DistLODStats: Distributed Computation of RDF Dataset Statistics

Gezim Sejdiu[1(✉)], Ivan Ermilov[2], Jens Lehmann[1,3], and Mohamed Nadjib Mami[1,3]

[1] Smart Data Analytics, University of Bonn, Bonn, Germany
{sejdiu,jens.lehmann,mami}@cs.uni-bonn.de
[2] Department of Computer Science, University of Leipzig, 04109 Leipzig, Germany
iermilov@informatik.uni-leipzig.de
[3] Fraunhofer IAIS, Sankt Augustin, Germany
{jens.lehmann,mohamed.nadjib.mami}@iais.fraunhofer.de

Abstract. Over the last years, the Semantic Web has been growing steadily. Today, we count more than 10,000 datasets made available online following Semantic Web standards. Nevertheless, many applications, such as data integration, search, and interlinking, may not take the full advantage of the data without having a priori statistical information about its internal structure and coverage. In fact, there are already a number of tools, which offer such statistics, providing basic information about RDF datasets and vocabularies. However, those usually show severe deficiencies in terms of performance once the dataset size grows beyond the capabilities of a single machine. In this paper, we introduce a software component for statistical calculations of large RDF datasets, which scales out to clusters of machines. More specifically, we describe the first distributed in-memory approach for computing 32 different statistical criteria for RDF datasets using Apache Spark. The preliminary results show that our distributed approach improves upon a previous centralized approach we compare against and provides approximately linear horizontal scale-up. The criteria are extensible beyond the 32 default criteria, is integrated into the larger SANSA framework and employed in at least four major usage scenarios beyond the SANSA community.

1 Introduction

Over the last two decades, the Semantic Web has grown from a mere idea for modeling data in the web, into an established field of study driven by a wide range of standards and protocols for data consumption, publication and exchange on the Web. For the record, today we count more than 10,000 datasets openly available online using Semantic Web standards[1]. Thanks to such standards, large datasets

Resource type: Software Framework.
Website: http://sansa-stack.net/distlodstats/.
Permanent URL: https://doi.org/10.6084/m9.figshare.6080711.

[1] http://lodstats.aksw.org/.

D. Vrandečić et al. (Eds.): ISWC 2018, LNCS 11137, pp. 206–222, 2018.
https://doi.org/10.1007/978-3-030-00668-6_13

became machine-readable [13]. Nevertheless, many applications such as data integration, search, and interlinking may not take full advantage of the data without having *a priori* statistical information about its internal structure and coverage. RDF dataset statistics can be beneficial in many ways, for example: (1) Vocabulary reuse (suggesting frequently used similar vocabulary terms in other datasets during dataset creation), (2) Quality analysis (analysis of incoming and outcoming links in RDF datasets to establish hubs similar to what pagerank has achieved in the traditional web), (3) Coverage analysis (verifying whether frequent dataset properties cover all similar entities and other related tasks), (4) privacy analysis (checking whether property combinations may allow to uniquely identify persons in a dataset) and (5) link target analysis (finding datasets with similar characteristics, e.g. similar frequent properties) for interlinking candidates.

A number of solutions have been conceived to offer users such statistics about RDF vocabularies [17] and datasets [7,9]. However, those efforts showed severe deficiencies in terms of performance when the dataset size goes beyond the main memory size of a single machine. This limits their capabilities to medium-sized datasets only, which paralyzes the role of applications in embracing the increasing volumes of the available datasets.

As the memory limitation was the main shortcoming in the existing works, we investigated parallel approaches that distribute the workload among several separate memories. One solution that gained traction over the past years is the concept of *Resilient Distributed Dataset (RDDs)*, initially suggested at [18], which are in-memory data structures. Using RDDs, we are able to perform operations on the whole dataset stored in a significantly enlarged distributed memory.

Apache Spark[2] is an implementation of the concept of RDDs. It allows performing coarse-grained operations over voluminous datasets in a distributed manner in parallel. It extends earlier efforts in the area such as Hadoop MapReduce.

In this paper, we introduce a software component "DistLODStats" for statistical evaluation of large RDF datasets, which scales out to clusters of multiple machines. We extend the approach proposed in [5] for computing 32 different statistical criteria for RDF datasets. Our contributions can be summarized as follows:

- We propose an algorithm for computing RDF dataset statistics and implement it using an efficient framework for large-scale, distributed and in-memory computations: Apache Spark.
- We perform an analysis of the complexity of the computational steps and the data exchange between nodes in the cluster.
- We evaluate our approach and demonstrate empirically its superiority over a previous centralized approach.
- We integrated the approach into the SANSA framework, where it is actively maintained and re-uses the community infrastructure (mailing list, issues trackers, website etc.).
- We briefly describe four usage scenarios for DistLODStats.

[2] http://spark.apache.org.

The paper is structured as follows: Our approach for the computation of RDF dataset statistics is detailed in Sect. 2 and evaluated in Sect. 3. Related work on the computation of RDF statistics is discussed in Sect. 5. Finally, we conclude and suggest planned extensions of our approach in Sect. 6.

2 Approach

In this paper, we adopted the 32 statistical criteria proposed in [5]. In contrast to [5], we perform the computation in a large-scale distributed environment using Spark and the concept of RDDs. Instead of processing the input RDF dataset directly, this approach requires the conversion to an RDD that is composed of three elements: *Subject*, *Property* and *Object*. We name such an RDD a *main dataset*.

The statistical criteria proposed in [5] are formalized as a triple (F, D, P) consisting of a filter condition F, a derived dataset D and a post processing operation P. In our approach, we adapt the definition of those elements to be applicable to RDDs.

Definition 1 (Statistical criterion). *A statistical criterion $\mathcal{C}$ is a triple $\mathcal{C} = (F, D, P)$, where:*

- *F is a SPARQL filter condition.*
- *D is a derived dataset from the main dataset (RDD of triples) after applying F.*
- *P is a post-processing filter operating on the data structure D.*

F acts as a filter operation, which determines whether a specific criterion is matched against a triple in the *main dataset*. D is the result of applying the criterion on the *main dataset*. P is an operation applied to D to (optionally) perform further computational steps. If no extra computation are needed, P just returns exactly the results from the intermediate dataset D.

2.1 Main Dataset Data Structure

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 in parallel on large clusters. By using RDDs, Spark abstracts away the differences of the underlying data sources. RDDs during their lifecycle are kept in-memory, which enables efficient reuse of RDDs during several consequent transformations. Spark provides fault-tolerance by keeping a lineage information (a Directed Acyclic Graph (DAG) of transformations) for each RDD. This way any RDD can be reconstructed in case of node failure by tracing back the lineage. Spark enables full control over the persistence state and partitioning of the RDDs in the cluster. Thus, we can further improve computational efficiency of statistical criteria by planning a suitable storage strategy (i.e. alternating between memory and disk). For example, we can precisely determine which RDDs will be reused, and manage the degree of parallelism by specifying how an RDD is partitioned across the available resources.

Definition 2 (Basic Operations). *All the statistical criteria can be represented in our approach using the following basic operations: map, filter, reduce-by, and group-by. These operations can be formalized as follows:*

- *map* : $I \rightarrow O$, *where* I *is an input RDD and* O *is an output RDD. Map transforms each value from an input RDD into another value, following a specified rule.*
- $filter : I \rightarrow O$, *where* I *is an input RDD and* O *is an output RDD, which contains only the elements that satisfy a condition.*
- $reduce : I \rightarrow O$, *where* I *is an input RDD of key-value* (K,V) *pairs and* O *is an output RDD of* $(K, list(V))$ *pairs.*
- $group\text{-}by : (I,F) \rightarrow O$, *where* I *is an input RDD of pairs* $(K, list(V))$, F *is a grouping function (e.g., count, avg), and* O *is an output RDD containing the values in* $list(V)$ *from* I *aggregated using the grouping function.*

2.2 Distributed LODStats Architecture

The computation of statistical criteria is performed as depicted in Fig. 1. Our approach consists of three steps: (1) saving RDF data in scalable storage, (2) parsing and mapping the RDF data into the *main dataset*, and (3) performing statistical criteria evaluation on the *main dataset* and generating results.

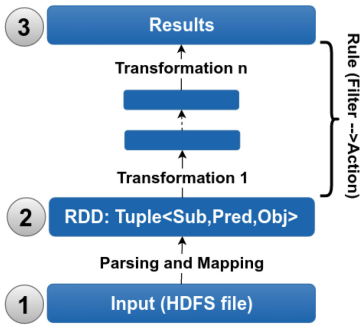

Fig. 1. RDD lineage of a criterion execution.

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)[3]. HDFS is able to accommodate any type of data in its raw format, horizontally scale to arbitrary number of 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.

[3] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

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*.

2.3 Algorithm

The DistLODStats algorithm (see Algorithm 1) constructs the *main dataset* from an RDF file (line 1). Afterwards, the algorithm iterates over the criteria defined inside the DistLODStats framework and evaluates them (lines 4, 6 and 8).

To define a statistical criterion inside the DistLODStats framework, one must specify *filter*, *action*, and *postProc* methods. The evaluation of the criterion then starts first by the *filter* method (line 4) that is used to apply the rule filters of the criterion (Rule Filter in Table 1). Applied on a *main dataset*, this latter will return a new RDD with a subset of the triples. Next, the *action* method is used to apply the criterion's rule action (Rule Action in Table 1). Applied on the filtered RDD, this either computes statistics directly or reorganizes the RDD so statistics can be computed in the next step. At the end, the *postProc* method is used as an optional operation to perform further statistical computations (e.g. average after count or sort).

Algorithm 1. DistLODStats.

```
   input : RDF: an RDF dataset, C: a list of criterion.
1  RDD mainDataset = RDF.toRDD < Triple > ()
2  mainDataset.cache()
3  foreach c ∈ C do
4      triples ← c.filter(mainDataset)
5      triples.cache()
6      triples ← c.action(triples)
7      if c.hasPostProc then
8          triples ← c.postProc(triples)
```

In our work, we make use of Spark caching techniques. Basically, if an RDD is constructed from a data source e.g. file, or through a lineage of RDDs, and then cached, there is no need to construct the RDD again the next time it is needed. We have used two different approaches for caching: (1) caching the *main dataset* entirely (line 2), and (2) caching a derived RDD after applying the criteria *filter* on the *main dataset* (line 5). In the first approach, the RDD is constructed from the RDF source during the first criteria computation, so the next criteria do not need to fetch it again. In the second approach, the RDD resulting from executing

the *filter* of one criterion is cached and used by any other criterion sharing the same *filter* pattern.

2.4 Complexity Analysis

The performance of criteria computation depends on two factors mainly:

- **Data Shuffling and Filtering.** In general, the computation can be expensive if there is data movement involved during the distributed execution, which is also known as shuffling. This generally happens when there is a data reduction (in the map-reduce sense). This entails cases like grouping together similar data or applying aggregation functions for SUM, AVG, COUNT, etc. Another factor influencing the performance of criteria computation are filters. The more data is filtered in early stages, the less processing is required in subsequent steps.
- **Data Scanning.** To execute the criterion filter on the same data, data is scanned only once for all criteria. However if data changes state, for example is mapped to another form with new columns added, then another scan of the new state is needed. Finally, if data is shuffled across cluster nodes, then a new scan is needed as well.

Per-criterion Complexity Analysis. Based on the two previous factors, we performed a complexity analysis of each statistical criterion. The results are reported in Table 2. We deem the complexity is mostly linear corresponding to cases where only one or limited number of scans is required. However there are situations where the complexity can increase when there are iterative executions, like the case of data sorting or graph-based computations (e.g. finding cycles or getting the path between two edges).

Below we give an overview of complexity analysis for our most operators used through our approach.

The complexity of $map()$ and $filter()$ itself is linear with respect to the number of input triples. The overall complexity depends on the functions passed to them. Consider an RDD as a single data structure on memory, any other operations (such as map and filter) are linear, or $O(n)$. The subsequent step is to split this RDD between s nodes, the complexity on each node then becomes $O(n/s)$. Let be f a function with complexity $O(f)$, then its complexity will be $O(n/s * O(f))$. As evident from the formula $O(n/s * O(f))$, the runtime increases linearly when the size of RDD increases and *decreases* linearly with the number of nodes in the cluster in case of a function f with with $O(f) = O(1)$.

The complexity of the $sortBy$ operation according to Spark[4] is a sampled $O(n)$, which means only the unique sample keys m (with $m \leq n$) are sorted and lead to a complexity of $O(m * log(m))$ plus the ranges of key sets. Afterwords, the data is shuffled around in $O(n)$ which is costly as sorting needs to be applied internally for the range of keys collected on a given partition p, i.e. $O(p * log(p))$ time is required.

[4] https://github.com/apache/spark/blob/d5b1d5fc80153571c308130833d0c0774de62c92/core/src/main/scala/org/apache/spark/Partitioner.scala#L101.

Table 1. Definition of Spark rules (using Scala notation) per criterion.

	Criterion	Rule (Filter → Action)		Postproc.
1	used classes	`p=RDF_TYPE && o.isURI()`	`→ map(_.o)`	–
2	class usage count	`p=RDF_TYPE && o.isURI()`	`→ map(o => (o, 1)).reduceByKey(_ + _)`	`take(100)`
3	classes defined	`p=RDF_TYPE && s.isURI()&& (o=RDFS_CLASS||o=OWL_CLASS)`	`→ map(_.s)`	–
4	class hierarchy depth	`p=RDFS_SUBCLASS_OF && s.isIRI() && o.isIRI()`	`→ G += (?s,?o)`	`depth(G)`
5	property usage		`→ map(f => (f.pred, 1)) .reduceByKey(_ + _)`	`take(100)`
6	property usage distinct per subj.		`→ groupBy(_.subj) .reduceByKey(_ + _)`	`count`
7	property usage distinct per obj.		`→ groupBy(_.obj) .reduceByKey(_ + _)`	`count`
8	properties distinct per subj.		`→ groupBy(_.subj) .combineByKey(_ + _)`	`sum/count`
9	properties distinct per obj.		`→ groupBy(_.obj) .combineByKey(_ + _)`	`sum/count`
10	outdegree		`→ map(_.s).map(f => (f, 1)) .combineByKey(_ + _)`	`sum/count`
11	indegree		`→ map(_.o).map(f => (f, 1)) .combineByKey(_ + _)`	`sum/count`
12	property hierarchy depth	`p=RDFS_SUBPROPERTY_OF && s.isIRI() && o.isIRI()`	`→ G += (?s,?o)`	`depth(G)`
13	subclass usage	`p=RDFS_SUBPROPERTY_OF`	`→ count()`	–
14	triples		`→ count()`	–
15	entities mentioned		`→ map(f=>(s.isURI(), p.isURI(),o.isURI())).count`	–
16	distinct entities		`→ map(f=>(s.isURI(), p.isURI(),o.isURI())).distinct`	–
17	literals	`o.isLiteral()`	`→ count()`	–
18	blanks as subj.	`s.isBlank()`	`→ count()`	–
19	blanks as obj.	`o.isBlank()`	`→ count()`	–
20	datatypes	`o.isLiteral()`	`→ map(o => (o.dataType(), 1)) .reduceByKey(_ + _)`	–
21	languages	`o.isLiteral()`	`→ map(o => (o.languageTag(), 1)) .reduceByKey(_ + _)`	–
22	average typed string length	`o.isLiteral() && obj .getDatatype()=XSD_STRING)`	`→ count(); len+=o.length()`	`len/count`
23	average untyped string length	`o.isLiteral() && o.getDatatype().isEmpty()`	`→ count(); len+=o.length()`	`len/count`
24	typed subject	`p=RDF_TYPE`	`→ count()`	–
25	labeled subject	`p=RDFS_LABEL`	`→ count()`	–
26	sameAs	`p=OWL_SAME_AS`	`→ count()`	–
27	links	`!s.getNS() =(o.getNS())`	`→ map(f => (s.getNS()+ o.getNS())) .map(f=> (f, 1)).reduceByKey(_ + _)`	–
28	max per property {int,float,time}	`o.getDatatype()={XSD_INT | XSD_float | XSD_datetime}`	`→ map(f => (f.p, f.o)) .maxBy(_._2)`	–
29	average per property {int,float,time}	`o.getDatatype()={XSD_INT | XSD_float | XSD_datetime}`	`→ m1=>map(_.o).count m2=>map(_.p).count`	`m1/m2`
30	subj. vocabularies		`→ map(f => (f.s.getNS())) .map(f => (f, 1)).reduceByKey(_ + _)`	–
31	pred. vocabularies		`→ map(f => (f.p.getNS())) .map(f => (f, 1)).reduceByKey(_ + _)`	–
32	obj. vocabularies		`→ map(f => (f.o.getNS())) .map(f => (f, 1)).reduceByKey(_ + _)`	–

Table 2. Complexity and data shuffling breakdown by statistical criterion. Notation conventions: n = number of triples; V = number of vertices; E = number of edges.

Criterion	Runtime complexity	Data shuffling and data scanning
(1, 3)	O(n)	Data is filtered locally and returned, i.e. no data exchange is needed
(2, 5)	As sorting is required to retrieve the top 100 results, i.e. the complexity depends on the sorting algorithm used	This operation can be implemented in a map-reduce fashion: classes initially are distributed across the cluster, so calculating their counts requires data to be shuffled and then reduced. The sorting in post-processing requires moving the data. Currently, data is sorted in each node and the union of the datasets is subsequently sorted as well
(6, 7, 8, 9)	O(n)	Following a map-reduce approach, the data is first mapped to <subject,property> pairs and then reduced by subject, so data needs to be shuffled prior to the grouping. De-duplication (distinct) is automatically achieved by the reduce function
(4, 12)	O(V+E)	The best representation of this criterion is a graph where data is already connected, and only linear traversal is required so no data transfer is needed
(10, 11, 20, 21)	O(n)	Following a map-reduce approach, data is first mapped to <subject,1> and then reduced by subject counting the 1s, so data needs to be shuffled prior to the grouping
(13, 14)	O(n)	The count is performed locally and the individual counts are summed up for the cluster, i.e. no data movement is needed
(15)	O(n)	Counting of entities with mentioned s, p and o is done in parallel, so the overall count uses individual counts and sums them. Hence, no data transfer is needed
(16)	O(n)	This is similar to 15, but instead of counting, just returning the triples, so data is saved directly after checking isURI and saved back, i.e. no data is moved
(17, 18, 19, 24, 25, 26, 27, 30, 31, 32)	O(n)	Data is filtered and then counted in each node, the overall count can be obtained by summing up individual counts, so no data movement
(23, 23)	O(n)	The computation requires to project out the objects only and map them to the length of themselves, then the average is computed by summing up the length dividing by the size of each map. The AVG count is done in parallel in each node and then the AVG of all AVGs is a matter of getting single values from each node, so no data movement is needed
(28)	O(n)	Obtaining the maximum per property requires also reducing data distributed in the cluster, so data movement needed
(29)	O(n)	The data here is also reduced by property, so the sum and the count, thus the average, can happen in the same time. Either way, data needs to be moved across the cluster

2.5 Implementation

DistLODStats comprises three main phases depicted in Fig. 2 and explained previously. The output of the *Computing* phase will be the statistical results represented in a human-readable format e.g. VoID, or row data. We expressed the three phases of the 32 criteria using the basic operations defined in Definition 2. Next, those have been mapped to Spark transformations and actions in Table 1, where: *map* is mapped directly to Spark `Map()`, *reduce* is mapped to `groupByKey()`, and *group-by* is mapped to `reduceByKey()`. Exceptions of this general strategy were done for the implementation of the post processing steps

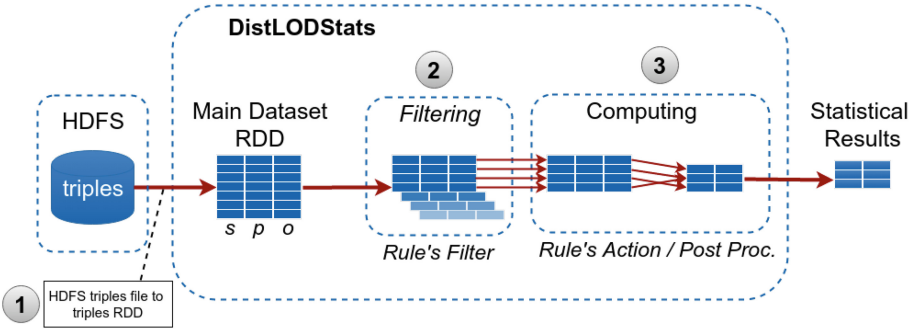

Fig. 2. Overview of DistLODStats's abstract architecture.

of Criteria 4 and 12, where we use a Spark GraphX[5], which is more suitable for this particular case of graph-oriented criterion computation.

Furthermore, we provide a Docker image of the system[6] available under *Apache License 2.0*, integrated within the BDE platform[7] - an open source Big Data Processing Platform allowing users to install numerous big data processing tools and frameworks and create working data flow applications.

We implemented DistLODStats using Spark-2.2.0, Scala 2.11.11 and Java 8. DistLODStats has meanwhile been integrated into SANSA [6,11], an open source[8] *data flow processing engine* for performing distributed computation over large-scale RDF datasets. It provides data distribution, communication, and fault tolerance for manipulating large RDF graphs and applying machine learning algorithms on the data at scale. Via this integration, DistLODStats can also leverage the developer and user community as well as infrastructure behind the SANSA project. This also ensure the sustainability of DistLODStats given that SANSA is backed by several grants until at least 2021.

3 Evaluation

The aim of our evaluation is to see how well our approach can perform against non-distributed approaches as well as analysing the scalability of the distributed approach. In particular, we addressed the following questions: (Q_1): How does the runtime of the algorithm change when more nodes in the cluster are added? (Q_2): How does the algorithm scale to larger datasets? (Q_3): How does the algorithm scale to a larger number of datasets?

In the following, we present our experimental setup including the datasets used. Thereafter, we give an overview of our results, which we subsequently discuss in the final part of this section.

[5] https://spark.apache.org/docs/latest/graphx-programming-guide.html.
[6] https://github.com/SANSA-Stack/Spark-RDF-Statistics.
[7] https://github.com/big-data-europe.
[8] https://github.com/SANSA-Stack.

3.1 Experimental Setup

We used one synthetic and two real world datasets for our experiments:

1. We chose the geospatial dataset LinkedGeoData [16] which offers a spatial RDF knowledge base derived from OpenStreetMap.
2. As a cross domain dataset, we selected *DBpedia* [10] (v 3.9). DBpedia is a knowledge base with a large ontology.
3. As a synthetic dataset, we chose to use the Berlin SPARQL Benchmark (BSBM) [2]. It is based on an e-commerce use case which is built around a set of products that are offered by different vendors. The benchmark provides a data generator, which can be used to create sets of connected triples of any particular size.

Properties of these datasets are given in Table 3.

Table 3. Dataset summary information (nt format).

⟶		DBpedia			BSBM		
	LinkedGeoData	en	de	fr	2 GB	20 GB	200 GB
#nr. of triples	1,292,933,812	812,545,486	336,714,883	340,849,556	8,289,484	81,980,472	817,774,057
Size (GB)	191.17	114.4	48.6	49.77	2	20	200

For the evaluation, all data is stored on the same HDFS cluster using Hadoop 2.8.0. All experiments were carried out on a 6 nodes cluster (1 master, 5 workers): Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10 GHz (32 Cores), 128 GB RAM, 12 TB SATA RAID-5. The experiments on a local mode are all performed on a single instance of the cluster. We ran two centralized versions of LODStats (explained below at Sect. 3.2) for comparison. The machines were connected via a Gigabit network. All experiments were executed three times and the average value is reported.

3.2 Results

We evaluate our approach using the above datasets to compare it against the original LODStats. We carried out two sets of experiments. First, we evaluate the execution time of our distributed approach against the original approach. Second, we evaluate the horizontal scalability via increasing nodes (machines) in the cluster. Results of the experiments are presented in Table 4, Figs. 3, 4 and 5.

Distributed Processing on Large-Scale Datasets
To address Q_1, we started our experiments by evaluating the *speedup* gained by adopting a distributed implementation of LODStats criteria using our approach, and compare it against the original centralized version. We run the experiments on four datasets ($DBpedia_{en}$, $DBpedia_{de}$, $DBpedia_{fr}$, and $LinkedGeoData$) in

Table 4. Distributed Processing on Large-Scale Datasets.

⟶	Runtime (h) (mean/std)				
	LODStats		DistLODStats		
	(a) files	(b) bigfile	(c) local	(d) cluster	(e) speedup ratio
LinkedGeoData	n/a	n/a	36.65/0.13	**4.37**/0.15	7.4x
$M^{en}_{DBpedia}$	**24.63**/0.57	fail	25.34/0.11	**2.97**/0.08	7.6x
$M^{de}_{DBpedia}$	n/a	n/a	10.34/0.06	**1.2**/0.0	7.3x
$M^{fr}_{DBpedia}$	n/a	n/a	10.49/0.09	**1.27**/0.04	7.3x

a local environment on a single instance with two configurations: (1) files of the dataset are considered separately, and (2) one big file–all files concatenated.

Table 4 shows the performance of two algorithms applied to the four datasets. The column LODStats $^{(a)}$ reports on the performance of LODStats on files separately (considering each file as a sequence of execution), the next columns LODStats$^{(b)}$ reports on the performance of LODStats using a single big file by concatenating each file, and the last columns reports on the performance of DistLODStats on the same case as previously i.e. the performance for one big dataset in local mode (c) and cluster mode (d). We observe that the execution in DistLODStats$^{(c),(d)}$ finishes with all the datasets (see Fig. 3). However, for LODStats$^{(a),(b)}$ the execution often fails at different stages of the execution. In particular, *n/a* indicates parser exceptions and *fail* out of memory exceptions. The only case where the execution finishes and actually slightly outperforms DistLODStats$^{(c)}$ on a single node is executing LODStats on the dataset $DBpedia^{en}$ split into files (25.34 h for DistLODStats$^{(c)}$ vs 24.63 h in LODStats$^{(a)}$). This is because the DistLODStats$^{(c)}$ considers the input dataset as a big file instead of evaluation it on each file separately. LODStats streams the criteria one by one, so having a large dataset streamed that way would lead to very high processing times. However, with small data as input, the processing can finish in short amount of time, but the results can be very inaccurate.

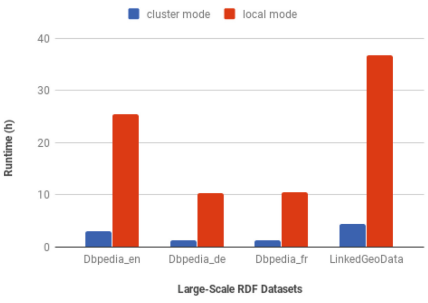

Fig. 3. Speedup performance evaluation of DistLODStats.

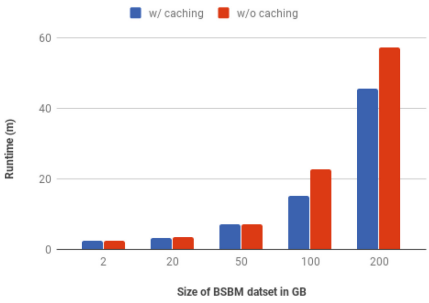

Fig. 4. Sizeup performance evaluation of DistLODStats.

Figure 3 shows the speedup performance evaluation for large-scale RDF Datasets for DistLODStats on local mode and cluster mode, respectively. All results illustrate consistent improvement for each dataset when running on a cluster. The maximum speedup is 7.6x and the geometric mean of the speedup is 7.4x.

For example, on $DBpedia_{en}$, the time on cluster mode is about 2.97 h which is 7.6 times faster than evaluating DistLODStats on local mode (about 25.34 h). The reason why the time spent on local mode extremely decreases is that the size of the working directory of worker processes is too large and Spark uses threads for distributing the tasks.

Scalability

Sizeup Scalability. To measure the performance of *size-up* i.e. scalability of our approach, we run experiments on three different sizes. This analysis keeps the number of nodes in a cluster constant, we fix the number of workers (nodes) to 5 and grow the size of datasets to measure whether a given algorithm can deal with larger datasets. Since real-world datasets are considered to be unique in the size and also on other aspects e.g. number of unique terms, we chose the BSBM benchmark tool to generate artificial datasets of different sizes. We started by generating a dataset of 2 GB. Then we iteratively increased the size of datasets by one order of magnitude.

On each dataset, we ran the distributed algorithm and the runtime is reported on Fig. 4. The x-axis is a generated BSBM dataset per each order of 10x magnitude.

By comparing the runtime (see Fig. 4), we note that the execution time cost grows linearly and is near-constant when the size of the dataset increases. As expected, it stays near-constant as long as the data fits in memory. This demonstrates one of the advantages of utilizing an in-memory approach in performing the statistics computation. The overall time spent in data read/write and network communication found in disk-based approaches is no present in distributed in-memory computing. The performance only starts to degrade when substantial amounts of data need to be written to disk due to memory overflows. The results show scalability of our algorithm in context of sizeup, which answers question Q_2.

Node Scalability. In order to measure node scalability, we use variations of the number of the workers on our cluster. The number of workers varies from 1, 2, 3 and 4 to 5.

Let T_N be the time required to complete the task on N workers. The speedup S is the ratio $S = \frac{T_L}{T_N}$, where T_L is the execution time of the algorithm on local mode. Efficiency measures the processing power being used (i.e speedup per worker). It is defined as the time to run the algorithm on N workers compared to the time to run algorithm on local mode: $E = \frac{S}{N} = \frac{T_L}{NT_N}$.

Figure 5 shows the speedup for $BSBM_{50GB}$. We can see that as the number of workers increases, the execution time cost is super-linear. As depicted in Fig. 6, the speedup performance trend is consistent as the number of workers increases.

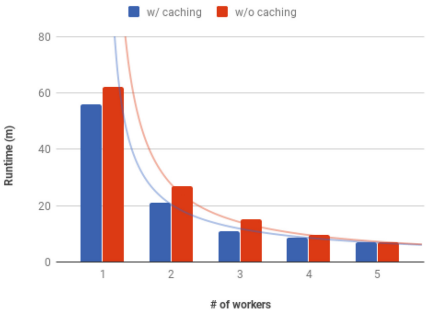

Fig. 5. Scalability performance evaluation on DistLODStats.

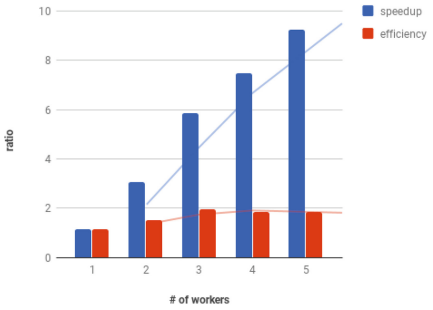

Fig. 6. Speedup ratio and efficiency of DistLODStats.

In contrast, as the number of workers was increased from 1 to 5, efficiency increased only up to the 4th worker for $BSBM_{50GB}$ dataset. This implies that the tasks generated from the given dataset were covered with almost 4 nodes. The results imply that DistLODStats can achieve near linear or even super linear scalability in performance, which answers question Q_3.

Breakdown by Criterion

Now we analyze the overall runtime of criteria execution. Fig. 7 reports on the runtime of each criterion on both $BSBM_{20GB}$ and $BSBM_{200GB}$ datasets.

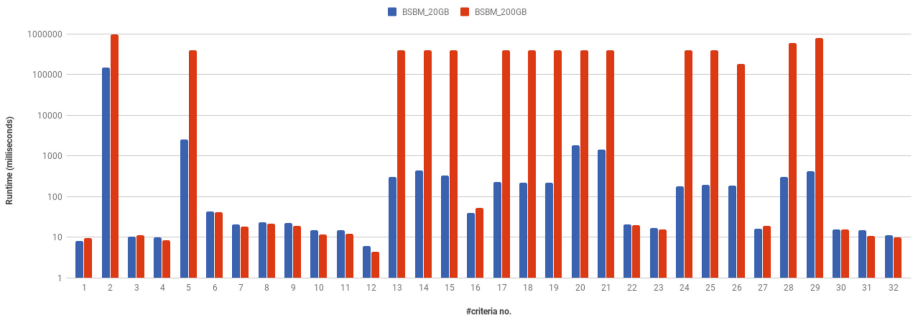

Fig. 7. Overall breakdown by criterion analysis (log scale).

Discussion. DistLODStats consists of 32 predefined criteria most of which have a runtime complexity of $O(n)$ where n is the number of input triples. The breakdown for BSBM with two instances is shown in Fig. 7. The results obtained confirm to a large extent the pre-analysis made in Subsect. 2.4. The execution is longer when there is data movement in the cluster compared to when data is processed without movement e.g. Criterion 2, 3 and 4. There are some criteria that are quite efficient to compute even with data movement e.g. 22, 23. This

is because data is largely filtered before the movement. Criterion 2 and 28 are the most expensive ones in terms of time of execution. This is most probably because of the sorting and maximum algorithm used by Spark. Criteria 20 and 21 are particularly expensive because of the extra overhead caused by extracting the data type and language for each particular object of type Literal. Criteria like 14 and 15 do not require movement of data, but yet are inefficient in execution. This is because the data is not filtered previously. The last three criteria do include data movement but are among the most efficient ones. This is because the low number of namespaces the chosen datasets have.

Overall, the evaluation study conducted demonstrates that parallel and distributed computation of the different statistical values is scalable, i.e. the execution finishes in reasonable time relative to the high volume of datasets.

4 Use Cases

DistLODStats is a generic tool for horizontally scalable statistics evaluation. We are aware of the following major users of the tool:

Comprehensive Statistics – LODStats. LODStats[9] is a project, which has crawled RDF data from metadata portals for the past seven years. It interacts with the CKAN dataset metadata registry to obtain a comprehensive picture of the current state of the Data Web. The drawback of the previous engine for LODStats is its inability to horizontally scale out, which naturally limited its scope to small and medium size datasets. For this reason, statistical criteria for several large-scale datasets were not reflected in the project website. Meanwhile, DistLODStats is used as underlying engine overcoming the previous limitations and generating statistical descriptions, including e.g. VoID, for large parts of the Linked Open Data Cloud.

Big Data Platform – BDE. Big Data Europe (BDE)[10] [1] is an open source big data processing platform allowing users to deploy Big Data processing tools and frameworks. Those tools and frameworks usually generate large amounts of log data. DistLODStats is used for computing statistics over those logs within the BDE platform. BDE uses the Mu Swarm Logger service[11] for detecting docker events and convert their representation to RDF. In order to generate visualisations of log statistics, BDE then calls DistLODStats from SANSA-Notebooks [6].

Blockchain – Alethio Use Case. Alethio is building an Ethereum analytics platform that strives to provide transparency over the transaction pool of the Ethereum p2p network. Their 5 billion triple data set contains large scale blockchain transaction data modelled as RDF according to the structure of the Ethereum ontology[12]. Alethio is using SANSA in general and DistLODStats

[9] http://lodstats.aksw.org/.
[10] https://github.com/big-data-europe.
[11] https://github.com/big-data-europe/mu-swarm-logger-service.
[12] https://github.com/ConsenSys/EthOn.

specifically in order to perform large-scale batch analytics, e.g. computing the asset turnover for sets of accounts, computing attack pattern frequencies and Opcode usage statistics. DistLODStats was run on a 100 node cluster with 400 cores to compute those statistics.

LOD Summaries – ABSTAT. ABSTAT[13][14] is a framework that aims to provide a better understanding of linked data sets. It implements an ontology-driven linked data summarization approach. DistLODStats is used for data set summarisation of large-scale RDF datasets in this context.

5 Related Work

In this section, we provide an overview of related work regarding RDF dataset statistics calculation. To the best of our knowledge, all but one existing approaches use small to medium scale datasets and do not horizontally scale. A dataset is large-scale w.r.t. a particular task in the scope of this article if the main memory on commodity hardware is insufficient to perform the task (without swapping to disk). We mention here, for example RDF_{Pro} [3], which offers a suite of stream-oriented, highly optimized processors for common tasks, such as data filtering, RDFS inference, smushing, as well as statistics extraction. The second related approach we are aware of is Aether [12], which is an application for generating, viewing and comparing extended VoID statistical descriptions of RDF datasets. The tool is useful, for example, in getting to know a newly encountered dataset, in comparing the different versions of a dataset, and in detecting outliers and errors. Luzzu [4] is a quality assessment framework for linked data. Its Quality Metric Language (LQML), is a domain specific language (DSL) that enables knowledge engineers to declaratively define quality metrics whose definitions can be understood more easily. LQML offers notations, abstractions and expressive power, focusing on the representation of quality metrics. However, only one work we came across that provided a distributed framework for RDF statistics computation: LODOP [8]. LODOP adopts a MapReduce approach for computing, optimizing, and benchmarking data profiling techniques. It uses Apache Pig as the underlying computation engine (Hadoop-based). LODOP implements 15 data profiling tasks comparing to 32 in our work. Because of the usage of MapReduce, the framework has a significant drawback: materialization of intermediate results between Map and Reduce and between two subsequent jobs is done on disk. DistLODStats does not use the disk-based MapReduce framework (Hadoop), but rather bases its computation mainly in-memory, so runtime performance is presumably better [15]. Unfortunately, we were unable to run LODOP for comparison. This is due to technical problems encountered, despite the very significant effort we devoted to deploy and run it. To the best of our knowledge, DistLODStats is the first software component for in-memory distributed computation of RDF dataset statistics.

[13] http://abstat.disco.unimib.it/.

6 Conclusions and Future Work

For obtaining an overview over the Web of Data as well as evaluating the quality of individual datasets, it is important to gather statistical information describing characteristics of the internal structure of datasets. However, this process is both data-intensive and computing-intensive and it is a challenge to develop fast and efficient algorithms that can handle large scale RDF datasets.

In this paper, we presented DistLODStats, a novel software component for distributed in-memory computation of RDF Datasets statistics implemented using the Spark framework. DistLODStats is maintained and has an active community due to its integration in SANSA. Our definition of statistical criteria provides a framework reducing the implementation effort required for adding further statistical criteria. We showed that our approach improves upon a previous centralized approach we compare against. Since Spark RDDs are designed to scale horizontally, cluster sizes can be adapted to dataset sizes accordingly. Although we achieved reasonable results in terms of scalability, we plan to further improve time efficiency by persisting the data to an even higher extend more in memory and perform load balancing.

Acknowledgment. This work was partly supported by the EU Horizon2020 projects BigDataEurope (GA no. 644564), QROWD (GA no. 723088), WDAqua (GA no. 642795) and BigDataOcean (GA no. 732310).

References

1. Auer, S., et al.: The BigDataEurope platform - supporting the variety dimension of big data. In: 17th International Conference on Web Engineering (2017)
2. Bizer, C., Schultz, A.: The berlin SPARQL benchmark. Int. J. Semant. Web Inf. Syst. **5**, 1–24 (2009)
3. Corcoglioniti, F., Rospocher, M., Mostarda, M., Amadori, M.: Processing billions of RDF triples on a single machine using streaming and sorting. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, pp. 368–375. ACM (2015)
4. Debattista, J., Auer, S., Lange, C.: Luzzu – a methodology and framework for linked data quality assessment. J. Data Inf. Qual. (JDIQ) **8**(1), 4 (2016)
5. Auer, S., Demter, J., Martin, M., Lehmann, J.: LODStats — an extensible framework for high-performance dataset analytics. In: ten Teije A., et al. (eds.) Knowledge Engineering and Knowledge Management, EKAW 2012, Lecture Notes in Computer Science, vol. 7603, pp. 353–362. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33876-2_31
6. Ermilov, I., et al.: The Tale of Sansa Spark. In: Proceedings of 16th International Semantic Web Conference, Poster and Demos (2017)
7. Ermilov, I., Martin, M., Lehmann, J., Auer, S.: Linked open data statistics: collection and exploitation. In: Proceedings of the 4th Conference on Knowledge Engineering and Semantic Web (2013)
8. Forchhammer, B., Jentzsch, A., Naumann, F.: LODOP - multi-query optimization for linked data profiling queries. In: International Workshop on Dataset PROFIling and fEderated Search for Linked Data (PROFILES), Heraklion, Greece (2014)

9. Langegger, A., Wöß, W.: RDFstats - an extensible RDF statistics generator and library. In: DEXA Workshops, pp. 79–83. IEEE Computer Society (2009)
10. Lehmann, J., et al.: DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. Semant. Web J. **6**(2), 167–195 (2015)
11. Lehmann, J., et al.: Distributed semantic analytics using the SANSA stack. In: Proceedings of 16th International Semantic Web Conference (2017)
12. Mäkelä, E.: Aether – generating and viewing extended VoID statistical descriptions of RDF datasets. In: Presutti, V., Blomqvist, E., Troncy, R., Sack, H., Papadakis, I., Tordai, A. (eds.) ESWC 2014. LNCS, vol. 8798, pp. 429–433. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11955-7_61
13. Ngonga Ngomo, A.-C., Auer, S., Lehmann, J., Zaveri, A.: Introduction to linked data and its lifecycle on the web. In: Reasoning Web (2014)
14. Palmonari, M., Rula, A., Porrini, R., Maurino, A., Spahiu, B., Ferme, V.: ABSTAT: linked data summaries with ABstraction and STATistics. In: Gandon, F., Guéret, C., Villata, S., Breslin, J., Faron-Zucker, C., Zimmermann, A. (eds.) ESWC 2015. LNCS, vol. 9341, pp. 128–132. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25639-9_25
15. Shi, J., et al.: Clash of the titans: mapreduce vs. spark for large scale data analytics. Proc. VLDB Endow. **8**(13), 2110–2121 (2015)
16. Stadler, C., Lehmann, J., Höffner, K., Auer, S.: LinkedGeoData: a core for a web of spatial open data. Semant. Web J. **3**(4), 333–354 (2012)
17. Vandenbussche, P.-Y., Atemezing, G.A., Poveda-Villalón, M., Vatant, B.: Linked open vocabularies (LOV): a gateway to reusable semantic vocabularies on the web. Semant. Web, 1–16 (2015). Preprint(Preprint)
18. Zaharia, M., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (2012)