]https://ncit.nci.nih.gov
[4]https://spark.apache.org/
[5]https://flink.apache.org/

can be summarized as follows: 1) We proposed OWLStats approach for computing 50 different statistical criteria for OWL dataset, 2) OWLStats - as an open-source implementation using Apache Spark RDD parallel programming model. 3) We prove that OWLStats is scalable in terms of node and data scalability. 4) OWLStats has been integrated into the SANSA[6] framework. SANSA is actively maintained and uses community services, e.g., mailing lists, websites, etc. The remainder of this paper is organized as follows: the proposed approach, its architecture, and implementation are described in Section III. The experimental setup and the discussion of the obtained results are presented in Section IV. Three use cases are presented in Section V. Section II gives a brief overview of the related work. Ultimately, we conclude and the potential extension of OWLStats in Section VI.

II. Related Work

The last few years have witnessed considerable growth in the Linked Open Data (LOD) datasets, which can be consumed by software agents. These agents can explore, stream, recommend, and organize information in intelligent ways to assist web users. Comprehensive statistics calculation on such datasets has become a vital factor in describing their internal structure and coverage. These statistics are increasingly important in many areas, involving data analysis (e.g., quality analysis and coverage analysis), query optimization, and data interlinking and reuse. In this section, we outline the work related to RDF datasets and OWL ontologies statistics computations. Previous work has tended to focus on calculating RDF statistics, ignoring to address OWL statistics, especially when the talk is about large-scale OWL ontologies.

RDF datasets statistics computations. Few researchers have addressed the problem of calculating RDF statistics. Make-void[7] is a tool, written in Java, that computes statistics about RDF files, such as the number of triples, classes, and properties, by running several SPARQL queries using the ARQ query engine provided in the Apache Jena framework. It generates statistics in RDF format using voiD vocabulary in order to make it machine-readable. RDFStat [8] is another framework, based on the Jena framework, for calculating statistics from RDF sources, such as documents and SPARQL endpoints. It can generate statistical data, such as instances count, as well as histograms for a variety of various data types. Contrary to make-void, RDFStats can utilize SPARQL endpoints for querying RDF data and visualizations for its statistics. Furthermore, it does not use VoID for statistics description, but rather describe them using SCOVO (Statistical Core Vocabulary [9]). LODStat [6] is an approach, written as a Python module and uses the Redland library [10], for computing 32 different statistical criteria, such as typed string length, max per property and class hierarchy depth, described using VoID. The main advantage of LODStats, when compared to current approaches, is its significantly better performance and scalability as well as low memory consumption. One of the limitations of LODStat is that it can operate only on a single triple pattern, i.e., it does not support, for example, star patterns [11]. Nevertheless, it provides several schema-level, such as RDFS sub-hierarchy depth, and data-level statistics, such as counting triples with literals. LODStats has been integrated with the Comprehensive Knowledge Archive (CKAN)[8] dataset metadata registry in order to get a general overview of the current state of the Data on the Web. *Concerning distributed processing based approaches*, Böhm et al. [12] developed a scalable approach that automatically generates voiD (Vocabulary of Interlinked Datasets) descriptions for large corpora of Linked Data in a distributed manner. Another distributed in-memory approach for the computation of large RDF datasets statistics is DistLODStats [7]. DistLODStats extends LODStats by calculating the same statistical criteria but in a distributed and scalable manner. DistLODStats is implemented using the Apache Spark framework and is integrated into SANSA.

III. Approach

OWLStats adopted 30 criteria proposed in [6], [7] and added 20 more criteria. In contrast to [7], we perform the statistical computation on the axiom structure of the OWL datasets. We carried out the computation in Spark distributed environment using RDDs (Definition 2). The proposed approach involves the conversion of the input OWL dataset to RDDs of OWLAxioms. The following definition formalizes the concept of a statistical criteria [6]:

Definition 1 (Statistical Criterion): *A statistical criterion C is a triple $C = (F, D, P)$, where: F is a SPARQL filter condition, D is a derived dataset from the input dataset (RDD of OWLAxioms) after applying F, and P is a post-processing filter operating on the data structure D.*

F serves as a filter operation, to decide whether an axiom matches the condition of a specific criterion. The dataset is processed axiom by axiom, where each axiom examined is matched against each triple design of each criterion. *D* is the result RDD after applying *F* on the input OWL dataset. In most cases, the post-processing step is not required. However, *P* returns values from the derived dataset *D*. The post-processing operation performs further computational steps, such as retrieving the top-n elements of *D*. A formal representation for each statistical criteria is shown in Table I.

Example (Criteria 29 in Table I). This example illustrates the statistical criterion average per property, which computes the entities with their average values in the dataset. Here, the *Criterion name* is "Average per property (int, float, time)", the *Description* is "Lists all entities and their average values in the dataset", the *Filter* is "Filter all the data property assertions and check if the object is literal and of type: *int*, or *float*, or *time*", the *Action* is "Calculate the number of object connected with each property $m1$ and the number of properties $m2$", and the *Post-Processing* is $m1/m2$.

[6]http://sansa-stack.net/

[7]https://github.com/cygri/make-void

[8]http://thedatahub.org/

TABLE I: Statistical criterion defined by Spark rules.

Criterion	**Rule Filter**	→ **Rule Action**	**Postproc.**
1. used classes	A=Class_Assertion	→ map(_.getClassExpression)	-
2. class usage count	A=Class_Assertion	→map(_.getClassExpression) .reduceByKey(_+_)	take(100)
3. classes defined	A=Declaration.isOWLClass	→ map (_.getEntity.getIRI)	-
4. class hierarchy depth	A=SubClass_Of	G += (a.getSubClass,a.getSuperClass)	depth(G)
5. data property usage	A=Data_Property_Assertion	→map (a => (a.getProperty, 1)).reduceByKey(_+_)	take(100)
6. object property usage	A=Object_Property_Assertion	→map (a => (a.getProperty, 1)) .reduceByKey(_+_)	take(100)
7. property usage distinct per subj.		→ groupBy(_.getSubject).reduceByKey(_+_)	count()
8. property usage distinct per obj.		→ groupBy(_.getObject).reduceByKey(_+_)	count()
9. properties distinct per subj.		→ groupBy(_.getSubject).combineByKey(_+_)	sum/count
10. properties distinct per obj.		→ groupBy(_.getObject).combineByKey(_+_)	sum/count
11. outdegree		→map(_.getSubject).map(a => (a, 1)) .combineByKey(_+_)	sum/count
12. indegree		→map(_.getObject).map(a => (a, 1)) .combineByKey(_+_)	sum/count
13. data Property hierarchy depth	A=Sub_Data_Property_Of	G += (a.getSubProperty, a.getSuperProperty)	depth(G)
14. object Property hierarchy depth	A=Sub_Object_Property_Of	G += (a.getSubProperty, a.getSuperProperty)	depth(G)
15. subclass usage	A=SubClass_Of	→count()	-
16. axioms		→count()	-
17. entities mentioned		→map(a => a.getSubject.isNamed && a.getObject.isNamed).count()	-
18. distinct entities		→map(a => a.getSubject.isNamed && a.getObject.isNamed).distinct()	-
19. literals	A=Data_Proeprty_Assertion && obj.isLiteral	→count()	-
20. datatypes	A=Data_Proeprty_Assertion && obj.isLiteral	→ map (a => (o.getDatatype, 1)) .reduceByKey(_+_)	-
21. languages	A=Data_Proeprty_Assertion && obj.isLiteral	→map (a => (o.getLang, 1)) .reduceByKey(_+_)	-
22. average typed string length	A=Data_Proeprty_Assertion && obj.isLiteral && obj.getDatatype = XSD_STRING	→count() len+=o.length	len/count
23. average untyped string length	A=Data_Proeprty_Assertion && obj.isLiteral && !obj.getDatatype.isEmpty()	→count() len+=o.length	len/count
24. typed subject	A=Axiom_TYPES && p=RDF_TYPE	→count()	-
25. labeled subject	A=Annotation_Assertion && a.getProperty.isLabel	→count()	-
26. sameAs	A=SAME_INDIVIDUAL	→count()	-
27. links	A=Axiom_TYPES && s.getNS != o.getNS	→map (a => ((s.getNS, o.getNS), 1)) .reduceByKey(_+_)	-
28. max per property {int,float,time}	A=Data_Proeprty_Assertion && o.isLiteral&& o.getDatatype = isInt | isFloat | isDateTime	→map (_.getProperty, _.getObject) .reduceByKey(_ max _)	-
29. avg per property {int,float,time}	A=Data_Proeprty_Assertion && o.isLiteral&& o.getDatatype = isInt | isFloat | isDateTime	→m1 => map(_.getObject).count() m2 => map(_.getProperty).count()	m1/m2
30. subj. vocabularies		→map (a => (a.getsubject.getNS, 1)) .reduceByKey(_+_)	-
31. pred.. vocabularies		→map (a => (a.getProperty.getNS, 1)) .reduceByKey(_+_)	-
32. obj. vocabularies		→map (a => (a.getObject.getNS, 1)) .reduceByKey(_+_)	-
33. subdata property usage	A=Sub_Data_Property	→count()	-
34. subobject property usage	A=Sub_Object_Property	→count()	-
35. subannotation property usage	A=Sub_Annotation_Property	→count()	-
36. classes count		→flatMap(a => a.classesInSignature) .count()	-
37. data properties count		→flatMap(a => a.dataProperties InSignature).count()	-
38. object properties count		→flatMap(a => a.objectProperties InSignature).count()	-
39. class assertion count	A=Class_Assertion	→count()	-
40. data property assertion count	A=Data_Property_Assertion	→count()	-
41. object property assertion count	A=Object_Property_Assertion	→count()	-
42. annotation property assertion count	A=Annotation_Property_Assertion	→count()	-
43. different individuals	A=Different_Individuals	→count()	-
44. object intersection	A=Equivalent_Classes && Type=Object_Intersection_Of	→map(_.getOperandsAsList.get(1))	count()
45. object union	A=Equivalent_Classes && Type=Object_Union_Of	→map(_.getOperandsAsList.get(1))	count()
46. object complement	A=Equivalent_Classes && Type=Object_Complement_Of	→map(_.getOperandsAsList.get(1))	count()
47. data SomeValuesFrom	A=Equivalent_Classes && Type=Data_Some_Values_From	→map(_.getOperandsAsList.get(1))	count()
48. data AllValuesFrom	A=Equivalent_Classes && Type=Data_All_Values_From	→map(_.getOperandsAsList.get(1))	count()
49. data Cardinality	A=Equivalent_Classes && Type=Data_Min_Cardinality ‖ Type=Data_Max_Cardinality	→map(_.getOperandsAsList.get(1))	count()
50. data HasValue	A=Equivalent_Classes && Type=Data_Has_Value	→map(_.getOperandsAsList.get(1))	count()

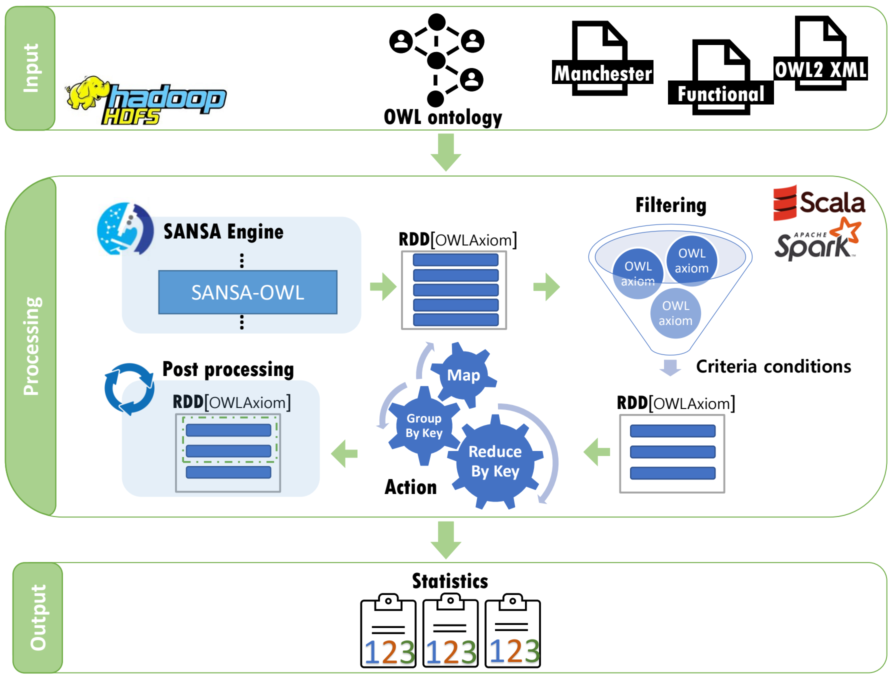

Fig. 1: OWLStats Architecture.

Definition 2 (RDD Operations): *All the statistical criteria implemented using the following operations: map, filter, reduceByKey, groupBy, and combineByKey. All RDD operations are documented in RDD Programming Guide*[9]*. A brief formalization for each operation is given as follows:*

- *map: The map function iterates over every line in RDD and converts it into new RDD based on a specific function.*
- *flatMap: flatMap is similar to map operation, except that map return one element, while flatMap return a list of elements based on a specific function.*
- *filter: New RDD returned containing only the elements that match a certain condition.*
- *reduceByKey: The input RDD for reduceByKey operation is a key-value (K, V) pairs, the pairs on the same machine with the same key are combined before the data is shuffled.*
- *groupBy: The input RDD is a key-value (K, V) pairs, returns a new RDD of (K, Iterable<V>) pairs after applying a grouping function (e.g., average, count) over the input RDD.*

A. OWLStats Architecture

Figure 1 illustrates the main workflow of the statistical computation proposed by OWLStats. OWLStats approach consists of three main steps: 1) Storing the OWL dataset into a scalable distributed storage, 2) converting the input dataset into the main data structure (i.e. `RDD[OWLAxiom]`), 3) Compute the statistical criteria evaluation and generating the results.

Step 1: *Storing the OWL dataset.* To read the OWL dataset efficiently, Spark needs the dataset to be stored in a large-scale storage system. The Hadoop Distributed File-System (HDFS) [13] is used for data storage. HDFS is designed to store and stream large datasets for user applications efficiently. HDFS splits the data into separate blocks when the data is loaded into HDFS, then it replicates and distributes the blocks to various nodes in a cluster, allowing highly efficient parallel processing and fault-tolerance. Consequently, the node information that crash can be found in a cluster elsewhere.

Step 2: *Dataset conversion.* To convert the input OWL dataset, we used SANSA-OWL[10] layer for the conversion. SANSA-OWL layer supports the conversion for three input formats: *Functional, Manchester, and OWL/XML.* The output of this step is `RDD[OWLAxiom]`.

Step 3: *Statistical criteria evaluation.* For each criterion, we start an execution plan to filter the input dataset and calculate the output. Spark transformations, i.e., *map, filter, groupBy, reduceByKey.*, perform the calculations. Computing phase output would be the statistical results expressed in a human-readable format e.g. VoID. VoID[11] is an RDF Schema vocabulary to represent metadata about RDF datasets.

B. Implementation

This section explains the implementation of OWLStats framework. All phases of the OWLStats have been implemented using Apache Spark. Scala[12] programming language

[9]https://spark.apache.org/docs/latest/rdd-programming-guide.html

[10]https://github.com/SANSA-Stack/SANSA-OWL

[11]https://www.w3.org/TR/void/

[12]https://www.scala-lang.org/

API has been used to provide a distributed implementation of the proposed approach. Algorithm 1 establishes the primary dataset from an OWL file (as constructed from line 2). The algorithm takes as input: the OWL dataset, the syntax of the OWL dataset (we support *Functional*, *Manchester* and *OWL/XML* syntax), and the list of the statistical criteria. Line 2 converts the input OWL file into `RDD[OWLAxiom]`. Afterwards, for each criterion defined inside OWLStats, the algorithm computes them using the *filter*, *action*, and *post-processing* operations (lines 5, 7, and 9). Every transformed RDD can be recalculated by default each time running an action on it. Nevertheless, an RDD could be persisted in memory for quicker access next time needed it instead of reconstructing the RDD. Spark caching techniques, persist or cache actions, can be used for faster access to RDD elements. In the OWLStats algorithm, caching is used twice to persist RDD elements in memory. In line 3, the OWLAxioms RDD to be used for each criterion is cached. Afterward, caching the derived RDD after applying the criterion filter condition on the input dataset (line 6).

Algorithm 1: OWLStats computation over set of statistical criteria

```
Input: spark: Spark Session,
       input: The OWL dataset,
       syntax: Syntax type (func, manch, owlxml),
       CL: List of criterion
1  begin
2      RDD axioms = input.convert(spark, syntax)
3      axioms.cache()
4      foreach  $c \in CL$ do
5          $rdd \leftarrow c.filter(axioms)$
6          rdd.cache()
7          $rdd \leftarrow c.action(rdd)$
8          if c.hasPostProc. then
9              $rdd \leftarrow c.postProc(rdd)$
10     end
11 end
```

Algorithm 2 describes the filter, action, and post processing operations for Criterion 2 listed in Table I. Function *filter()* extracts all the `Class_Assertion` axioms from the input OWLAxioms RDD and then convert it to `RDD[OWLClassAssertionAxioms]` instead of `RDD[OWLAxiom]` (lines 3-4), then get all the class expressions within the `OWLClassAssertionAxioms` (line 5). Afterwards, *action()* function is applied to calculate the frequency of each `OWLClassExpression` (lines 8-9). Finally, *postProc()* function is called to sort the calculated *action()* and return the first 100 `OWLClassExpression` with their corresponding frequencies (line 12).

Algorithm 2: Classes usage count criterion

```
Input  : axioms: rdd of the converted OWL dataset
Output: result: first 100 Classes and there usage frequency
1  begin
2      Function filter():RDD[OWLClassExpression]
3          val f = extractAxioms(axioms,
             AxiomType.CLASS_ASSERTION)
4
            .asInstanceOf[RDD[OWLClassAssertionAxiom]]
5                .map(_.getClassExpression)
6          return f
7      Function action():RDD[(OWLClassExpression,
         Int)]
8          val a = filter().map(e => (e, 1))
9                .reduceByKey(_ + _)
10         return a
11     Function postProc():RDD[(OWLClassExpression,
         Int)]
12         val p = action().sortBy(_._2, false).take(100)
13         return p
14 end
```

IV. Evaluation

In this section, we describe the evaluation of OWLStats. We are aiming to address the following questions concerning scalability and flexibility: Q1) How does OWLStats scale to larger datasets? Q2) How is the speedup ration affected with respect to change the number of worker nodes? Q3) How does OWLStats process different datasets sizes? Q4) How fast is OWLStats in computing the introduced computational statistics? We start with the experimental setup, afterward present the results and then discuss details.

A. Experimental Setup

System configuration. All distributed experiments ran on a cluster with five nodes. Among these nodes, one is reserved to act as the master and four nodes used as computing workers. Each node has AMD Opteron 2.3 GHz processors (64 Cores), 250.9 GB memory, and the configured capacity is 1.7 TB. The nodes are connected with 1 Gb/s Ethernet. Also, Spark v2.4.4 and Hadoop v2.8.1 with Java 1.8.0 is installed on this cluster. Local-mode experiments are all carried out on a single cluster instance. All distributed experiments run three times, and the results indicate average execution time.

Benchmark. Lehigh University (LUBM) [14] synthetic benchmark has been used for the experiment. For the evaluation of Semantic Web repositories, LUBM is a commonly used benchmark for evaluating the efficiency of such repositories regarding extensional queries over a large dataset. LUBM generator generates many A-Box axioms but no T-Box axioms. We use the LUBM data generator in our experiment to generate five datasets of different sizes: LUBM-50, LUBM-200, LUBM-500, LUBM-1000, and LUBM-2000. The numbers attached to the benchmark name is the number of generated universities. Properties of the generated datasets, loading time to the HDFS, and the number of axioms of each dataset are listed in Table II. To create larger datasets from the ontology files created from the LUBM benchmark, we implemented a merge tool.

B. Results and Discussion

We evaluate our approach using the aforementioned datasets to study its performance as well as scalability. In this evaluation, we measure the scalability of OWLStats based on node and data scalability. We conducted two types of experiments. First, we evaluate the execution time of our distributed approach with different data sizes (Experiment 1). Second, by increasing worker nodes (machines) in the cluster, we evaluate the horizontal scalability (Experiment 2). Figure 2 and Figure 4 reports the results of efficiency analysis for data and node scalability, respectively.

Experiment 1 (Data Scalability). In this experiment, we measure the efficiency of OWLStats by increasing the size of the input dataset. We retain a constant number of nodes (workers) at five in the cluster and increase the size of datasets to assess whether the proposed approach can handle larger datasets. To test the data scalability for OWLStats, we run the experiments on the LUBM benchmark on five different sizes. We begin by generating a dataset of 50 universities (LUBM-50), then we iteratively increase the number of universities (i.e., scaling up the size). Figure 2 displays the run time of the proposed distributed algorithm with and without caching for each dataset. Caching is a mechanism to speed up applications that have multiple access to the same RDD, which keeps the data in memory and accelerates the computations. It is apparent from Figure 2, the reduction in execution time between OWLStats with caching (blue columns) and without caching (red columns). The x-axis represents the LUBM datasets produced with an increase in the number of universities, while the y-axis represents the execution time within minutes. For example, calculating the 50 statistics criteria with LUBM-2000 costs around 31 minutes without using the caching mechanism, while the time after caching was triggered, decreased to 24 minutes. Spark has the performance advantage of using in-memory data storage. The use of data storage in memory contributes to a decrease in the average time spent on network communication and data read/write using disk-based approaches It is evident that the execution time increases linearly as the size of the dataset increases. The results show that our algorithm can be scaled according to dataset size, which answers *Q1* and *Q3*. Figure 3 shows the output obtained from running OWLStats in a multi-machine (five-machine) cluster environment. For more illustrations, consider the LUBM-1000 dataset; the execution time decreased from 48.83 minutes in a single machine environment (local) down to 19.67 minutes in multiple machines environment (cluster).

TABLE II: LUBM benchmark datasets (functional syntax)

Dataset	Size (GB)	Load Time (m)	#Axioms
LUBM-50	1.3	8	6,654,756
LUBM-200	4.8	23	31,178,382
LUBM-500	10.5	82	77,577,078
LUBM-1000	20.3	126	151,066,104
LUBM-2000	41.2	230	306,002,524

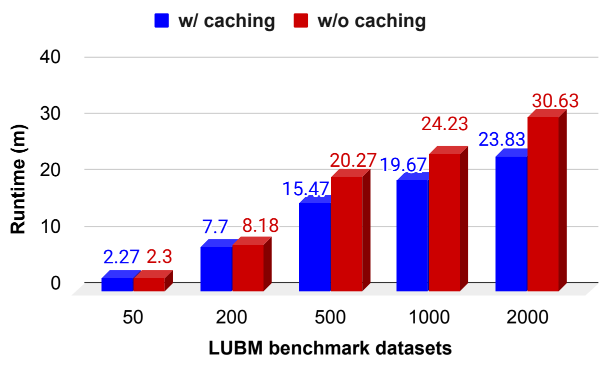

Fig. 2: OWLStats sizeup performance evaluation

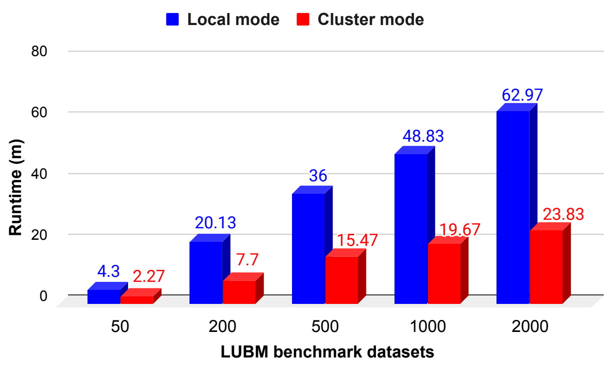

Fig. 3: OWLStats speedup performance evaluation in cluster and local environments.

The observed time decrease can be interpreted as a result of multiple machine distribution of the computation. For LUBM-1000, the use of cluster mode speeds up performance by two times, which addresses *Q2*.

Experiment 2 (Node Scalability). In this experiment, we increase the number of cluster workers in order to measure the node scalability. We have increased the number of workers from one to four (with one step each time). Figure 4 shows LUBM-200's speedup efficiency by raising the number of worker nodes from one to five. The execution time decreased around three times (from 26.3 min to 8.2 min). It is evident that as the number of workers increases, the execution time decreases linearly. The speedup ratio (S) is an essential metric for calculating the parallel algorithms (addressing Q4). The speedup ratio is the ratio $S = T_L/T_N$, where T_L is the execution time of the algorithm in local mode, and T_N is the time on N workers. Efficiency E measures the speedup per worker. It is the time taken to run the algorithm in local mode against the time on N workers, $E = S/N$. Figure 5 illustrates the speedup and efficiency ratios for the LUBM-200 dataset. The speedup for the selected data is increased sequentially with respect to the number of workers. In conclusion, the findings illustrate that OWLStats would achieve near-linear scalability of output in the sense of speedup.

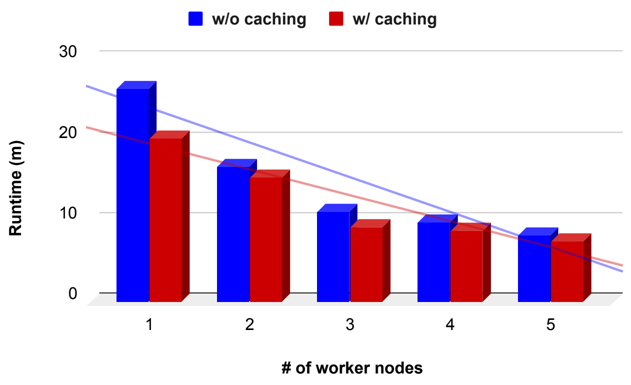

Fig. 4: OWLStats scalability performance evaluation

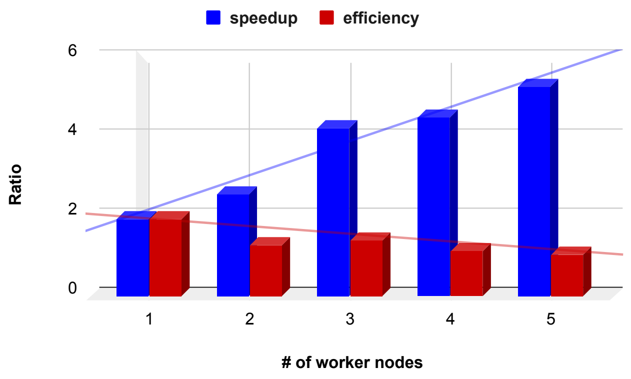

Fig. 5: OWLStats efficiency and speedup ratio.

Criteria Execution. The overall execution time of OWLStats per each criterion is illustrated in Figure 6. The runtime for each criterion is reported for both LUBM-50 and LUBM-200 datasets. OWLStats consists of 50 statistical criteria; each criterion execution time depends on the number of the input OWLAxiom. The findings obtained from both datasets' execution show that runtime is less when there is no data shuffle inside the cluster. Criteria 25, requires less execution time since it relies on `OWLAnnotationAssertions` of type `owl:literal` and the number of annotation assertions in LUBM benchmark not too much. The longest execution time is taken by Criteria 17. The reason refers to that even though it requires no data shuffling, but no filter applied to the input data; therefore, the whole dataset is processed to evaluate the criteria. LUBM benchmark generates many A-box axioms (i.e., assertions) but no T-Box axioms; thus, criteria from 33 to 43 consume long execution time because they are all evaluated on assertion axioms. Criteria 30, 31, and 32 concerning the vocabularies used in the dataset are considered efficient since there is no data movement among the cluster nodes. Criteria 44 to 50 were not evaluated because the LUBM benchmark contains only six `OWLEquivalentClasses` such that those consume almost no time. Overall, the experiments led us to conclude that OWLStats can complete statistical computation execution in a reasonable time, proving that OWLStats distributed statistical criteria computation is scalable.

To evaluate OWLStats over more complex ontologies (i.e., ontologies with more T-Box axioms), we use it over Gene Ontology (GO)[13]. GO is considered as the leading source of gene information. It consists of 601,235 axioms. OWLStats computes all the criteria within 326 seconds on a cluster of five worker nodes. Furthermore, we evaluate OWLStats over the University Ontology Benchmark (UOBM)[14], which generates more complex and realistic datasets. We generate a dataset with ten universities (maximum number of universities offered by the benchmark), which contains 1,475,83 axioms. Within 195 seconds, OWLStats measures all statistical criteria on a cluster of five worker nodes.

V. Use Cases

OWLStats is a generic software component for computing statistical information about large-scale OWL datasets. In this section, we present three use cases for our proposed approach.

SANSA-Stack: OWLStats has been successfully integrated into Scalable Semantic Analytics Stack (SANSA-Stack) framework [15], [16]. SANSA is an open source[15] large-scale processing engine for efficient processing of large-scale RDF datasets. The need for utilizing fault-tolerant big data frameworks such as Apache Spark and Flink is raised, to process this massive amount of data efficiently. SANSA is built on top of Spark, offering a set of facilities for the representation (RDF and OWL), querying and inference of semantic data. Currently, SANSA-RDF[16] layer supports 32 statistical criteria for RDF datasets. Some of the machine learning algorithms in the inference and machine learning layers are built on top axioms level. Therefore, we integrated OWLStats into the SANSA framework to support the SANSA-OWL layer to compute 50 statistical criteria based on OWL datasets (Functional, Manchester, and OWL/XML formats).

PLATOON Project[17]: PLATOON is an EU-funded H2020 project that digitizes the energy sector with the adoption of AI techniques. The objective is to increase renewable energy consumption, smart grids management, and energy efficiency. PLATOON reference architecture is used to construct and deploy scalable and replicable energy management solutions. OWLStats will be used in PLATOON to collect statistical information to assist in the development of prediction algorithms.

ABSTAT: ABSTAT [17] is a summarization framework to support linked set understanding. ABSTATS[18] framework can provide compact as well as concise summaries for a given dataset. To derive summarization for more input datasets of OWL formats, OWLStats will be used.

[13]http://geneontology.org/
[14]https://www.cs.ox.ac.uk/isg/tools/UOBMGenerator/
[15]https://github.com/SANSA-Stack
[16]https://github.com/SANSA-Stack/SANSA-RDF
[17]https://platoon-project.eu/
[18]http://abstat.disco.unimib.it/

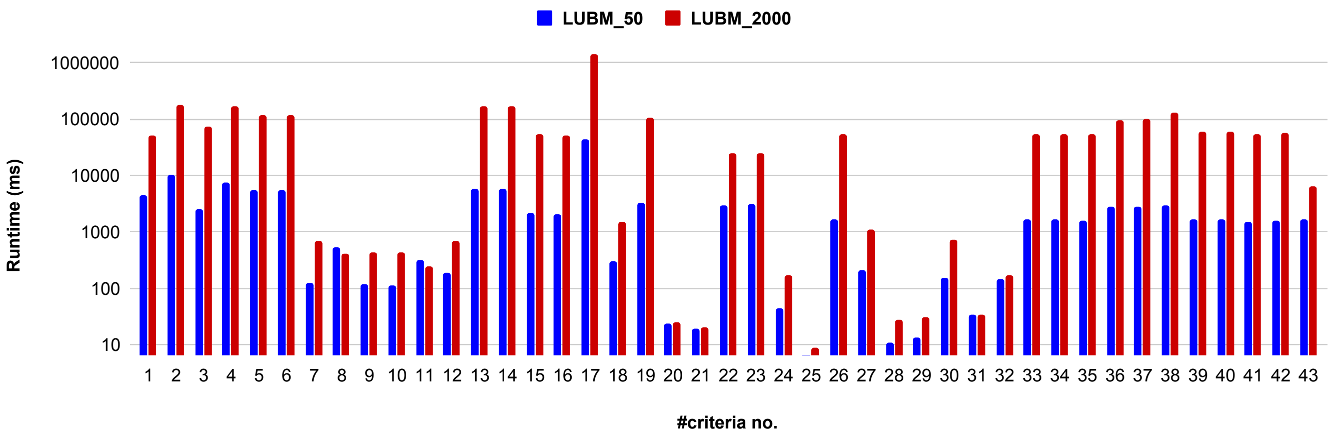

Fig. 6: Overall execution analysis per criteria (log scale)

VI. Conclusion and Future Work

In this paper, we proposed a novel approach *OWLStats*, to evaluate 50 statistical criteria for OWL datasets. OWLStats is an open-source distributed framework for computing comprehensive statistics over OWL datasets. Interestingly, after reviewing the literature, we found no tool which could calculate such statistics. In conclusion, the evaluation of OWLStats achieved near-linear scalability of output in the sense of speedup. OWLStats has been successfully integrated into the SANSA framework. We carried out two experiments to test OWLStats's efficiency as well as the data and node scalability. To further our research, we are planning to add more criteria that cover further OWL axioms structures. Moreover, we aim to introduce further improvements in terms of code optimization, such as using different persisting strategies and using Alluxio[19] to bridge the gap between application and storage system.

References

[1] M. Martínez-Romero, J. M. Vázquez-Naya, J. R Rabunal, S. Pita-Fernández, R. Macenlle, J. Castro-Alvariño, L. López-Roses, J. L Ulla, A. V Martinez-Calvo, S. Vázquez *et al.*, "Artificial intelligence techniques for colorectal cancer drug metabolism: ontologies and complex networks," *Current drug metabolism*, vol. 11, no. 4, pp. 347–368, 2010.

[2] P. Hohenecker and T. Lukasiewicz, "Ontology reasoning with deep neural networks," *arXiv preprint arXiv:1808.07980*, 2018.

[3] D. Gocheva, H. Eminova, and I. Batchkova, "Ontology based data and information integration in biomedical domain," *Machines. Technologies. Materials.*, vol. 10, no. 2, pp. 35–38, 2016.

[4] G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, and R. Rosati, "Using ontologies for semantic data integration," in *A Comprehensive Guide Through the Italian Database Research Over the Last 25 Years*. Springer, 2018, pp. 187–202.

[5] K. Banujan and S. Vasanthapriyan, "Knowledge sharing system for dental extraction in order to assist dental doctors and assistants," 2018.

[6] S. Auer, J. Demter, M. Martin, and J. Lehmann, "Lodstats–an extensible framework for high-performance dataset analytics," in *International Conference on Knowledge Engineering and Knowledge Management*. Springer, 2012, pp. 353–362.

[7] G. Sejdiu, I. Ermilov, J. Lehmann, and M. N. Mami, "Distlodstats: Distributed computation of rdf dataset statistics," in *International Semantic Web Conference*. Springer, 2018, pp. 206–222.

[8] A. Langegger and W. Woss, "Rdfstats-an extensible rdf statistics generator and library," in *2009 20th International Workshop on Database and Expert Systems Application*. IEEE, 2009, pp. 79–83.

[9] M. Hausenblas, D. Ayers, L. Feigenbaum, T. Heath, W. Halb, and Y. Raimond, "The statistical core vocabulary (scovo)," *Digital Enterprise Research Institute (DERI), DERI Specification*, 2012.

[10] D. Beckett, "The design and implementation of the redland rdf application framework," in *Proceedings of the 10th international conference on World Wide Web*, 2001, pp. 449–456.

[11] T. Gottron, M. Knauf, S. Scheglmann, and A. Scherp, "A systematic investigation of explicit and implicit schema information on the linked open data cloud," in *Extended Semantic Web Conference*. Springer, 2013, pp. 228–242.

[12] C. Böhm, J. Lorey, and F. Naumann, "Creating void descriptions for web-scale data," *Journal of Web Semantics*, vol. 9, no. 3, pp. 339–345, 2011.

[13] K. Shvachko, H. Kuang, S. Radia, R. Chansler *et al.*, "The hadoop distributed file system." in *MSST*, vol. 10, 2010, pp. 1–10.

[14] Y. Guo, Z. Pan, and J. Heflin, "Lubm: A benchmark for owl knowledge base systems," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 3, no. 2-3, pp. 158–182, 2005.

[15] J. Lehmann, G. Sejdiu, L. Bühmann, P. Westphal, C. Stadler, I. Ermilov, S. Bin, N. Chakraborty, M. Saleem, A.-C. N. Ngomo *et al.*, "Distributed semantic analytics using the sansa stack," in *International Semantic Web Conference*. Springer, 2017, pp. 147–155.

[16] I. Ermilov, J. Lehmann, G. Sejdiu, L. Bühmann, P. Westphal, C. Stadler, S. Bin, N. Chakraborty, H. Petzka, M. Saleem *et al.*, "The tale of sansa spark." in *International Semantic Web Conference (Posters, Demos & Industry Tracks)*, 2017.

[17] B. Spahiu, R. Porrini, M. Palmonari, A. Rula, and A. Maurino, "Abstat: ontology-driven linked data summaries with pattern minimalization," in *European Semantic Web Conference*. Springer, 2016, pp. 381–395.

[19]https://docs.alluxio.io/os/user/stable/en/Overview.html