Efficient Distributed In-Memory Processing of RDF Datasets

Dissertation
zur
Erlangung des Doktorgrades (Dr. rer. nat.)
der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität Bonn

von
Gezim Sejdiu
aus
Smire, Kosovo

Bonn, 06.05.2020

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

1. Gutachter:	Prof. Dr. Jens Lehmann
2. Gutachter:	Prof. Dr. Sören Auer

Tag der Promotion:	29.09.2020
Erscheinungsjahr:	2020

Abstract

Over the past decade, vast amounts of machine-readable structured information have become available through the automation of research processes as well as the increasing popularity of knowledge graphs and semantic technologies. Today, we count more than 10,000 datasets made available online following Semantic Web standards. A major and yet unsolved challenge that research faces today is to perform scalable analysis of large-scale knowledge graphs in order to facilitate applications in various domains including life sciences, publishing, and the internet of things. The main objective of this thesis is to lay foundations for efficient algorithms performing analytics, i.e. exploration, quality assessment, and querying over semantic knowledge graphs at a scale that has not been possible before. First, we propose a novel approach for statistical calculations of large RDF datasets, which scales out to clusters of machines. In particular, we describe the first distributed in-memory approach for computing 32 different statistical criteria for RDF datasets using Apache Spark. Many applications such as data integration, search, and interlinking, may take full advantage of the data when having a priori statistical information about its internal structure and coverage. However, such applications may suffer from low quality and not being able to leverage the full advantage of the data when the size of data goes beyond the capacity of the resources available. Thus, we introduce a distributed approach of quality assessment of large RDF datasets. It 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. Based on the knowledge of the internal statistics of a dataset and its quality, users typically want to query and retrieve large amounts of information. As a result, it has become difficult to efficiently process these large RDF datasets. Indeed, these processes require, both efficient storage strategies and query-processing engines, to be able to scale in terms of data size. Therefore, we propose a scalable approach to evaluate SPARQL queries over distributed RDF datasets by translating SPARQL queries into Spark executable code. We conducted several empirical evaluations to assess the scalability, effectiveness, and efficiency of our proposed approaches. More importantly, various use cases i.e. Ethereum analysis, Mining Big Data Logs, and Scalable Integration of POIs, have been developed and leverages by our approach. The empirical evaluations and concrete applications provide evidence that our methodology and techniques proposed during this thesis help to effectively analyze and process large-scale RDF datasets. All the proposed approaches during this thesis are integrated into the larger SANSA framework.

Acknowledgements

This work would not have been possible without the support and guidance of many people. First and foremost, I would like to express my sincere thanks to my supervisor Prof. Dr. Jens Lehmann for his constant guidance and support throughout the PhD studies. At the same time, I am also greatly appreciated by his kindness, patience, and encouragement that let me feel more confident and grow gradually as an independent researcher. I was fortunate to have Prof. Lehmann as an advisor during the development of this thesis. I am also thankful to Prof. Dr. Sören Auer for his support during this thesis. His insightful guidance helped me to see new ideas when tackling the research problem. They are both inspiring mentors and continue to lead by example. They always challenge me to push ideas and work further, and share their advice and experience on life and research. Being able to learn from both of them has been my great fortune.

I would like to thank all the staff members of the Smart Data Analytics (SDA) group at the University of Bonn, for the great time I had. I really enjoyed the atmosphere and also their friendship. Special thanks go to Monish Dubey, Harsh Thakkar, Dr. Diego Esteves, Mehdi Ali, Ass. Prof. Maria Maleshkova, Denis Lukovnikov, Dr. Günter Kniesel, Dr. Sahar Vahdati, Dr. Giulio Napolitano, Hamid Zafar, Debanjan Chaudhuri, Rostislav Nedelchev, Nilesh Chakraborty, Gaurav Maheshwari, Priyansh Trivedi, Mohamed Nadjib Mami, and Debayan Banerjee. Within our research group, I had the opportunity to be involved and responsible with regard to managing and maintaining some parts of the SANSA project as well as teaching and supervising master students. I want to thank all my colleagues in the SDA group and am glad to have the opportunity to be part of this group.

Draft versions of the thesis were read by Dr. Hajira Jabeen, Dr. Damien Graux, and Dr. Anisa Rula and I thank them for their valuable feedback and support, which helped to improve the overall quality of the thesis. I am also grateful for previous collaborations and discussions we had, as a result, it helped me to acquire and improve my academic skills.

As most of the research ideas described in this thesis were implemented, evaluated and integrated into the open-source SANSA project. I thank everyone working on this project. In particular, Ivan Emirlov for his DevOps help when it was needed, Lorenz Bühmann for his constructive feedback and help while working in SANSA, Claus Stadler, Simon Bin, Patrick Westphal and many more.

I would like to sincerely thank Shendrit Bytyqi and Ali Salihu for their support when we arrived in Germany. Finally, I would like to express my gratitude to my family and friends, for their persistent support and love and enriching my life beyond my scientific endeavors. In particular, I thank my lovely wife - Mimoza Sejdiu, for her constant support, sacrifices and understanding in the past years. Also, I would like to thank my beloved sons - Jon Sejdiu and Nil Sejdiu, for their love and motivation throughout the years of my PhD work. Thank you all.

This PhD thesis is dedicated
to my lovely wife, Mimoza Sejdiu
and my beloved sons, Jon Sejdiu and Nil Sejdiu.
Love you.

Contents

1 Introduction **1**
- 1.1 Problem Definition and Challenges . . . 2
  - 1.1.1 Challenge 1: Scalable Computation of RDF Dataset Statistics . . . 2
  - 1.1.2 Challenge 2: Quality Assessment of RDF Dataset at Scale . . . 3
  - 1.1.3 Challenge 3: Efficient and Scalable SPARQL Query Evaluation . . . 3
- 1.2 Research Questions . . . 3
- 1.3 Thesis Overview . . . 4
  - 1.3.1 Contributions . . . 4
  - 1.3.2 List of Publications . . . 7
- 1.4 Thesis Outline . . . 9

2 Preliminaries **11**
- 2.1 Semantic Technologies . . . 11
  - 2.1.1 RDF Data . . . 12
  - 2.1.2 SPARQL . . . 16
- 2.2 Hadoop Ecosystem . . . 17
  - 2.2.1 Apache Hadoop and MapReduce . . . 18
  - 2.2.2 Apache Spark . . . 19

3 Related Work **23**
- 3.1 RDF Dataset Statistics Systems . . . 23
- 3.2 RDF Quality Assessment Frameworks . . . 25
- 3.3 SPARQL Query Evaluators . . . 27

4 Large-Scale RDF Dataset Statistics **31**
- 4.1 A Scalable Distributed Approach for Computation of RDF Dataset Statistics . . . 32
  - 4.1.1 Main Dataset Data Structure . . . 33
  - 4.1.2 Distributed LODStats Architecture . . . 34
  - 4.1.3 Algorithm . . . 34
  - 4.1.4 Complexity Analysis . . . 35
  - 4.1.5 Implementation . . . 38
  - 4.1.6 Evaluation . . . 39
- 4.2 STATisfy: A REST Interface for DistLODStats . . . 45
  - 4.2.1 System Design Overview . . . 46
- 4.3 Summary . . . 47

5 Quality Assessment of RDF Datasets at Scale **49**
5.1 A Scalable Framework for Quality Assessment of RDF Datasets 50
5.1.1 Quality Assessment Pattern 50
5.1.2 System Overview 52
5.1.3 Implementation 53
5.2 Evaluation 55
5.2.1 Experimental Setup 55
5.2.2 Results 56
5.3 Summary 61

6 Scalable RDF Querying **63**
6.1 Sparklify: A Scalable Software for SPARQL Evaluation of Large RDF Data 64
6.1.1 System Architecture Overview 65
6.1.2 Evaluation 66
6.2 A Scalable Semantic-Based Distributed Approach for SPARQL Query Evaluation . . 71
6.2.1 System Architecture Overview 71
6.2.2 Distributed Algorithm Description 73
6.2.3 Evaluation 75
6.3 Summary 80

7 Implementation and Use Cases **83**
7.1 The SANSA framework 84
7.1.1 Architecture Overview 84
7.1.2 SANSA-Notebooks: Developer friendly access to SANSA 86
7.2 Leveraging Blockchain RDF Data Using the SANSA Framework 88
7.2.1 The Hubs and Authorities Transaction Network Analysis 89
7.2.2 Profiting From Kitties on Ethereum 92
7.3 Mining Big Data Applications Logs Using the SANSA Framework 93
7.4 Scalable Integration of Big POI Data Using the SANSA Framework 95
7.4.1 Proposed Solution: Architecture Overview 96
7.5 Summary 98

8 Conclusion and Future Directions **99**
8.1 Review of the Contributions 99
8.2 Limitations and Future Directions 102
8.3 Closing Remarks 104

Bibliography **105**

A SANSA Framework Release History **115**

B SPARQL Benchmark Queries **117**
B.1 LUBM SPARQL Queries 117
B.2 WatDiv SPARQL Queries 119

C List of Publications **123**

List of Figures 127

List of Tables 131

CHAPTER 1

Introduction

One of the key features of Big Data is its complexity. We can define complexity in different ways. It could be that data is coming from different sources, it could be the same data source representing different aspects of a resource, it could be different data sources representing the same property; this difference in representation, structure, or association makes it difficult to introduce common methodologies or algorithms to learn and predict from different types of data. The state of the art to handle this ambiguity and complexity of data is its representation or modeling using Semantic Web Technologies.

Semantic Web Technologies follows a set of standards for the integration of data and information in addition to searching and querying it. To create such data, the information represented in unstructured form or referring to other structured or semi-structured representation is mapped to the Resource Description Framework (RDF) data model. RDF has a very flexible data model comprised of triples (subject, predicate, object), that can be interpreted as a labelled directed graph (s, p, o) with s and o being arbitrary resources (vertices) and p being the property (edge from s to o) among these two resources. Thus, a set of RDF triples forms an inter-linkable graph whose flexibility allows to represent a large variety of highly to loosely structured datasets.

RDF, which was standardized by World Wide Web Consortium (W3C), is increasingly being adapted to model data in a variety of scenarios, partly due to the popularity of projects like linked open data and schema.org. This linked or semantically annotated data has grown steadily towards a massive scale[1].

Nevertheless, most existing solutions are limited to standalone environments only. In order to deal with the massive data being produced at scale, the existing big data frameworks like Apache Spark[2] and Apache Flink[3] offer fault-tolerant, high available and scalable approaches to process this data efficiently. These frameworks have matured over recent years and offer a proven and reliable method for processing of large scale unstructured data.

In the past few years, MapReduce based, and related frameworks for Big Data processing have been explored for distributed processing of RDF data as well. Some examples include the Spark-based S2RDF [1] which rewrites SPARQL Protocol And RDF Query Language (SPARQL) queries to SQL by using prior research by the RDB2RDF community and augments this approach by using

[1] http://lodstats.aksw.org/
[2] http://spark.apache.org/
[3] https://flink.apache.org/

precomputed semi-join tables. Approaches like SparkRDF [2], H2RDF [3] and H2RDF+ [4] use triple dataset statistics to find best merge-join orders for efficient querying. But, they are rather focused on one key element of the semantic stack, i.e. querying. Therefore, there is a need for a comprehensive framework that offers capabilities of exploring, validating and querying a large amount of RDF data at scale. The main motivations behind using distributed computing are being able to handle data that does not fit on a single machine and achieve a speed-up and scalability. Systems like *Apache Spark* employ the Bulk Synchronous Parallel (BSP) synchronization approach, i.e. each parallel iteration/task has to wait for a synchronization step - all *sub-tasks* must finish. This ensures correctness and fault tolerance. However many applications, i.e. ranking resources (as PageRank is for web pages) are usually iteratively convergent in nature and this synchronization barrier at the end of each iteration overshadows the speed-up gained by distributed computation. In this thesis, we aim to exploit the existing communication, synchronization and distribution techniques to optimize the performance of Distributed Processing of RDF Datasets when dealing with large amounts of data.

1.1 Problem Definition and Challenges

Processing large-scale RDF datasets is considered as one of the most challenging tasks in the Semantic Web [5]. The increase of the RDF data in a rapid manner brings multiple challenges when exploring and getting more insight from the data. More specifically, we face (i) a knowledge exploration problem, i.e. knowing the internal characteristics of the dataset. (ii) a data quality problem, i.e. which dataset is considered fit for use. (iii) a processing challenge, i.e. can we retrieve and manage RDF data when the size of the dataset increases.

In the following sections, we define the challenges that need to be addressed while designing a scalable and efficient processing framework for RDF datasets.

1.1.1 Challenge 1: Scalable Computation of RDF Dataset Statistics

The first challenge to overcome when dealing with large-scale RDF datasets is to have *a priori* statistical information about its internal structure and coverage. A significant fraction of RDF data available online[4] today are stored as Linked Open Data (LOD). Large RDF datasets, i.e. DBpeida [6] are often collaborative and contain data that has been extracted semi-automatically or has been ingested from different sources. Hence, such large-scale RDF datasets do not have an apriori view of the data or a strict unified view for structuring the instances. As a result, these processes derive noisily and, in a wrong case incomplete data [7]. In particular, for many applications such as data integration [8], semantic search [9], interlinking [10], or RDF data partitioning do not take full advantage of the data without knowing the internal structure of the data. In fact, there are already a number of tools, which offer such statistics, providing basic information about RDF vocabularies [11] and datasets [12, 13]. However, those efforts showed severe deficiencies in terms of performance when the dataset size goes beyond the main memory size of a single machine. Therefore, to produce type information about large-scale RDF datasets, we need a scalable computation of RDF dataset statistics that is able to deal with massive RDF datasets.

[4] https://lod-cloud.net/

1.1.2 Challenge 2: Quality Assessment of RDF Dataset at Scale

Apart from knowing the internals of a given dataset, deciding how quality and what information is considered "*fit for use*" is a challenge when the size of a dataset goes beyond the capacity of a single machine. Assessing the quality of RDF datasets is a crucial step to enhance the quality of the data being processed and published. The process of assessing the quality of the data should be efficient and made available in order to facilitate the difference when it comes to finding the right information that is fit for use. Some efforts have been made to provide a mechanism to assess the quality of the RDF datasets [14–17]. However, these methods can either be used on a small portion of large datasets [15] or narrow down to specific problems e.g., syntactic accuracy of literal values [16], or accessibility of resources [18]. Existing efforts show severe deficiencies in terms of performance when data grows beyond the capabilities of a single machine. This limits the applicability of existing solutions to medium-sized datasets only, in turn, paralyzing the role of applications in embracing the increasing volumes of the available datasets.

1.1.3 Challenge 3: Efficient and Scalable SPARQL Query Evaluation

More and more structured data is generated by an increasing number of organizations that are using RDF as a model for data representation. Therefore, analytics over such large-scale RDF datasets lead us to a completely new level of computational complexity. As a consequence, it becomes difficult to process such datasets using conventional approaches. Many standalone SPARQL query evaluators have been introduced in the past, nevertheless, as the volume of RDF data increases, these single-machine solutions encounter performance bottlenecks in terms of data processing, loading, and querying. For that reason, there is a need for a scalable and efficient framework that can handle large-scale RDF datasets. With that in mind, several approaches for distributed RDF data processing have been proposed, e.g [1, 19]. We want to investigate and study implementations of current SPARQL query evaluators through system-level characterizations and consequently propose our distributed approaches for RDF data processing.

1.2 Research Questions

As stated in the motivation section above and identified challenges, we define the main research question:

> *Is it possible to process large-scale RDF datasets efficiently and effectively?*

This research question then breaks down into three specific research questions.

Each challenge is mapped to specific research questions and altogether contribute to the overall research problem definition tackled throughout this thesis.

> **RQ1**: How can we efficiently explore the structure of large-scale RDF datasets?

To address this question, we evaluate existing solutions that deal with statistical information of RDF datasets. In particular, we investigate the definitions proposed on [20] and consider these 32 statistical criteria as a base for our system. As our main goal is to offer a scalable and efficient approach for the statistical computation of large RDF datasets, as an underlying engine, we consider one of the most prominent distributed frameworks, Apache Spark. Finally, we study the use of novel distributed data structure representations – known as Resilient Distributed Dataset (RDD) [21]. Within the scope of the thesis, we introduce a novel scalable approach for RDF dataset statistics computation. The results of the research question RQ1 allow us to address the defined challenge (cf. Section 1.1.1).

> **RQ2**: Can we scale RDF dataset quality assessment horizontally?

In order to answer this question, we investigate state-of-the-art quality assessment approaches with their metric definitions which can be used as a building block for quality measurements of RDF datasets. With the focus on the scalability, we derive quality metrics defined in [7] and propose a scalable and efficient quality assessment framework that compute different quality metrics for large RDF datasets. Results obtained during the research question RQ2 allow us to address the defined challenge (cf. Section 1.1.2).

> **RQ3**: Can distributed RDF datasets be queried efficiently and effectively?

With the objective of answering this research question, we investigate different data storage representation and SPARQL query evaluation and propose two different approaches for scalable and efficient RDF data processing. First, we introduce Sparklify: a scalable software component for efficient evaluation of SPARQL queries over distributed RDF datasets. It uses SPARQL-to-SQL rewriter techniques for translating SPARQL queries into Spark executable code. The second approach we investigated and developed with the scope of this thesis is a scalable approach to evaluate SPARQL queries over distributed RDF datasets using a semantic-based partitioning. It groups the facts based on the subject and its associated triples. Results obtained during the research question RQ3 allow us to address the defined challenge (cf. Section 1.1.3).

1.3 Thesis Overview

This section gives an overview of our main contributions conducted during this thesis and the research areas investigated. References to scientific publications covering this study and an overview of the thesis outline are also covered.

1.3.1 Contributions

Our contributions cover a spectrum of research areas in the scope of distributed RDF processing from the Scalable RDF Datasets, RDF Quality Assessment at Scale, and Scalable and Efficient SPARQL evaluators, as depicted in Figure 1.1.

1. *A Scalable Distributed Approach for Computation of RDF Dataset Statistics*.

   For a better view and type of information when dealing with large-scale RDF dataset we introduce DistLODStats, 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

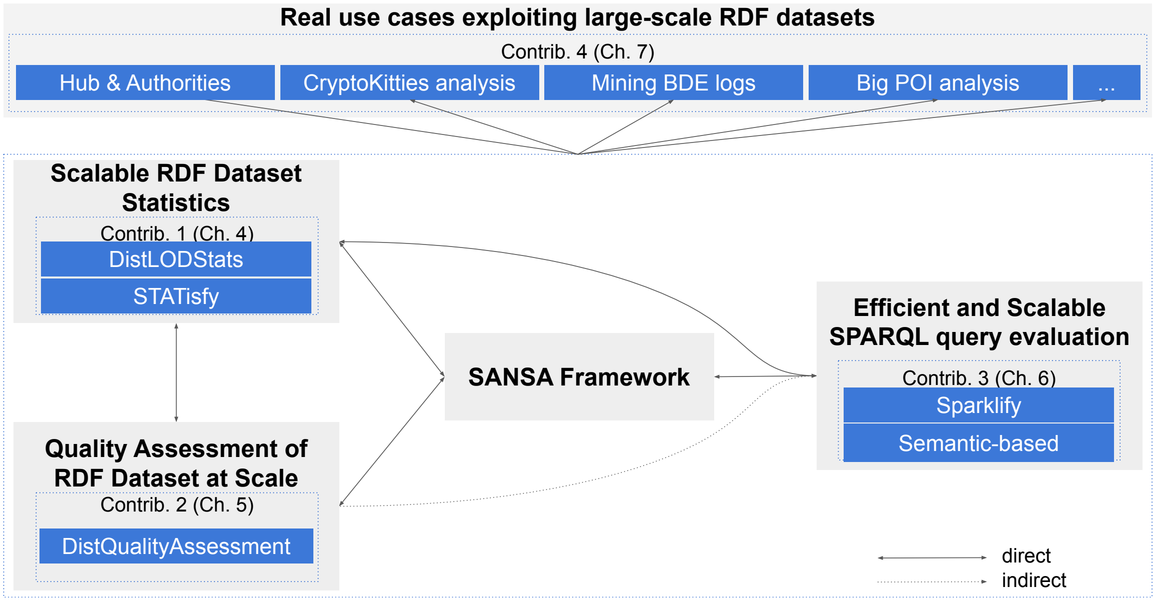


Figure 1.1: **Thesis Contributions**. Four are the main contributions of this thesis: (1) a scalable distributed approach for evaluation of RDF dataset statistics; (2) a scalable framework for quality assessment of RDF datasets; (3) a scalable framework for SPARQL evaluation of large RDF data; (4) a comprehensive, open-source RDF processing and analytics stack for distributed in-memory computing with the real use cases where the thesis results are applicable.

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. More details on this contribution are provided in Chapter 4, and publications [22, 23], which answer RQ1.

2. *A Scalable Framework for Quality Assessment of RDF Datasets*.

   Quality of the data is one of the key components when designing and performing RDF processing tasks. However, when dealing with large amounts of RDF data, it becomes a challenge processing and exploring such quantitative and qualitative information. There exist a few approaches for the quality assessment of RDF datasets, but their performance degrades with the increase in data size and quickly grows beyond the capabilities of a single machine. To address this, we present DistQualityAssessment – 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. The work presented here is integrated with the SANSA framework and has been applied to at least three use cases beyond the SANSA community. The results show that our approach is more generic, efficient, and scalable as compared to previously proposed approaches. See Chapter 5 for more information about this contribution, and publication [24]. The DistQualityAssessment approach contributes to answering the research question RQ2.

3. *A Scalable Framework for SPARQL Evaluation of Large RDF Data.*

   Over the last two decades, the amount of data that has been created, published and managed using Semantic Web standards and especially via RDF has been increasing. As a result, the efficient processing of such big RDF datasets has become challenging. Indeed, these processes require, both efficient storage strategies and query-processing engines, to be able to scale in terms of data size. In order to overcome this, we propose two different techniques that scale up to the cluster of machines. First, Sparklify: a scalable software component for efficient evaluation of SPARQL queries over distributed RDF datasets. It uses Sparqlify[5] as a SPARQL-to-SQL rewriter for translating SPARQL queries into Spark executable code. Our preliminary results demonstrate that Sparklify is more extensible, efficient, and scalable as compared to state-of-the-art approaches. Sparklify is integrated into a larger SANSA framework and it serves as a default query engine and has been used by at least three external use scenarios. The second approach we investigated and developed with the scope of this thesis is a scalable approach to evaluate SPARQL queries over distributed RDF datasets using a semantic-based partition and is implemented inside the state-of-the-art RDF processing framework: SANSA. An evaluation of the performance of a semantic-based approach in processing large-scale RDF datasets is also presented. The preliminary results of the conducted experiments show that it can scale horizontally and perform well as compared with the previous Hadoop-based system. It is also comparable with the in-memory SPARQL query evaluators when there is less shuffling involved. The Sparklify and semantic-based approaches contribute to answering the research question RQ3. A more detailed information on these contributions is given in Chapter 6, and publications [25–27].

4. *A comprehensive, open-source RDF processing and analytics stack for distributed in-memory computing.*

   We collaborated with many different stockholders and research projects during the development of this thesis in order to solve the real-world scalable knowledge analysis and RDF processing use cases. First, we mention here, Hub & Authorities and CryptoKitties analysis use cases. Alethio[6], is an advanced analytics platform making Ethereum more accessible and digestible for everyone. Their extensive data set contains large-scale blockchain transaction data modelled in RDF (currently encompassing more than 20B triples[7]) according to the structure of the Ethereum ontology [28]. As the blockchain is evolving, many users want to know more about the important players of the chain. With Hub & Authorities' use case, we investigate and analyze the Ethereum blockchain network in order to identify the major entities across the transaction network. By leveraging the rich data available through Alethio's platform in the form of RDF triples we learn about the Hubs and Authorities of the Ethereum transaction network. Alethio uses our approach for efficient reading and processing of such large-scale RDF data (transactions on Ethereum blockchain) in order to perform analytics e.g. finding top accounts, or typical behavior patterns of exchanges' deposit wallets and more. In another use case where Alethio is involved is the CryptoKitties analysis use case. CryptoKitties[8] is one of the first games to

[5] https://github.com/SmartDataAnalytics/Sparqlify
[6] https://aleth.io/
[7] https://linkeddata.aleth.io/
[8] https://www.cryptokitties.co/

be built on blockchain technology. Our solution empowers Alethio to read and query the data at scale for further analysis: game performance and customer behaviors. Within our solution, Alethio is able to get more insight from the CryptoKitties analyses, i.e the number of active users and the amount of spent Ether or correlation between indicators (e.g. to determine whether richer owners have the tendency to collect special/rare kitties which are more expensive). The second use case we were involved in was about mining BigDataEurope project logs. Big Data Europe (BDE)[9] [29] 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[10] 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 [30]. Finally, Big Points Of Interests (POI) analysis use case is developed. Among the various domains using large RDF graphs, applications often rely on geographical information which is often represented via POIs. In particular, one challenge is to extract patterns from POI sets to discover Areas of Interest (AOI)s. To tackle this challenge, a typical method is to aggregate various points according to specific distances (e.g. geographical) via clustering algorithms. In this study, we present a flexible architecture to design pipelines able to aggregate POIs from contextual to geographical dimensions in a single run. This solution allows any kind of clustering algorithm combinations to compute AOIs and is built on top of a Semantic Web stack which allows multiple-source querying and filtering through SPARQL. The architecture is embedded inside a state-of-the-art Semantic Web stack, SANSA, and then benefits from the advantages of it. The best practices, guidelines, easy to deploy and use in a lightweight allows us to quickly adapt the SANSA framework from the semantic web community and other fields of data science. Some of the use cases are described in Chapter 7, and publications [29–34].

1.3.2 List of Publications

In this thesis, part of the work is based on the following publications [22–27, 29–35]:

- *Conference Papers (peer reviewed)*
  1. **Gezim Sejdiu**; Anisa Rula; Jens Lehmann; and Hajira Jabeen, "A Scalable Framework for Quality Assessment of RDF Datasets," in Proceedings of 18th International Semantic Web Conference (ISWC), 2019. URL: `http://jens-lehmann.org/files/2019/iswc_dist_quality_assessment.pdf`
  2. Claus Stadler; **Gezim Sejdiu**; Damien Graux; and Jens Lehmann, "Sparklify: A Scalable Software Component for Efficient evaluation of SPARQL queries over distributed RDF datasets," in Proceedings of 18th International Semantic Web Conference (ISWC), 2019. URL: `http://jens-lehmann.org/files/2019/iswc_sparklify.pdf` This article is a joint work with Claus Stadler, a PhD student at the University of Leipzig. In this article, I devised the implementation of the conceptual architecture, helped on the implementation of the proposed approach, reviewed related work, and prepared of the experiments and analysis of the obtained results.

[9] https://github.com/big-data-europe

[10] https://github.com/big-data-europe/mu-swarm-logger-service

3. **Gezim Sejdiu**; Damien Graux; Imran Khan; Ioanna Lytra; Hajira Jabeen; and Jens Lehmann, "Towards A Scalable Semantic-based Distributed Approach for SPARQL query evaluation," 15th International Conference on Semantic Systems (SEMANTiCS), Research & Innovation , 2019. URL: https://gezimsejdiu.github.io/publications/semantic_based_query_paper_SEMANTICS2019.pdf

4. **Gezim Sejdiu**; Ivan Ermilov; Jens Lehmann; and Mohamed Nadjib-Mami, "DistLODStats: Distributed Computation of RDF Dataset Statistics," in Proceedings of 17th International Semantic Web Conference (ISWC), 2018. URL: http://jens-lehmann.org/files/2018/iswc_distlodstats.pdf

5. Jens Lehmann; **Gezim Sejdiu**; Lorenz Bühmann; Patrick Westphal; Claus Stadler; Ivan Ermilov; Simon Bin; Nilesh Chakraborty; Muhammad Saleem; Axel-Cyrille Ngomo Ngonga; and Hajira Jabeen, "Distributed Semantic Analytics using the SANSA Stack,"; in Proceedings of 16th International Semantic Web Conference - Resources Track (ISWC'2017), 2017. URL: http://svn.aksw.org/papers/2017/ISWC_SANSA_SoftwareFramework/public.pdf

6. Ivan Ermilov; Axel-Cyrille Ngonga Ngomo; Aad Versteden; Hajira Jabeen; **Gezim Sejdiu**; Giorgos Argyriou; Luigi Selmi; Jürgen Jakobitsch; and Jens Lehmann, "Managing Lifecycle of Big Data Applications,"; in KESW, 2017. URL: https://svn.aksw.org/papers/2017/KESW_BDE_Workflow/public.pdf This article is a joint work with Ivan Ermilov, a PhD student at the University of Leipzig. In this article, I helped with the implementation of the proposed approach and SC4 (Transport) use case, reviewed related work, and preparation of the experiments and analysis of the obtained results.

7. Sören Auer; Simon Scerri; Aad Versteden; Erika Pauwels; Angelos Charalambidis; Stasinos Konstantopoulos; Jens Lehmann; Hajira Jabeen; Ivan Ermilov; **Gezim Sejdiu**; Andreas Ikonomopoulos; Spyros Andronopoulos; Mandy Vlachogiannis; Charalambos Pappas; Athanasios Davettas; Iraklis A. Klampanos; Efstathios Grigoropoulos; Vangelis Karkaletsis; Victor Boer; Ronald Siebes; Mohamed Nadjib Mami; Sergio Albani; Michele Lazzarini; Paulo Nunes; Emanuele Angiuli; Nikiforos Pittaras; George Giannakopoulos; Giorgos Argyriou; George Stamoulis; George Papadakis; Manolis Koubarakis; Pythagoras Karampiperis; Axel-Cyrille Ngonga Ngomo; and Maria-Esther Vidal, "The BigDataEurope Platform – Supporting the Variety Dimension of Big Data," in 17th International Conference on Web Engineering (ICWE2017), 2017. URL: http://jens-lehmann.org/files/2017/icwe_bde.pdf This article is a joint work with the BDE consortium. In this article, I contributed within the semantic layer, more specifically; bringing the Big Data Analytics for RDF into the BDE platform and co-contributing into dockerizing BDE components.

- *Demo & Poster Papers (peer reviewed)*

8. Claus Stadler; **Gezim Sejdiu**; Damien Graux; and Jens Lehmann. "Querying large-scale RDF datasets using the SANSA framework". In Proceedings of 18th International Semantic Web Conference (ISWC), Poster & Demos, 2019. URL: https://gezimsejdiu.github.io/publications/sansa-sparklify-ISWC-demo.pdf This demonstration article is a joint work with Claus Stadler, a PhD student at the University of Leipzig. In this article, I helped in describing the architecture and implementation of the running example.

9. Danning Sui; **Gezim Sejdiu**; Damien Graux; and Jens Lehmann. "The Hubs and Authorities Transaction NetworkAnalysis using the SANSA framework". In 15th International Conference on Semantic Systems (SEMANTiCS), Poster & Demos, 2019. URL: `http://tiny.cc/4ukxcz`
10. Rajjat Dadwal; Damien Graux; **Gezim Sejdiu**; Hajira Jabeen; and Jens Lehmann. "Clustering Pipelines of large RDF POI Data" in Proceedings of 16th Extended Semantic Web Conference (ESWC), Poster & Demos, 2019. URL: `https://gezimsejdiu.github.io/publications/piping-clustering-eswc19-poster.pdf`
11. **Gezim Sejdiu**; Ivan Ermilov; Jens Lehmann; and Mohamed-Nadjib Mami, "STATisfy Me: What are my Stats?," in Proceedings of 17th International Semantic Web Conference (ISWC), Poster & Demos, 2018. URL: `http://jens-lehmann.org/files/2018/iswc_statisfy_pd.pdf`
12. Damien Graux; **Gezim Sejdiu**; Hajira Jabeen; Jens Lehmann; Danning Sui; Dominik Muhs; and Johannes Pfeffer, "Profiting from Kitties on Ethereum: Leveraging Blockchain RDF with SANSA," in 14th International Conference on Semantic Systems, Poster & Demos, 2018. URL: `http://jens-lehmann.org/files/2018/semantics_ethereum_pd.pdf`
13. Ivan Ermilov; Jens Lehmann; **Gezim Sejdiu**; Lorenz Bühmann; Patrick Westphal; Claus Stadler; Simon Bin; Nilesh Chakraborty; Henning Petzka; Muhammad Saleem; Axel-Cyrille Ngomo Ngonga; and Hajira Jabeen, "The Tale of Sansa Spark," in Proceedings of 16th International Semantic Web Conference, Poster & Demos, 2017 (**Best Demo Award**). URL: `http://jens-lehmann.org/files/2017/iswc_pd_sansa.pdf` This demonstration article is joint work with Ivan Ermilov, a PhD student at the University of Leipzig. In this article, I helped in describing the architecture, implementation of the examples and demonstration of the prototype.

Appendix C contains the complete list of publications finished during the PhD studies.

1.4 Thesis Outline

The thesis consists of eight chapters. Chapter 1 introduces the thesis starting with the main research problem and challenges, motivation, research questions, scientific contributions addressing research questions, and a list of published scientific papers describing these contributions. Chapter 2 presents basic concepts and background about Semantic Web technologies and the Hadoop Ecosystem for a comprehensive overview of the research problem. Chapter 3 describes state-of-the-art efforts in the field of processing RDF datasets w.r.t research problem. We provide an overview of existing RDF dataset statistics systems, quality assessment systems, and SPARQL query evaluators in order to provide a thorough knowledge of their limitations, and the identified gaps we cover in this thesis. In Chapter 4 we introduce a scalable approach for the statistical calculation of large RDF datasets, which scales out to a cluster of machines. More specifically, we describe the first distributed in-memory approach for computing 32 different statistical criteria for RDF dataset using the Apache Spark framework. Chapter 5 introduces a scalable approach for quality assessment of RDF datasets. The presented approach offers generic features to solve common data quality checks. As a consequence, this can enable further applications to build trusted data utilities. We have demonstrated empirically that our approach improves upon the previous centralized approach that we have compared against. We also provide a quality assessment pattern that can be used to generate new scalable metrics that can be applied to big data. Chapter 6 proposes two storage strategies and query engine implementations for efficient and scalable querying and processing RDF datasets. First, Sparklify: a scalable software component for efficient evaluation of SPARQL queries over distributed RDF datasets. It uses a SPARQL-to-SQL rewriter technique for translating SPARQL queries into Spark executable code. The second approach we investigated and developed with the scope of this thesis is a scalable approach to evaluate SPARQL queries over distributed RDF datasets using a semantic-based partition. Moreover, in this chapter, we present the evaluations of our implementations as compared with state of the art SPARQL query evaluators. Chapter 7 presents real world use-cases powered by our solutions. More specifically, we show the usage of SANSA in general and the solutions proposed during this work , and, consequently, validate the solutions proposed for the problems RQ1, RQ2, and RQ3. Chapter 8 conclude the thesis with an overall overview of the contributions made during this research work and a discussion on the future work based on the limitations of the actual solutions.

CHAPTER 2

Preliminaries

This chapter covers the foundation technologies used throughout the thesis. First, Section 2.1 gives an overview of Semantic Technologies, i.e RDF model as a standard model for representing the data and its accompanying query language SPARQL. It also covers different RDF serialization formats. Later, Section 2.2 gives an introduction to *Hadoop*, its core technologies *Hadoop Distributed File-System (HDFS)*, *MapReduce* and *Apache Spark* with its libraries that have been used in the course of this thesis.

2.1 Semantic Technologies

Originally web was considered to be a hub for sharing web pages or documents that could be understood by humans. In addition, interlinking with other web pages or records could also be generated anywhere on the web. Most of this data was intended solely for human consumption. Machines could process and show such information but did not understand it.

Semantic Web [36], introduced by Tim Berners-Lee is an attempt to describe and link the web content into more meaningful to the machines. The main idea is to extend the existing web considered as "Web of Documents" towards "Web of Things" a.k.a Semantic Web where things are connected and able to be exchanged with each other in an understandable way. Semantic Web tries to give meaning to the data and thus turn the current web of documents into a more global and decentralized knowledge which is understandable and suitable for machines besides exclusively designed for human consummation. Therefore, Semantic Web can be seen as an extension of the classical World Wide Web (WWW). The Semantic Web vision is to build community-driven technologies and tools (known as standards) which allows data to be shared and reused. As a consequence, the W3C consortium was built and is mainly in charge of leading such standards.

Figure 2.1 depict various layers of Semantic Web related technologies. Here we focus only on *Data interchange: RDF* and *Query: SPARQL* layers, which are relevant to the work presented in this thesis and are therefore discussed in this chapter.

Semantic Web's core technology is the so-called Resource Description Framework (RDF) which serves as the main data representation. It represents information about resources. A resource is identified with a globally unique identifier (Unique Resource Identifiers (URI)s). The RDF data model can be interpreted as a directed labeled graph where resources identified by URI are nodes in the graph and edges represent the relationships between resources labeled with the type of relationship

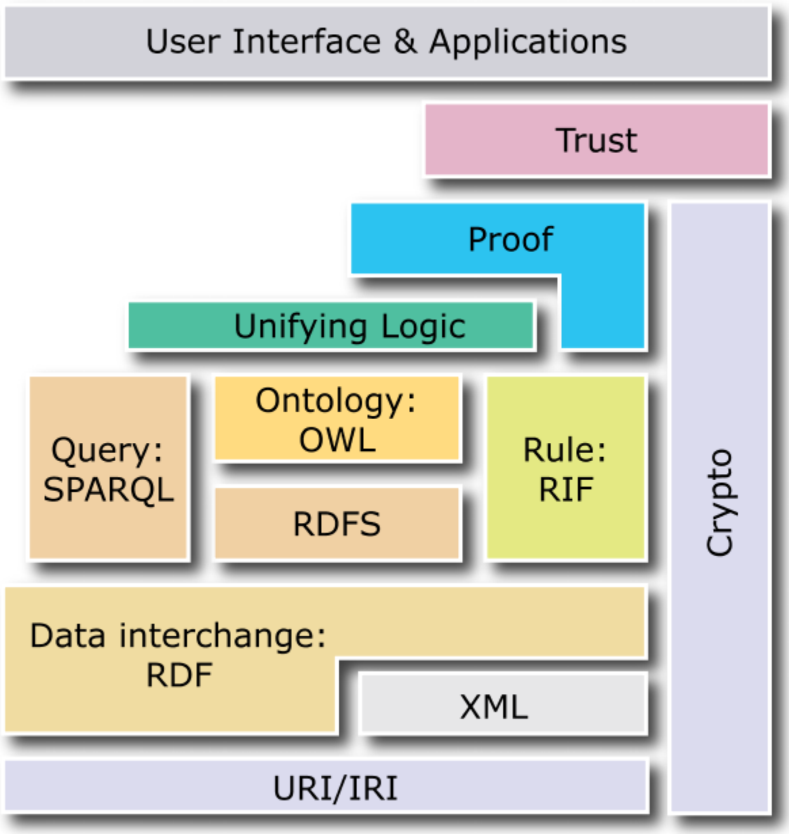


Figure 2.1: **Semantic Web Stack**[2]. The Semantic Web Stack, also known as Semantic Web Cake or Semantic Web Layer Cake, illustrates the architecture of the Semantic Web, according to W3C.

known as predicates, also identified by URIs.

SPARQL is the W3C standard for querying RDF data. It uses a graph pattern mechanism to be matched against an RDF graph and its syntax is similar to SQL.

More details about RDF (cf. Section 2.1.1) and SPARQL (cf. Section 2.1.2) is given in the following sections.

2.1.1 RDF Data

The Resource Description Framework (RDF) [37] is a W3C standard for describing resources. A resource is a fact or a thing that can be described and identified. A person, a home page, this thesis is a resource. An RDF resource is identified by a *URI* reference, while literals are used to represent a respective data values. Literals consist of either a string and its language tag or value and its data type.

An RDF graph is a set of RDF triples (s, p, o) where s is called the *subject*, p is the *predicate* and o is the *object*, each of which can be an URI, subjects and objects can alternatively be blank nodes and objects can also represent literal data values. It can be also seen as a directed graph containing of vertices and edges. A vertex represents subjects and objects and an edge represents predicates.

[2] https://www.w3.org/2007/03/layerCake.png

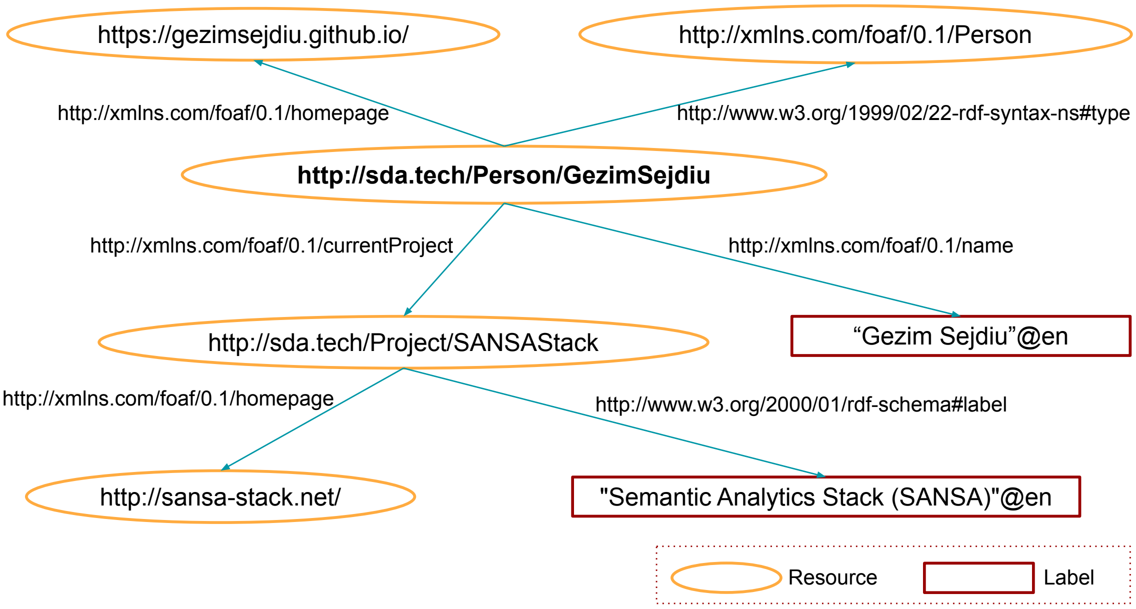


Figure 2.2: **Sample RDF Graph representation**. Small knowledge base about 'Gezim Sejdiu' represented as a graph.

Figure 2.2 represent an RDF graph sample about `"Gezim Sejdiu"` as a resource. One of the RDF statements (triples) from the Figure 2.2 is:

```
<http://sda.tech/Person/GezimSejdiu> <http://xmlns.com/foaf/0.1/currentProject>
    <http://sda.tech/Project/SANSAStack> .
```

which simply states "The subject identified by `<http://sda.tech/Person/GezimSejdiu>` has a property identified by `<http://xmlns.com/foaf/0.1/currentProject>` whose value is equal to `<http://sda.tech/Project/SANSAStack>`". In a more natural statement representation, it means that a person "Gezim Sejdiu" has a "current-project" which is "SANSA-Stack".

Below we give some necessary notions about RDF.

Definition 2.1.1 (RDF Term) *Let $\mathcal{U}$, be a set of URIs, $\mathcal{B}$ set of blank nodes and $\mathcal{L}$ set of literals, an RDF term ($\mathcal{T}$) is a set of $\mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$.*

Definition 2.1.2 (RDF Triple) *Let $\mathcal{U}$, be a set of URIs, $\mathcal{B}$ set of blank nodes and $\mathcal{L}$ set of literals, an RDF triple is a ternary tuple in the form of $(s, p, o) \in (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (U \cup B \cup L)$, where the subject $s \in (U \cup B)$ is a resource, the predicate $p \in U$ is a property, and the object $o \in (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$ is either another resource $(\mathcal{U} \cup \mathcal{B})$ or a literal $(\mathcal{L})$.*

Definition 2.1.3 (RDF Graph) *An RDF Graph ($\mathcal{G} = \{t_1, t_2, \ldots, t_n\}$) is defined as a finite set of RDF triples t_i.*

Definition 2.1.4 (RDF Dataset) *An RDF dataset is a collection of RDF graphs*

$$D = \{G_0, \langle u_1, G_1 \rangle, \ldots, \langle u_n, G_n \rangle\}$$

where $u_1, \ldots, u_n \in \mathcal{U}$. G_0 is considered as a default graph that does not have a name and can be empty, whereas $\langle u_i, G_i \rangle$ are called named graphs.

RDF Serialization Formats

As described in Section 2.1.1, RDF is modeled as a graph where the triple notation is used mostly for such representation. In this section, we will cover some of the most common RDF serialization/syntax formats. We focus primarily on those used during this work.

N-Triples The N-Triples [38] RDF serialization format is a plain-text, line-based syntax for an RDF graph. Each triple is written into a single line. As a consequence, each element of the triple (*subject*, *predicate*, and *object*) is represented without any abbreviation i.e. prefixes. These elements then are separated with white space (spaces or tabs) and this sequence ends with a dot '`.`' and a new line (optional at the end of a file).

```
<http://sda.tech/Person/GezimSejdiu> <http://www.w3.org/1999/02/22-rdf-syntax-
    ns#type> <http://xmlns.com/foaf/0.1/Person> .
<http://sda.tech/Person/GezimSejdiu> <http://xmlns.com/foaf/0.1/name> "Gezim
    Sejdiu"@en .
<http://sda.tech/Person/GezimSejdiu> <http://xmlns.com/foaf/0.1/homepage> <
    https://gezimsejdiu.github.io/> .
<http://sda.tech/Person/GezimSejdiu> <http://xmlns.com/foaf/0.1/currentProject>
     <http://sda.tech/Project/SANSAStack> .
<http://sda.tech/Project/SANSAStack> <http://www.w3.org/1999/02/22-rdf-syntax-
    ns#type> <http://xmlns.com/foaf/0.1/Project> .
<http://sda.tech/Project/SANSAStack> <http://www.w3.org/2000/01/rdf-schema#
    label> "Semantic Analytics Stack (SANSA)"@en .
<http://sda.tech/Project/SANSAStack> <http://xmlns.com/foaf/0.1/homepage> <http
    ://sansa-stack.net> .
```

Listing 2.1: **N-Triples syntax example**. Representation of the example in Figure 2.2 using the N-Triples syntax.

Listing 2.1 is an N-Triples representation of the example depicted in Figure 2.2.

As we see from the N-Triples basic example above, URIs are written between angle brackets i.e. '<' and '>'. Literals are enclosed by double-quotes. Sometimes, literals include language tags using a '`@`' symbol and if typed, with '`^^`'. Blank nodes are identified by '`_:`'.

Turtle The Turtle [39] syntax is basically a textual syntax for an RDF. It is a more compact and natural form to write an RDF graph as compared i.e. N-Triples syntax. Turtle can be seen as an extension of the N-Triples representation, with abbreviations for common usage patterns and datatypes. Triples written in Turtle are a sequence of *subject*, *predicate* and *object* separated by a white space (spaces or tabs) this sequence ends with a dot '`.`' like in N-Triples.

With the Turtle syntax, RDF statements can be written in a more compact way as compared to N-Triples. Often, triples are grouped (1) if several *predicates* share the common *subject*, and (2) if the same tuple (*subject*, *predicate*) have multiple object values. The following example depicts an RDF graph represented in Turtle syntax.

```
@base           <http://sda.tech/> .
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
```

```
@prefix rdfs:       <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf:       <http://xmlns.com/foaf/0.1/> .
@prefix sdaperson:  <http://sda.tech/Person/> .
@prefix sdaproject: <http://sda.tech/Project/> .

sdaperson:GezimSejdiu a foaf:Person ;
                  foaf:name "Gezim Sejdiu"@en ;
                  foaf:homepage <https://gezimsejdiu.github.io/> ;
                  foaf:currentProject sdaproject:SANSAStack .

sdaproject:SANSAStack a foaf:Project ;
                rdfs:label "Semantic Analytics Stack (SANSA)"@en ;
                foaf:homepage <http://sansa-stack.net> .
```

Listing 2.2: **Turtle syntax example**. Representation of the example in Figure 2.2 using the Turtle syntax.

Listing 2.2 represent the example depicted in Figure 2.2 in the Turtle syntax. This example introduces some of the features of the Turtle language: prefixes defined by the '@' symbol, predicated lists separated by ';', and literals. The object lists are separated by ',' in case they share the same tuple (*subject*, *predicate*).

RDF/XML The RDF/XML [40] is an Extensible Markup Language (XML) representation of an RDF graph. It is considered a normative syntax and the RDF graph is encoded using XML terms – element names, attribute names, element contents and attribute values. It exploits a hierarchical structure for the representation of an RDF graph. An RDF graph using the RDF/XML representation is considered as a collection of paths (in the hierarchical structure) of the form $node \longrightarrow predicate\ arc \longrightarrow node \longrightarrow predicate\ arc \longrightarrow node \longrightarrow predicate\ arc, \cdots \longrightarrow node$ which cover the entire graph. These paths then become a sequence of elements within elements that alternate node elements with arcs predicates.

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
        xmlns:foaf="http://xmlns.com/foaf/0.1/"
        xmlns:sdaperson="http://sda.tech/Person/"
        xmlns:sdaproject="http://sda.tech/Project/">

<rdf:Description rdf:about="http://sda.tech/Person/GezimSejdiu">
        <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
        <foaf:name xml:lang="en">Gezim Sejdiu</foaf:name>
        <foaf:homepage rdf:resource="https://gezimsejdiu.github.io/"/>
        <foaf:currentProject rdf:resource="http://sda.tech/Project/SANSAStack
            "/>
</rdf:Description>

<rdf:Description rdf:about="http://sda.tech/Project/SANSAStack">
        <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Project"/>
```

```
        <rdfs:label xml:lang="en">Semantic Analytics Stack (SANSA)</rdfs:label>
        <foaf:homepage rdf:resource="http://sansa-stack.net"/>
</rdf:Description>

</rdf:RDF>
```

Listing 2.3: **RDF/XML syntax example**. Representation of the example in Figure 2.2 using the RDF/XML syntax.

Listing 2.3 represent an RDF/XML syntax of the example in Figure 2.2. The `rdf:RDF` node is considered as a root node of an RDF/XML document. RDF triples are grouped according to their subject and encoded using the XML elements. The `rdf:Description` is the node element and is used to describe subjects and objects of the RDF graph. The `rdf:about` attribute is used for the unique identifier of a resource representation, whereas the literal values are encoded using the separate tags (e.g. `rdfs:label`, `foaf:name`). The property elements (predicates) can either be encoded using the XML attributes or as a separate resources i.e using the `rdf:resource` element.

2.1.2 SPARQL

An RDF graph is considered being a directed, labeled graph data format that represents information on the Web. SPARQL [41] is a W3C standard query language for retrieving and manipulating RDF data. Its core component is the graph pattern mechanism which allows users to write queries in the form of triple patterns, conjunctions, disjunctions and/or a set of optional patterns (e.g. `FILTER`) which are matched against an RDF graph. This is done by replacing the variables in the triple pattern with elements of the RDF graph such that the resulting graph is contained in the original RDF graph, known as pattern matching. The results of SPARQL queries are a set of binding or an RDF graph.

In the following, we cover the foundation of SPARQL and its syntax as an analog to the definitions in [42]. More details can also be found in the W3C specification of SPARQL [41].

Definition 2.1.5 (Triple Pattern) *Let $\mathcal{V}$ be a set of variables such that $\mathcal{V} \cap \mathcal{T} = \emptyset$. A triple pattern tp is member of the set $(\mathcal{T} \cup \mathcal{V}) \times (\mathcal{U} \cap \mathcal{V}) \times (\mathcal{T} \cup \mathcal{V})$.*

Definition 2.1.6 (Query Variable) *A query variable is a member of the set $\mathcal{V}$ where $\mathcal{V}$ is considered infinite and disjoint from $\mathcal{T}$.*

Definition 2.1.7 (Basic Graph Pattern (BGP)) *Let $tp = \{tp_1, tp_2, tp_3, \ldots, tp_n\}$ be a set of triple patterns. A Basic Graph Pattern BGP is a conjunction of triple patterns, i.e $BGP = tp_1 \wedge tp_2 \wedge tp_3 \wedge, \ldots, \wedge tp_n$.*

Definition 2.1.8 (Solution Modifiers) *A solution modifier is a mapping from a set of $\mathcal{V}$ to a set of $\mathcal{T}$. More formally, $SM = \{(v, modifier(v)) | v \in \mathcal{V}$, where $modifer$ is one of the* `project`, `distinct`, `order`, `limit`, *and* `offset` *modifiers.*

Definition 2.1.9 (Result Set) *Given $Q = (BGP, D, SM, SELECT\ \mathcal{V})$, then a result set QS is a solution formed by matching dataset D with graph pattern BGP.*

Definition 2.1.10 (SPARQL Query) *A SPARQL query is a tuple (BGP, D, SM, QS).*

Let us consider an example for a better understanding of SPARQL. Assume that we want to know "*What is the project (and its homepage) that Gezim Sejdiu is currently working on?*" from our small knowledge base (as depicted in Figure 2.2). Listing 2.4 depicts a simple SPARQL query to retrieve information about the project and its homepage of Gezim Sejdiu's current project.

```
1  PREFIX sda:       <http://sda.tech/>
2  PREFIX sdaperson: <http://sda.tech/Person/>
3  PREFIX foaf:      <http://xmlns.com/foaf/0.1/>
4
5  SELECT ?project ?homepage
6  WHERE {
7       sdaperson:GezimSejdiu foaf:currentProject ?project.
8       ?project foaf:homepage ?homepage.
9  }
```

Listing 2.4: **A SPARQL query example**. A SPARQL query to retrieve the project name and its homepage of Gezim Sejdiu's current project (as depicted in Figure 2.2).

We see that (from Listing 2.4) SPARQL query has a similar SQL-like syntax. Mainly a SPARQL query contains four parts. First, prefixes as optional headers are given. It helps the reader to make the rest of the query more readable. Second, the query form is defined. In our case, we use `SELECT` query form. Then, the `WHERE` clause is used which is the main definition of the SPARQL query. It involves a set of conditions/patterns as a composition of the result set. Finally, optional solution modifiers are set in order to adjust the selection before retrieving the results.

More specifically, in Listing 2.4, lines 1-3 define prefixes as a shortness version of URIs. The upcoming statement (line 5) is the `SELECT` clause which declares the variables that should be retrieved as an output when executing the query. There are two variables `?project` and `?homepage`. We see that variables are defined with a `?` symbol. The following statements (lines 7-8) include two Basic Graph Pattern (BGP)s. The first one (line 7) states that the statement with subject `sdaperson:GezimSejdiu` and property `foaf:currentProject`, we assign the value of its object to a variable called `?project`. When evaluated, this variable will contain the value of `sdaproject:SANSAStack`. Afterwords (line 8), the same variable `?project` with an associated value will be the subject of the next statement. That is, the statement will be `sdaproject:SANSAStack foaf:homepage ?homepage`. The remaining variable `?homepage` then will take the value `http://sansa-stack.net`. As an output, both values of the variables `?project` and `?homepage` will be rendered.

2.2 Hadoop Ecosystem

Apache Hadoop [43] is a collection of distributed processing and storage frameworks of large-scale datasets across a cluster of computers. Its ecosystem contains build-in mechanisms in order to guarantee fault tolerance and high availability on top of commodity hardware. Therefore, specific hardware involvement is not needed, making it highly scalable and cost-effective.

As of today, the Hadoop ecosystem has been enriched with extensive tools and libraries that are either built on top of Hadoop or use it for different application fields, including but not limited to: data mining, querying, data analysis, processing, and data warehousing. It has become the de-facto industry standard in Big Data management all thanks to its high degree of parallelism, fault-tolerant, reliability, and scalability.

In this section, we provide a brief overview of the Hadoop ecosystem projects used in the course of this thesis. We focus mostly on the aspects needed to understand the content of the following chapters without going into the technical details.

2.2.1 Apache Hadoop and MapReduce

HDFS

The Hadoop Distributed File System (HDFS) [44] is one of the main components of the Hadoop. It is a popular file system capable of handling the distribution of the data across multiple nodes in the cluster. HDFS serve as a common, distributed and fault-tolerant data pool for all applications on top of the Hadoop in order to minimize the data movement and duplication. Furthermore, it also leverages the distributed processing of large-scale datasets by adopting advanced and automatically partitioning techniques across all the nodes in the cluster. HDFS was originally built as infrastructure for the Apache Nutch[3] web search engine project and was inspired by the Google File System (GFS) [45]. HDFS is an integral component of the Apache Hadoop ecosystem.

HDFS is designed in a way that it doesn't require highly reliable and costly hardware but instead, it can be run on a cluster of computers with commodity hardware. HDFS splits data (files) into blocks that can be replicated across the cluster in order to ensure fault-tolerance and efficiency.

The HDFS architecture follows the master/slave model. The *namenode* (master) is responsible for managing a file system namespace or a directory structure, coordinating the replication process, keep track and maintain metadata about the replicated blocks. The *datanodes* (slaves) are the machines where these blocks are physically stored. A *datanode* instance allows access for storing and retrieving the data. To increase the availability, the *namenode* maintains multiple copies of the metadata of the replicated blocks. In the earlier version of HDFS, the *namenode* was considered being a single point of failure. The latest versions support deploying two instances configured being *namenode* with the mechanism active/passive for high availability. An active *namenode* is the *namenode* which is running in the cluster, and a passive *namenode* is kept synchronized and stand by. In case of a failure, the passive *namenode* can replace the active *namenode*. Hence, the cluster can be recovered faster and it never fails.

MapReduce

Besides its distributed file system, Hadoop contains computing system so-called MapReduce [46]. MapReduce is a distributed framework that allows for the distributed processing of large data sets across a cluster of computers. It enables scalable, fault-tolerant and massively parallel computations over a cluster of machines. The core of MapReduce is a distributed file system GFS which split larger size of files into equal-sized blocks of records across the cluster.

The workflow of a MapReduce job is a sequence of *map* and *reduce* phases performed in an iterative way. These phases contains a *shuffle* and *sort* operations (as depicted in Figure 2.3) as an intermediate phase. Usually, the input data is split into distributed blocks across the cluster for parallel execution. Topically, a program should contain the *map* and *reduce* operations which are then evaluated in a

[3] https://nutch.apache.org/

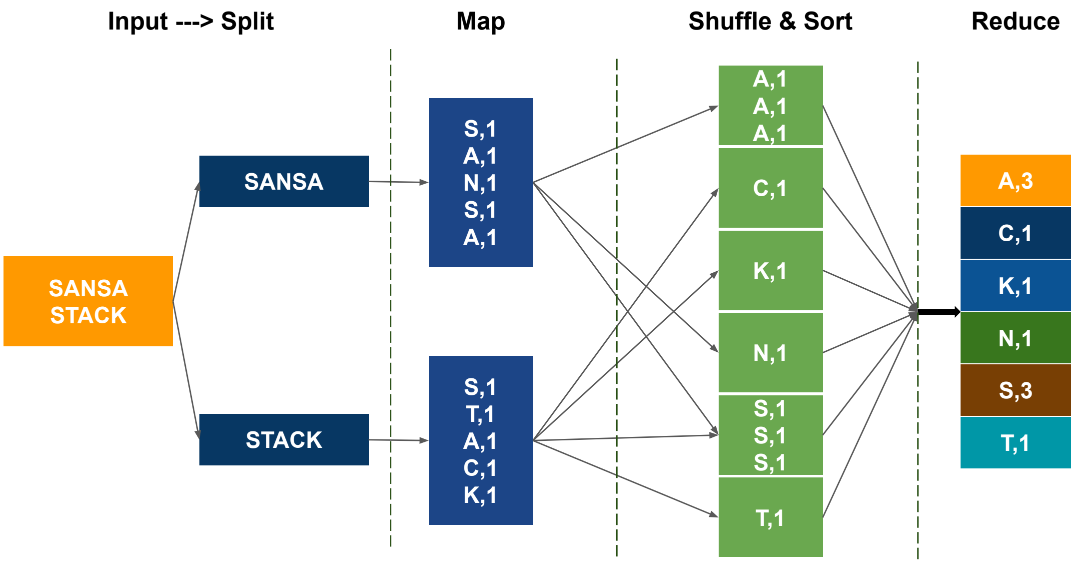


Figure 2.3: **MapReduce dataflow**. A MapReduce dataflow illustrated with the "Character Count" example.

parallel setting on a partition of the data. During the *map* phase, every record of the input dataset as key/value pairs are read and another set of intermediate key/value pairs is generated. Later, during the *reduce* phase these key/value pairs are ingested and evaluated in order to return a single set of results. While performing such map/reduce phases, intermediate results are generated and need to be shuffled and/or sorted across the cluster.

The user has to implement the *map* and *reduce* functions with a signature as follows:

```
map:    <k1, v1> --> Map() --> list(<k2, v2>)
reduce: <k2, list(v2)> --> Reduce() --> list(<k3, v3>)
```

Figure 2.3 illustrates an example of the MapReduce dataflow. With such an example, the user wants to count the number of occurrences for all characters in the dataset. First, the input is split into small subsets of the dataset, e.g. a line of text. The map function splits the input into a set of characters and outputs the key/value pairs, i.e. *(character, 1)* for every character occurrence. Later, the shuffle & sort phase is used to combine and partition all the pairs with the same key (character) to the same reducer and thus the reduce function is executed with a list of values for a single character. Finally, the sum of all these values is returned.

2.2.2 Apache Spark

Apache Spark[4] is a fast and generic-purpose cluster computing engine that is built over the Hadoop ecosystem. It started as a research project in 2009 within the AMPLab[5] at the University of California,

[4] http://spark.apache.org/
[5] https://amplab.cs.berkeley.edu/

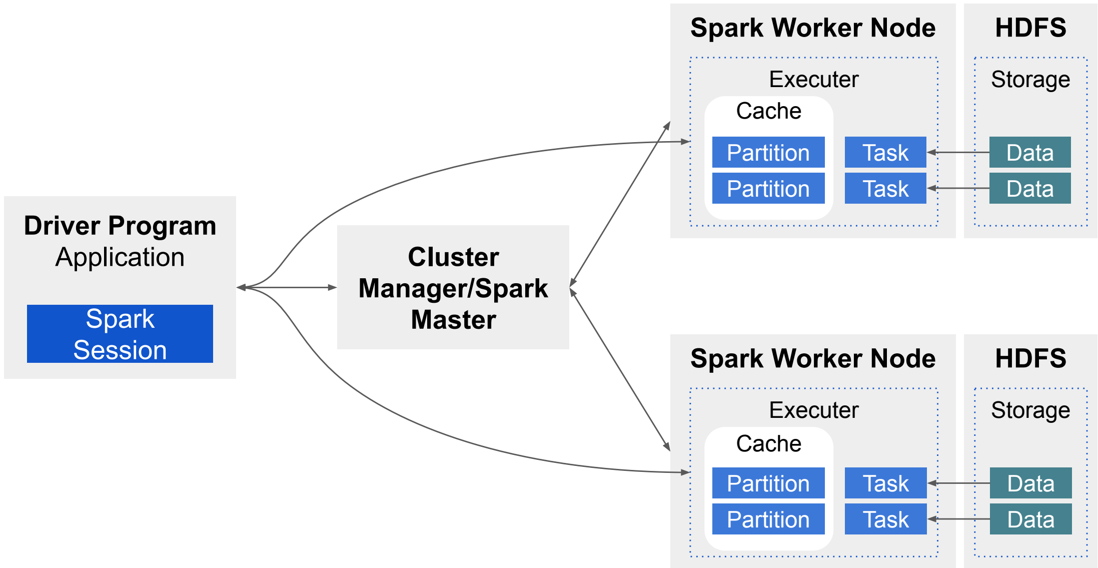


Figure 2.4: **Spark Architecture Diagram**. A Spark Cluster Mode Overview.

Berkeley. The main goal of the project was to keep the benefits of MapReduce's scalable, distributed, and fault-tolerant processing framework while making it more efficient and much easier to use.

Apache Spark follows a master/slave architecture, i.e. one central coordinator and many distributed workers. A Spark cluster contains a single master, a cluster manager and any number of workers (slaves). Figure 2.4 depict a cluster mode overview architecture of Spark. Spark applications can run as an independent set of processes on the cluster, coordinated by the so-called *SparkSession* in the *driver program*. More specifically, a Spark session connects to a cluster manager (e.g. Spark's own standalone cluster manager), which allocates resources across applications when running on the cluster. Once connected to the cluster manager, it acquires executors on the worker nodes, which are processes that run computations and store data for the submitted application. Afterword, it sends the application code to the executors. Hence, the tasks are triggered to run on those executors, one task per partition. Such a task applies its workload to a dataset in its partition and outputs a new partition dataset. Some of the tasks performed on Spark may involve iterative operations where operations are run repeatedly to data, they benefit from caching datasets across iterations. Finally, the results are sent back to the driver application. They can be kept in-memory for further processing or pushed back and saved to disk.

The main data structures that Spark operates with are so-called RDD [21] which are fault-tolerant and immutable collections of records that can be operated in a parallel setting. RDDs are considered being *resilient* – fault-tolerant and capable of rebuilding data on failure, *distributed* – able to distribute the data among the multiple nodes in the cluster, and *dataset* – collection of partitioned data with their values. An RDD splits the data into chunks based on a key. They are considered being highly resilient i.e being able to recover quickly from any failure as the same data chunks are replicated across multiple executor nodes in the cluster. Thus, even a node failure occurs, the other nodes will still process the data. Moreover, an RDD, once created becomes immutable – not able to be modified after it is created. RDD follow the concept of transformation and are considered being lazy evaluation.

In a distributed setting, each dataset in RDD is split into logical partitions which are computed on different nodes in the cluster. This allows us to perform any transformation or action on the whole dataset in a parallel manner. The distribution of the workload is taken care of by Spark. Such an RDD can be created using an existing collection of the data or by loading a dataset from an external storage system, such as HDFS, or even a file system. With RDDs, we can perform two types of operations: (i) *Transformations* – operations which are applied when creating an RDD or transforming it to another one, and (ii) *Actions* – operations applied on an RDD and retrieve the result.

Apache Spark provides a rich set of Application Programming Interface (API)s for faster, in-memory processing of RDDs. It also provides a rich functional programming model and comes with higher level libraries, e.g. for structured querying (*Spark SQL*), machine learning (*MLlib*), streaming (*Spark Streaming*), and graph parallel processing (*GraphX*).

In the following sections, we will cover those libraries we make use of.

GraphX

GraphX [47] is a Spark library for graphs and parallel graph computation. It extends the RDD abstraction and thus introduces Resilient Distributed Graph (RDG), which relates records with vertices (VertexRDD) and edges (EdgeRDD) in a graph and provides an expressive set of computational primitives. In addition, GraphX simplifies the conventional Extract, Transform, Load (ETL) processes and analysis significantly by providing new operations for viewing, filtering, and transforming graphs.

The GraphX RDG leverages advances in distributed graph representation by combining the best of both worlds; benefits of graph-parallel and data-parallel systems. It exploits the graph structure in order to minimize network communication and storage overhead.

It uses the so-called efficient vertex-cut partitioning strategy (as described in [48]) and data-parallel partitioning heuristics by assigning edges to machines and allowing vertices to span multiple machines in order to minimize the vertex span per machine. By adding abstraction to the core of Spark (RDDs) it eases the usage of graph data. GraphX contains a set of common graph operations, i.e *filter*, *map*, *reduceByKey*, *join*, etc. By using such graph-parallel and data-parallel operations, GraphX performs its computation. Usually, these operators take graphs and collections as input and produce new graphs and collections as an output.

SparkSQL

Spark SQL [49] is a Spark library for SQL and structured data processing which allows querying structured data inside Spark programs. Essentially, the main abstraction in Spark SQL's API is a *DataFrame* which are distributed collections of rows with a homogeneous schema. A *DataFrame* is an RDD with a schema. They can be seen as tables in a relational database and can also be manipulated in a similar way to RDDs. They are represented using a columnar storage format (while kept in-memory caching) which allows access to only those columns required, therefore it reduces the memory footprint by applying columnar compression schemas, i.e dictionary encoding, and run-length encoding. The main purpose of using DataFrames as compared to RDDs is that it offers a built-in optimizer for Spark SQL operators, the *Catalyst*. It leverages advanced programming language features (e.g. Scala pattern matching) in order to build an extensible query optimizer by scanning the data schema and its query semantics.

Spark DataFrames are considered being lazy, as a consequence, each DataFrame object represents a logical plan to compute a dataset, but no real execution occurs until an action is called, i.e. count. By this, Spark enables rich optimization across all operations which has been used in order to build a DataFrame.

CHAPTER 3

Related Work

This chapter reviews the related work to our research, according to the research problem and research questions defined in Chapter 1. We first discuss and compare the state-of-the-art RDF dataset statistics systems. Then, we give an overview and discuss previous work related to RDF quality assessment frameworks. Finally, we cover existing SPARQL query evaluators and position our proposed solutions.

This chapter is based on the related work sections from following publications [22, 24–26]:

- **Gezim Sejdiu**; Ivan Ermilov; Jens Lehmann; and Mohamed Nadjib-Mami, "DistLODStats: Distributed Computation of RDF Dataset Statistics," in Proceedings of 17th International Semantic Web Conference (ISWC), 2018.
- **Gezim Sejdiu**; Anisa Rula; Jens Lehmann; and Hajira Jabeen, "A Scalable Framework for Quality Assessment of RDF Datasets," in Proceedings of 18th International Semantic Web Conference (ISWC), 2019.
- **Gezim Sejdiu**; Damien Graux; Imran Khan; Ioanna Lytra; Hajira Jabeen; and Jens Lehmann, "Towards A Scalable Semantic-based Distributed Approach for SPARQL query evaluation," 15th International Conference on Semantic Systems (SEMANTiCS), Research & Innovation , 2019.
- Claus Stadler; **Gezim Sejdiu**; Damien Graux; and Jens Lehmann, "Sparklify: A Scalable Software Component for Efficient evaluation of SPARQL queries over distributed RDF datasets," in Proceedings of 18th International Semantic Web Conference (ISWC), 2019. This article is a joint work with Claus Stadler, a PhD student at the University of Leipzig. In this article, I devised the implementation of the conceptual architecture, helped on the implementation of the proposed approach, reviewed related work, and preparation of the experiments and analysis of the obtained results.

3.1 RDF Dataset Statistics Systems

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 thesis if the main memory on commodity hardware is insufficient to perform the task (without swapping to disk). We mention here, for example RDFStats [12], which is a framework for generating statistics from RDF data that can be used for SPARQL query optimization while processing RDF data over SPARQL endpoints. Such statistics include histograms about subjects (URIs, and blank nodes), properties, and their corresponding ranges. The tool can be integrated into user interfaces and other applications that utilize the Jena toolkit in order to provide such statistics for better performance when processing RDF data. But, the main purpose of the tool is to collect statistics for query optimization rather than generating VoID [50].

RDF_{Pro} [51] offers a suite of stream-oriented, highly optimized processors for common tasks, such as data filtering, Resource Description Framework Schema (RDFS) inference, smushing, as well as statistics extraction. The main component of the tool is a so-called *RDF processor*, a Java component that consumes an input stream of RDF quads containing RDF triples with an optional fourth named graph component in one or more passes. It does by downloading and filtering the desired RDF quads and place them into a separate graph in order to track the provenance. A metadata file is added as a link between each graph generated during the process, to the URI of the associated sources (e.g. DBpedia). Afterward, it extracts the TBox information from such filtered data and then sorts them. The consequence step drop unnecessary top-level classes and vocabulary alignments. The process follows the smushing step – using of canonical URIs for each `owl:sameAs` equivalence class, producing intermediate results (file) containing smushed data. The inference of smushed data is computed and saved. These intermediate results contain duplicate data, e.g. the same subject, predicate, and object. RDF_{Pro} does a deduplication process, by removing such duplicates. Finally, RDF dataset statistics are extracted and merged with the TBox data.

ExpLOD [52] explores summaries of RDF usage and interlinking among datasets. These summaries include information about the structure of the RDF graph, such as the instantiated RDF classes of a resource or property usage. The tool also provides statistics about the number of corresponding entities connected using the `owl:sameAs` predicate to describe the interlinking between datasets. The tool can also produce SPARQL queries from a summary.

ProLOD [53] is a web-based profiling tool, with a possibility to analyze RDF data and thus provide a deeper understanding of the underlying structure and semantics. It analyzes the object values of RDF triples and generates statistics upon them such as data type and pattern distribution. ProLOD uses regular expression rules for type detection and such patterns are normalized on the later stage for better visualization of a large number of different patterns. It also generates a statistical description of the literal values and external links. ProLOD++ [54] is an interactive web-based tool that offers a set of methods with the aim of computing different profiling, mining or cleansing tasks. The tool is divided into two primary views, a cluster view, and a detailed view. The cluster view enables users to explore and navigate through the cluster tree with more information for statistics for the selected cluster. ProLOD++ is an extension of ProLOD. In addition to the mining and the cleansing tasks, ProLOD++ generates profiling features like finding frequencies and distribution of distinct subjects, predicates, and objects, range of the predicates, string pattern analysis, link analysis, and data type analysis.

Loupe [55] is a configurable RESTful web service for generating Linked Data profiles in RDF using the Loupe ontology[1]. A tool provides summarized information about explicit vocabulary, class and

[1] https://github.com/nandana/loupe-ontology

property usage. Besides that, it also facilitates the analysis of implicit data patterns by providing a set of metrics including the ratio of instances of a given class, and property distribution.

Another related approach we are aware of is Aether [56], 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. By giving a SPARQL endpoint, the Aether tool can generate an extended VoID description containing a wide variety of characteristics describing the dataset. Later, these statistics can then be viewed in order to get a better overview of the dataset. The viewer component of the Aether can be also useful on comparing dataset descriptions to each other so that the changes between two different versions of the dataset can be captured.

However, only one work we came across that provided a distributed framework for RDF statistics computation: LODOP [57]. 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: the 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 [58]. 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.

3.2 RDF Quality Assessment Frameworks

Even though quality assessment of big datasets is an important research area, it is still largely under-explored. There have been a few works discussing the challenges and issues of big data quality [59–61]. Only recently, a few of them have started to address the problem from a practical point of view [17], which is the focus of our work w.r.t the quality assessment of RDF datasets. In the following, we divide the section between conceptual and practical approaches proposed in the state of the art for big data quality assessment.

In [62] the authors propose a big data processing pipeline and a big data quality pipeline. For each of the phases of the processing pipeline, they discuss the corresponding phase of the big data quality pipeline. Relevant quality dimensions such as accuracy, consistency, and completeness are discussed for the quality assessment of RDF datasets as part of an integration scenario. Given that the quality dimensions and metrics have somehow evolved from relational to RDF data, it is relevant to understand the evolution of quality dimensions according to the differences between the structural characteristics of the two data models [63]. This allows managing the huge variability of methods and techniques needed to manage data quality and understand which are the quality dimensions that prevail when assessing large-scale RDF datasets.

Most of the existing approaches can be applied to small/medium scale datasets and do not horizontally scale [17, 64]. The work in [64] presents a methodology for assessing the quality of RDF data based on a test case generation analogy used for software testing. The idea of this approach is to generate templates of the SPARQL queries (i.e., quality test case patterns) and then instantiate them by using the vocabulary or schema information, thus producing quality test case queries.

Luzzu [17] is similar in spirit with our approach in that its objective is to provide a framework for quality assessment. 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. In contrast to our approach, where data is distributed and also the evaluation of metrics is distributed, Luzzu does not provide any large-scale processing of the data. It only uses Spark streaming for loading the data which is not part of the core framework.

Another approach proposed for assessing the quality of large-scale medical data implements Hadoop Map/Reduce [65]. It takes advantage of query optimization and joins strategies that are tailored to the structure of the data and the SPARQL queries for that particular dataset. In addition, this work, differently from our approach, does not assess any data quality metric defined in [7]. The work in [66] proposes a reasoning approach to derive inconsistency rules and implements a Spark-based implementation of the inference algorithm for capturing and cleaning inconsistencies in RDF datasets. The inference generally incurs higher complexity. Our approach is designed for scalability, and we also use Spark-based implementation for capturing inconsistencies in the data. While the approach in [66] needs manual definitions of the inconsistency rules, our approach runs automatically, not only for consistency metrics but also for other quality metrics. In addition, we test the performance of our approach to large-scale RDF datasets while their approach is not experimentally evaluated.

LD-Sniffer [18], is a tool for assessing the accessibility of Linked Data resources according to the metrics defined in the Linked Data Quality Model. The limitation of this tool, besides that it is a centralized version, is that it does not provide most of the quality assessment metrics defined in [7]. In addition to the above, there is a lack of unified structure to propose and develop new quality metrics that are scalable and less computationally expensive.

LiQuate [67] is another tool that combines Bayesian Networks and rule-based systems for analyzing the quality of the data and links in the LOD cloud. It uses the probabilistic methods for exploring the assessed datasets for completeness, redundancies, and inconsistencies. It has a two-fold approach. First, it detects the ambiguities and then, links to solve these ambiguities are inferred and suggested to the user for resolving the identified quality problems. The domain expert is required for identifying such rules for the Bayesian Network.

WIQA [68] is another quality assessment framework that provides a mechanism for creating and applying a number of policies driven by the provenance and background context related to the data providers. WIQA provides a SPARQL- like a language (WIQA-PL) for applying any assessment metric over the defined quality metric. It does not report any quality metadata or quality problem reports but rather an assessment result that includes the set of matching triples with a description of why such triple attain the policy.

LINK-QA [69] is a quality assessment framework that allows for the assessment of Linked Data mappings using network metrics i.e. degree, clustering coefficient, centrality, Web Ontology Language (OWL) `sameAs` chains, and descriptive richness through OWL `sameAs`. These metrics have been proposed using the framework on a set of known good and bad links generated by a common mapping system, and show the behavior of those metrics. The system generates HTML reports for the results of the quality assessment.

RDFUnit [70] is another quality assessment system for Linked Data via test-driven quality checks. It follows the test-driven software development concept by providing a set of test-cases, which help to ensure a basic level of quality. The proposed methodology assesses the quality of the RDF data resources, based on a formalization of bad smells and data quality issues. Such a formalization employs SPARQL queries templates into concrete quality test queries. The main focus of RDFUnit is to perform an integrity check via SPARQL patterns. The quality of the data is assessed by executing custom SPARQL queries against different datasets using SPARQL endpoints. Test case results including quality values and quality problems reported from RDFUnit are represented in a form of RDF visualized as HTML.

Based on the identified limitations of these aforementioned approaches, we have introduced DistQualityAssessment which bases its computation and evaluations mainly in-memory. As a result the computation of the quality metrics show a high performance for large-scale datasets (cf. Chapter 5).

3.3 SPARQL Query Evaluators

Partitioning of RDF Data In recent years, significant effort has been made on the development and designing of efficient solutions for managing and processing RDF data. Centralized RDF stores use relational (e.g., Sesame [71]), property (e.g., Jena [72]), or binary tables (e.g., SW-Store [73]) for storing RDF triples or maintain the graph structure of the RDF data (e.g., gStore [74]). These tools have achieved high performance on processing RDF data over a single computation (centralized) node, neither by designing novel data representation of the underlying data or applying different rational optimization techniques w.r.t to the data storage or processing. For dealing with big RDF datasets, vertical partitioning and exhaustive indexing are commonly employed techniques. For instance, Abadi et al. [75] introduce a vertical partitioning approach in which each predicate is mapped to a two-column table containing the subject and object. This approach has been extended in Hexastore [76] to include all six permutations of subject, predicate, and object (s, p, o). To improve the efficiency of SPARQL queries RDF-3X [77] has adopted exhaustive indices not only for all (s, p, o) permutations but also for their binary and unary projections. While some of these techniques can be used in distributed configurations as well, storing and querying RDF datasets in distributed environments pose new challenges such as scalability. In our approach, we tackle partitioning and querying of big RDF datasets in a distributed manner.

Partitioning-based approaches for distributed RDF systems propose to partition an RDF graph in fragments that are hosted in centralized RDF stores at different sites. Such approaches use either standard partitioning algorithms like METIS [78] or introduce their own partitioning strategies. For instance, Lee et al. [79] define a partition unit as a vertex with its closest neighbors based on heuristic rules while DiploCloud [80] and AdPart [81] use physiological RDF partitioning based on RDF molecules. In our proposal, we use both, vertical partitioning and semantic-based partitioning approaches.

Hadoop-Based Systems Cloud-based approaches for managing large-scale RDF mainly use NoSQL distributed data stores or employ various partitioning approaches on top of Hadoop infrastructure, i.e., the HDFS and its MapReduce implementation, in order to leverage computational resources of multiple nodes. For instance, Sempala [82] is a Hadoop-based approach that serves as the SPARQL-to-SQL approach on top of Hadoop. It uses Impala[2] as a distributed SQL processing engine. Sempala uses unified vertical partitioning based on a single property table to improve the runtime of the star-shaped queries by excluding the joins. The limitation of Sempala is that it was designed only for that particular shape of the queries. PigSPARQL [83] uses Hadoop based implementation of

[2] https://impala.apache.org/

vertical partitioning for data representation. It translates SPARQL queries into Pig[3] LATIN queries and runs them using the Pig engine. A most recent approach based on MapReduce is RYA [84]. It is a Hadoop based scalable RDF store that uses Accumulo[4] as a distributed key-value store for indexing the RDF triples. RYA indexes triples into three tables and replicate them across the cluster for leveraging the indexes over all the possible records. It has the mechanism of performing join reorder, but it lacks the in-memory computation, which makes it not comparable with other systems. One of RYA's advantages is the power of performing join reorder. The main drawback of RYA is that it relies on disk-based processing increasing query execution times. Other RDF systems like JenaHBase [85] and H2RDF+ [86] use the Hadoop database HBase for storing triple and property tables. JenaHBase represents triples in the form of three index tables: SPO, POS, and OSP. It maps RDF URIs and most literals to numerical ids and uses the same table structure for all indices: the row key is built from the concatenation of the ids, and leaving the rest i.e. column qualifiers and cell values empty. This is done in order to leverage the lexicographical sorting of the row keys, covering multiple triple patterns with the same table. The main idea behind indexing is reducing network and disk I/O overhead, for fast joins. H2RDF+ is conceptually similar to Rya and JenaHBase as it stores RDF data in HBase. It does that by storing triples in the row key which uses six tables for all possible triple permutations thus creates six different indexes. In addition, it also maintains index statistics for triple pattern selectivity estimation as well as join output size and cost. H2RDF+ is able to answer selective queries efficiently as it is able to determine the scale for non-selective queries to be executed centrally but is slower when done through distributed execution. SHARD [87] is one approach that groups RDF data into a dedicated partition so-called semantic-based partition. It groups these RDF data by subject and implements a query engine which iterates through each of the clauses used on the query and performs a query processing. A MapReduce job is created while scanning each of the triple patterns and generates a single plan for each of the triple patterns which leads to a larger query plan, therefore, it contains too many Map and Reduces jobs. Our partitioning algorithm implemented on the Semantic-based query engine is based on SHARD, but instead of creating MapReduce jobs we employ the Spark framework in order to increase scalability.

While the MapReduce paradigm has been realized for disk-based as well as in-memory processing, the concept is not concerned with controlling aspects of generally distributed workflows, such as which intermediate results to cache. As a consequence, high-level frameworks were devised which may use MapReduce as a building block. Apache Spark is one of them [21]. Below, we will list some of the approaches which make use of the Apache Spark (in-memory computation) framework.

In-Memory Systems S2RDF [1] and SPARQLGX [19] approaches are considered the most recent distributed SPARQL evaluators over large-scale RDF datasets. S2RDF [1] is a distributed query engine that translates SPARQL queries into SQL ones while running them on Spark-SQL [49]. It introduces a data partitioning strategy that extends vertical partitioning with additional statistics, containing pre-computed semi-joins for query optimization. While doing so, S2RDF avoids tuples that do not have counterparts in the referenced relation (join) which reduces the query input size and thus execution runtime. By pre-computing the possible join relations between partitions i.e. tables of Vertical Partitioning (VP), the S2RDF query processor can directly access the subset of a specific table where the object also exists as a subject in at least one tuple in the other table and join it with the equivalent subset of that table. This avoids dangling tuples, tuples that do not find a corresponding join

[3] https://pig.apache.org/

[4] accumulo.apache.org

partner, to be used as input and thus also reduces I/O overhead and the number of join comparisons that lead to overall speeds up. S2RDF query processor is based on the algebra representation of SPARQL expressions. It uses Jena ARQ for parsing the SPARQL query into a corresponding algebra tree. It traverses through the algebra tree and generates the corresponding Spark SQL expressions mapped to the extended vertical partitioning schema as described above. As a consequence, such an equivalent Spark SQL query is then executed by the Spark engine. SPARQLGX [19] is similar to S2RDF, but instead of translating SPARQL to SQL, it maps SPARQL into direct Spark RDD operations. It is a scalable query engine that is capable of evaluating efficiently the SPARQL queries over distributed RDF datasets [88]. It uses a simplified VP approach, where each predicate is assigned to a specific parquet file. As an addition, it is able to assign RDF statistics for further query optimization while also providing the possibility of directly query files on the HDFS using SDE (its direct SPARQL evaluator).

Nevertheless, these engines lack one important information derived from the knowledge, *RDF terms*. RDF terms includes information about a statement such as *language*, *typed literals* and *blank nodes* which are omitted from most of the engines. Beside *RDF terms*, we also wanted to investigate different partitioning mechanisms while querying a large amount of RDF. During this thesis, we propose two different SPARQL query evaluator. Sparklify – a scalable software component for efficient evaluation of SPARQL queries over distributed RDF datasets. The conceptual foundation is the application of *ontology-based data access* (OBDA) tooling, specifically SPARQL-to-SQL rewriting, for translating SPARQL queries into Spark executable code. We demonstrate our approach using Sparqlify, which has been used in the LinkedGeoData[5] community project to serve more than 30 billion triples on-the-fly from a relational OpenStreetMap database. As we mentioned previously, we wanted to see if different partitioning strategies improve the execution time while evaluating SPARQL queries over large-scale RDF datasets and propose a Semantic-based approach which partitions the data into subject-based grouping (e.g. all entities which are associated with a unique subject). For more details on the proposed approaches, see Chapter 6.

[5] http://linkedgeodata.org

CHAPTER 4

Large-Scale RDF Dataset Statistics

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 became machine-readable [89]. 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 [11] and datasets [12, 13]. 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 RDD, initially suggested at [21], 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 chapter we address the following research question:

[1] http://lodstats.aksw.org/
[2] http://spark.apache.org

RQ1: How can we efficiently explore the structure of large-scale RDF datasets?

Contributions of this chapter are summarize 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.).
- An approach for triggering RDF statistics calculation remotely simply using HTTP requests. DistLODStats is built as a plugin into the larger SANSA framework and makes use of Apache Livy, a novel lightweight solution for interacting with the Spark cluster via a REST Interface.

This chapter is based on the following publications ([22, 23]):

- **Gezim Sejdiu**; Ivan Ermilov; Jens Lehmann; and Mohamed Nadjib-Mami, "DistLODStats: Distributed Computation of RDF Dataset Statistics," in Proceedings of 17th International Semantic Web Conference (ISWC), 2018.
- **Gezim Sejdiu**; Ivan Ermilov; Jens Lehmann; and Mohamed-Nadjib Mami, "STATisfy Me: What are my Stats?," in Proceedings of 17th International Semantic Web Conference (ISWC), Poster & Demos, 2018.

The remainder of this chapter is organized as follows: Our approach for the computation of RDF dataset statistics is detailed in Section 4.1. An analysis of the complexity of the computational steps and the data exchange between nodes is conducted in Subsection 4.1.4 to assess the complexity of each statistical criterion. The evaluation of the approach is elaborated in Subsection 4.1.6. STATisfy, a component for triggering RDF statistics calculation remotely by using HTTP request has been described in Section 4.2. Finally, we summarize our work in Section 4.3.

4.1 A Scalable Distributed Approach for Computation of RDF Dataset Statistics

We adopted the 32 statistical criteria proposed in [20]. In contrast to [20], 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 [20] 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 4.1.1 (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 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 is needed, P just returns exactly the results from the intermediate dataset D.

4.1.1 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 the 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 4.1.2 (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:*

- *$map : I \longrightarrow O$, 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.*
- *$filter : I \longrightarrow O$, where I is an input RDD and O is an output RDD, which contains only the elements that satisfy a condition.*
- *$reduce : I \longrightarrow O$, where I is an input RDD of key-value (K,V) pairs and O is an output RDD of (K, list(V)) pairs.*
- *group-by : $(I, F) \longrightarrow O$, 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.*

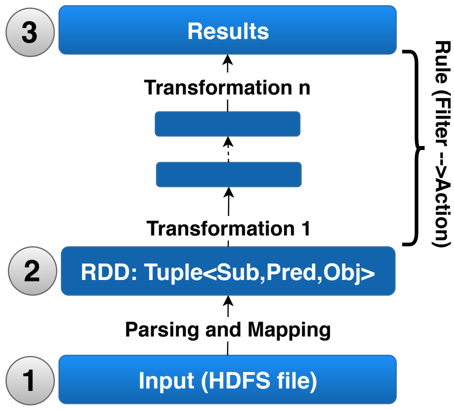


Figure 4.1: **RDD lineage of a Criterion execution**. It consists of three steps: (1) saving RDF data into a scalable storage, (2) parsing and mapping RDF into the main dataset (RDD of triples), and (3) performing statistical criteria evaluation on the main dataset.

4.1.2 Distributed LODStats Architecture

The computation of statistical criteria is performed as depicted in Figure 4.1. 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.

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

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*.

4.1.3 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 4.1). Applied on a *main dataset*, this latter will return a new RDD with a subset of the triples. Next, the *action* method is

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

used to apply the criterion's rule action (Rule Action in Table 4.1). 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).

Algorithmus 1 : DistLODStats.

```
  input :RDF: an RDF dataset, C: a list of criterion.
  /* Iterate through the list of criteria                       */
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.

4.1.4 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 the 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 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 4.2. We deem the complexity is mostly linear corresponding to cases where only one or a limited number of scans is required.

	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(f => (f.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.p, 1)).reduceByKey(_ + _)	take(100)
6	prop. usage per subj.		→ groupBy(_.s).reduceByKey(_ + _)	count
7	prop. usage per obj.		→ groupBy(_.o).reduceByKey(_ + _)	count
8	prop. distinct per subj.		→ groupBy(_.s).combineByKey(_ + _)	sum/count
9	prop. distinct per obj.		→ groupBy(_.o).combineByKey(_ + _)	sum/count
10	outdegree		→ map(f => (f.s, 1)).combineByKey(_ + _)	sum/count
11	indegree		→ map(f => (f.o, 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(_.(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 4.1: **Definition of Spark rules (using Scala notation) per criterion**. A list of statistical criteria following the Rule (Filter->Action) -> Postproc paradigm using the Spark/Scala notation.

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.

Table 4.2: **Complexity and data shuffling breakdown by statistical criterion**. Notation conventions: n = number of triples; V = number of vertices; E = number of edges.

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

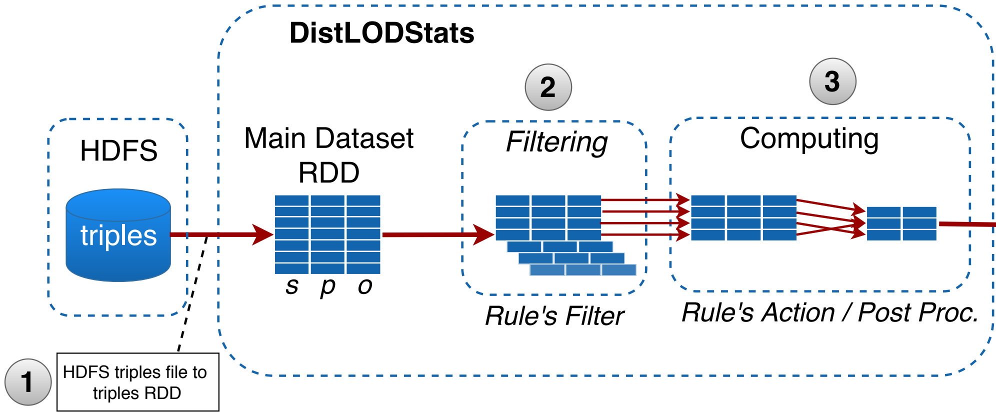


Figure 4.2: **Overview of DistLODStats's abstract architecture**. It is composed of three steps: First, it reads RDF data from HDFS and converts them into RDD of triples. Second, this latter undergoes a Filtering operation applying the Rule's Filter and producing a new filtered RDD. Third, the filtered RDD will serve as an input to the next step: Computing where the rule's action and/or post-processing are effectively applied. As a result, a statistical representation is generated.

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. Afterword, 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.

4.1.5 Implementation

DistLODStats comprises three main phases depicted in Figure 4.2 and explained previously. The output of the *Computing* phase will be the statistical results represented in a human-readable format e.g. VoID [50], or row data.

We expressed the three phases of the 32 criteria using the basic operations defined in Definition 4.1.2. Next, those have been mapped to Spark transformations and actions in Table 4.1, 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 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.

[4] http://tiny.cc/jn91iz
[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

We implemented DistLODStats using Spark-2.2.0, Scala 2.11.11 and Java 8. DistLODStats has meanwhile been integrated into SANSA [30, 31], 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 ensures the sustainability of DistLODStats given that SANSA is backed by several grants until at least 2021.

4.1.6 Evaluation

The aim of our evaluation is to see how well our approach can perform against non-distributed approaches as well as analyzing 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.

Experimental Setup

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

1. We chose the geospatial dataset LinkedGeoData [90] which offers a spatial RDF knowledge base derived from OpenStreetMap.
2. As a cross-domain dataset, we selected *DBpedia* [6] (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) [91]. 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 4.3.

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 @ 2.10GHz (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. The machines were connected via a Gigabit network. All experiments were executed three times and the average value is reported.

[8] https://github.com/SANSA-Stack

⟶	LinkedGeoData	DBpedia			BSBM		
		en	de	fr	2GB	20GB	200GB
#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

Table 4.3: **Dataset summary information (nt format)**. Lists dataset characteristics used on the evaluation of the DistLODStats. The size (in GB) and the number of triples are given.

⟶	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

Table 4.4: **Distributed Processing on Large-Scale Datasets**. Reports the performance analysis of the *speedup* gained by DistLODStats as compared with the original centralized version. The experiments were run on four datasets ($DBpedia_{en}$, $DBpedia_{de}$, $DBpedia_{fr}$, and $LinkedGeoData$) in 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.

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 4.4, Figure 4.3, 4.4 and 4.5.

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 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 4.4 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 Figure 4.3). 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

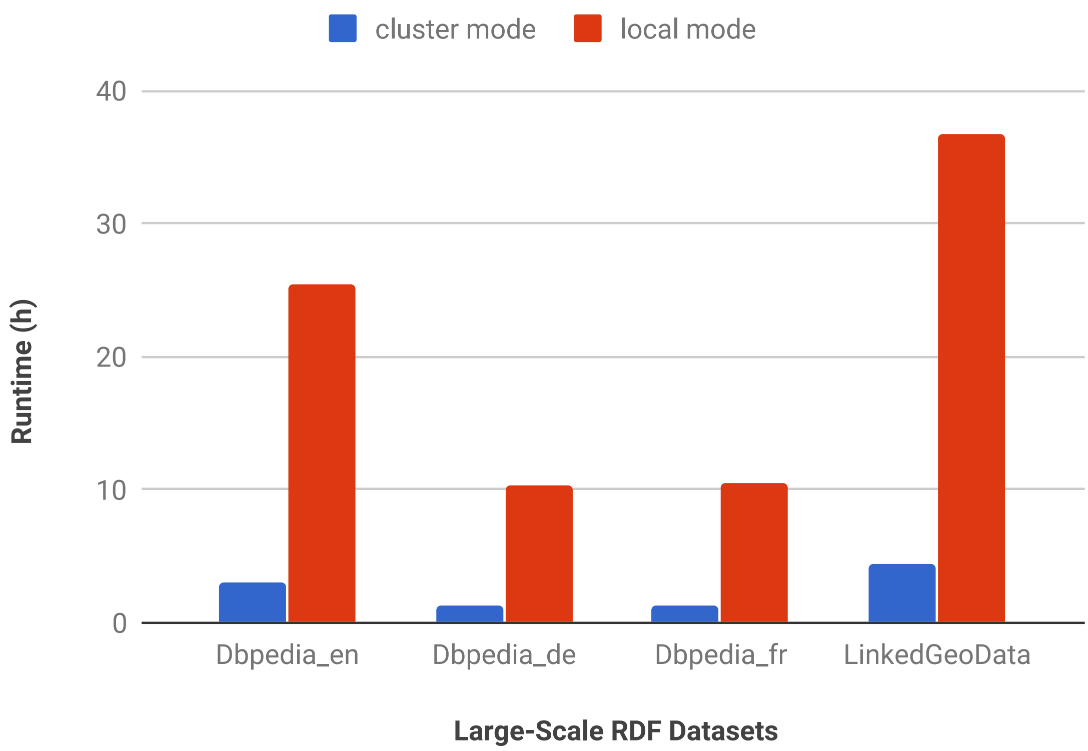


Figure 4.3: **Speedup performance evaluation of DistLODStats**. Reports speedup performance analysis for large-scale RDF datasets for DistLODStats on local mode and cluster mode, respectively. All results illustrate consistent improvement for each dataset when running on a cluster. The geometric mean of the speedup is 7.4x.

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 (25.34h for DistLODStats$^{c)}$ vs 24.63h 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.

Figure 4.3 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 hours which is 7.6 times faster than evaluating DistLODStats on local mode (about 25.34 hours). 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

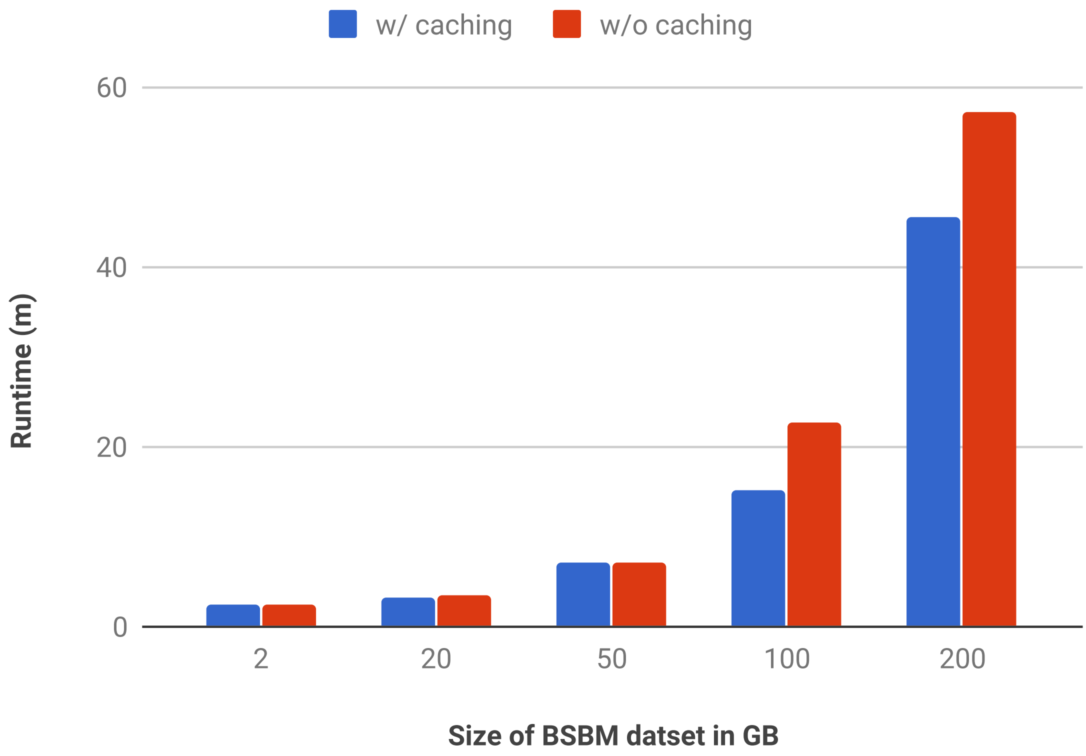


Figure 4.4: **Sizeup performance evaluation of DistLODStats**. The analysis keeps the number of nodes in a cluster constant (5 worker nodes) and grows the size of datasets (BSBM) to measure whether our approach can deal with larger datasets. We see that the execution time cost grows linearly and is near-constant when the size of the dataset increases. It stays near-constant as long as the data fits in memory which demonstrates one of the advantages of utilizing an in-memory approach in performing the statistics computation.

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 2GB. 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 Figure 4.4. The x-axis is a generated BSBM dataset per each order of 10x magnitude.

By comparing the runtime (see Figure 4.4), 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 the scalability of our algorithm in the context of sizeup, which answers question Q_2.

Node scalability In order to measure node scalability, we use variations of the number of workers on

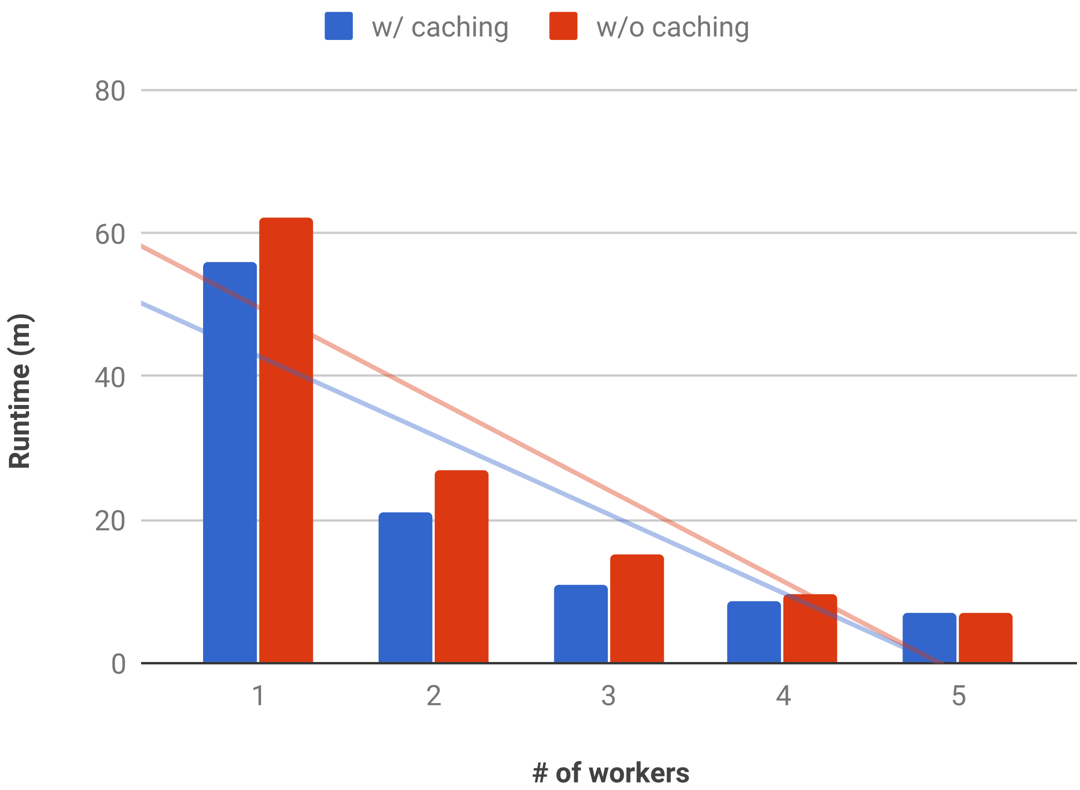


Figure 4.5: **Scalability performance evaluation on DistLODStats**. The analysis keeps the size of the dataset constant ($BSBM_{50GB}$) and varies the number of workers on the cluster. The number of workers varies from 1, 2, 3, and 4 to 5. We can see that as the number of workers increases, the execution time cost is super-linear on $BSBM_{50GB}$ dataset.

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 4.5 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 Figure 4.6, the speedup performance trend is consistent as the number of workers increases.

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 superlinear scalability in performance, which answers question Q_3.

Breakdown by Criterion

Now we analyze the overall runtime of criteria execution. Figure 4.7 reports on the runtime of each

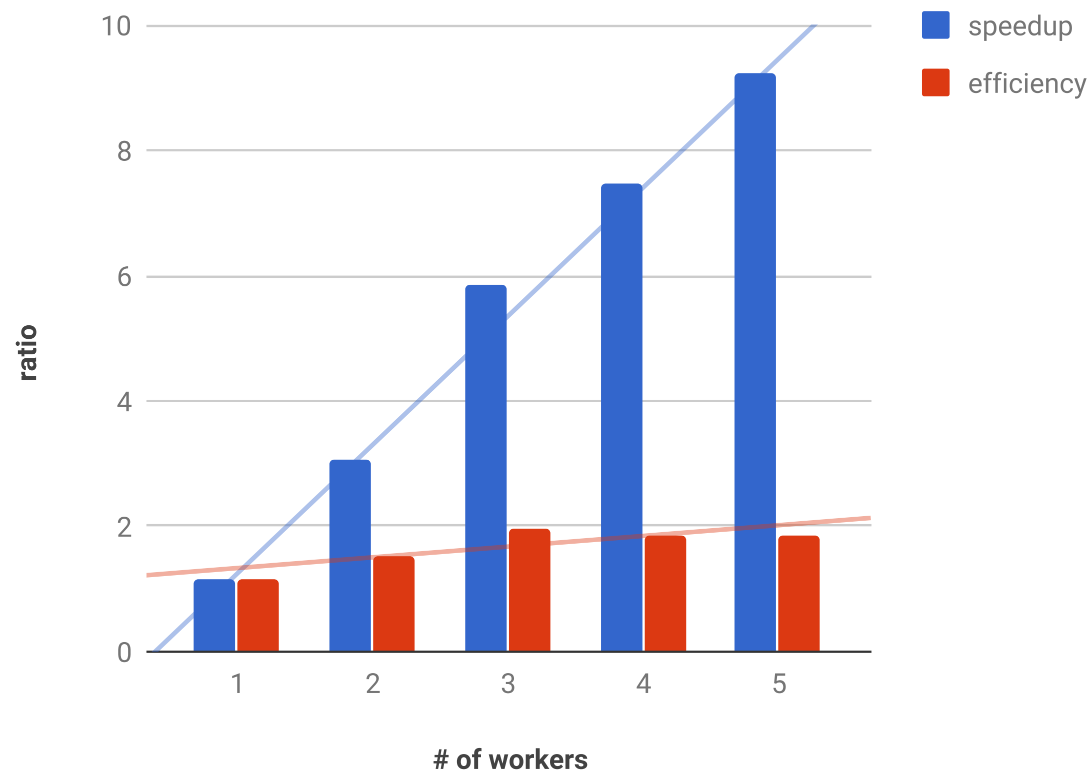


Figure 4.6: **Speedup Ratio and Efficiency of DistLODStats**. The speedup performance trend is consistent as the number of workers increases. Efficiency increased only up to the 4th worker for $BSBM_{50GB}$ dataset. The results imply that DistLODStats can achieve near-linear or even superlinear scalability in performance.

criterion on both $BSBM_{20GB}$ and $BSBM_{200GB}$ datasets.

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 Figure 4.7. The results obtained confirm to a large extent the pre-analysis made in Figure 4.1.4. 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 a reasonable time relative to the

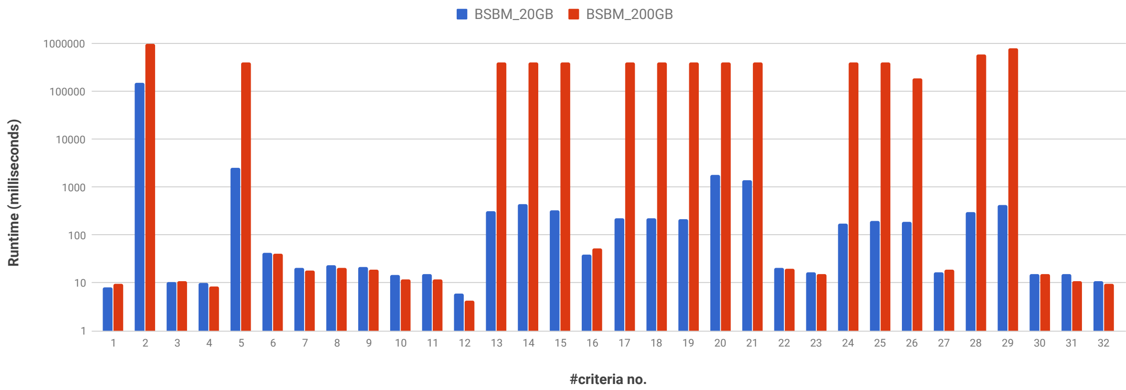


Figure 4.7: **Overall Breakdown by Criterion Analysis (log scale)**. The execution time is longer when there is data movement in the cluster compared to when data is processed without movement. There are some criteria that are quite efficient to compute even with data movement e.g. 22, 23. This is because data is largely filtered before the movement.

high volume of datasets.

4.2 STATisfy: A REST Interface for DistLODStats

The increasing adoption of the Linked Data format, RDF, over the last two decades has brought new opportunities. It has also raised new challenges though, especially when it comes to managing and processing large amounts of RDF data. In particular, assessing the internal structure of a data set is important, since it enables users to understand the data better. One prominent way of assessment is computing statistics about the instances and schema of a data set. However, computing statistics of large RDF data is computationally expensive. To overcome this challenging situation, we previously built DistLODStats, a framework for parallel calculation of 32 statistical criteria over large RDF datasets, based on Apache Spark. Running DistLODStats is, thus, done via submitting jobs to a Spark cluster. Often times, this process is done manually, either by connecting to the cluster machine or via a dedicated resource manager.

SANSA and DistLODStats use Apache Spark[9] as an underlying engine, which is a popular framework for processing large datasets in-memory. Spark provides two possibilities of running and interacting with applications:

- *Interactive* - via a Command Line Interface (CLI) called *Spark Shell*, or via Spark *Notebooks* (e.g. SANSA-Notebooks [30]),
- *Batch* - which includes a bash script called *spark-submit* used to submit a Spark application to the cluster without interaction during run time.

Spark application is usually launched by logging first into a cluster, either in the premises or remotely in the cloud. This process presents several difficulties:

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

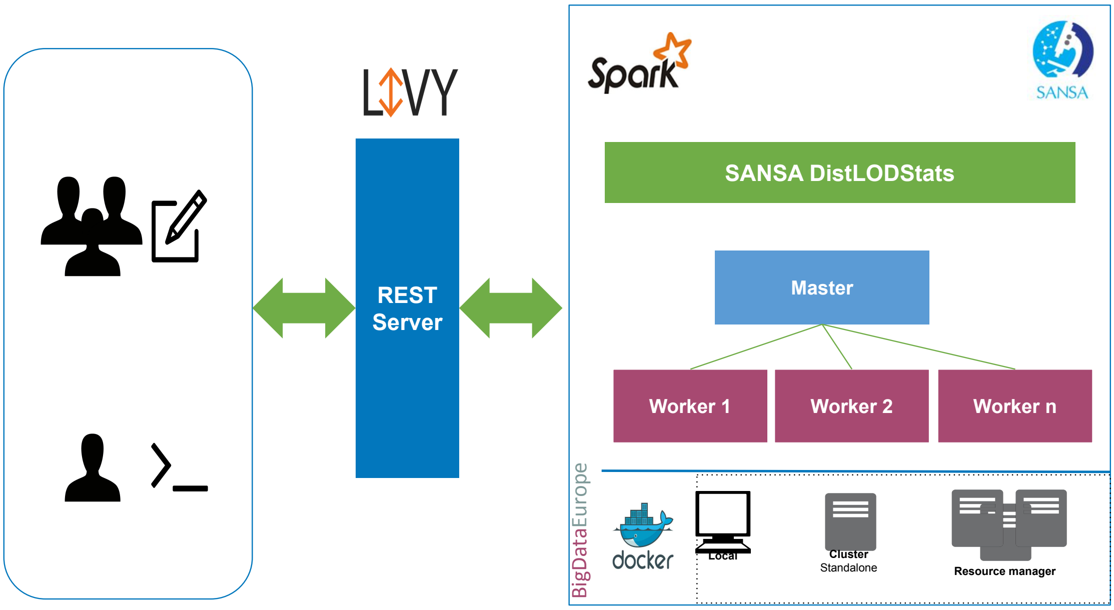


Figure 4.8: **STATisfy overview architecture**. Main services of STATisfy: *Client* – will create a remote Spark cluster for initialization, and submit jobs through REST APIs. *Livy REST Server* – it will then discover this job and sent through Remote Procedure Call (RCP) to SparkSession, where the code will be initialized and executed using DistLODStats.

- It requires a sophisticated user access control management, which may become hard to maintain with multiple users.
- It raises the chances of exhausting the cluster or even causing its failure.
- It exposes cluster and its configurations to all the users with access.

In order to elevate those, we have investigated Apache Livy [10] – a novel open-source REST interface for interacting remotely with Apache Spark. It supports executing snippets of code or programs in a Spark context that runs locally, in a Spark cluster or in Apache Hadoop YARN.

4.2.1 System Design Overview

Traditionally, when running a Spark job, submitting it to a Spark cluster is done via a *spark-shell* or *spark-submit*. Usually, this process is done manually either entering the cluster gateway machines or via a dedicated resource manager (e.g. SLURM, OpenStack).

For users with little experience in cluster management and the Hadoop infrastructure, it can be challenging to run Spark. As an alternative, we introduce **STATisfy**[11]: REST Interface for DistLODStats.

[10] https://livy.incubator.apache.org/
[11] https://github.com/GezimSejdiu/STATisfy

Instead of computing RDF statistics directly on the cluster, the interaction is done via REST APIs (as it is depicted in the Figure 4.8).

The client-side will create a remote Spark cluster for initialization, and submit jobs through REST APIs. Livy REST Server will then discover this job and send it through Remote Procedure Call (RCP) to SparkSession, where the code will be initialized and executed. In the meantime, the client will be waiting for the result of this job coming from the same direction.

Running the STATisfy is similar to using DistLODStats via *spark-submit*. The difference is that this shell is not running locally, instead, it runs in a cluster and transfers the data back and forth through the network.

For demonstrating the usage of the tool, we have deployed it on the comprehensive statistics catalog LODStats[12] which crawls RDF data from metadata portals such as CKAN dataset metadata registry. By doing this, it obtains a comprehensive picture of the current state of the Web of Data. As we use DistLODStats as an underlying engine for computing RDF statistics afterward, the limitation was that the user has to interact with the cluster manually and initiate the job for computing such statistics. By using STATisfy REST interface, LODStats will interact with the cluster from anywhere which provides the capabilities necessary to do this without compromising on ease of use or security.

As it is shown in Figure 4.8, the user starts a session via REST API using Livy for submitting a job to the Spark cluster. With the POST request, the user could submit a request to DistLODStats using the Livy server. Using Livy, STATisfy will then help to launch this request in the cluster. As a result, the output will be curled by their end in the format of the VoID description.

4.3 Summary

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 chapter, 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.

DistLODStats is a prominent solution, however, it requires setup and managing of the cluster configuration and job submission. To make the process easier, we have introduced STATisfy, a tool for interacting with DistLODStats via a REST Interface. This way DistLODStats can be provided as-a-service, where users only send (HTTP) requests to the remote cluster and obtain the wished results, without having any knowledge about system access or cluster management. STATisfy is used for the LODStats project and inclusion in the new DBpedia[13] community release processes is ongoing.

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

[13] https://wiki.dbpedia.org/

CHAPTER 5

Quality Assessment of RDF Datasets at Scale

Large amounts of data are being published openly to Linked Data by different data providers. A multitude of applications such as semantic search, query answering, and machine reading [89] depend on these large-scale[1] RDF datasets. The quality of underlying RDF data plays a fundamental role in large-scale data consuming applications. Measuring the quality of linked data spans a number of dimensions including but not limited to: accessibility, interlinking, performance, syntactic validity or completeness [7]. Each of these dimensions can be expressed through one or more quality metrics. Considering that each quality metric tries to capture a particular aspect of the underlying data, numerous metrics are usually provided against the given data that may or may not be processed simultaneously.

On the other hand, the limited number of existing techniques of quality assessment for RDF datasets are not adequate to assess data quality at large-scale and these approaches mostly fail to capture the increasing volume of big data. To date, a limited number of solutions have been conceived to offer a quality assessment of RDF datasets [14–17]. But, these methods can either be used on a small portion of large datasets [15] or narrow down to specific problems e.g., syntactic accuracy of literal values [16], or accessibility of resources [18]. In general, these existing efforts show severe deficiencies in terms of performance when data grows beyond the capabilities of a single machine. This limits the applicability of existing solutions to medium-sized datasets only, in turn, paralyzing the role of applications in embracing the increasing volumes of the available datasets.

To deal with big data, tools like Apache Spark[2] have recently gained a lot of interest. Apache Spark provides scalability, resilience, and efficiency for dealing with large-scale data. Spark uses the concepts of RDD [21] and performs operations like transformations and actions on this data in order to effectively deal with large-scale data.

To handle large-scale RDF data, it is important to develop flexible and extensible methods that can assess the quality of data at scale. At the same time, due to the broadness and variety of quality assessment domain and resulting metrics, there is a strong need to provide a generic pattern to characterize the quality assessment of RDF data in terms of scalability and applicability to big data.

In this chapter, we borrow the concepts of data *transformation* and *action* from Spark and present a pattern for designing quality assessment metrics over large RDF datasets, which is inspired by design patterns. In software engineering, design patterns are general and reusable solutions to common

[1] http://lodstats.aksw.org/
[2] https://spark.apache.org/

problems. Akin to design pattern, where each pattern acts like a blueprint that can be customized to solve a particular design problem, the introduced concept of Quality Assessment Pattern ($\mathcal{QAP}$) represents a generalized blueprint of scalable quality assessment metrics. In this way, the quality metrics designed following $\mathcal{QAP}$ can exhibit the ability to achieve scalability to large-scale data and work in a distributed manner. In addition, we also provide an open source implementation and assessment of these quality metrics in Apache Spark following the proposed $\mathcal{QAP}$.

In this chapter we address the following research question:

> **RQ2**: Can we scale RDF dataset quality assessment horizontally?

Contributions of this chapter are summarize as follows:

- We present a Quality Assessment Pattern $\mathcal{QAP}$ to characterize scalable quality metrics.
- We provide DistQualityAssessment – a distributed (open source) implementation of quality metrics using Apache Spark.
- We perform an analysis of the complexity of the metric evaluation 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. SANSA is actively maintained and uses the community ecosystem (mailing list, issues trackers, continuous integration, website, etc.).

This chapter is based on the following publication ([24]):

- **Gezim Sejdiu**; Anisa Rula; Jens Lehmann; and Hajira Jabeen, "A Scalable Framework for Quality Assessment of RDF Datasets," in Proceedings of 18th International Semantic Web Conference (ISWC), 2019.

The remainder of this chapter is organized as follows: Our approach for the computation of RDF dataset quality metrics is detailed in Section 5.1 and evaluated in Section 5.2. Finally, we summarize our work in Section 5.3.

5.1 A Scalable Framework for Quality Assessment of RDF Datasets

In this section, we first introduce the basic notions used in our approach, the formal definition of the proposed quality assessment pattern and then describe the workflow.

5.1.1 Quality Assessment Pattern

Data quality is commonly conceived as a multi-dimensional construct [92] with a popular notion of 'fitness for use' and can be measured along many dimensions $\mathcal{D}$ such as accuracy ($d_{accu} \in \mathcal{D}$), completeness ($d_{comp} \in \mathcal{D}$) and timeliness ($d_{tmls} \in \mathcal{D}$). The assessment of a quality dimensions d is based on quality metrics $\mathcal{QM} = \{m_1, m_2, \ldots m_k\}$ where m_i is a heuristic that is designed to fit a specific assessment dimension. The following definitions form the basis of $\mathcal{QAP}$.

`Quality Metric`	`:=`	`Action` | `(Action` $\mathcal{OP}$ `Action)`
$\mathcal{OP}$	`:=`	$*$ | $-$ | $/$ | $+$
`Action`	`:=`	`Count(Transformation)`
`Transformation`	`:=`	`Rule(Filter)` | `(Transformation BOP Transformation)`
`Filter`	`:=`	`getPredicates` $\sim ?p$ | `getSubjects` $\sim ?s$ | `getObjects` $\sim ?o$ | `getDistinct(Filter)` | `Filter or Filter` | `Filter && Filter)`
`Rule`	`:=`	`isURI(Filter)` | `isIRI(Filter)` | `isInternal(Filter)` | `isLiteral(Filter)` | `!isBroken(Filter)` | `hasPredicateP` | `hasLicenceAssociated(Filter)` | `hasLicenceIndications(Filter)` | `isExternal(Filter)` | `hasType((Filter)` | `isLabeled(Filter)`
`BOP`	`:=`	$\cap$ | $\cup$

Table 5.1: **Quality Assessment Pattern**. A reusable template for quality metric implementation composed of transformations and actions.

Definition 5.1.1 (Filter) *Let $\mathcal{F} = \{f_1, f_2, \ldots f_l\}$ be a set of filters where each filter f_i sets a criteria for extracting predicates, objects, subjects, or their combination. A filter f_i takes a set of RDF triples as input and returns a subgraph that satisfies the filtering criteria.*

Definition 5.1.2 (Rule) *Let $\mathcal{R} = \{r_1, r_2, \ldots r_j\}$ be a set of rules where each rule r_i sets a conditional criteria. A rule takes a subgraph as input and returns a new subgraph that satisfies the conditions posed by the rule r_i.*

Definition 5.1.3 (Transformation) *A transformation $\tau : \mathcal{G} \to \mathcal{G}'$ is an operation that applies rules defined by $\mathcal{R}$ on the RDF graph $\mathcal{G}$ and returns an RDF subgraph $\mathcal{G}'$. A transformation τ can be a union $\cup$ or intersection $\cap$ of other transformations.*

Definition 5.1.4 (Action) *An action $\alpha : \mathcal{G} \to \mathbb{R}$ is an operation that triggers the transformation of rules on the filtered RDF graph $\mathcal{G}'$ and generates a numerical value. Action α is the count of elements obtained after performing a τ operation.*

Definition 5.1.5 (Quality Assessment Pattern $\mathcal{QAP}$) *The Quality Assessment Pattern $\mathcal{QAP}$ is a reusable template to implement and design scalable quality metrics. The $\mathcal{QAP}$ is composed of transformations and actions. The output of a $\mathcal{QAP}$ is the outcome of an action returning a numeric value against the particular metric.*

$\mathcal{QAP}$ is inspired by Apache Spark operations and designed to fit different data quality metrics (for more details see Table 5.1).

Each data *quality metric* can be defined following the $\mathcal{QAP}$. Any given data quality metric m_i that is represented through the $\mathcal{QAP}$ using transformation τ and action α operations can be easily transformed into Spark code to achieve scalability.

Table 5.2 demonstrates a few selected quality metrics defined against proposed $\mathcal{QAP}$.

As shown in Table 5.2, each quality metric can contain multiple rules, filters or actions. It is worth mentioning that action `count(triples)` returns the total number of triples in the given data. This can also be seen that the action can be an arithmetic combination of multiple actions i.e. ratio, sum,

	Metric	Transformation τ	Action α
L1	Detection of a Machine Readable License	`r = hasLicenceAssociated(?p)`	α `= count(r)` α `> 0 ? 1 : 0`
L2	Detection of a Human Readable License	`r = isURI(?s)` ∩ `hasLicenceIndications(?p)` ∩ `isLiteral(?o)` ∩ `isLicenseStatement(?o)`	α `= count(r)` α `> 0 ? 1 : 0`

I2	Linkage Degree of Linked External Data Providers	`r_1 = isIRI(?s)` ∩ `internal(?s)` ∩ `isIRI(?o)` ∩ `external(?o)`
		`r_2 = isIRI(?s)` ∩ `external(?s)` ∩ `isIRI(?o)` ∩ `internal(?o)`
		`r_3 = r_1` ∪ `r_2`
			α`_1 = count(r_3)`
			α`_2 = count(triples)`
			α `= a_1/a_2`

U1	Detection of a Human Readable Labels	`r_1 = isURI(?s)` ∩ `isInternal(?s)` ∩ `isLabeled(?p)`
		`r_2 = isInternal(?p)` ∩ `isLabeled(?p)`
		`r_3 = isURI(?o)` ∩ `isInternal(?o)` ∩ `isLabeled(?p)`
			α`_1 = count(r_1) + count(r_2) + count(r_3)`
			α`_2 = count(triples)`
			α`_1/` α`_2`

RC1	Short URIs	`r_1 = isURI(?s)` ∪ `isURI(?p)` ∪ `isURI(?o)` `r_2 = resTooLong(?s, ?p, ?o)`	α`_1 =count(r_2)` α`_1/count(triples)`
SV3	Identification of Literals with Malformed Datatypes	`r = isLiteral(?o)` ∩ `getDatatype(?o)` ∩ `isLexicalFormCompatibleWithDatatype(?o)`	α `= count(r)`
CN2	Extensional Conciseness	`r = isURI(?s)` ∩ `isURI(?o)`	α`_1 = count(r)` α`_2 = count(triples)` `(`α`_2-` α`_1)/` α`_2`

Table 5.2: **Definition of selected metrics following $\mathcal{QAP}$.** List of few selected quality metrics defined against proposed QAP.

etc. We illustrate our proposed approach on some metrics selected from [7, 17]. Given that the aim of this chapter is to show the applicability of the proposed approach and comparison with existing methods, we have only selected those which are already provided out-of-box in Luzzu.

5.1.2 System Overview

In this section, we give an overall description of the data model and the architecture of DistQualityAssessment. We model and store RDF graphs $\mathcal{G}$ based on the basic building block of the Spark framework, RDDs. RDDs are in-memory collections of records that can be operated in parallel on a large distributed cluster. RDDs provide an interface based on *coarse-grained* transformations (e.g *map*, *filter* and *reduce*): operations applied on an entire RDD. A *map* function transforms each value from an input RDD into another value while applying τ rules. A *filter* transforms an input RDD to an output RDD, which contains only the elements that satisfy a given condition. *Reduce* aggregates the RDD elements using a specific function from τ.

The computation of the set of quality metrics $\mathcal{QM}$ is performed using Spark as depicted in Figure 5.1. Our approach consists of four steps:

Defining quality metrics parameters (Step 1) The metric definitions are kept in a dedicated file that contains most of the configurations needed for the system to evaluate quality metrics and gather result sets.

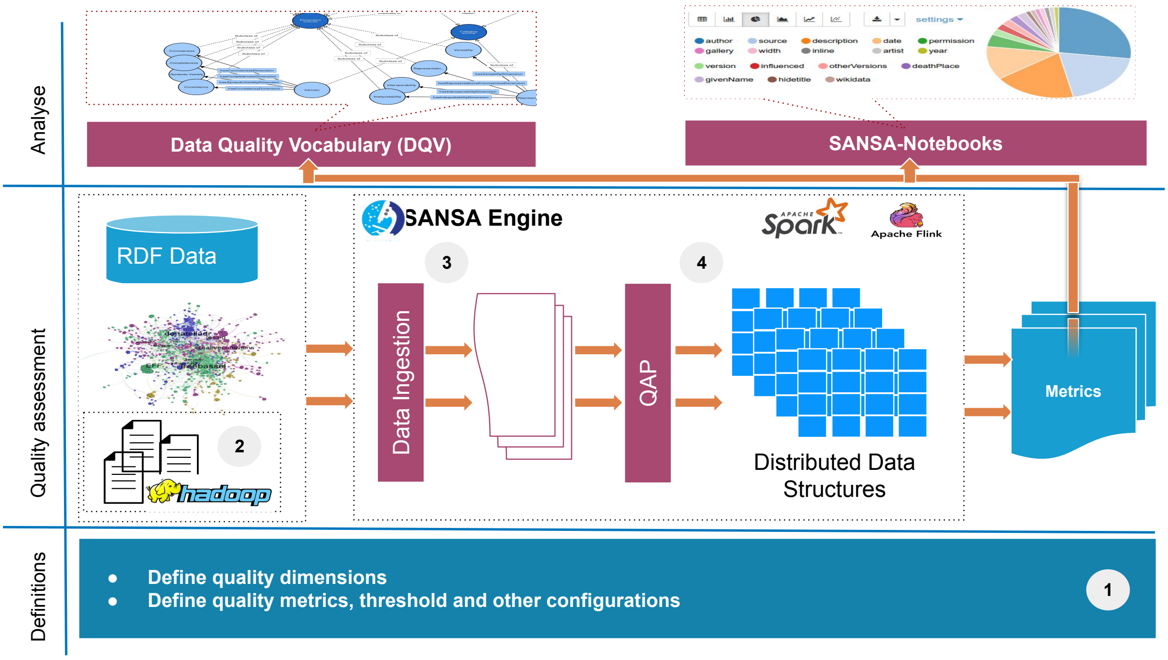


Figure 5.1: **Overview of distributed quality assessment's abstract architecture**. Main components of DistQualityAssessment: 1) Definitions – defining quality metrics parameters, 2) Retrieving the RDF data, 3) Parsing and mapping RDF data into the main dataset (RDD of triples), and 4) Quality metric evaluation.

Retrieving the RDF data (Step 2) RDF data first needs to be loaded into a large-scale storage that Spark can efficiently read from. We use HDFS. HDFS is able to fit and stores any type of data in its Hadoop-native format and parallelizes 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 (Step 3) 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 (Step 4) Considering the particular quality metric, Spark generates an execution plan, which is composed of one or more τ transformations and α actions. The numerical output of the final action is the quality of the input RDF corresponding to the given metric.

5.1.3 Implementation

We have used the Scala[3] programming language API in Apache Spark to provide the distributed implementation of the proposed approach.

The DistQualityAssessment (see Algorithm 2) constructs the *main dataset* (Line 1) while reading RDF data (e.g. NTriples file or any other RDF serialization format) and converts it into an RDD of triples. This latter undergoes the transformation operation of applying the filtering through rules in *R*

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

Algorithmus 2 : Spark-based parallel quality assessment algorithm.

```
   input  : RDF: an RDF dataset, param: quality metrics parameters.
   output : dqv description or metric numerical value
1  triples = spark.rdf(lang)(input)
2  triples.persist()
3  dqv ← ∅
4  foreach m ∈ param.getListOfMetrics do
5      triples ← triples.Tranform { t =>
6          rule ← m.Rule
7          t.apply(rule) }
8      metric ← triples.apply(m.Action)
9      if m.hasDQVdescription then
10         dqvify ← metric.dqvify()
11     dqv.add(dqvify)
12 return (dqv, metric)
```

and producing a new *filtered* RDD ($\mathcal{G}'$) (Line 5). At the end, $\mathcal{G}'$ will serve as an input to the next step which applies a set of α actions (Line 8). The output of this step is the metric output represented as a numerical value (Line 8). The result set of different quality metrics (Line 12) can be further visualized and monitored using SANSA-Notebooks [30].
The user can also choose to extract the output in a machine-readable format (Line 10). We have used the data quality vocabulary (DQV) [93] to represent the quality metrics.

Furthermore, we also provide a Docker image of the system integrated within the BDE platform[4] - an open-source Big Data processing platform allowing users to install numerous big data processing tools and frameworks and create working data flow applications.

The work done here (available under *Apache License 2.0*) has been integrated into SANSA [31], an open source[5] *data flow processing engine* for scalable processing of large-scale RDF datasets. SANSA uses Spark offering fault-tolerant, highly available and scalable approaches to process massive sized datasets efficiently. SANSA provides the facilities for semantic data representation, querying, inference, and analytics at scale. Being part of this integration, DistQualityAssessment can take advantage of having the same user community as well as infrastructure build via the SANSA project. Doing so, it can also ensure the sustainability of the tool given that SANSA is supported by several grants until at least 2021.

Complexity Analysis We deem that the overall time complexity of the distributed quality assessment evaluation is $O(n)$. The performance of metrics computation depends on data shuffling (while filtering using rules in R) and data scanning. Our approach performs a direct mapping of any quality metric designed using $\mathcal{QAP}$ into a sequence of Spark-compliant Scala-commands, as a consequence, most of the operators used are a series of transformations like map, $filter$ and $reduce$. The complexity of map and $filter$ is considered to be linear with respect to the number of triples associated with it. The

[4] https://github.com/big-data-europe

[5] https://github.com/SANSA-Stack

complexity of a metric then depends on the α operation that returns the count of the filtered output. This later step works on the distributed RDD between p nodes which implies that the complexity of each node then becomes $O(n/p)$, where n is a number of input triples. Let be $O(\tau)$ a complexity of τ, then the complexity of the metric will be $O(n/p * O(\tau))$. This indicates that the runtime increases linearly when the size of an RDD increases and decreases linearly when more nodes p are added to the cluster.

5.2 Evaluation

The main aim of DistQualityAssessment is to serve massive large-scale real-life RDF datasets. We are interested in addressing the following additional questions.

- **Flexibility**: How fast our approach processes different types of metrics?
- **Scalability**: How large are the RDF datasets that DistQualityAssessment can scale to? What is the system speedup w.r.t the number of nodes in a cluster mode?
- **Efficiency**: How well our approach performs compared with other state-of-the-art systems on real-world datasets?

In the following, we present our experimental setup including the datasets used. Thereafter, we give an overview of our results.

5.2.1 Experimental Setup

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

1. *DBpedia* [6] (v 3.9) – a cross domain dataset. DBpedia is a knowledge base with a large ontology. We build a set of 3 pipelines of increasing complexity: (i) $M^{en}_{DBpedia}$ ($\approx$ 813M triples); (ii) $M^{de}_{DBpedia}$ ($\approx$ 337M triples); (iii) $M^{fr}_{DBpedia}$ ($\approx$ 341M triples). DBpedia has been chosen because of its popularity in the Semantic Web community.
2. *LinkedGeoData* [90] – a spatial RDF knowledge base derived from OpenStreetMap.
3. *Berlin SPARQL Benchmark (BSBM)* [91] – a synthetic dataset based on an e-commerce use case containing a set of products that are offered by different vendors and reviews posted by consumers about products. The benchmark provides a data generator, which can be used to create sets of connected triples of any particular size.

Properties of the considered datasets are given in Table 5.3.

We implemented DistQualityAssessment using Spark-2.4.0, Scala 2.11.11 and Java 8, and all the data were stored on the HDFS cluster using Hadoop 2.8.0. The experiments in local mode are all performed on a single instance of the cluster. Specifically, we compare our approach with Luzzu [17] v4.0.0, a state-of-the-art quality assessment system[6]. All distributed experiments were carried out on a small cluster of 7 nodes (1 master, 6 workers): Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz (32 Cores), 128 GB RAM, 12 TB SATA RAID-5. The machines were connected via a Gigabit network. All experiments have been executed three times and the average value is reported in the results.

[6] https://github.com/Luzzu/Framework

$\longrightarrow$	LinkedGeoData	DBpedia			BSBM		
		en	de	fr	2GB	20GB	200GB
#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

Table 5.3: **Dataset summary information (nt format)**. Lists dataset information used on the evaluation of the DistQualityAssessment. The size (in GB) and the number of triples are given.

5.2.2 Results

We evaluate the proposed approach using the above datasets to compare it against Luzzu [17]. We carry out two sets of experiments. First, we evaluate the runtime of our distributed approach in contrast to Luzzu. Second, we evaluate the horizontal scalability via increasing nodes in the cluster. Results of the experiments are presented in Table 5.4, Figure 5.2 and 5.3. Based on the metric definition, some metrics make use of external access (e.g. Dereferenceability of Forward Links) which leads to a significant increase in Spark processing due to network latency. For the sake of the evaluation, we have suspended such metrics. As of that, we choose seven metrics (see Table 5.2 for more details) where the level of difficulty varies from simple to complex according to the combination of transformation/action operations involved.

Performance evaluation on large-scale RDF datasets

We started our experiments by evaluating the *speedup* gained by adopting a distributed implementation of quality assessment metrics using our approach, and compare it against Luzzu. We run the experiments on five datasets ($DBpedia_{en}$, $DBpedia_{de}$, $DBpedia_{fr}$, $LinkedGeoData$ and $BSBM_{200GB}$). Local mode represents a single instance of the cluster without any tuning of Spark configuration and the cluster mode includes further tuning. Luzzu was run in a local environment on a single machine with two strategies: (1) streaming the data for each metric separately, and (2) one stream/load – all metrics evaluated just once.

Table 5.4 shows the performance of two approaches applied to five datasets. In Table 5.4 we indicate "Timeout" whenever the process did not complete within a certain amount of time[7] and "Fail" when the system crashed before this timeout delay. Column Luzzu$^{a)}$ represents the performance of Luzzu on bulk load – considering each metric as a sequence of the execution, on the other hand, the column Luzzu$^{b)}$ reports on the performance of Luzzu using a joint load by evaluating each metric using one load. The last columns reports on the performance of DistQualityAssessment run on a local mode c), cluster mode d) and speedup ratio of our approach compared to Luzzu$^{b)}$ $(d)/b) - 1)$ and itself evaluated on local mode $(d)/c) - 1)$ is reported on the column e). We observe that the execution of our approach finishes with all the datasets whereas this is not the case with Luzzu which either timeout or fail at some point.

Unfortunately, Luzzu was not capable of evaluating the metrics over large-scale RDF datasets from Table 5.4 (part one). For that reason, we run yet another set of experiments on very small datasets that Luzzu was able to handle. The second part of the Table 5.4 shows a performance evaluation of our

[7] We set the timeout delay to 24 hours of the quality assessment evaluation stage.

⟶		Runtime (m) (mean/std)				
		Luzzu		**DistQualityAssessment**		
		a) single	b) joint	c) local	d) cluster	e) speedup ratio w.r.t Luzzu |DistQualityAssessment$^{c)}$
Large-scale	$LinkedGeoData$	Fail	Fail	446.9/63.34	7.79/0.54	n/a|56.4x
	$DBpedia_{en}$	Fail	Fail	274.31/38.17	1.99/0.04	n/a|136.8x
	$DBpedia_{de}$	Fail	Fail	161.4/24.18	0.46/0.04	n/a|349.9x
	$DBpedia_{fr}$	Fail	Fail	195.3/26.16	0.38/0.04	n/a|512.9x
	$BSBM_{200GB}$	Fail	Fail	454.46/78.04	7.27/0.64	n/a|61.5x
Small to medium	$BSBM_{0.01GB}$	2.64/0.02	2.65/0.01	0.04/0.0	0.42/0.04	65x|(-0.9x)
	$BSBM_{0.02GB}$	5.9/0.16	5.66/0.02	0.04/0.0	0.43/0.03	146.5x|(-0.9x)
	$BSBM_{0.05GB}$	16.38/0.44	15.39/0.21	0.05/0.0	0.46/0.02	326.6x|(-0.9x)
	$BSBM_{0.1GB}$	40.59/0.56	37.94/0.28	0.06/0.0	0.44/0.05	675.5x|(-0.9x)
	$BSBM_{0.2GB}$	101.8/0.72	101.78/0.64	0.07/0.0	0.4/0.03	1453.3|(-0.8x)
	$BSBM_{0.5GB}$	459.19/18.72	468.64/21.7	0.15/0.01	0.48/0.03	3060.3x|(-0.7x)
	$BSBM_{1GB}$	1454.16/10.55	1532.95/51.6	0.4/0.02	0.56/0.02	3634.4x|(-0.3x)
	$BSBM_{2GB}$	Timeout	Timeout	3.19/0.16	0.62/0.04	n/a|4.1x
	$BSBM_{10GB}$	Timeout	Timeout	29.44/0.14	0.52/0.01	n/a|55.6x
	$BSBM_{20GB}$	Fail	Fail	34.32/9.22	0.75/0.29	n/a|44.8x

Table 5.4: **Performance evaluation on large-scale RDF datasets**. A *speedup* analysis gained by DistQualityAssessment as compared with Luzzu. The experiments were run on five datasets ($DBpedia_{en}$, $DBpedia_{de}$, $DBpedia_{fr}$, $LinkedGeoData$ and $BSBM_{200GB}$). Luzzu was run in a local environment on a single machine with two strategies: (1) streaming the data for each metric separately, and (2) one stream/load – all metrics evaluated just once.

approach compared with Luzzu on very small RDF datasets. In some cases (e.g. RC1, SV3) for a very small dataset, Luzzu performs better than our approach with a small margin of runtime in the local mode. It is due to the fact that in the streaming model when Luzzu$^{a)}$ finds the first statement which fulfills the condition (e.g.finding the shortest URIs), it stops the evaluation and returns the results. On the contrary, our approach evaluates the metrics over the whole dataset exploiting the fault-tolerance and resilient features build-in Spark. In other cases, Luzzu suffers from significant slowdowns, which are several orders of magnitude slower. Therefore, its average runtime over all metrics is worst as compared to our approach. It is important to note that our approach to these very small datasets degrades while running on the cluster mode. This is because of the network overhead while shuffling the data, but it outperforms Luzzu$^{a),b)}$ when considering "average runtime" over all the metrics (even for very small datasets).

Findings shown in Table 5.4 depicts that our approach starts outperforming when the size of the dataset grows (e.g. $BSBM_{2GB}$). The runtime in the cluster mode stays constant when the size of the data fits into the main memory of the cluster. On other hand, Luzzu is not able to evaluate the metrics when the size of data starts increasing, the time taken lasts beyond the delay we set for small datasets. Because of the large differences, we have used a logarithmic scale to better visualize these results.

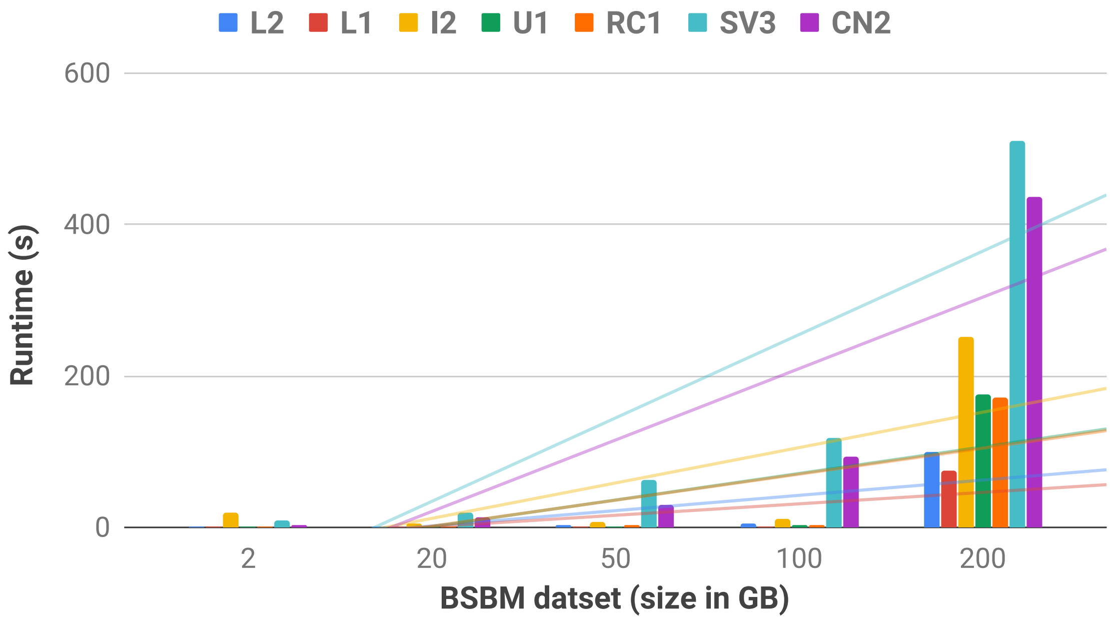


Figure 5.2: **Sizeup performance evaluation of DistQualityAssessment**. The analysis fixes the number of nodes to 6 and grows the size of datasets to measure whether DistQualityAssessment can deal with larger datasets. We see that the execution time increases linearly and is near-constant when the size of the dataset increases. As expected, it stays near-constant as long as the data fits in memory.

Scalability performance analysis

In this experiment, we evaluate the efficiency of our approach. Figure 5.2 and 5.3 illustrates the results of the comparative efficiency analysis.

Data scalability To measure the performance of *size-up* scalability of our approach, we run experiments on five different sizes.

We fix the number of nodes to 6 and grow the size of datasets to measure whether DistQualityAssessment can deal with larger datasets. For this set of experiments, we consider BSBM benchmark tool to generate synthetic datasets of different sizes since the real-world dataset is considered to be unique in their size and attributes.

We start by generating a dataset of 2GB. Then, we iteratively increase the size of datasets. On each dataset, we run our approach and the runtime is reported on Figure 5.2. The x-axis shows the size of the BSBM dataset with an increasing order of 10x magnitude.

By comparing the runtime (see Figure 5.2), we note that the execution time increases 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 the in-memory approach for performing quality assessment computation. The overall time spent in data read/write and network communication found in disk-based approaches is saved. However, when the data overflows the memory, and it is spilled to disk, the performance degrades. These results show the scalability of our algorithm in the context of size-up.

Node scalability In order to measure node scalability, we vary the number of workers on our cluster.

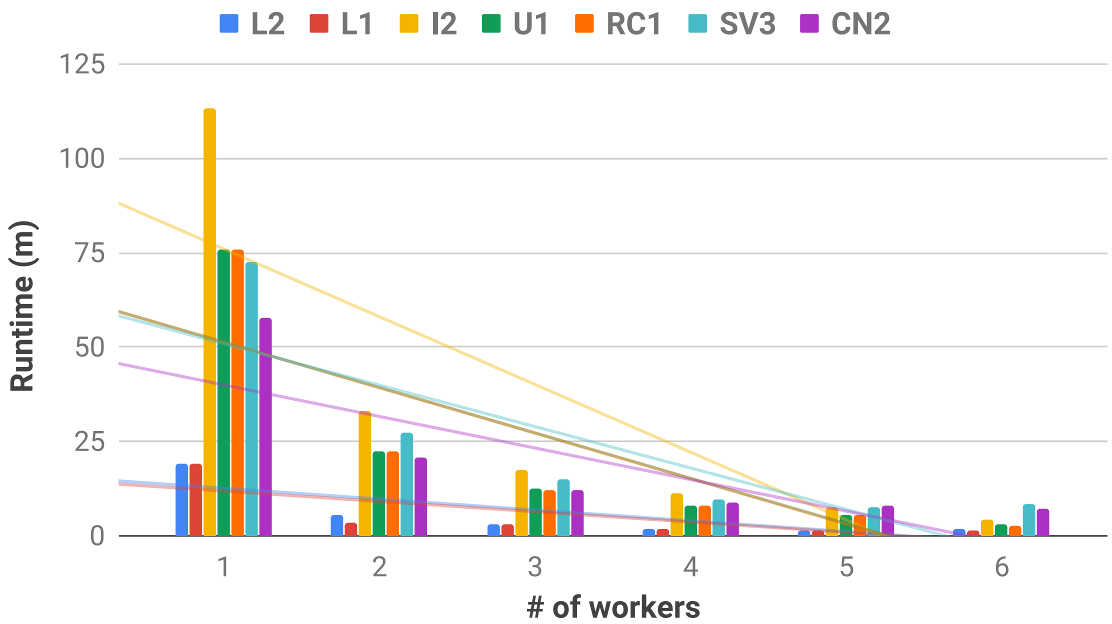


Figure 5.3: **Node scalability performance evaluation of DistQualityAssessment**. The analysis keeps the size of the dataset constant ($BSBM_{200GB}$) and varies the number of workers on the cluster. The number of workers varies from 1, 2, 3, 4 and 5 to 6. We can see that as the number of workers increases, the execution time cost-decrease is almost linear. It decreases about 14 times (from 433.31 minutes down to 28.8 minutes) as cluster nodes increase from one to six worker nodes. The results shown here imply that our approach can achieve near-linear scalability in performance in the context of speedup.

The number of workers has varied from 1, 2, 3, 4 and 5 to 6.

Figure 5.3 shows the speedup for $BSBM_{200GB}$ with the various number of worker nodes. We can see that as the number of workers increases, the execution time cost-decrease is almost linear. The execution time decreases about 14 times (from 433.31 minutes down to 28.8 minutes) as cluster nodes increase from one to six worker nodes. The results shown here imply that our approach can achieve near-linear scalability in performance in the context of speedup.

Furthermore, we conduct the effectiveness evaluation of our approach. Speedup S is an important metric to evaluate a parallel algorithm. It is defined as a ratio $S = T_s/T_n$, where T_s represents the execution time of the algorithm run on a single node and T_n represents the execution time required for the same algorithm on n nodes with the same configuration and resources. Efficiency is defined as a ratio $E = S/n = T_s/nT_n$ which measures the processing power being used, in our case the speedup per node.

The speedup and efficiency curves of DistQualityAssessment are shown in Figure 5.4. The trend shows that it achieves almost linear speedup and even superlinear in some cases. The upper curve in the Figure 5.4 indicates superlinear speedup. The speedup grows faster than the number of worker nodes. This is due to the computation task for the metric being computationally intensive, and the data does not fit in the cache when executed on a single node. But it fits into the caches of several machines when the workload is divided amongst the cluster for parallel evaluation. While using Spark, the superlinear speedup is an outcome of the improved complexity and runtime, in addition to efficient

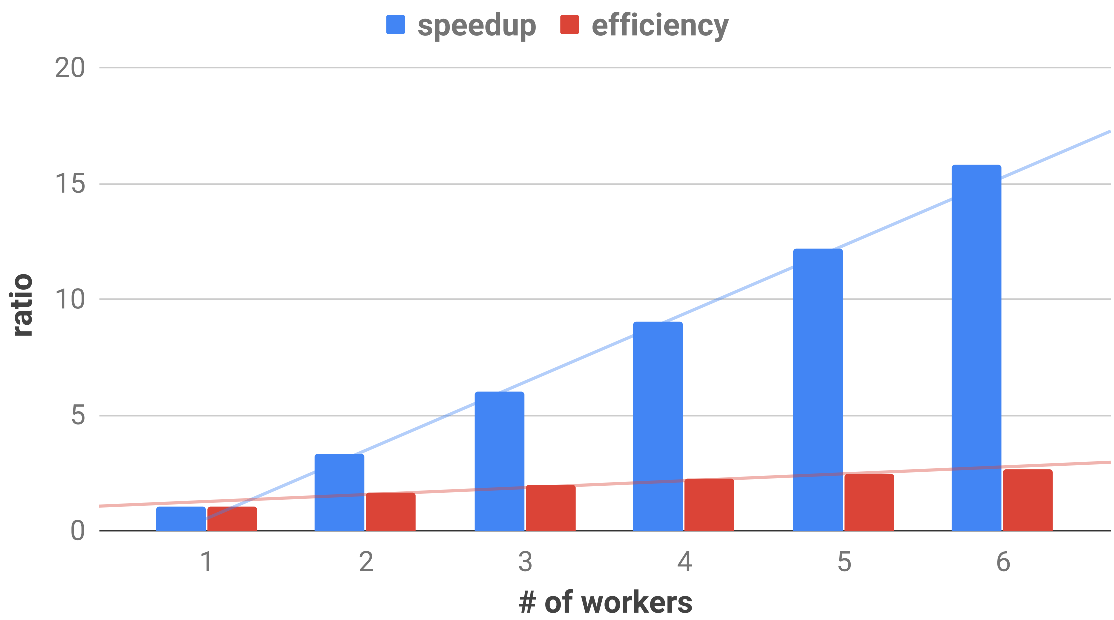


Figure 5.4: **Effectiveness of DistQualityAssessment**. The speedup performance trend shows that it achieves almost linear speedup and even superlinear in some cases. The speedup grows faster than the number of worker nodes due to the computation task for the metric being computationally intensive, and the data does not fit in the cache when executed on a single node but fits into several machines when the workload is divided amongst the cluster for parallel evaluation.

memory management behavior of the parallel execution environment.

Correctness of metrics

In order to test the correctness of implemented metrics, we assess the numerical values for metrics like L1, L2, and RC1 on very small datasets and the results are found correct w.r.t Luzzu. For metrics like I2 and CN2, Luzzu uses approximate values for faster performance, and that is not the same as getting the exact number as in the case of our implementation.

Overall analysis by metrics

We analyze the overall run-time of the metric evaluation. Figure 5.5 reports on the run-time of each metric considered (see Table 5.2) on both $BSBM_{20GB}$ and $BSBM_{200GB}$ datasets.

DistQualityAssessment implements predefined quality assessment metrics from [7]. We have implemented these metrics in a distributed manner such that most of them have a run-time complexity of $O(n)$ where n is the number of input triples. The overall performance of analysis for the BSBM dataset with two instances is shown in Figure 5.5. The results obtained show that the execution is sometimes a little longer when there is a shuffling involved in the cluster compared to when data is processed without movement e.g. Metric L2 and L1. Metric SV3 and CN2 are the most expensive ones in terms of runtime. This is due to the extra overhead caused by extracting the literals for objects, and checking the lexical form of its datatype.

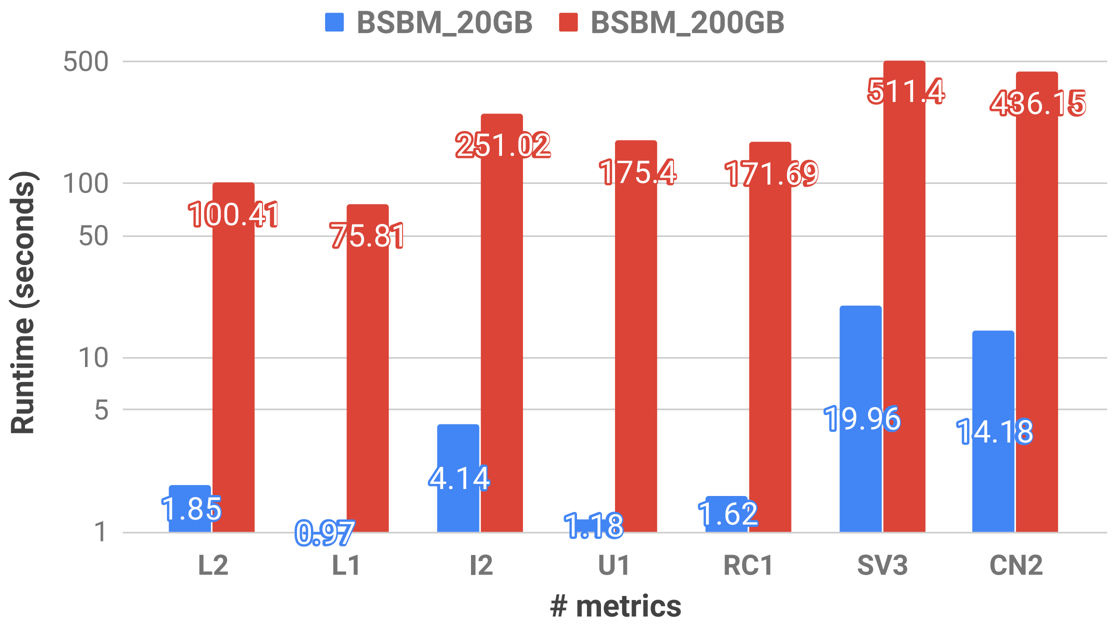


Figure 5.5: **Overall analysis of by metric in the cluster mode (log scale)**. It shows that the execution is sometimes a little longer when there is a shuffling involved in the cluster compared to when data is processed without movement e.g. Metric L2 and L1. Metric SV3 and CN2 are the most expensive ones in terms of runtime. This is due to the extra overhead caused by extracting the literals for objects and checking the lexical form of its datatype.

Overall, the evaluation study carried out demonstrates that distributed computation of different quality measures is scalable and the execution ends in reasonable time given the large volume of data.

5.3 Summary

The data quality assessment becomes challenging with the increasing sizes of data. Many existing tools mostly contain a customized data quality functionality to detect and analyze data quality issues within their own domain. 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 thesis, we have introduced DistQualityAssessment, a novel approach for distributed in-memory evaluation of RDF quality assessment metrics implemented on top of the Spark framework. The presented approach offers generic features to solve common data quality checks. As a consequence, this can enable further applications to build trusted data utilities.

We have demonstrated empirically that our approach improves upon the previous centralized approach that we have compared against. The benefit of using Spark is that its core concepts (RDDs) are designed to scale horizontally. Users can adapt the cluster sizes corresponding to the data sizes, by dropping when it is not needed and adding more when there is a need for it.

CHAPTER 6

Scalable RDF Querying

In recent years, our information society has reached the stage where it produces billions of data records, amounting to multiple quintillions of bytes[1], on a daily basis. Extraction, cleansing, enrichment and refinement of information are key to fuel value-adding processes, such as analytics as a premise for decision making. Devising appropriate (ideally uniform) representations and facilitating efficient querying of data, metadata, and provenance arising from such phases constantly poses challenges, especially when data volumes are vast. The most prominent and promising effort is the W3C consortium with encouraging RDF as a common data representation and vocabularies (e.g. RDFS, OWL) as a way to include meta-information about the data. These data and meta-data can be further processed and analyzed using the de-facto query language for RDF data, SPARQL. SPARQL serves as a standard query language for manipulating and retrieving RDF data.

Querying RDF data becomes challenging when the size of the data increases. This has motivated a considerable amount of work on designing distributed RDF systems able to efficiently evaluate SPARQL queries ([1, 19]). Being able to query a large amount of data in an efficient and faster way is one of the key requirements for every SPARQL engine.

To address these challenges, in this thesis, we propose a scalable RDF querying engine based on two different partitioning strategies. First, Sparklify – SPARQL-to-SQL rewriter based on the vertical partitioning [94] implemented on top of Apache Spark. As a second approach, we investigated and developed the so-called Semantic-based query system. Both approaches are a scalable and efficient evaluation of SPARQL queries over distributed RDF datasets. The main component of the both systems are the data partitioning and query evaluation over this data representation.

In this chapter we address the following research question:

> **RQ3**: Can distributed RDF datasets be queried efficiently and effectively?

Contributions of this chapter are summarized as follows:

- We present a novel approach for vertical partitioning including RDF terms using the distributed computing framework, Apache Spark.
- We developed a scalable query system using Sparqlify – a SPARQL-to-SQL rewriter on top of Apache Spark (under the *Apache Licence 2.0*).

[1] https://www.domo.com/learn/data-never-sleeps-5

- We evaluate Sparklify with state-of-the-art engines and demonstrate it empirically.
- A scalable approach for semantic-based partitioning using the distributed computing framework, Apache Spark.
- A scalable semantic-based query engine (*SANSA.Semantic*) on top of Apache Spark.
- Comparison of the semantic-based system with state-of-the-art engines and demonstrate the performance empirically.
- We integrated the proposed approaches into the SANSA [31][2] larger framework. Sparklify serves as a default query engine in SANSA. SANSA is an active project and maintained, including issue tracker, mailing list, changelogs, website, etc.

This chapter is based on the following publications ([25–27]):

- **Gezim Sejdiu**; Damien Graux; Imran Khan; Ioanna Lytra; Hajira Jabeen; and Jens Lehmann, "Towards A Scalable Semantic-based Distributed Approach for SPARQL query evaluation," 15th International Conference on Semantic Systems (SEMANTiCS), Research & Innovation , 2019.
- Claus Stadler; **Gezim Sejdiu**; Damien Graux; and Jens Lehmann, "Sparklify: A Scalable Software Component for Efficient evaluation of SPARQL queries over distributed RDF datasets," in Proceedings of 18th International Semantic Web Conference (ISWC), 2019. This article is a joint work with Claus Stadler, a PhD student at the University of Leipzig. In this article, I devised the implementation of the conceptual architecture, helped on the implementation of the proposed approach, reviewed related work, and preparation of the experiments and analysis of the obtained results.
- Claus Stadler; **Gezim Sejdiu**; Damien Graux; and Jens Lehmann. "Querying large-scale RDF datasets using the SANSA framework". In Proceedings of 18th International Semantic Web Conference (ISWC), Poster & Demos, 2019. This demonstration article is a joint work with Claus Stadler, a PhD student at the University of Leipzig. In this article, I helped in describing the architecture and implementation of the running example.

The rest of the chapter is structured as follows: Sparklify, a scalable software component for SPARQL evaluation of large RDF data is presented in Section 6.1. Its data modeling, data partitioning, and query translation using a distributed framework (Apache Spark) are detailed in Subsection 6.1.1 and evaluated in Subsection 6.1.2. Second part of the chapter, Section 6.2 elaborate the Semantic-based approach, including its system architecture overview as presented in Section 6.5. The Semantic-based approach is evaluated in Subsection 6.2.3. Finally, we summarize our work in Section 6.3.

6.1 Sparklify: A Scalable Software for SPARQL Evaluation of Large RDF Data

In this section, we present the overall architecture of Sparklify, the SPARQL-to-SQL rewriter, and mapping to Spark Scala-compliant code.

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

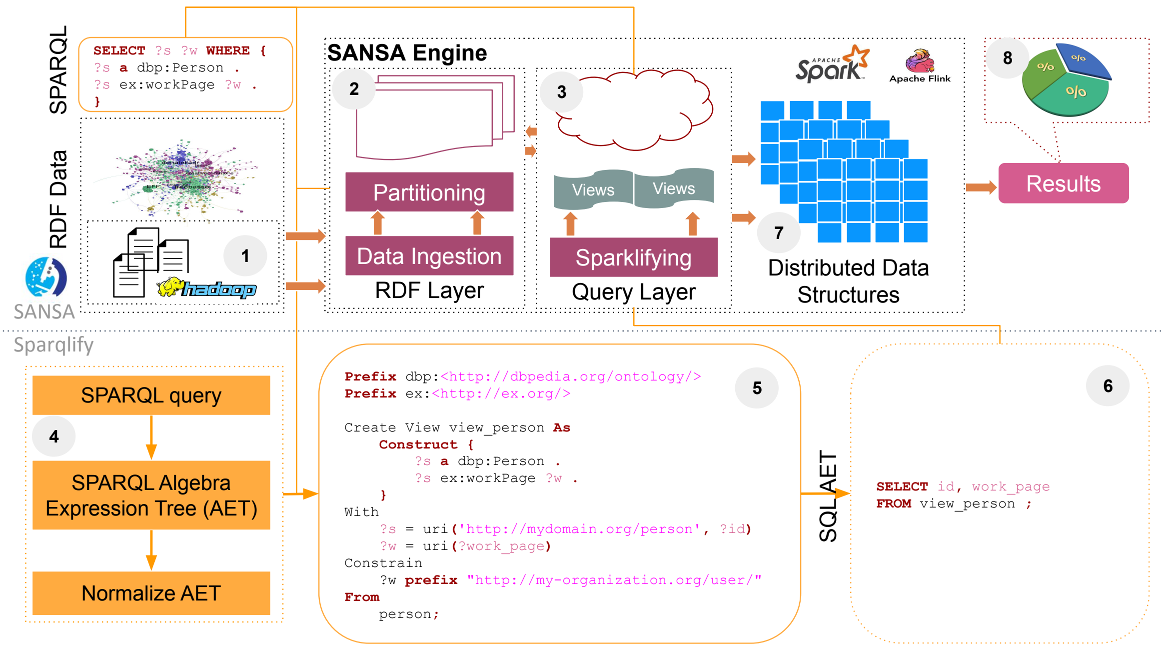


Figure 6.1: **Sparklify Architecture Overview**. It consists of four main components: Data modeling – data ingestion and data partitioning (using the extensible VP), Mappings/Views – the relational-to-RDF mapping, Query Translator – SQL query generator from the SPARQL query, and Query Evaluator - SQL query evaluated directly into the Spark SQL engine.

6.1.1 System Architecture Overview

The overall system architecture is shown in Figure 6.1. It consists of four main components: Data Model, Mappings, Query Translator and Query Evaluator. In the following, each component is discussed in details.

Data Model SANSA [31] comes with different data structures and different partitioning strategies. We model and store RDF graph following the concept of RDDs – a basic building blocks of the Spark Framework. RDDs are in-memory collections of records that are capable of operating in a parallel overall larger cluster. Sparklify makes use of the SANSA bottom layer which corresponds with the extended VP including RDF terms. This partition model is the most convenient storage model for the fast processing of RDF datasets on top of HDFS.

Data Ingestion (Step 1) RDF data first needs to be loaded into a large-scale storage that Spark can efficiently read from. We use HDFS. Spark employ different data locality scheme in order to accomplish computations nearest to the desired data in HDFS, as a result avoiding i/o overhead.

Data Partition (Step 2) VP approach in SANSA is designed to support the 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 [95].

The main entities defined with SML are *view definitions*. See *Step 5* in the Figure 6.1 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[3] 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 International Resource Identifiers (IRI)s with disjoint 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 Evaluator 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 distributed data structure of Spark (e.g. DataFrame) (*Step 7*) which then is mapped into a SPARQL bindings. The result set can further used for analysis and visualization using the SANSA-Notebooks (*Step 8*) [30].

6.1.2 Evaluation

The goal of our evaluation is to observe the impact of the extensible VP as well as analyzing its scalability when the size of the dataset increases. At the same time, we also want to measure the effect of using Sparqlify optimizer for improving query performance. Especially, we want to verify and answer the following questions:

Q1) : Is the runtime affected when more nodes are added in the cluster?

Q2) : Does it scale to a larger dataset?

Q3) : How does it scale when adding a larger number of datasets?

In the following, we present our experiment setting including the benchmarks used and server configurations. Afterword, we elaborate on our findings.

Experimental Setup

We used two well-known SPARQL benchmarks for our evaluation. The *Lehight University Benchmak (LUBM)* v3.1 [96] and *Waterloo SPARQL Diversity Test Suite (WatDiv)* v0.6 [97]. Characteristics of the considered datasets are given in Table 6.1.

[3] https://jena.apache.org/documentation/query/

$\longrightarrow$	LUBM			Watdiv		
	1K	5K	10K	10M	100M	1B
#nr. of triples	138,280,374	690,895,862	1,381,692,508	10,916,457	108,997,714	1,099,208,068
size (GB)	24	116	232	1.5	15	149

Table 6.1: **Summary information of used datasets (nt format)**. Lists dataset characteristics used on the evaluation. The size (in GB) and the number of triples are given.

LUBM comes with a *Data Generator (UBA)* which generates synthetic data over the *Univ-Bench* ontology in the unit of a university. Our *LUBM* datasets consist of 1000, 5000, and 10000 universities. The number of triples varies from 138M for 1000 universities, to 1.4B triples for 10000 universities. *LUBM*'s test suite is comprised of 14 queries.

We have used *WatDiv* datasets with approximate 10K to 1B triples with scale factors 10, 100 and 1000, respectively. *WatDiv* provides a test suite with different query shapes, therefore, it allows us to compare the performance of Sparklify and the other approach we compare within a more compact way. We have generated these queries using the *WatDiv Query Generator* and report the average mean runtime in the overall results presented below. It comes with a set of 20 predefined query templates so-called *Basic Testing Use Case* which is grouped into four categories, based on the query shape : *star (QS)*, *linear (QL)*, *snowflake (QF)*, and *complex (QC)*.

We implemented Sparklify using Spark-2.4.0, Scala 2.11.11, Java 8, and Sparqlify 0.8.3 and all the data were stored on the HDFS cluster using Hadoop 2.8.0. All experiments were carried out on a commodity cluster of 7 nodes (1 master, 6 workers): Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz (32 Cores), 128 GB RAM, 12 TB SATA RAID-5, connected via a Gigabit network. The experiments have been executed three times and the average runtime has been reported into the results.

Results

We evaluate Sparklify using the above datasets and compare it with the chosen state-of-the-art distributed SPARQL query evaluator. Since our approach does not involve any pre-processing of the RDF data before being able to evaluate SPARQL queries on it, Sparklify is thereby closer to the so-called direct evaluators. Indeed, Sparklify only needs to virtually partition the data prior. As a consequence, we omit other distributed evaluators (such as e.g. S2RDF [1]) and compare it with SPARQGX [19] as it outperforms other approaches as noted by Graux et.al [19]. We compare our approach with *SPARQLGX*'s direct evaluator named SDE and report the loading time for partitioning and query execution time, see Table 6.2. We specify "fail" whenever the system fails to complete the task and "n/a" when the task could not be completed due to a failure in one of the intermediate phase. In some cases e.g. in Table 6.2, *QC in Watdiv-1B* dataset, we define "partial fail" due to the failure of one of the queries, therefore the sum-up is not possible.

Findings of the experiments are depicted in Table 6.2, Figure 6.2, 6.3, and 6.4.

To verify Q1, we analyze the *speedup* and compare it with SPARQLGX. We run the experiments on three datasets, *Watdiv-10M*, *Watdiv-1B* and *LUBM-10K*.

Table 6.2 shows the performance analysis of two approaches run on three different datasets. Column SPARQLGX-SDE^a reports on the performance of SPARQLGX-SDE considering the total runtime to evaluate the given queries. Column Sparklify^b lists the times required for Sparklify to perform the VP

		Runtime (s) (mean)			
⟶		**SPARQLGX-SDE**	**Sparklify**		
		a) total	b) paritioning	c) querying	d) total
Watdiv-10M	QC	103.24	134.81	61	195.84
	QF	157.8	241.24	107.33	349.51
	QL	102.51	236.06	134	370.3
	QS	131.16	237.12	108.56	346
Watdiv-1B	QC	partial fail	778.62	2043.66	2829.56
	QF	6734.68	1295.31	2576.52	3871.97
	QL	2575.72	1275.22	610.66	1886.73
	QS	4841.85	1290.72	1552.05	2845.3
LUBM-10K	$Q1$	1056.83	627.72	718.11	1346.8
	$Q2$	fail	595.76	fail	n/a
	$Q3$	1038.62	615.95	648.63	1267.37
	$Q4$	2761.11	632.93	1670.18	2303.18
	$Q5$	1026.94	641.53	564.13	1206.67
	$Q6$	537.65	695.74	267.48	963.62
	$Q7$	2080.67	630.44	1331.13	1967.25
	$Q8$	2636.12	639.93	1647.57	2288.48
	$Q9$	3124.52	583.86	2126.03	2711.24
	$Q10$	1002.56	593.68	693.73	1287.71
	$Q11$	1023.32	594.41	522.24	1118.58
	$Q12$	2027.59	576.31	1088.25	1665.87
	$Q13$	1007.39	626.57	6.66	633.26
	$Q14$	526.15	633.39	258.32	891.89

Table 6.2: **Performance analysis on large-scale RDF datasets**. Comparison analysis of Sparklify as compared with *SPARQLGX*'s direct evaluator named SDE. The loading time for partitioning and query execution time is reported.

and then the query execution time is reported on the Sparklify^c. Total runtime for Sparklify is shown in the last column, Sparklify^d.

We observe that the execution of both approaches fails for the *Q2* in the *LUBM-10K* dataset while evaluating the query. We believe that it is due to the reason that *LUBM Q2* involves a triangular pattern which is often resource consuming. As a consequence, in both cases, Spark performs the shuffling (e.g. data scanning) while reducing the result set. It is interesting to note that for the *Watdiv-1B* dataset, SPARQLGX-SDE fails for the query *C3* when data scanning is performed. Sparklify is capable of evaluating it successfully. Due to the Spark SQL optimizer in conjunction with Sparqlify's approach of rewriting a SPARQL query typically into only a single SQL query – effectively offloading all query planning to Spark – Sparklify performs better than SPARQLGX-SDE when the size of the dataset increases (see *Watdiv-1B results* in the Table 6.2) and when there are more joins involved

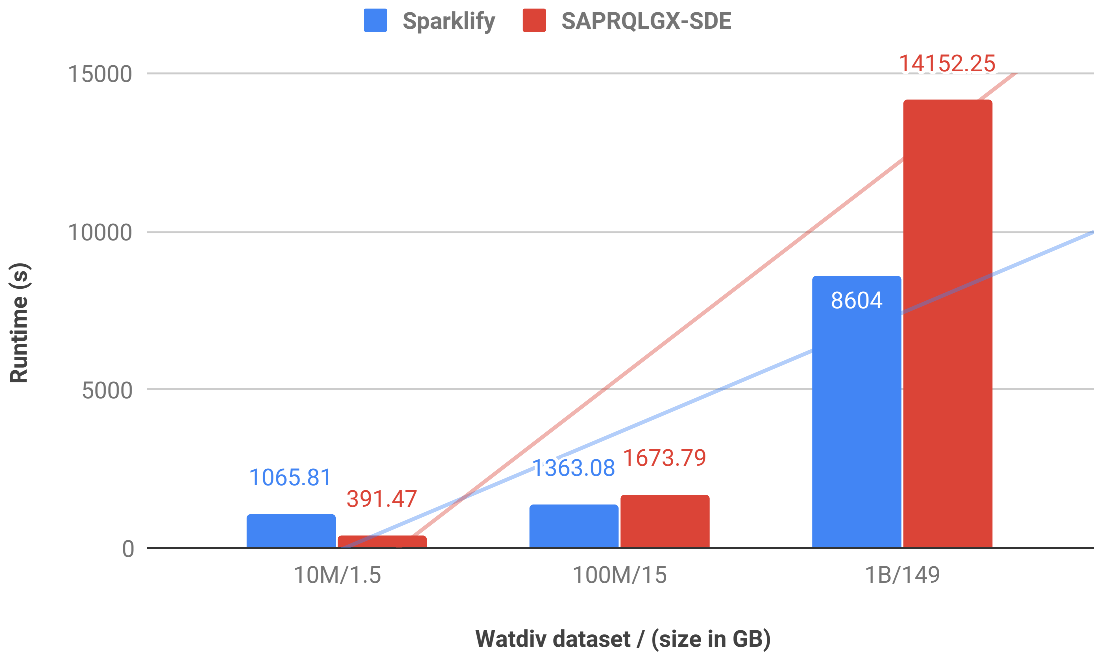


Figure 6.2: **Sizeup analysis (on Watdiv dataset)**. The analysis keeps the number of nodes constant i.e. 6 worker nodes and grow the size of the dataset (Watdiv) in order to measure whether the approaches chosen for evaluation can deal with larger datasets. As depicted, the execution time for Sparklify grows linearly as compared with SPARQLGX-SDE, and keep staying near-linear when the size of the dataset increases.

(see *Watdiv-1B* and *LUBM-10K* results in the Table 6.2). SPARQLGX-SDE evaluates the queries faster when the size of the datasets is smaller, but it degrades when the size of the dataset increases. The likely reason for Sparklify's worse performance on smaller datasets is its higher partitioning overhead. Figure 6.2 shows that Sparklify starts outperforming when the size of the datasets grows (e.g. *Watdiv-100M*).

Size-up scalability analysis To measure the performance of the data scalability (e.g. size-up) of both approaches, we run experiments on three different sizes of *Watdiv* (see Figure 6.2).

We keep the number of nodes constant i.e 6 worker nodes and grow the size of the datasets to measure whether both approaches can deal with larger datasets. We see that the execution time for Sparklify grows linearly compared with SPARQLGX-SDE, which keeps staying as near-linear when the size of the datasets increases. The results presented show the scalability of Sparklify in the context of the sizeup, which addresses the question Q2.

Node scalability analysis To measure the node scalability of Sparklify, we vary the number of worker nodes. We vary them from 1, 3 to 6 worker nodes.

Figure 6.3 depict the speedup performance of both approaches run on *Watdiv-100M* dataset when the number of worker nodes varies. We can see that as the number of nodes increases, the runtime cost for the Sparklify decreases linearly. The execution time for Sparklify decreases about 0.6 times (from 2547.26 seconds down to 1588.4 seconds) as worker nodes increase from one to three nodes. We see that the speedup stays constant when more worker nodes are added since the size of the data is not that

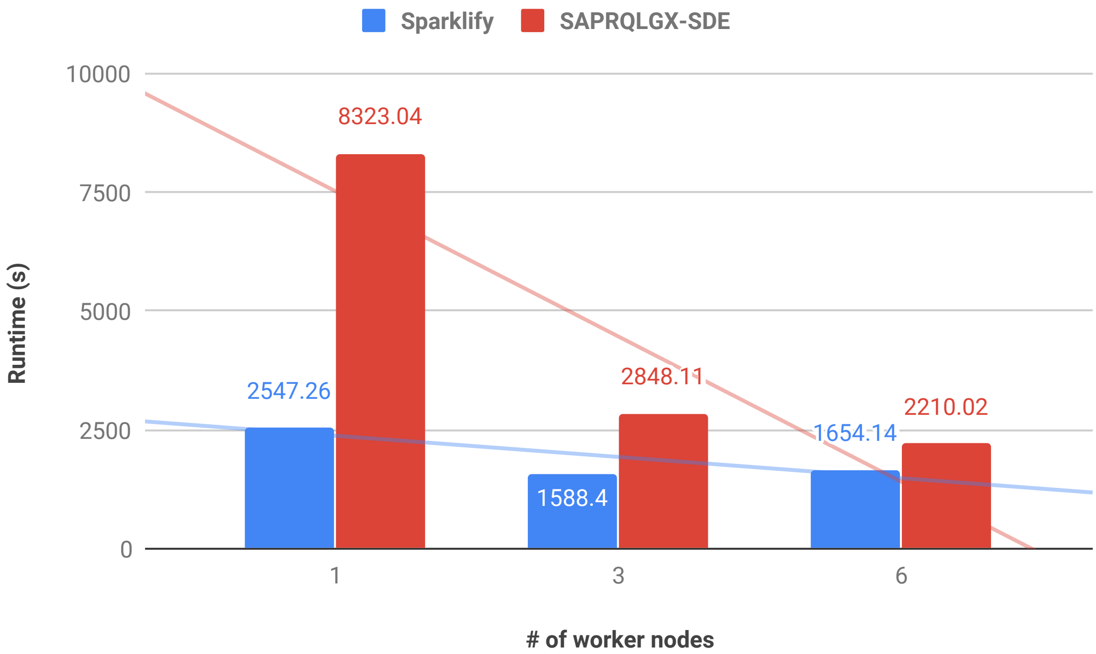


Figure 6.3: **Node scalability (on Watdiv-100M)**. The analysis varies the number of worker nodes e.g. from 1, 3, to 6 worker nodes and keeps the size of the dataset constant i.e. *Watdiv-100M*. It shows that as the number of nodes increases, the runtime cost for Sparklify decreases linearly. It decreases about 0.6 times (from 2547.26 seconds down to 1588.4 seconds) as worker nodes increase from one to three nodes.

large and the network overhead increases a little the runtime when it runs over six worker nodes. This implies that our approach is efficient up to three worker nodes for the *Watdiv-100M* (15GB) dataset. In another hand, SPARQLGX-SDE takes longer to evaluate the queries when running on one worker node but it improves when the number of worker nodes increases.

Result presented here shows that Sparklify can achieve linear scalability in the performance, which addresses Q3.

Correctness of the result set In order to assess the correctness of the result set, we computed the count of the result set for the given queries and compare it within both approaches. We conclude that both approaches return exactly the same result set which implies the correctness of the results.

Overall analysis by SPARQL queries Here we analyze Watdiv queries run on *Watdiv-100M* dataset in a cluster mode on both approaches.

According to Figure 6.4, SPARQLGX-SDE performance decreases as the number of triple patterns involved in the query increase. This might be due to the fact that SPARQLGX-SDE has to read the whole triple file each time. In contrast to SPARQLGX-SDE, Sparklify seems to perform well when there are more triple patterns involved (see queries *QC*, *QF* and *QS* in the Figure 6.4) but slightly worst when there are linear queries (see *QL*) evaluated. This may be due to the reason that Sparqlify typically rewrites a SPARQL query into a single SQL query, thus maximizing the opportunities given to the Spark SQL optimizer. Conversely, SPARQLGX-SDE constructs the workflow by chaining Scala API calls, which may restrict the possibilities e.g. in regard to join ordering. Based on our findings

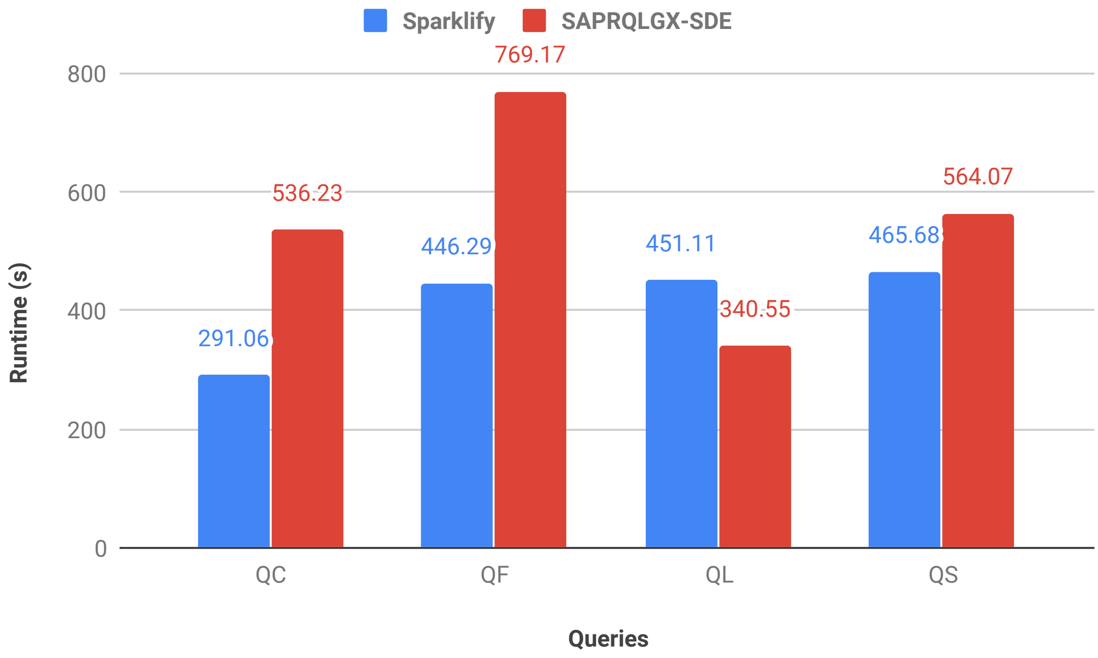


Figure 6.4: **Overall analysis of queries on the Watdiv-100M dataset (cluster mode)**. This analysis gives more insights about running Watdiv queries on *Watdiv-100M* dataset in a cluster mode on both approaches, Sparklify and SPARQLGX-SDE. The findings show that SPARQLGX-SDE performance decreases as the number of triple patterns involved in the query increase. In contrast to SPARQLGX-SDE, Sparklify seems to perform well when there are more triple patterns involved (i.e. *QC*, *QF* and *QS*) but slightly worst when there are linear queries (see *QL*) evaluated.

and the evaluation study carried out, we show that Sparklify is scalable and the execution time ends in a reasonable time given the size of the dataset.

6.2 A Scalable Semantic-Based Distributed Approach for SPARQL Query Evaluation

In this section, we present the system architecture of the Semantic-based approach, the semantic-based partitioning, and mapping SPARQL to Spark Scala-compliant code.

6.2.1 System Architecture Overview

The system architecture overview is shown in Figure 6.5.

It consists of three main facets: Data Storage Model, SPARQL Query Fragments Translator, and Query Evaluator. Below, each facet is discussed in more details.

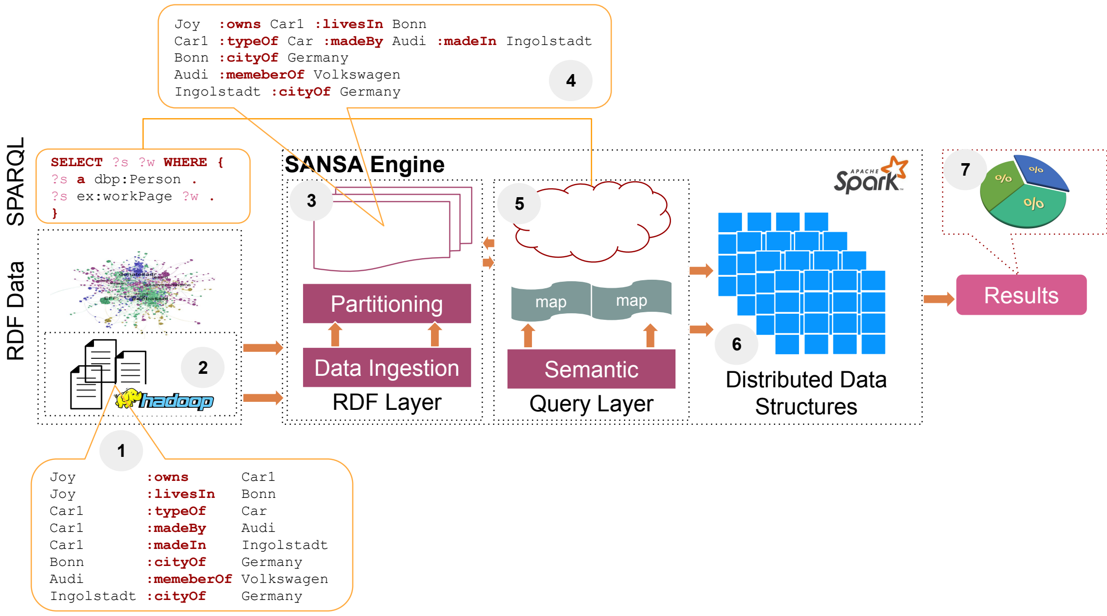


Figure 6.5: **Semantic-based System Architecture Overview**. It consists of three main facets: Data Storage Model – model and partition the data using the semantic-based approach, SPARQL Query Fragments Translator – the process of generating the Scala code in the format of Spark RDD operations, and Query Evaluator – the SPARQL evaluation using the Spark RDD executable code (generated from the previous step).

Data Storage Model

We model the RDF data following the concept of RDDs. RDDs are immutable collections of records, which represent the basic building blocks of the Spark framework. RDDs can be kept in-memory and are able to operate in parallel throughout the Spark cluster. We make use of SANSA [31]'s data representation and distribution layer for such representation.

Data Partitioning 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 for improving the system availability, load balancing and query processing time. There are many different data partitioning techniques proposed in the literature. We choose to investigate the so-called *semantic-based partitioning* behaviors when dealing with large-scale RDF datasets. This partitioned technique was proposed in the SHARD [87] system. We have implemented this technique using in-memory processing engine, Apache Spark for better performance. 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.

Data Model First, the RDF data (see *Step 1* as an example) needs to be loaded into a large-scale distributed storage (*Step 2*). We use HDFS. We choose HDFS as Spark is capable of performing operations based on data locality in order to choose the nearest data for faster and efficient computation over the cluster. Second, we partition (*Step 3*) the data using semantic-based partitioning (see *Step 4* as an example of such partition). Instead of working with table-wise representation where the triples are kept in the format of `RDD<Triple>`, data is partitioned into subject-based grouping (e.g. all entities which are associated with a unique subject). Consider the example in the Figure 6.5 (*Step 2*, first line), which represents two triples associated with the entity `Joy`:

```
Joy :owns Car1 :livesIn Bonn
```

This line represents that the entity `Joy` owns a car entity `Car1`, and that `Joy` lives in `Bonn`.

Often flattening data is considered immature with respect to other data representation, we want to explore and investigate if it improves the performance of the query evaluation. We choose this representation for the reason of easy-storage and reuse while designing a query engine. Although, it slightly degrades the performance when it comes to multiple scans over the table when there are multiple predicates involved in the query. However, this is minimal, as Spark uses in-memory, caching operations. We will discuss this in Section 6.2.3 into more detail.

SPARQL Query Fragments Translation

This process generates the Scala code in the format of Spark RDD operations using the key-value pairs mechanism. With Spark pairRDD, one can manipulate the data by splitting it into key-value pairs and group all associated values with the same keys. It walks through the SPARQL query (*Step 4*) using the Jena ARQ[4] and iterate through clauses in the SPARQL query and bind the variables into the RDF data while fulfilling the clause conditions. Such iteration corresponds to a single clause with one of the Spark operations (e.g. *map*, *filter*, *reduce*). Often this operation needs to be materialized i.e the result set of the next iteration depends on the previous clauses and therefore a *join* operation is needed. This is a bottleneck since scanning and shuffling is required. In order to keep these joins as small as possible, we leverage the caching techniques of the Spark framework by keeping the intermediate results in-memory while the next iteration is performed. Finally, the Spark-Scala executable code is generated (*Step 5*) using the bindings corresponding to the query. Besides simple BGP translation, our system supports ***UNION***, ***LIMIT*** and ***FILTER*** clauses.

Query Evaluator

The mappings created as shown in the previous section can now be evaluated directly into the Spark RDD executable code. The result set of these operations is distributed data structure of Spark (e.g. RDD) (*Step 6*). The result set can be used for further processing and visualization using the SANSA-Notebooks (*Step 7*) [30].

6.2.2 Distributed Algorithm Description

We implement our approach using the Apache Spark framework (see Algorithm 3). It constructs the graph (*Line 1*) while reading RDF data and converts it into an RDD of triples. Later, it partitions the data (*Line 2*, for more details see Algorithm 4) using the semantic-based partitioning strategy. Finally, the query evaluator is constructed (*Line 3*) which is detailed in Algorithm 5.

The partition algorithm (see Algorithm 4) transforms the RDF graph into a convenient semantic-based partitioning (*Line 2*). For each unique triple in the graph in a distributed fashion, it does the

[4] https://jena.apache.org/documentation/query/

Algorithmus 3 : Spark parallel semantic-based query engine.

```
  input  : q: a SPARQL query, input: an RDF dataset
  output : result an RDD – list of result set
  /* Loading the graph                                              */
1 graph = spark.rdf(lang)(input)
  /* Partitioning the graph.  See Algorithm 4 for more details.     */
2 partitionGraph ← graph.partitonAsSemanticGraph()
  /* Querying the graph.  See Algorithm 5 for more details.         */
3 result ← partitionGraph.sparql(q)
4 return result
```

following: It gets the values about subjects and objects (*Line 3*) and local name of the predicate (*Line 4*). It generates the key-value pairs of the subject and its associated triples with predicate and objects separated with space in between (*Line 5*). After the mapping is done, the data is grouped by key (in our case *subject*) (*Line 6*). Afterward, when this information is collected, the block is partitioned using the *map* transformation function of Spark to refactor the format of the lines based on the above information (*Line 7*).

Algorithmus 4 : partitonAsSemanticGraph: Semantic-based partition algorithm.

```
  input  : graph: an RDD of triples
  output : partionedData: an RDD of partitions
1 partitonedData ← ∅
2 foreach ∀!triple ∈ graph  &&  triple.getSubject ≠ ∅ do
3   | s ← triple.getSubject;   o ← triple.getObject
4   | p ← triple.getPredicate.getLocalName
5   | partitonedData + = (s, p + " " + o + " ")
6 partitonedData.reduceByKey(_ + _)
7               .map(f → (f._1 + " " + f._2))
8 return partitonedData
```

This SPARQL query rewriter includes multiple Spark operations. First, partitioned data is mapped to a list of variable bindings satisfying the first BGP of the query (*Line 2*). During this process, the duplicates are removed and the intermediate result is kept in-memory (RDD) with the variable bindings as a key. The consequent step is to iterate through other variables and bind them by processing the upcoming query clauses and/or filtering the other ones unseen on the new clause. These intermediate steps perform Spark operations over both, the partitioned data and the previously bound variables which were kept on Spark RDDs.

The ith step discovers all variables in the partitioned data which satisfy the ith clause appeared and keep this intermediate result in-memory with the key being any variable in the ith step which has been introduced on the previous step. During this iteration, the intermediate results are reconstructed in the way that the variables not seen in this iteration are mapped (*Line 5*) with the variables of the previous clause and generate a key-value pair of variable bindings. Afterward, the *join* operation is performed over the intermediate results from the previous clause and the new ones with the same key.

This process iterates until all clauses are seen and variables are assigned. Finally, the variable binding (*Line 7*) to fulfill the *SELECT* clause of the SPARQL query happens and returns the result (*Line 8*) of only those variables which are present in the SELECT clause.

Algorithmus 5 : sparql: Semantic-based query algorithm.

```
  input  : partitonedData: an RDD of partitions
  output : result an RDD of result set
1 foreach p ∈ partitionedData do
2     1stVariable ← assignVariablesFor1stClaues()
3     foreach i ∈ getClauses() do
4         iVariable ← assignVariablesForiClaues()
5         mapResult ← mapByKey(getCommonVariables())
6         joinResult ← join(mapResult)
7     joinResult.filter(getSelectVariables())
8     result ← result.join(joinResult)
9 return result
```

6.2.3 Evaluation

In our evaluation, we observe the impact of semantic-based partitioning and analyze the scalability of our approach when the size of the dataset increases.

In the following subsections, we present the benchmarks used along with the server configuration setting, and finally, we discuss our findings.

Experimental Setup

We make use of two well-known SPARQL benchmarks for our experiments: the *Waterloo SPARQL Diversity Test Suite (WatDiv)* v0.6 [97] and *Lehigh University Benchmark (LUBM)* v3.1 [96]. The dataset characteristics of the considered benchmarks are given in Table 6.3.

WatDiv comes with a test suite with different query shapes which allows us to compare the performance of our approach and the other approaches. In particular, it comes with a predefined set of 20 query templates which are grouped into four categories, based on the query shape: *star-shaped* queries, *linear-shaped* queries, *snowflake-shaped* queries, and *complex-shaped* queries. We have used *WatDiv* datasets with 10M to 100M triples with scale factors 10 and 100, respectively. In addition, we have generated the SPARQL queries using *WatDiv Query Generator*.

LUBM comes with a *Data Generator (UBA)* which generates synthetic data over the *Univ-Bench* ontology in the unit of a university. *LUBM* provides Test Queries, more specifically 14 test queries. Our *LUBM* datasets consist of 1000, 2000, and 3000 universities. The number of triples varies from 138M for 1000 universities, to 414M triples for 3000 universities.

We implemented our approach using Spark-2.4.0, Scala 2.11.11, Java 8, and all the data were stored on the HDFS cluster using Hadoop 2.8.0. All experiments were carried out on a commodity cluster of 6 nodes (1 master, 5 workers): Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz (32 Cores), 128

	LUBM			Watdiv	
	1K	2K	3K	10M	100M
#nr. of triples	138,280,374	276,349,040	414,493,296	10,916,457	108,997,714
size (GB)	24	49	70	1.5	15

Table 6.3: **Dataset characteristics (nt format)**. Lists dataset information used on the evaluation. The size (in GB) and the number of triples are given.

GB RAM, 12 TB SATA RAID-5. We executed each experiment three times and the average query execution time has been reported.

Preliminary Results

We run experiments on the same cluster and evaluate our approach using the above benchmarks. In addition, we compare our proposed approach with selected state-of-the-art distributed SPARQL query evaluators. In particular, we compare our approach with SHARD [87] – the original approach implemented on Hadoop MapReduce, *SPARQLGX* [19]'s direct evaluator SDE, and Sparklify [25] and report the query execution time (cf. Table 6.4). We have selected these approaches as they do not include any pre-processing steps (e.g. statistics) while evaluating the SPARQL query, similar to our approach.

Our evaluation results for performance analysis, sizeup analysis, node scalability, and breakdown analysis by SPARQL queries are shown in Table 6.4, Figure 6.6, 6.7, and 6.8 respectively. In Table6.4 we use "fail" whenever the system fails to complete the task and "n/a" when the task could not be completed due to a parser error (e.g. not able to translate some of the basic patterns to RDDs operations).

In order to evaluate our approach with respect to the *speedup*, we analyze and compare it with other approaches.

This set of experiments was run on three datasets, *Watdiv-10M*, *Watdiv-100M* and *LUBM-1K*.

Table 6.4 presents the performance analysis of the systems on three different datasets. We can see that our approach evaluates most of the queries as opposed to SHARD. SHARD system fails to evaluate most of the *LUBM* queries and its parser does not support *Watdiv* queries. On the other hand, SPARQLGX-SDE performs better than both Sparklify and our approach, when the size of the dataset is considerably small (e.g. less than 25GB). This behavior is due to the large partitioning overhead for Sparklify and our approach. However, Sparklify performs better compared to SPARQLGX-SDE when the size of the dataset increases (see *Watdiv-100M* results in the Table 6.4) and the queries involve more joins (see *LUBM-1K* results in the Table 6.4). This is due to the Spark SQL optimizer and Sparqlify self-joins optimizers. Both SHARD and SPARQLGX-SDE fail to evaluate query *Q2* in the *LUBM-1K* dataset. Sparklify can evaluate the query but takes longer as compared to our approach. This is due to the fact that our approach uses Spark's lazy evaluation and join optimization by keeping the intermediate results in memory.

Scalability analysis In order to evaluate the scalability of our approach, we conducted two sets of experiments. First, we measure the data scalability (e.g. size-up) of our approach and position it with other approaches. As SHARD fails for most of the LUBM queries, we omit other queries on this set

	Queries	Runtime (s) (mean)			
		SHARD	**SPARQLGX-SDE**	**SANSA.Sparklify**	**SANSA.Semantic**
Watdiv-10M	$C3$	n/a	38.79	72.94	90.48
	$F3$	n/a	38.41	74.69	n/a
	$L3$	n/a	21.05	73.16	72.84
	$S3$	n/a	26.27	70.1	79.7
Watdiv-100M	$C3$	n/a	181.51	96.59	300.82
	$F3$	n/a	162.86	91.2	n/a
	$L3$	n/a	84.09	82.17	189.89
	$S3$	n/a	123.6	93.02	176.2
LUBM-1K	$Q1$	774.93	103.74	103.57	226.21
	$Q2$	fail	fail	3348.51	329.69
	$Q3$	772.55	126.31	107.25	235.31
	$Q4$	988.28	182.52	111.89	294.8
	$Q5$	771.69	101.05	100.37	226.21
	$Q6$	fail	73.05	100.72	207.06
	$Q7$	fail	160.94	113.03	277.08
	$Q8$	fail	179.56	114.83	309.39
	$Q9$	fail	204.62	114.25	326.29
	$Q10$	780.05	106.26	110.18	232.72
	$Q11$	783.2	112.23	105.13	231.36
	$Q12$	fail	159.65	105.86	283.53
	$Q13$	778.16	100.06	90.87	220.28
	$Q14$	688.44	74.64	100.58	204.43

Table 6.4: **Performance analysis on large-scale RDF datasets**. A comparison of our approach with SHARD – the original approach implemented on Hadoop MapReduce, SPARQLGX's direct evaluator SDE, and Sparklify w.r.t query execution time.

of experiments and choose only Q1, Q5, and Q14. Q1 has been chosen due to its complexity while bringing large inputs of the data and high selectivity, Q5 since it has considerably larger intermediate results due to the triangular pattern in the query, and Q14 mainly for its simplicity. We run experiments on three different sizes of *LUBM* (see Figure 6.6).

We keep the number of nodes constant i.e. 5 worker nodes and increase the size of the datasets to measure whether our approach deals with larger datasets.

We see that the query execution time for our approach grows linearly when the size of the datasets increases. This shows the scalability of our approach as compared to SHARD, in the context of the sizeup. SHARD suffers from the expensive overhead of MapReduce joins which impacts its performance, as a result, it is significantly worse than other systems.

Second, in order to measure the node scalability of our approach, we increase the number of worker nodes and keep the size of the dataset constant. We vary them from 1, 3 to 5 worker nodes.

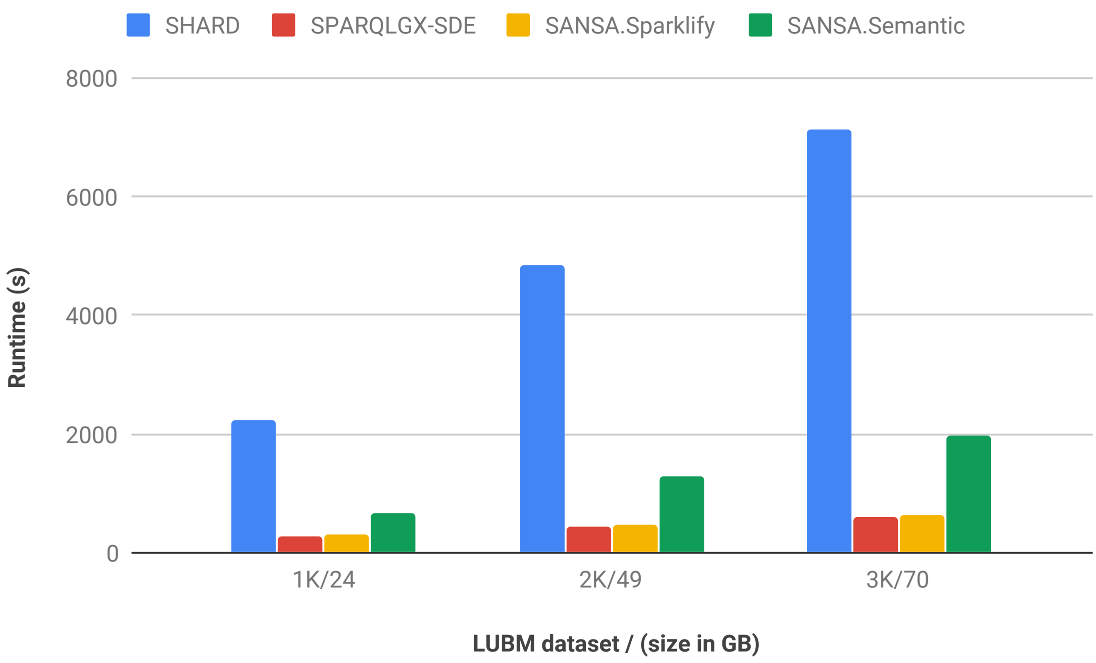


Figure 6.6: **Sizeup analysis (on LUBM dataset)**. The analysis keeps the number of nodes constant i.e. 5 worker nodes and increases the size of the datasets to measure whether a semantic-based approach deals with larger datasets. The query execution time for our approach grows linearly when the size of the datasets increases. This shows the scalability of our approach as compared to SHARD, in the context of the sizeup. SHARD suffers from the expensive overhead of MapReduce joins which impacts its performance, as a result, it is significantly worse than other systems.

Figure 6.7 shows the performance of systems on *LUBM-1K* dataset when the number of worker nodes varies. We see that as the number of nodes increases, the runtime cost of our query engine decreases linearly as compared with the SHARD, which keeps staying constant. SHARD performance stays constant (high) even when more worker nodes are added. This trend is due to the communication overhead SHARD needs to perform between map and reduce steps. The execution time of our approach decreases about 1.7 times (from 1,821.75 seconds down to 656.85 seconds) as the worker nodes increase from one to five nodes. SPARQLGX-SDE and Sparklify perform better when the number of nodes increases compared to our approach and SHARD.

Our main observation here is that our approach can achieve linear scalability in the performance.

Correctness In order to assess the correctness of the result set, we computed the count of the result set for the given queries and compare it with other approaches. As a result of it, we conclude that all approaches return exactly the same result set. This implies the correctness of the results.

Breakdown by SPARQL queries Here we analyze some of the LUBM queries (Q1, Q5, Q14) run on a *LUBM-1K* dataset in a cluster mode on all the systems.

We can see from Figure 6.8 that our approach performs better compared to the Hadoop-based system,

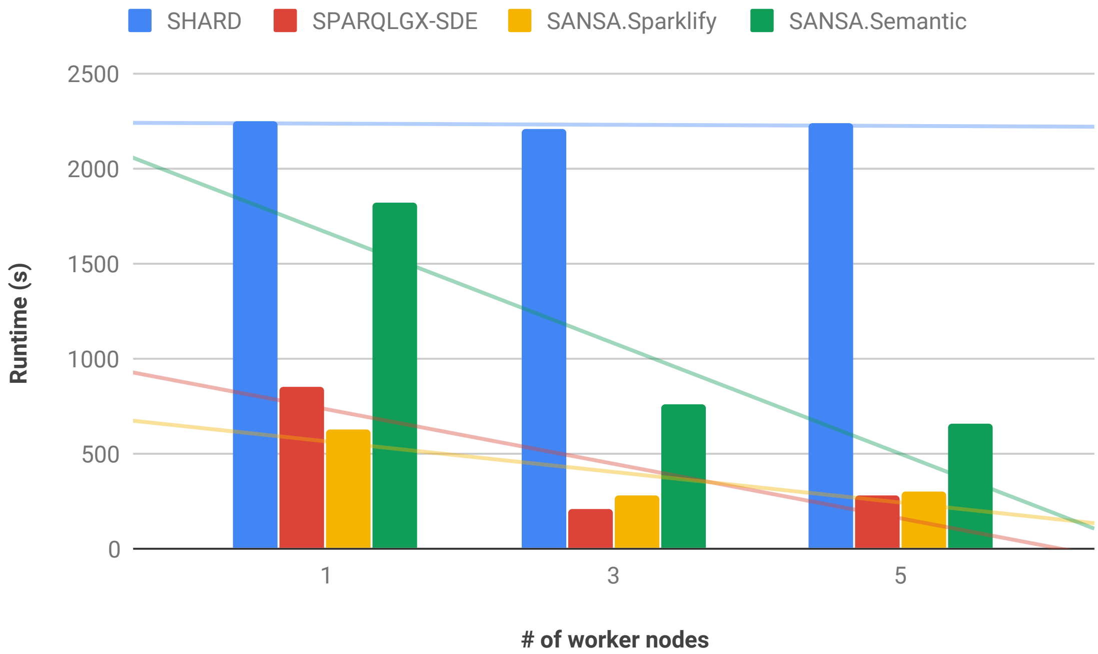


Figure 6.7: **Node scalability (on LUBM-1K)**. The analysis increases the number of worker nodes and keeps the size of the dataset constant. We vary them from 1, 3 to 5 worker nodes. As the number of nodes increases, the runtime cost of our query engine decreases linearly as compared with the SHARD, which keeps staying constant. SHARD performance stays constant (high) even when more worker nodes are added. This trend is due to the communication overhead SHARD needs to perform between map and reduce steps. The execution time of our approach decreases about 1.7 times (from 1,821.75 seconds down to 656.85 seconds) as the worker nodes increase from one to five nodes.

SHARD. This is due to the use of the Spark framework which leverages the in-memory computation for faster performance. However, the performance declines as compared to other approaches that use vertical partitioning (e.g., SPARQLGX-SDE on RDD and Sparklify on Spark SQL). This is due to the fact that our approach performs de-duplication of triples that involves shuffling and incurs network overhead. The results show that the performance of SPARQLGX-SDE decreases as the number of triple patterns involved in the query increases (see *Q5*) when compared to Sparklify. However, SPARQLGX-SDE performs better when there are simple queries (see *Q14*). This occurs because SPARQLGX-SDE must read the whole RDF graph each time when there is a triple pattern involved. In contrast to SPARQLGX-SDE, Sparklify performs better when there are more triple patterns involved (see *Q5*) but slightly worse when linear queries (see *Q14*) are evaluated.

Based on our findings and the evaluation study carried out, we show that our approach can scale up with the increasing size of the dataset.

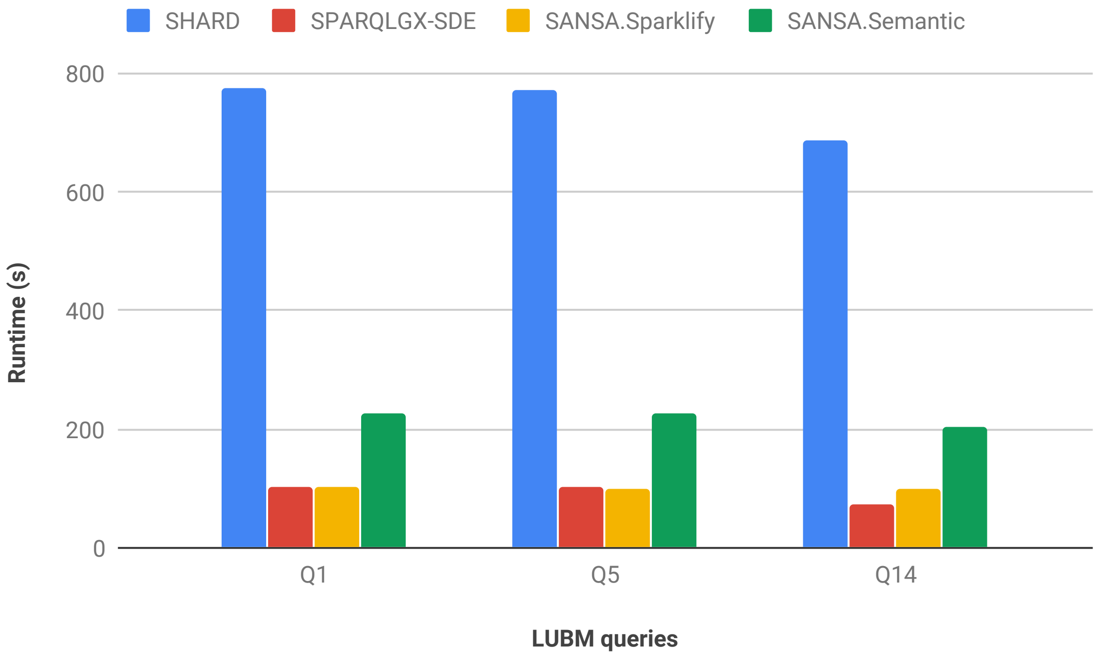


Figure 6.8: **Overall analysis of queries on the LUBM-1K dataset (cluster mode)**. This analysis depicts some of LUBM queries (Q1, Q5, Q14) run on a *LUBM-1K* dataset in a cluster mode on all the systems. Overall, our approach performs better compared to the Hadoop-based system, SHARD due to the use of the Spark framework which leverages the in-memory computation for faster performance. However, the performance declines as compared to other approaches that use vertical partitioning (e.g., SPARQLGX-SDE on RDD and Sparklify on Spark SQL). This is due to the fact that our approach performs de-duplication of triples that involves shuffling and incurs network overhead.

6.3 Summary

Querying RDF data becomes challenging when the size of the data increases. Existing Spark-based SPARQL systems mostly do not retain all RDF term information consistently while transforming them into a dedicated storage model such as using vertical partitioning. Often, this process is both data and computing-intensive and raises the need for a scalable, efficient and comprehensive query engine that can handle large scale RDF datasets.

In this chapter, we propose scalable approaches for SPARQL query evaluation over distributed RDF data. First, *Sparklify*: a scalable software component for efficient evaluation of SPARQL queries over distributed RDF datasets. It uses Sparqify as a SPARQL-to-SQL rewriter for translating SPARQL queries into Spark executable code. By doing so, it leverages the advantages of the Spark framework. SANSA features methods to execute SPARQL queries directly as part of Spark workflows instead of writing the code corresponding to those queries (sorting, filtering, etc.). It also provides a command-line interface and a W3C standard-compliant SPARQL endpoint for externally querying data that has been loaded using the SANSA framework. We have shown empirically that Sparklify can scale horizontally and perform well w.r.t to the state-of-the-art approaches.

With this work, we showed that the application of OBDA tooling to Big Data frameworks achieves promising results in terms of scalability. We present a working prototype implementation that can serve as a baseline for further research.

As a second approach, we investigated and implemented a scalable semantic-based query engine for efficient evaluation of SPARQL queries over distributed RDF datasets. It uses a semantic-based partitioning strategy as the data distribution and converts SPARQL to Spark executable code. By doing so, it leverages the advantages of the Spark framework's rich APIs. We have shown empirically that a semantic-based approach can scale horizontally and perform well as compared with the previous Hadoop-based system: the SHARD triple store. It is also comparable with other in-memory SPARQL query evaluators when there is less shuffling involved i.e. less duplicate values.

CHAPTER 7

Implementation and Use Cases

In this chapter, we give a more detailed overview of the SANSA framework and the components developed during this thesis. It also shows how they can be applied to various use cases.

The chapter is organized as follows: First, in Section 7.1, we give an overview of the SANSA framework, which contains the implementation of the methods presented in this thesis. Later, we demonstrate the use of our components in real use cases in Section 7.2.

This chapter is based on the following publications [31]:

- Danning Sui; **Gezim Sejdiu**; Damien Graux; and Jens Lehmann. "The Hubs and Authorities Transaction NetworkAnalysis using the SANSA framework". In 15th International Conference on Semantic Systems (SEMANTiCS), Poster & Demos, 2019.
- Rajjat Dadwal; Damien Graux; **Gezim Sejdiu**; Hajira Jabeen; and Jens Lehmann. "Clustering Pipelines of large RDF POI Data" in Proceedings of 16th Extended Semantic Web Conference (ESWC), Poster & Demos, 2019.
- Damien Graux; **Gezim Sejdiu**; Hajira Jabeen; Jens Lehmann; Danning Sui; Dominik Muhs; and Johannes Pfeffer, "Profiting from Kitties on Ethereum: Leveraging Blockchain RDF with SANSA," in 14th International Conference on Semantic Systems, Poster & Demos, 2018.
- Jens Lehmann; **Gezim Sejdiu**; Lorenz Bühmann; Patrick Westphal; Claus Stadler; Ivan Ermilov; Simon Bin; Nilesh Chakraborty; Muhammad Saleem; Axel-Cyrille Ngomo Ngonga; and Hajira Jabeen, "Distributed Semantic Analytics using the SANSA Stack,"; in Proceedings of 16th International Semantic Web Conference - Resources Track (ISWC'2017), 2017.
- Ivan Ermilov; Jens Lehmann; **Gezim Sejdiu**; Lorenz Bühmann; Patrick Westphal; Claus Stadler; Simon Bin; Nilesh Chakraborty; Henning Petzka; Muhammad Saleem; Axel-Cyrille Ngomo Ngonga; and Hajira Jabeen, "The Tale of Sansa Spark," in Proceedings of 16th International Semantic Web Conference, Poster & Demos, 2017 (**Best Demo Award**). This demonstration article is joint work with Ivan Ermilov, a PhD student at the University of Leipzig. In this article, I helped in describing the architecture, implementation of the examples and demonstration of the prototype.
- Ivan Ermilov; Axel-Cyrille Ngonga Ngomo; Aad Versteden; Hajira Jabeen; **Gezim Sejdiu**; Giorgos Argyriou; Luigi Selmi; Jürgen Jakobitsch; and Jens Lehmann, "Managing Lifecycle of

Big Data Applications,"; in KESW, 2017. This article is a joint work with Ivan Ermilov, a PhD student at the University of Leipzig. In this article, I helped with the implementation of the proposed approach and SC4 (Transport) use case, reviewed related work, and preparation of the experiments and analysis of the obtained results.

- Sören Auer; Simon Scerri; Aad Versteden; Erika Pauwels; Angelos Charalambidis; Stasinos Konstantopoulos; Jens Lehmann; Hajira Jabeen; Ivan Ermilov; **Gezim Sejdiu**; Andreas Ikonomopoulos; Spyros Andronopoulos; Mandy Vlachogiannis; Charalambos Pappas; Athanasios Davettas; Iraklis A. Klampanos; Efstathios Grigoropoulos; Vangelis Karkaletsis; Victor Boer; Ronald Siebes; Mohamed Nadjib Mami; Sergio Albani; Michele Lazzarini; Paulo Nunes; Emanuele Angiuli; Nikiforos Pittaras; George Giannakopoulos; Giorgos Argyriou; George Stamoulis; George Papadakis; Manolis Koubarakis; Pythagoras Karampiperis; Axel-Cyrille Ngonga Ngomo; and Maria-Esther Vidal, "The BigDataEurope Platform – Supporting the Variety Dimension of Big Data," in 17th International Conference on Web Engineering (ICWE2017), 2017. This article is a joint work with the BDE consortium. In this article, I contributed within the semantic layer, more specifically; bringing the Big Data Analytics for RDF into the BDE platform and co-contributing into dockerizing BDE components.

7.1 The SANSA framework

In this section, we introduce SANSA[1], an open-source[2] *structured data processing engine* for performing distributed computation over large-scale RDF datasets. It provides data distribution, scalability, and fault tolerance for manipulating large RDF datasets, and facilitates analytics on the data at scale by making use of cluster-based big data processing engines. It comes with: (i) specialized serialization mechanisms and partitioning schemata for RDF, using vertical partitioning strategies, (ii) a scalable query engine for large RDF datasets and different distributed representation formats for RDF, namely graphs, tables, and tensors, (iii) an adaptive reasoning engine which derives an efficient execution and evaluation plan from a given set of inference rules, (iv) several distributed structured machine learning algorithms that can be applied on large-scale RDF data, and (v) a framework with a unified API that aims to combine distributed in-memory computation technology with semantic technologies.

To achieve the goal of storing and manipulating large RDF datasets, we leverage existing big data frameworks like Apache Spark[3] and Apache Flink[4], which have matured over the years and offer a proven and reliable method for general-purpose processing of large-scale data.

7.1.1 Architecture Overview

We now give an overview of the SANSA framework. Figure 7.1 shows the overall architecture of SANSA that consists of four layers: ***Knowledge Distribution & Representation Layer***, ***Query Layer***, ***Inference Layer*** and ***Machine Learning Layer***.

In the following, we explain the role of each layer.

[1] http://sansa-stack.net/
[2] https://github.com/SANSA-Stack
[3] http://spark.apache.org/
[4] http://flink.apache.org/

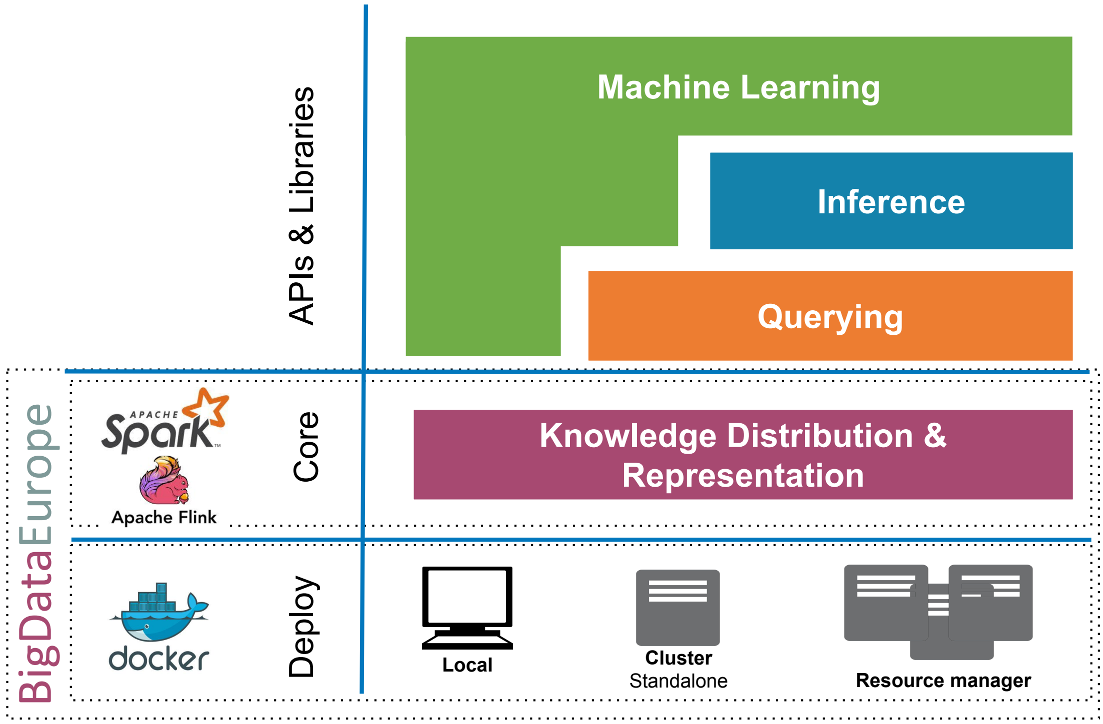


Figure 7.1: **Overview of the SANSA stack**. The SANSA framework combines distributed analytics and semantic technologies into a scalable semantic analytics stack.

Knowledge Distribution & Representation Layer It is the lowest layer on top of the existing distributed frameworks (Apache Spark or Apache Flink). This layer mainly provides the facility to read and write native RDF or OWL data from HDFS or a local drive and represent it in the native distributed data structures of the frameworks.

In addition, it also provides a dedicated serialization mechanism for faster I/O. SANSA aim to support 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 and thus would require less time to get productive with the SANSA stack.

Moreover, it allows users to compute RDF statistics (cf. Chapter 4) and quality assessment (cf. Chapter 5) in a distributed manner.

Query Layer Querying an RDF graph is the primary method for searching, exploring, and extracting information from the underlying RDF data. SPARQL is the W3C standard for querying RDF graphs. Our aim is to have cross-representational transformations and partitioning strategies for efficient query answering. We are investigating the performance of different data structures (e.g., graphs, tables, tensors) in the context of different types of queries and workflows. SANSA provides APIs for performing SPARQL queries directly in Spark and Flink programs (cf. Chapter 6). It also features a W3C standard-compliant HTTP SPARQL endpoint server component for enabling externally querying the data that has been loaded using its APIs. These queries are eventually transformed into lower-level Spark/Flink programs executed on the Distribution & Representation Layer. At present, SANSA implements flexible triple-based partitioning strategies on top of RDF (such as predicate tables with sub-partitioning by datatypes), which will be complemented with sub-graph based partitioning strategies. In addition, it also support a so-called semantic-based query engine – a scalable approach to evaluate SPARQL queries over distributed RDF datasets. Based on the partitioning and the SQL dialects supported by Spark and Flink, SANSA provides an infrastructure for the integration of existing SPARQL-to-SQL rewriting tools. This bears the potential advantage of leveraging the optimizers of both the rewriters as well as those of the underlying frameworks for SQL. Currently, the Sparklify implementation serves as the baseline. It uses Sparqlify[5] as a SPARQL-to-SQL rewriter for translating SPARQL queries into Spark executable code. Query results can then be further processed by other modules in the SANSA Framework.

Inference Layer Both RDFS and OWL contain schema information in addition to links between different resources. This additional information and rules allow to perform reasoning on the knowledge bases in order to infer new knowledge and expanding the existing one. 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 in the knowledge base. It is well known that there is always a trade-off between the expressiveness of a formal language and the efficiency of reasoning in that language. SANSA contains an adaptive rule engine that can use a given set of arbitrary rules and derive an efficient execution plan from a given set of inference rules.

By using SANSA, applications will be able to fine-tune the rules they require and – in case of scalability problems – adjust them accordingly.

Machine Learning Layer While most machine learning algorithms are based on processing simple features, the machine learning algorithms in SANSA exploit the graph structure and semantics of the background knowledge specified using the RDF and OWL standards. In many cases, this allows obtaining either more accurate or more human-understandable results. There exist a wide range of machine learning algorithms for structured data. However, the challenging task would be to distribute the data and to devise distributed versions of these algorithms to fully exploit the underlying frameworks. We are exploring different algorithms namely, tensor factorization, association rule mining, decision trees and clustering on structured data. The aim is to provide out-of-the-box algorithms to work with the structured data in a distributed, fault-tolerant and resilient fashion. Based on those advances, we will also be able to efficiently perform analytics to gain insights into the data for relevant trends, predictions or detection of anomalies.

7.1.2 SANSA-Notebooks: Developer friendly access to SANSA

SANSA provides Notebooks for an easy local deployment for development and demonstration purposes. SANSA-Notebooks is an interactive toolkit on top of Hadoop-Spark-Workbench[6] with Apache Zeppelin[7], which allows the copying of files from/to HDFS and an interactive Spark code execution via a web GUI. The architecture of SANSA-Notebooks is depicted in Figure 7.2.

We utilize SANSA-Notebooks (see Figure 7.3) in Big Data labs[8] and courses as they alleviate the complicated Hadoop/Spark setup and allow the students to focus on developing distributed algorithms

[5] https://github.com/AKSW/Sparqlify
[6] https://github.com/big-data-europe/docker-hadoop-spark-workbench
[7] https://zeppelin.apache.org/
[8] https://github.com/SmartDataAnalytics/MA-INF-4223-DBDA-Lab

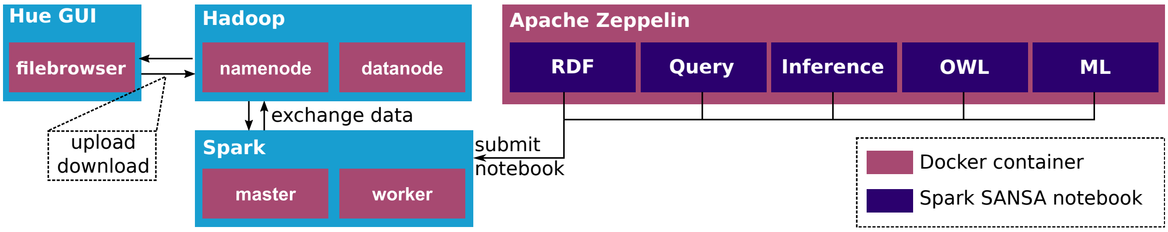


Figure 7.2: **SANSA-Notebooks architecture**. An interactive toolkit on top of dockerized Hadoop-Spark-Workbench with Apache Zeppelin.

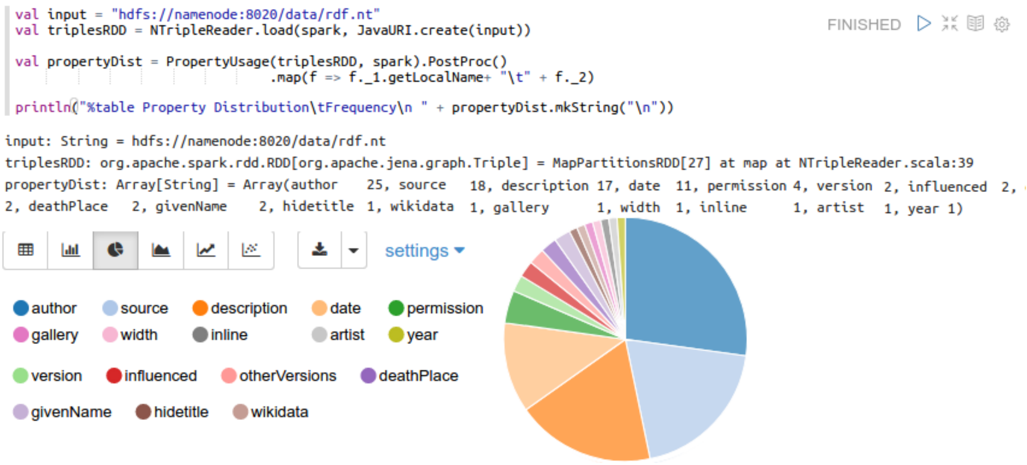


Figure 7.3: **SANSA Notebooks example**. RDF-Stats Spark application running in SANSA-Notebooks with statistics visualization.

on top of SANSA. Cluster deployment of the examples is also possible through Docker images (see SANSA-Examples Github repository[9]). Additionally, SANSA is readily available from the Maven Central Repository. Thus it is straightforward to include it in other projects using Maven or SBT – the most popular build managers for Scala – for both Spark- and Flink-based setups.

The notebooks present a compiled list of the SANSA examples[10]. These examples give a quick overview of the SANSA APIs. SANSA is build on the concepts of distributed datasets (i.e RDD, DataFrame, DataSet). A dataset is inferred from the external data, then parallel operations e.g. *transformations* and *actions* are applied which trigger a job execution on a cluster. In the following, we provide a concise description of the examples grouped by the SANSA layers.

1. ***RDF***.
   a) Reading and writing triple files from HDFS or file system and some basic triple operations.
   b) A distributed evaluation of numerous RDF Dataset Statistics dubbed RDF-Stats (see Figure 7.3), for example, property distribution, class distribution, distinct subjects/objects/entities as well as statistics summary.

[9] https://github.com/SANSA-Stack/SANSA-Examples
[10] The source code for all of them is provided at https://github.com/SANSA-Stack/SANSA-Examples

c) A distributed evaluation of numerous RDF Dataset quality assessment metrics i.e schema completeness, conciseness, interlinking, etc.

d) Assigning weights to a given entity based on the Spark GraphX PageRank algorithm after triples have been transformed to a graph representation (i.e. PageRank for resources).

2. ***Query***. The example applies Sparqlify[11], which is a SPARQL-to-SQL rewriter, for data partitioning and schema extraction. The queries are executed using the SparkSQL engine.

3. ***RDF inference***. The examples apply a reasoning profile (RDFS Full, RDFS Simple, OWL Horst, Transitive) on a given input file with an optimised execution plan.

4. ***OWL***. The examples provided for the OWL layer demonstrate the process of loading an OWL file into Spark RDD, a Spark Dataset, or a Flink DataSet.

5. ***Machine Learning***.

   a) Clustering algorithms. Three examples for different clustering algorithms are provided, namely power iteration clustering, BorderFlow and modularity clustering. They all take an RDF graph as input and return the list of triples for each of the different clusters.

   b) Rule mining. This example applies association rule mining on a given RDF knowledge base. The output is the set of closed Horn rules that satisfy a support-confidence threshold.

One of the powerful features of the SANSA Notebooks is that you can view the result set of the previous session within the Spark framework and, in case you have found some insight for your data and would like to share, you can easily create a report and either print or send it.

The main goal of the SANSA framework is to build a generic stack that can work with large amounts of linked data, offering algorithms for scalable, i.e. horizontally distributed, semantic data analysis. To validate this, we have developed use case implementations in several domains and projects.

A more detailed list of use cases with technical details and implementation is given on the following sections (cf. Section 7.2, 7.3, and 7.4).

7.2 Leveraging Blockchain RDF Data Using the SANSA Framework

With the hype on blockchain technologies and in particular in the Ethereum blockchain [98], many participants wanted to know more about the most impactful players across the blockchains transaction network. In parallel, as the number of statements, actions, and transactions in the network is increasing quickly, many "Big Data" challenges arise. First, transactions are raw data and one cannot take advantage of them for further analysis. To do so, Alethio designed EthOn (The Ethereum Ontology) [28] which models such raw data as triples using the RDF standard. This ontology describes all Ethereum terms including blocks, transactions, contract messages, event logs, etc., as well as their relationships. Afterword, performing querying and analysis on such large-scale RDF datasets is computing-intensive. To overcome these challenges, we have explored the potential of the SANSA [31] framework. With SANSA on Spark, RDF triples are loaded into Spark distributed and resilient data structured, namely the data frames, for further analysis.

[11] http://aksw.org/Projects/Sparqlify.html

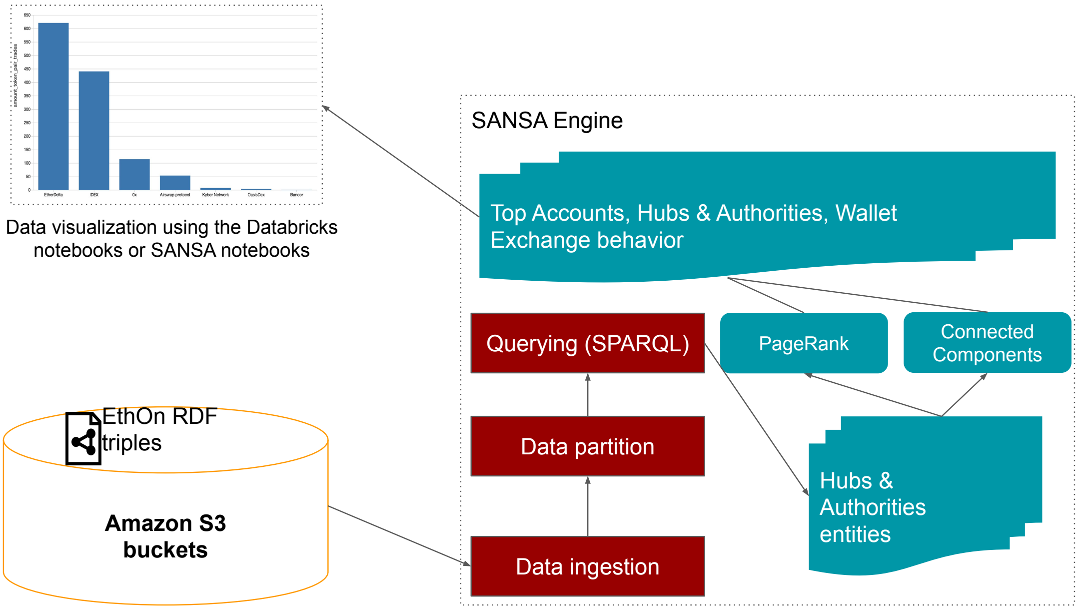


Figure 7.4: **Hubs and Authorities analysis workflow**. The architecture overview for gaining insight about Hubs and Authorities using the SANSA framework.

7.2.1 The Hubs and Authorities Transaction Network Analysis

In this work, we perform an analysis (using well-known graph processing algorithms) of the value transaction network graph with the main focus on the Hubs and Authorities behaviors. "Authorities" are accounts that payout to a large crowd of addresses, with high volume; while "Hubs" are entities who receive extensive Ether (ETH) flow into their accounts. In this study, we do not differentiate these two roles but rank them all together as the biggest players/entities.

Finding big Ethereum players with SANSA

The Ethereum network graph contains nodes of external accounts which have had a transaction on the Ethereum blockchain. The connection (edges) between such nodes on the network indicate the transaction relationship between them; when a node (an external account) sends ETH to another, a transaction record is written, and an edge between them is added in the network with the direction of the ETH flow. When we encounter multiple edges between same pairs of nodes, we summarize the edges as a single one[12]. The edge weight is the total transaction value in Ether. As an example, if address A sends x ETH to address B in total, there will be an edge of weight x from node A to node B. In this study, self-loops i.e. transactions from an address to itself are omitted.

SANSA framework has been used for efficient reading and querying of RDF datasets using SPARQL as depicted on Figure 7.4. First, the data need to be loaded on an efficient storage that SANSA can read from. For that purpose, we use Amazon S3 buckets containing the whole RDF Ethereum network transactions. Afterword, the SANSA data representation layer loads the data in a form of RDD of

[12] This optimization is also convenient practically as it is easier not to have duplicated edges in a graph.

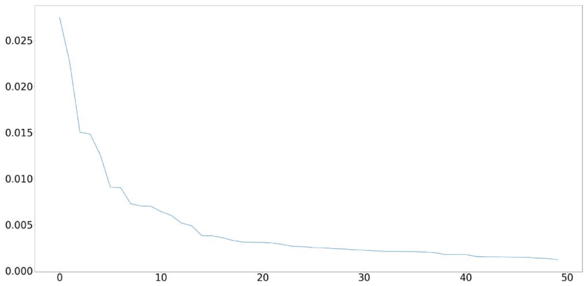


Figure 7.5: **PageRank Score Distribution of Top-50 Accounts**.

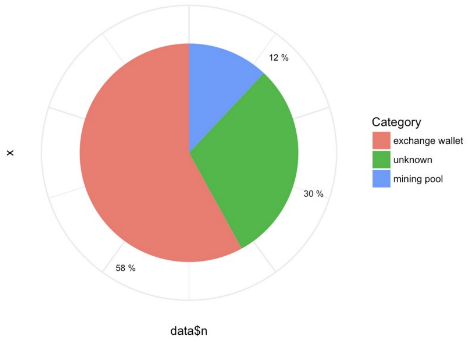


Figure 7.6: **Category Distribution of Top-50 Accounts**.

triples. During this process, SANSA performs a data partition for fast processing and then aggregate and filter the data using its query layer. Further, we applied two classic graph analysis algorithms via Apache GraphX: Connected Components and Page Rank. Connected Components algorithm enables us to find the largest cluster of connected nodes, regardless of transaction direction. Within this largest cluster, we can derive the page rank score of all nodes. Top-ranked entities and their relation are visualized.

Results

Datasets The Ethereum dataset in the format of RDF contains more than 17B triples. For the sake of the experiment, we limited the dataset to 10,000 blocks which contain around 38M triples, including both value transactions and contract messages.

Top Accounts Analysis The PageRank algorithm was run over the largest connected component of 185,741 nodes (accounts) and 250,637 edges (aggregated transaction relations).

Figure 7.5 plots the top 50 account's distribution. Based on the findings, we can see that these accounts are grouped on two different types: mining pool wallets, and (mostly centralized) exchange wallets.

Figure 7.6 shows that 58% of the addresses are controlled by exchanges, while another 12% with convincing tags related to the mining pools. The exchange and mining pool wallets can be found in the top position of our ranking, underlining the effectiveness of PageRank: Addresses related to mining pools allocate extensive amounts of payouts to their subscribed miners, resulting in large out-degrees, as well as high accumulated transaction value. We can see that the main wallets are centralized exchanges which distribute (and receive) large volumes of the transaction to (and from) their deposit wallets, token contracts, etc.

Our PageRank implementation successfully detects the most influential accounts across the network, corresponding to the Hubs and Authorities, connecting various transactors and carrying heavy flow weights.

Focusing on those known accounts (with labels from Etherscan[13]), we present (see Figure 7.7) the network overview of top hubs and authorities with transactions as edges surrounding them.

[13] https://etherscan.io/

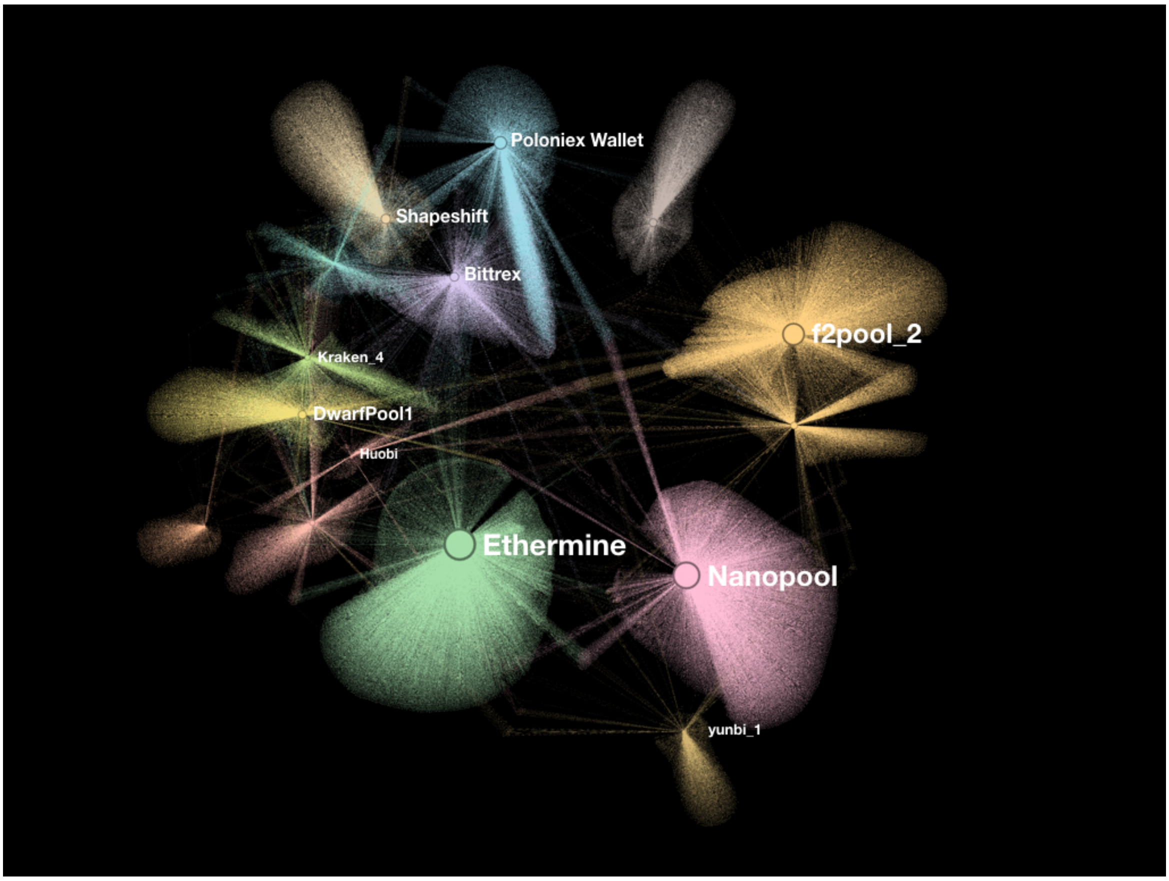


Figure 7.7: **Transaction Network of Top Hubs and Authorities**.

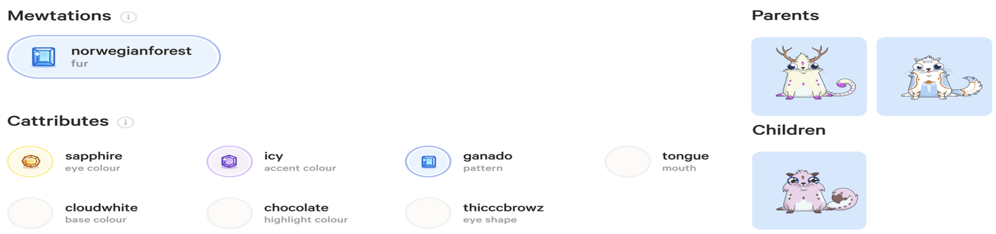
Typical Behavior Patterns of Exchanges' Deposit Wallets We investigated the associated transaction behavior of the exchange wallets. Based on our finding, these behaviors can be grouped into three categories:

1. *Frequently paying out to certain exchanges' main wallets with a fixed, large value* – From the scatter plot, the payout amount is always around the same value.

2. *Frequently receiving funds from the same exchange main wallets, and paying out to various token contracts* – This is due to the activity which is associated with exchanges as they use external accounts as deposit addresses for collecting tokens based on trading needs.

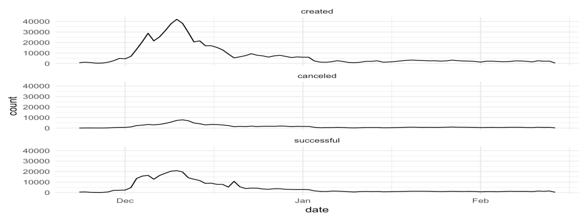
3. *Frequently receiving funds from a group of "miner" accounts, with "proxy" accounts in between, which clean out their received ETH within a short time frame* – Usually, these addresses receive funds from miner accounts, which again get paid reasonable amounts by known mining pools, which we assume are mining rewards (usually around 0.11-0.12 ETH).

Despite pointing out the three typical behaviors above, they are not necessarily mutually exclusive. There are addresses that share more than one of the deducted patterns. These behavior patterns explored here are based on the labels we have gathered, and this may be different for other use cases.

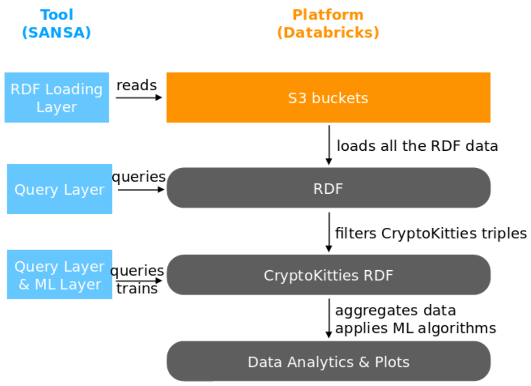


(a) The unique attributes of a Kitty.

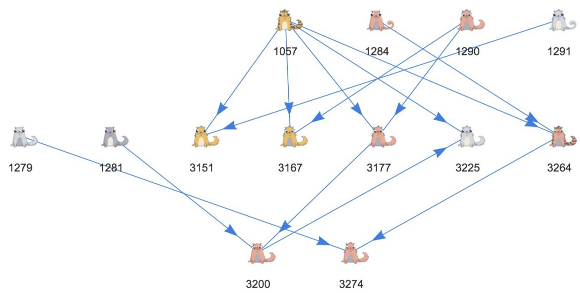
(b) An instance of a Kitty.



(c) History of three types of auction events.



(d) The process pipeline.



(e) An illustration of a small family tree.

Figure 7.8: **Leveraging Blockchain RDF Data with SANSA: CryptoKitties as a Use Case**.

7.2.2 Profiting From Kitties on Ethereum

The Ethereum ecosystem generates a large amount of data, including but not limited to protocol-level data (e.g. average block time, gas prices), as well as application-level data (e.g. account interactions, smart contract deployments). To efficiently handle this volume of data, Alethio has investigated different tools and frameworks with one focus: the infrastructure should be resilient, load-bearing, and most importantly, scalable. And so, for that reason to overcome the variety of the different data sources, Alethio introduces semantification of the Ethereum network and uses SANSA as an underlying engine for large scale distributed RDF based querying, reasoning, and machine learning on top of these RDF datasets. To show the joint effort between SANSA and Alethio, we describe a use case on how SANSA can be used to analyze Ethereum at new scales, as depicted in Figure 7.8.

CryptoKitties[14] is one of the first games to be built on blockchain technology. In particular, CryptoKitties initiated and released the first generation virtual kitties, with delicately designed icons and genes sequences. All the kitties are virtual with some biological feature settings. Shown in Figure 7.8(b) is a kitty with its specific biological attributes displayed in Figure 7.8(a). The attributes are stored in a sequence, succeeded from its parents' gene sequences, with the possibility of *mewtations*. An owner can sell, breed or gift it to other users. When users sell or breed it, they will send transactions to the CryptoKitties smart contracts, which will complete the execution of either transferring ownership between users or generating a new kitty. Based on that, game users can trade or breed kitties like traditional collectibles, while having the guarantee that the blockchain will track ownership securely. Moreover, one can breed two kitties to create a brand-new, genetically unique offspring.

Data Challenges. Alethio has been exploring efficient means of processing large RDF data sets. SANSA empowers Alethio to read and query the data at scale as described in Figure 7.8(d). Indeed, once the complete RDF data set is loaded, SANSA filters it to retain only the CryptoKitties triples –transactions, contract messages, and log information– before performing more specific analyses.

Practically, the challenges tackled with SANSA can be divided into two groups: game performance and customer behaviors. The first one focuses on time series metrics: throughput time, the event volume, number of active users and amount of spent Ether, which can jointly estimate the trend of popularity for the game. In Figure 7.8(c), the history of CryptoKitties auctions events shows clearly that there was a peak of traffic in December after the game was launched for around one month. By this time series, we can estimate the popularity of the game throughout history. The second one requires machine learning algorithms to detect correlations between indicators (e.g. to determine whether richer owners have the tendency to collect special/rare kitties which are more expensive) and topology from a network view. In Figure 7.8(e), we present a small subset of the kitty family tree, where incest happened during the reproduction: kitty 1057 is the secondary-degree relative (grandparent) of kitty 3200, while later it bred with kitty 3200 and gave birth to kitty 3225.

7.3 Mining Big Data Applications Logs Using the SANSA Framework

Big Data Europe (BDE)[15] [29] is a large Horizon2020 funded EU project which offers an open-source big data processing platform allowing users to install numerous big data processing tools and frameworks. The platform has been tested and used by the 17 different partners of the project scattered across Europe and its 7 different use cases cover a variety of societal challenges like climate, health, weather, etc.

More specifically, BDE also allows the creation of a workflow for a stack containing many applications, each serving a particular data value chain. An important feature of the integrator interface is the `mu.semte.ch` microservice which transforms docker events to RDF and stores them in a triple store[16]. The work is also being done towards storing the network logs in the triple store, by translating the HTTP network traffic as triples as they occur in the network. This network log data combined with the docker event data grows over time and provides a useful source that can help in analyzing

[14] https://www.cryptokitties.co/
[15] https://github.com/big-data-europe
[16] https://github.com/big-data-europe/mu-swarm-logger-service

the event-call-time proximity. SANSA has been used to perform useful analytics over this data and provide a possibility to create user profiles for the BDI platform.

Application Example: A Smart Green and Integrated Transport

The H2020 Societal Challenge 4[17], Smart Green and Integrated Transport, covers a broad topic ranging from urban mobility to safety, logistics, transport system integration, infrastructure monitoring, and planning. Transport systems consume huge flows of data to provide services, monitor infrastructures

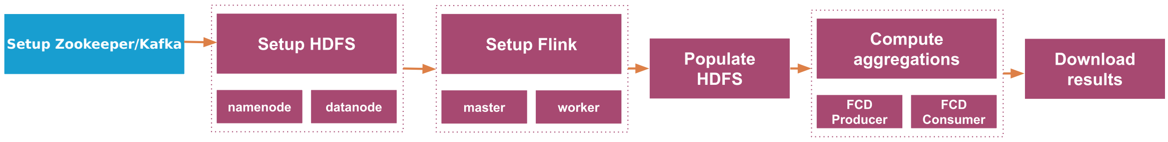


Figure 7.9: **Transport pilot initialization workflow**. The SC4 initialization pipeline including different BDE dockerized components.

and discover the usage patterns in order to forecast what will be the status in the near or distant future. All these systems consume streams of data from different sources and in different formats. In the SC4 pilot, we have therefore decided to build a pilot that can ingest, transform, integrate and store streams of data that have spatial and temporal dimensions. One of the project partners, CERTH-HIT, is managing a system that monitors the traffic flow in Thessaloniki, Greece, using floating car data from a transport company. The legacy system is based on a relational database, stored procedures and R scripts to map-match the location of the vehicles to the road segments and compute the traffic flow and average speed among other statistical parameters. The result of the computation is used for monitoring and as input for forecasting the value of the parameters in the near future and is made available through a web service. The aim of the pilot is to address the scalability issues of the current system leveraging the availability of distributed frameworks and the containerization technology for the deployment of services in different environments.

The pilot is based on the microservices architecture where different software components, producers and consumers, communicate through a messaging system connecting data sources to data sinks. Producers and consumers are implemented as Flink[18] jobs while Kafka[19] has been chosen as the messaging system. The producer fetches the data every two minutes from the web service, stores the records sets into HDFS, transforms the records into a binary format, using a schema shared with the consumer, and finally sends the records to a Kafka topic. The consumer reads the records from the Kafka topic and process them at event time applying the map matching function. The consumer must connect to an R server where an R script has been installed to perform the computation for the map matching using the road network data from Open Street Map stored in a PostGis database. The consumer adds the identifier of the road segment as an additional field to the original record and finally aggregates the records per road segment and in time windows to compute the traffic flow and the average speed in each road segment. The result of the aggregation can be sent to HDFS or

[17] `https://www.big-data-europe.eu/pilot-transport/`
[18] `http://flink.apache.org/`
[19] `https://kafka.apache.org/`

to Elasticsearch[20]. From Elasticsearch different visualizations can be created easily with Kibana[21]. The records with the aggregated values stored in Elasticsearch will be used as input to a forecasting algorithm to predict the traffic flow. All the components are available as Docker images and a docker-compose file has been created adding the initialization service and the UI provided by the BDI Stack in order to start the services in the right sequence from the browser (e.g. Zookeeper before Kafka and PostGis and Elasticsearch before the consumer)[22].

With such a chain of technologies used, it is obvious that many logs are being generated. The Mu Swarm Logger Service provided within the BDE platform collects the events generated by the Docker API and trigger code to log to the database the following events: i) Container's events (including the environments variable and labels), ii) Container's logs (STDOUT, and STDERR), and iii) Docker stats i.e. health status, CPU and Memory Usage footprint, I/O, etc. The events generated by the service are modeled using the Events Ontology[23]. Each pipeline (or stack of services) build within the BDE pipeline has the possibility to tag services with *LOG* label in order to generate log events (as described above). The service then waits for such logs and write them back into an RDF. One can imagine such a process run over millions of events spread across multiple big data stacks containing multiple services that will generate a large amount of RDF data (events). Analyzing this valuable information via traditional RDF data management systems was not possible. Therefore, we integrated our approaches via SANSA for analyzing log events on the BDE platform. More specifically, BDE run RDF dataset statistics from SANSA-Notebooks [30] and provide visualization of events generated e.g. finding the most frequent errors happening at the specific event.

7.4 Scalable Integration of Big POI Data Using the SANSA Framework

Various organizations like DBpedia [6], Wikidata [99] etc. are constantly working for gathering information from different sources and storing it in a structured form, e.g. RDF. RDF data allow to model various domains and this characteristic helps to solve problems in different areas i.e., from the medical domain to the geographical domain.

In this study, we are focusing on POIs. POIs are generally characterized by their geospatial coordinates along with their thematic/contextual attributes. A common POI use-case is to find hot zones according to specific topics: i.e. discovering AOIs as a result of the aggregation of POIs. With the assistance of AOIs, one can identify other similar areas in the same or a different city, recognize the distinguishing characteristics of this area, and determine potential types of users (or customers) that would be interested in that area.

In this use case, we propose a flexible architecture to design clustering pipelines for POI semantic datasets at once. Indeed, using large and detailed RDF vocabularies allow richer POI descriptions. For example, one POI related to a restaurant might be described by its latitude, longitude, food specialty, reviews, address, phone number, etc. which could represent up to 50 distinct triples[24] leading then to billions of RDF records overall. As a consequence, we require scalability and build our solution

[20] https://www.elastic.co/products/elasticsearch
[21] https://www.elastic.co/products/kibana
[22] https://github.com/big-data-europe/pilot-sc4-fcd-applications
[23] https://github.com/big-data-europe/mu-swarm-logger-service/blob/master/docs/docker-engine-events/events.owl
[24] See e.g. the SLIPO ontology: https://github.com/SLIPO-EU/poi-data-model/

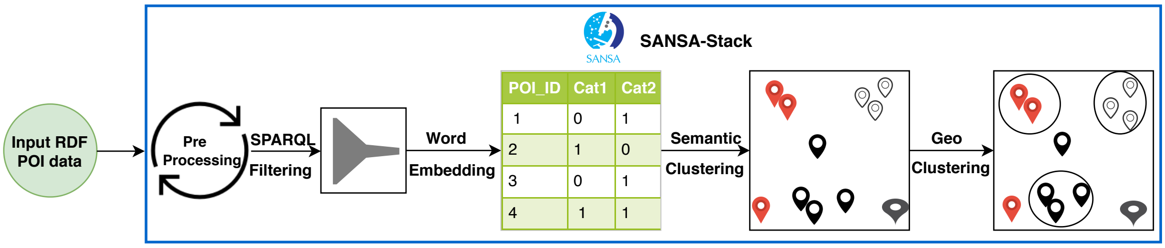


Figure 7.10: **A Semantic-Geo Clustering flow**. It consists of five main components: data pre-processing, SPARQL filtering, word embedding, semantic clustering, and geo-clustering.

on top of the distributed semantic stack SANSA which benefits from Apache Spark. The proposed architecture then enables any kind of clustering algorithm combinations on POI RDF data.

7.4.1 Proposed Solution: Architecture Overview

In order to process RDF (containing POIs) datasets in an efficient and scalable way, we first have to adopt a convenient processing framework. SANSA is a data-flow engine for distributed computing of large-scale RDF datasets. It provides APIs for faster reading, querying, inferencing and apply analytics at scale. It uses Apache Spark as an underlying engine. SANSA contains features which are utilized for processing RDF data with thematic and spatial information.

Our proposed 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 particular, in Figure 7.10, we present an example of the Semantic-Geospatial clustering pipeline. Indeed, we consider two types of clustering algorithms: the semantic-based ones and the geo-based ones.

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, we can select any word embedding technique among the three available ones namely one-hot encoding, Word2Vec, and Multi-Dimensional Scaling. All the above-mentioned 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 in another one. On the other hand, the geo-clustering methods help to group the spatially closed coordinates within each semantic cluster.

More generically, our 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. In addition, we directly embedded the state-of-the-art clustering algorithms into the SANSA Machine Learning layer[25] so that these pipelines are prone to be built out of the box.

[25] https://github.com/SANSA-Stack/SANSA-ML

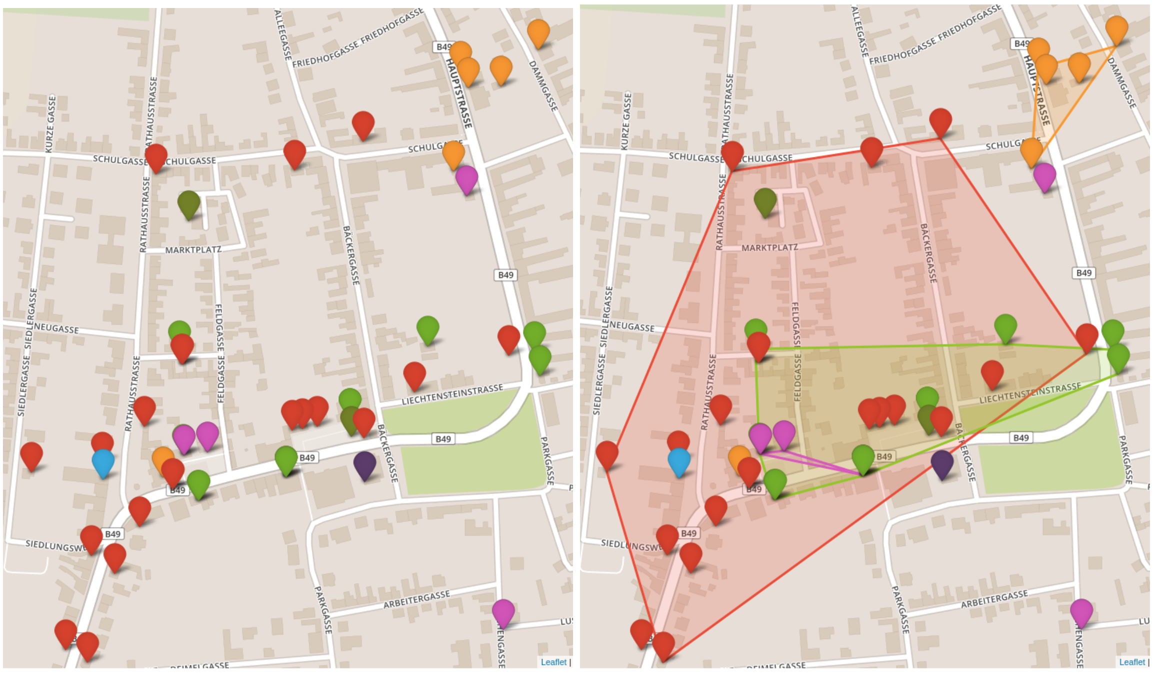


Figure 7.11: **Visualizations (on a map) of the Semantic-Geo clustering pipeline steps**. Visualizations of a zoom over a particular Austrian region with K-means results of POIs (left) and geographical clustering with relevant AOIs (right).

Application Example: A Semantic-Geo Clustering Pipeline

To illustrate the feasibility of our approach and demonstrate the potential of the RDF POI clustering library we developed in SANSA, we present –as an example– in this section the implementation results of the specific architecture presented in Figure 7.10 i.e. a Semantic-Geo clustering pipeline.

In order to test the process and validate the approach, we used an RDF POI dataset which follows the ontology described in [100] containing around 18 000 triples which represent information on 623 POIs (i.e. around 28 triples per POI). We then chose Word2Vec [101] as embedding for the K-means [102] semantic-clustering algorithm, before running DBSCAN [103] as geo-clustering method. In detail, we gave the following parameters to the algorithms: 8 clusters within 5 iterations for K-means and $\epsilon = 0.002$ with at least 2 points per cluster for DBSCAN. The complete process took around 20 seconds using an 8GB-memory laptop running a single-node SANSA & Spark stack.

We present the results obtained at the various steps in Figure 7.11 on a map, the figure presents a zoom over a particular Austrian region. The figure is twofold, we first display (left side) the only result of the K-means where POIs are pinned on a map and where each color corresponds to a specific cluster. As expected, the semantic clusters are distributed over the entire country since POIs of color are sharing common “sense” with regards to the categories in the ontology. As a consequence, the geographical step of aggregation allows then to break those country-spread clusters into pieces and obtain (right side of Figure 7.11) relevant AOIs. In particular, four AOIs are visible: an orange one in the corner, a large red one which also embeds a green one and a little magenta.

7.5 Summary

SANSA provides a scalable solution for reading and querying large scale RDF data, providing compatibility with machine learning libraries on Spark including GraphX as a graph processing library.

With conventional graph analysis tools, we successfully identified Hubs and Authorities in the Ethereum transaction network and discovered that they are mainly related to exchange wallet and mining pool activities.

This pipeline also provides a possibility to filter out top accounts, which are likely to exchange' deposit wallets. Furthermore, with the filtered top rank accounts, the "mixing" patterns of exchanges' deposit wallets become recognizable. This can be a promising tool for detecting previously unknown exchange wallets and lead to a deeper understanding of their behavior patterns for future analyses. Alethio is investigating DistQualityAssesment as well, for performing large-scale batch quality checks, e.g. analysing the quality while merging new data, computing attack pattern frequencies and fraud detection. Alethio uses our approaches on a cluster of 100 worker nodes to assess the quality of their ≈20B[26] of RDF data.

In addition, we showed the solution of collecting event logs generated by multiple dockerized big data components in the BDE platform and analyze them using our approach. The idea behind collecting reach information about docker logs and other services is for providing a better monitoring view of the running services. Usually, such historical information may lead to new knowledge for early detection of failures of the running processes or even by just categorizing the most frequent error types happening in the past. It is helpful for providing performance and diagnostics information to the user.

Finally, we presented a solution to extract AOIs from big POI data while considering several dimensions at the same time. The architecture is embedded inside a state-of-the-art Semantic Web stack (i.e. SANSA) and then benefits from the advantages of it. For instance, it allows source aggregation or datasets filtering via SPARQL to only focus on some interesting regions, e.g., a specific country can be selected. Moreover, even if we restricted our description in this study to a Semantic-Geo clustering pipeline, our architecture allows any kind of clustering combinations. The above-presented pipeline is also openly available from a demonstrating notebook[27] on the SANSA repository.

[26] https://linkeddata.aleth.io/

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

CHAPTER 8

Conclusion and Future Directions

In this chapter, we summarize the work done during this thesis and highlight the main results. During this thesis, we studied the research problem of efficient distributed in-memory processing of RDF datasets.

In particular, we addressed the problems of Scalable Computation of RDF Dataset Statistics (cf. Chapter 4), Quality Assessment of RDF Datasets at Scale (cf. Chapter 5), Scalable and Efficient SPARQL Query Evaluation (cf. Chapter 6), and usage of such scalable approaches into real-world use cases (cf. Chapter 7).

In the following sections, we provide a summary of our contributions and elaborate on the main findings that validate our research questions.

8.1 Review of the Contributions

In this section, we give an overview of the thesis' contributions in terms of the problems solved and how they offer concrete and valid solutions to the research questions. The main goal of the thesis is to advance the area of distributed processing of RDF datasets by providing a novel set of approaches in order to solve the main challenges in a distributed and scalable setting. In this respect, our contributions answer three research questions. Let us revisit the research questions defined during this thesis.

First, we tackled the problem of exploring the structure of the large-scale RDF datasets and answering the following research question.

RQ1: How can we efficiently explore the structure of large-scale RDF datasets?

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. To address RQ1, in Chapter 4 we introduced a software component for statistical calculations of large RDF datasets, which scales out to clusters of machines. More specifically, we described 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. Overall, we provide the following contributions to the state-of-the-art:

- We proposed 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 performed an analysis of the complexity of the computational steps and the data exchange between nodes in the cluster.
- We evaluated 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.).
- An approach for triggering RDF statistics calculation remotely simply using HTTP requests. DistLODStats is built as a plugin into the larger SANSA framework and makes use of Apache Livy, a novel lightweight solution for interacting with the Spark cluster via a REST Interface.

The second problem we tried to address was the possibility of assessing the quality of large-scale RDF datasets efficiently in a distributed manner and answers the following research question.

> **RQ2**: Can we scale RDF dataset quality assessment horizontally?

Over the last years, Linked Data has grown continuously. Today, we count more than 10,000 datasets being available online following Linked Data standards. These standards allow data to be machine-readable and interoperable. Nevertheless, many applications, such as data integration, search, and interlinking, cannot take full advantage of Linked Data if it is of low quality. There exist a few approaches for the quality assessment of Linked Data, but their performance degrades with the increase in data size and quickly grows beyond the capabilities of a single machine. To answer question RQ2, in this thesis, we present DistQualityAssessment (cf. Chapter 5) – 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. The work presented here is integrated with the SANSA framework and has been applied to at least three use cases beyond the SANSA community. The results show that our approach is more generic, efficient, and scalable as compared to previously proposed approaches. Overall, we provide the following contributions to the state-of-the-art:

- We present a Quality Assessment Pattern $\mathcal{QAP}$ to characterize scalable quality metrics.
- We provide DistQualityAssessment – a distributed (open source) implementation of quality metrics using Apache Spark.

- We performed an analysis of the complexity of the metric evaluation 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. SANSA is actively maintained and uses the community ecosystem (mailing list, issues trackers, continuous integration, website, etc.).

The third problem we tackled in this thesis was the problem of querying and retrieving distributed RDF datasets in an efficient and effective way and answers the following research question.

RQ3: Can distributed RDF datasets be queried efficiently and effectively?

One of the key features of Big Data is its complexity in terms of representation, structure, or formats. One existing way to deal with it is offered by Semantic Web standards. Among them, RDF –which proposes to model data with triples representing edges in a graph– has received a large success and the semantically annotated data has grown steadily towards a massive scale. Therefore, there is a need for scalable and efficient query engines capable of retrieving such information. To answer RQ3, in Chapter 6 we proposed scalable approaches for SPARQL query evaluation over distributed RDF data. First, Sparklify – a scalable software component for efficient evaluation of SPARQL queries over distributed RDF datasets. It uses Sparqlify as a SPARQL-to-SQL rewriter for translating SPARQL queries into Spark executable code. Our preliminary results demonstrate that our approach is more extensible, efficient, and scalable as compared to state-of-the-art approaches. As a second approach, we investigated and implemented a scalable semantic-based query engine for efficient evaluation of SPARQL queries over distributed RDF datasets. It uses a semantic-based partitioning strategy as the data distribution and converts SPARQL to Spark executable code. We have shown empirically that a semantic-based approach can scale horizontally and perform well as compared with the previous Hadoop-based system: the SHARD triple store. It is also comparable with other in-memory SPARQL query evaluators when there is less shuffling involved i.e. less duplicate values. Both approaches are integrated into a larger SANSA framework and Sparklify serves as a default query engine and has been used by at least three external use scenarios. Overall, we provide the following contributions to the state-of-the-art:

- We present a novel approach for vertical partitioning including RDF terms using the distributed computing framework, Apache Spark.
- We developed a scalable query system using Sparqlify – a SPARQL-to-SQL rewriter on top of Apache Spark.
- We evaluated Sparklify with state-of-the-art engines and demonstrate it empirically.
- A scalable approach for semantic-based partitioning using the distributed computing framework, Apache Spark.
- A scalable semantic-based query engine (*SANSA.Semantic*) on top of Apache Spark.

- Comparison of the semantic-based system with state-of-the-art engines and demonstrate the performance empirically.
- We integrated the proposed approaches into the SANSA larger framework. Sparklify serves as a default query engine in SANSA. SANSA is an active project and maintained, including issue tracker, mailing list, changelogs, website, etc.

8.2 Limitations and Future Directions

In this section, we discuss the limitations we identified during this study and potential future directions to take in order to overcome such limitations.

In the following, we summarize the limitations and future directions on each of the main contributions of this thesis.

- *Large-scale RDF Dataset Statistics* – In Chapter 4 we have demonstrated that our approach is scalable when computing statistics over a large amount of RDF data as compared with a centralized approach. Nevertheless, we plan to further improve time efficiency by persisting the data to an even higher extent more in memory and perform load balancing, which could further improve the performance. Moreover, as our implementation is purely batch processing, in which the data chunks are normally very large we plan to investigate additional techniques for lowering the network overhead and I/O footprint. In this regard, efficient compression (e.g. Header, Dictionary, Triples (HDT) [104]) methods for lowering data communication would be very relevant. Finally, as our main focus is on applying distributed techniques to RDF data processing, we plan to port the existing solution for near real-time computation of RDF dataset statistics.
- *Assessment of RDF Datasets at Scale* – Although we have achieved reasonable results in terms of scalability (cf. Chapter 5), we plan to further improve time efficiency by applying intelligent partitioning strategies and persist the data to an even higher extent in memory and perform dependency analysis in order to evaluate multiple metrics simultaneously. We also plan to explore near real-time interactive quality assessment of large-scale RDF data using Spark Streaming. Finally, in the future, we intend to develop a declarative plugin for the current work using Quality Metric Language (QML) [17], which gives users the ability to express, customize and enhance quality metrics. Besides the above mentioned future direction, as a long-term vision, we plan to offer DistQualityAssessment as a Service. It is obvious that the quality assessment of RDF is not considered a one-off event but, on the contrary, intends to be constantly evolving. Therefore, these changes have to be reflected as well. However, given the large-scale of such RDF datasets, one should consider various strategies for crawling and assessing the quality of the data. Currently, there is a number of crawlers available, such as the LODStats[1] project, which has crawled RDF data from metadata portals for the past eight years. It interacts with the CKAN dataset metadata registry to obtain a comprehensive picture of the current state of the Data Web. While crawling the data, and specifically over large-scale RDF datasets, data quality check is a must. The current solution does not provide such option,

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

therefore, integration of our approach with the LODStats project could bring another view w.r.t to the quality of the data

- *Scalable RDF Querying* – In this thesis, we showed that the application of OBDA tooling to Big Data frameworks achieves promising results in terms of scalability. We present a working prototype implementation that can serve as a baseline for further research. Our next steps include evaluating other tools, such as Ontop [105], and analyze how their performance in the Big Data setting can be improved further. For example, we intend to investigate how OBDA tools can be combined with dictionary encoding of RDF terms as integers and evaluate the effects. In addition to that, we plan to further extend our parser to support more SPARQL fragments and adding statistics to the query engine while evaluating queries. We want to analyze the query performance in the large-scale RDF datasets and explore prospects for improvement. For example, we intend to investigate the re-ordering of the BGPs and evaluate the effects on query execution time. In this regard, efficient strategies, as well as a detailed cost function for query plan optimization, have to be considered. In addition, we also plan to consider other data management operations i.e. additions, updates, deletions and materialization of the results. One solution could be considering the Delta[2] lake solution as an alternative for storage layer that brings ACID transactions to RDF data management solutions.

In addition to the future work mentioned above, we see a potential future direction as a long term vision of this work, in an attempt to foster the interest in scalable processing of RDF datasets.

- *Adaptive Distributed RDF Querying* – Often the power of freedom while designing SPARQL queries leads to very complex and performance deficits in SPARQL query evaluation. Within our SPARQL query evaluators, we will go beyond that by developing adaptive data distribution strategies, that generate and optimize index structures and distribute data based on anticipated query workloads of particular inference or machine learning algorithms.

- *Efficient Recommendation System for RDF Partitioners* – In order to store and query big RDF datasets efficiently in distributed environments, different partitioning techniques need to be implemented. Several techniques have been proposed for splitting Big RDF Data, ranging from vertical (cf. Section 6.1), hash, graph to semantic-based (cf. Section 6.2) partitioners. However, the selection of the "best partitioner" depends highly on the structure of the dataset and the query efficiency and effectiveness are coupled to the query engine used. We aim to develop a recommender system that will suggest the "best partitioner" for both of our SPARQL query evaluators based on the structure of the data gathered from DistLODStats (cf. Chapter 4) and specific requirements.

- *A Powerful Benchmarking Suite* – In order to decide which distributed SPARQL query evaluator performs best for specific query loads over a large-scale RDF dataset, it is required to perform benchmarks. Benchmarking is an extremely tedious task demanding repetitive manual effort, therefore it is required to automate the whole process. However, there are currently no benchmarking frameworks that support benchmarking and comparing diverse distributed SPARQL query evaluators. To this end, we will make use of the existing benchmarking platform i.e. LITMUS [106], HOBBIT [107] and extend them toward supporting distributed settings.

[2] https://delta.io/

8.3 Closing Remarks

With the increasing amount of the RDF data, processing large-scale RDF datasets are constantly facing challenges and a lot of potential for exploration. During this thesis, we have shown the benefits of distributed computing frameworks to successfully tackle the problem of scalable and efficient processing of RDF datasets. More specifically, we have presented the details of three core components: 1) scalable RDF dataset statistics evaluation, 2) distributed quality assessment of large amounts of RDF data, and 3) efficient and scalable SPARQL query evaluators. In addition, we have shown the usage of the proposed techniques into real-world use cases. Future research work can build upon the contributions presented during this thesis as a starting point for a comprehensive and out-of-the-box scalable processing of large-scale RDF datasets. The main contributions of this thesis have been integrated within the SANSA framework and are making an impact on the semantic web community and several semantic web applications in the big data era – resulting in a SANSA framework and being used in many European research projects.

Bibliography

[1] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen, *S2RDF: RDF Querying with SPARQL on Spark*, Proc. VLDB Endow. **9** (2016) 804, ISSN: 2150-8097, URL: http://dx.doi.org/10.14778/2977797.2977806 (cit. on pp. 1, 3, 28, 63, 67).

[2] Z. Xu, W. Chen, L. Gai, and T. Wang, "Sparkrdf: In-memory distributed rdf management framework for large-scale social data," *International Conference on Web-Age Information Management*, Springer, 2015 337 (cit. on p. 2).

[3] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris, "H 2 RDF+: High-performance distributed joins over large-scale RDF graphs," *Big Data, 2013 IEEE International Conference on*, IEEE, 2013 255 (cit. on p. 2).

[4] N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris, "H2RDF: adaptive query processing on RDF data in the cloud.," *Proceedings of the 21st International Conference on World Wide Web*, ACM, 2012 397 (cit. on p. 2).

[5] V. R. Benjamins, J. Contreras, O. Corcho, and A. Gómez-pérez, "Six Challenges for the Semantic Web," *In KR2002 Semantic Web Workshop*, 2002 2004 (cit. on p. 2).

[6] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer, *DBpedia - A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia*, Semantic Web Journal **6** (2015) 167, URL: http://jens-lehmann.org/files/2014/swj_dbpedia.pdf (cit. on pp. 2, 39, 55, 95).

[7] A. Zaveri, A. Rula, A. Maurino, R. Pietrobon, J. Lehmann, and S. Auer, *Quality assessment for linked data: A survey*, Semantic Web **7** (2015) 63 (cit. on pp. 2, 4, 26, 49, 52, 60).

[8] F. Michel, *Integrating heterogeneous data sources in the Web of data*, Theses: Université Côte d'Azur, 2017, URL: https://tel.archives-ouvertes.fr/tel-01508602 (cit. on p. 2).

[9] A. Tonon, G. Demartini, and P. Cudré-Mauroux, "Combining Inverted Indices and Structured Search for Ad-hoc Object Retrieval," *Proceedings of the 35th International ACM SIGIR Conference on Research and Development*

in Information Retrieval, SIGIR '12, ACM, 2012 125, ISBN: 978-1-4503-1472-5, URL: http://doi.acm.org/10.1145/2348283.2348304 (cit. on p. 2).

[10] A. Dutta, C. Meilicke, and S. P. Ponzetto, "A Probabilistic Approach for Integrating Heterogeneous Knowledge Sources," *The Semantic Web: Trends and Challenges*, ed. by V. Presutti, C. d'Amato, F. Gandon, M. d'Aquin, S. Staab, and A. Tordai, Springer International Publishing, 2014 286, ISBN: 978-3-319-07443-6 (cit. on p. 2).

[11] P.-Y. Vandenbussche, G. A. Atemezing, M. Poveda-Villalón, and B. Vatant, *Linked Open Vocabularies (LOV): a gateway to reusable semantic vocabularies on the Web*, Semantic Web **Preprint** (2015) 1 (cit. on pp. 2, 31).

[12] A. Langegger and W. Wöß, "RDFStats - An Extensible RDF Statistics Generator and Library.," *DEXA Workshops*, IEEE Computer Society, 2009 79, ISBN: 978-0-7695-3763-4, URL: http://dblp.uni-trier.de/db/conf/dexaw/dexaw2009.html#LangeggerW09 (cit. on pp. 2, 24, 31).

[13] I. Ermilov, M. Martin, J. Lehmann, and S. Auer, "Linked Open Data Statistics: Collection and Exploitation," *Proceedings of the 4th Conference on Knowledge Engineering and Semantic Web*, 2013, URL: http://svn.aksw.org/papers/2013/KESW_LODStats_Demo/public.pdf (cit. on pp. 2, 31).

[14] J. Debattista, C. Lange, S. Auer, and D. Cortis, *Evaluating the quality of the LOD cloud: An empirical investigation*, Semantic Web **9** (2018) 859, URL: https://doi.org/10.3233/SW-180306 (cit. on pp. 3, 49).

[15] M. Färber, F. Bartscherer, C. Menne, and A. Rettinger, *Linked data quality of DBpedia, Freebase, OpenCyc, Wikidata, and YAGO*, Semantic Web **9** (2018) 77 (cit. on pp. 3, 49).

[16] W. Beek, F. Ilievski, J. Debattista, S. Schlobach, and J. Wielemaker, *Literally better: Analyzing and improving the quality of literals*, Semantic Web **9** (2018) (cit. on pp. 3, 49).

[17] J. Debattista, S. Auer, and C. Lange, *Luzzu—A Methodology and Framework for Linked Data Quality Assessment*, Journal of Data and Information Quality (JDIQ) **8** (2016) 4 (cit. on pp. 3, 25, 26, 49, 52, 55, 56, 102).

[18] N. Mihindukulasooriya, R. García-Castro, and A. Gómez-Pérez, "LD Sniffer: A Quality Assessment Tool for Measuring the Accessibility of Linked Data," *Knowledge Engineering and Knowledge Management*, Springer International Publishing, 2016 149, ISBN: 978-3-319-58694-6 (cit. on pp. 3, 26, 49).

[19] D. Graux, L. Jachiet, P. Genevès, and N. Layaïda,
"SPARQLGX: Efficient Distributed Evaluation of SPARQL with Apache Spark,"
The Semantic Web – ISWC 2016,
ed. by P. Groth, E. Simperl, A. Gray, M. Sabou, M. Krötzsch, F. Lecue, F. Flöck, and Y. Gil,
Springer International Publishing, 2016 80, ISBN: 978-3-319-46547-0
(cit. on pp. 3, 28, 29, 63, 67, 76).

[20] J. Demter, S. Auer, M. Martin, and J. Lehmann,
"LODStats—An Extensible Framework for High-performance Dataset Analytics,"
Proceedings of the EKAW 2012, Lecture Notes in Computer Science (LNCS) 7603,
Springer, 2012, URL: http://svn.aksw.org/papers/2011/RDFStats/public.pdf
(cit. on pp. 3, 32).

[21] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin,
S. Shenker, and I. Stoica,
"Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,"
Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation,
2012 (cit. on pp. 4, 20, 28, 31, 49).

[22] G. Sejdiu, I. Ermilov, J. Lehmann, and M. Nadjib-Mami,
"DistLODStats: Distributed Computation of RDF Dataset Statistics,"
Proceedings of 17th International Semantic Web Conference, 2018,
URL: http://jens-lehmann.org/files/2018/iswc_distlodstats.pdf
(cit. on pp. 5, 7, 23, 32).

[23] G. Sejdiu, I. Ermilov, J. Lehmann, and M.-N. Mami, "STATisfy Me: What are my Stats?"
Proceedings of 17th International Semantic Web Conference, Poster & Demos, 2018,
URL: http://jens-lehmann.org/files/2018/iswc_statisfy_pd.pdf
(cit. on pp. 5, 7, 32).

[24] G. Sejdiu, A. Rula, J. Lehmann, and H. Jabeen,
"A Scalable Framework for Quality Assessment of RDF Datasets,"
Proceedings of 18th International Semantic Web Conference, 2019, URL:
http://jens-lehmann.org/files/2019/iswc_dist_quality_assessment.pdf
(cit. on pp. 5, 7, 23, 50).

[25] C. Stadler, G. Sejdiu, D. Graux, and J. Lehmann, "Sparklify: A Scalable Software Component
for Efficient evaluation of SPARQL queries over distributed RDF datasets,"
Proceedings of 18th International Semantic Web Conference, 2019,
URL: http://jens-lehmann.org/files/2019/iswc_sparklify.pdf
(cit. on pp. 6, 7, 23, 64, 76).

[26] G. Sejdiu, D. Graux, I. Khan, I. Lytra, H. Jabeen, and J. Lehmann,
"Towards A Scalable Semantic-based Distributed Approach for SPARQL query evaluation,"
15th International Conference on Semantic Systems (SEMANTiCS), 2019,
URL: https://gezimsejdiu.github.io/publications/semantic_based_query_paper_SEMANTICS2019.pdf (cit. on pp. 6, 7, 23, 64).

[27] C. Stadler, G. Sejdiu, D. Graux, and J. Lehmann,
"Querying large-scale RDF datasets using the SANSA framework,"
Proceedings of 18th International Semantic Web Conference (ISWC), Poster & Demos, 2019,
URL: https://gezimsejdiu.github.io/publications/sansa-sparklify-ISWC-demo.pdf (cit. on pp. 6, 7, 64).

[28] J. Pfeffer, A. Beregszazi, C. Detrio, H. Junge, J. Chow, M. Oancea, M. Pietrzak,
S. Khatchadourian, and S. Bertolo, *EthOn - An Ethereum ontology*, 2016 (cit. on pp. 6, 88).

[29] S. Auer, S. Scerri, A. Versteden, E. Pauwels, A. Charalambidis, S. Konstantopoulos,
J. Lehmann, H. Jabeen, I. Ermilov, G. Sejdiu, A. Ikonomopoulos, S. Andronopoulos,
M. Vlachogiannis, C. Pappas, A. Davettas, I. A. Klampanos, E. Grigoropoulos, V. Karkaletsis,
V. de Boer, R. Siebes, M. N. Mami, S. Albani, M. Lazzarini, P. Nunes, E. Angiuli, N. Pittaras,
G. Giannakopoulos, G. Argyriou, G. Stamoulis, G. Papadakis, M. Koubarakis,
P. Karampiperis, A.-C. N. Ngomo, and M.-E. Vidal,
"The BigDataEurope Platform - Supporting the Variety Dimension of Big Data,"
17th International Conference on Web Engineering, 2017,
URL: http://jens-lehmann.org/files/2017/icwe_bde.pdf (cit. on pp. 7, 93).

[30] I. Ermilov, J. Lehmann, G. Sejdiu, L. Bühmann, P. Westphal, C. Stadler, S. Bin,
N. Chakraborty, H. Petzka, M. Saleem, A.-C. N. Ngonga, and H. Jabeen,
"The Tale of Sansa Spark,"
Proceedings of 16th International Semantic Web Conference, Poster & Demos,
Best Demo Award, 2017,
URL: http://jens-lehmann.org/files/2017/iswc_pd_sansa.pdf
(cit. on pp. 7, 39, 45, 54, 66, 73, 95).

[31] J. Lehmann, G. Sejdiu, L. Bühmann, P. Westphal, C. Stadler, I. Ermilov, S. Bin,
N. Chakraborty, M. Saleem, A.-C. N. Ngonga, and H. Jabeen,
"Distributed Semantic Analytics using the SANSA Stack,"
Proceedings of 16th International Semantic Web Conference - Resources Track (ISWC'2017),
2017, URL: http://svn.aksw.org/papers/2017/ISWC_SANSA_SoftwareFramework/public.pdf
(cit. on pp. 7, 39, 54, 64, 65, 72, 83, 88).

[32] D. Sui, G. Sejdiu, D. Graux, and J. Lehmann,
"The Hubs and Authorities Transaction NetworkAnalysis using the SANSA framework,"
15th International Conference on Semantic Systems (SEMANTiCS), Poster & Demos, 2019,
URL: https://gezimsejdiu.github.io/publications/sansa-hubs-and-authorities-transaction-semantics19-poster.pdf (cit. on p. 7).

[33] R. Dadwal, D. Graux, G. Sejdiu, H. Jabeen, and J. Lehmann,
"Clustering Pipelines of large RDF POI Data,"
Proceedings of 16th Extended Semantic Web Conference (ESWC 2019), Poster & Demos,
2019, URL: https://gezimsejdiu.github.io/publications/piping-clustering-eswc19-poster.pdf (cit. on p. 7).

[34] D. Graux, G. Sejdiu, H. Jabeen, J. Lehmann, D. Sui, D. Muhs, and J. Pfeffer, "Profiting from Kitties on Ethereum: Leveraging Blockchain RDF with SANSA," *14th International Conference on Semantic Systems, Poster & Demos*, 2018, URL: http://jens-lehmann.org/files/2018/semantics_ethereum_pd.pdf (cit. on p. 7).

[35] I. Ermilov, A.-C. N. Ngomo, A. Versteden, H. Jabeen, G. Sejdiu, G. Argyriou, L. Selmi, J. Jakobitsch, and J. Lehmann, "Managing Lifecycle of Big Data Applications," *KESW*, 2017, URL: https://svn.aksw.org/papers/2017/KESW_BDE_Workflow/public.pdf (cit. on p. 7).

[36] T. Berners-Lee, J. Hendler, and O. Lassila, *The Semantic Web*, Scientific American **284** (2001) 34, URL: http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21 (cit. on p. 11).

[37] D. Wood, M. Lanthaler, and R. Cyganiak, *RDF 1.1 Concepts and Abstract Syntax*, W3C Recommendation, W3C, 2014, URL: http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/ (cit. on p. 12).

[38] A. Seaborne and G. Carothers, *RDF 1.1 N-Triples*, W3C Recommendation, W3C, 2014, URL: http://www.w3.org/TR/2014/REC-n-triples-20140225/ (cit. on p. 14).

[39] G. Carothers and E. Prud'hommeaux, *RDF 1.1 Turtle*, W3C Recommendation, W3C, 2014, URL: http://www.w3.org/TR/2014/REC-turtle-20140225/ (cit. on p. 14).

[40] G. Schreiber and F. Gandon, *RDF 1.1 XML Syntax*, W3C Recommendation, W3C, 2014, URL: http://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/ (cit. on p. 15).

[41] A. Seaborne and E. Prud'hommeaux, *SPARQL Query Language for RDF*, W3C Recommendation, W3C, 2008, URL: http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/ (cit. on p. 16).

[42] J. Pérez, M. Arenas, and C. Gutierrez, *Semantics and Complexity of SPARQL*, ACM Trans. Database Syst. **34** (2009) 16:1, ISSN: 0362-5915, URL: http://doi.acm.org/10.1145/1567274.1567278 (cit. on p. 16).

[43] T. White, *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale*, 4th, O'Reilly Media, Inc., 2015 (cit. on p. 17).

[44] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, IEEE Computer Society, 2010 1, ISBN: 978-1-4244-7152-2, URL: http://dx.doi.org/10.1109/MSST.2010.5496972 (cit. on p. 18).

[45] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, ACM, 2003 29, ISBN: 1-58113-757-5, URL: http://doi.acm.org/10.1145/945445.945450 (cit. on p. 18).

[46] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 2004 137 (cit. on p. 18).

[47] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph Processing in a Distributed Dataflow Framework," *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, USENIX Association, 2014 599, ISBN: 978-1-931971-16-4, URL: http://dl.acm.org/citation.cfm?id=2685048.2685096 (cit. on p. 21).

[48] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs," *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, USENIX, 2012 17, ISBN: 978-1-931971-96-6, URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez (cit. on p. 21).

[49] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational Data Processing in Spark," *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, ACM, 2015 1383, ISBN: 978-1-4503-2758-9, URL: http://doi.acm.org/10.1145/2723372.2742797 (cit. on pp. 21, 28).

[50] J. Zhao, M. Hausenblas, K. Alexander, and R. Cyganiak, *Describing Linked Datasets with the VoID Vocabulary*, W3C Note, http://www.w3.org/TR/2011/NOTE-void-20110303/: W3C, 2011 (cit. on pp. 24, 38).

[51] F. Corcoglioniti, M. Rospocher, M. Mostarda, and M. Amadori, "Processing billions of RDF triples on a single machine using streaming and sorting," *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, ACM, 2015 368 (cit. on p. 24).

[52] S. Khatchadourian and M. P. Consens, "ExpLOD: Summary-Based Exploration of Interlinking and RDF Usage in the Linked Open Data Cloud," *The Semantic Web: Research and Applications*, ed. by L. Aroyo, G. Antoniou, E. Hyvönen, A. ten Teije, H. Stuckenschmidt, L. Cabral, and T. Tudorache, Springer Berlin Heidelberg, 2010 272, ISBN: 978-3-642-13489-0 (cit. on p. 24).

[53] C. Böhm, F. Naumann, Z. Abedjan, D. Fenz, T. Grütze, D. Hefenbrock, M. Pohl, and D. Sonnabend, *Profiling linked open data with ProLOD*, 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010) (2010) 175 (cit. on p. 24).

[54] Z. Abedjan, T. Grütze, A. Jentzsch, and F. Naumann, *Profiling and mining RDF data with ProLOD++*, 2014 IEEE 30th International Conference on Data Engineering (2014) 1198 (cit. on p. 24).

[55] N. Mihindukulasooriya, R. García-Castro, F. Priyatna, E. Ruckhaus, and N. Saturno, "A Linked Data Profiling Service for Quality Assessment," *MEPDaW/LDQ@ESWC*, 2017 (cit. on p. 24).

[56] E. Mäkelä,
"Aether–generating and viewing extended VoID statistical descriptions of RDF datasets," *European Semantic Web Conference*, Springer, 2014 429 (cit. on p. 25).

[57] B. Forchhammer, A. Jentzsch, and F. Naumann,
"LODOP - Multi-Query Optimization for Linked Data Profiling Queries," *International Workshop on Dataset PROFIling and fEderated Search for Linked Data (PROFILES)*, 2014 (cit. on p. 25).

[58] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan,
Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics,
Proc. VLDB Endow. **8** (2015), ISSN: 2150-8097,
URL: https://doi.org/10.14778/2831360.2831365 (cit. on p. 25).

[59] D. Becker, T. D. King, and B. McMullen, "Big data, big data quality problem," *International Conference on Big Data*, IEEE, 2015 2644 (cit. on p. 25).

[60] D. Rao, V. N. Gudivada, and V. V. Raghavan, "Data quality issues in big data," *International Conference on Big Data*, IEEE, 2015 2654 (cit. on p. 25).

[61] L. Cai and Y. Zhu,
The challenges of data quality and data quality assessment in the big data era,
Data Science Journal **14** (2015) (cit. on p. 25).

[62] T. Catarci, M. Scannapieco, M. Console, and C. Demetrescu, "My (fair) big data," *International Conference on Big Data*, IEEE, 2017 2974 (cit. on p. 25).

[63] C. Batini, A. Rula, M. Scannapieco, and G. Viscusi, *From Data Quality to Big Data Quality*, J. Database Manag. **26** (2015) 60, URL: https://doi.org/10.4018/JDM.2015010103 (cit. on p. 25).

[64] D. Kontokostas, P. Westphal, S. Auer, S. Hellmann, J. Lehmann, R. Cornelissen, and A. Zaveri, "Test-driven evaluation of linked data quality," *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014*, 2014 747 (cit. on p. 25).

[65] S. Bonner, A. S. McGough, I. Kureshi, J. Brennan, G. Theodoropoulos, L. Moss, D. Corsar, and G. Antoniou, "Data quality assessment and anomaly detection via map/reduce and linked data: A case study in the medical domain," *International Conference on Big Data*, IEEE, 2015 (cit. on p. 26).

[66] S. Benbernou and M. Ouziri,
"Enhancing data quality by cleaning inconsistent big RDF data,"
International Conference on Big Data, IEEE, 2017 74 (cit. on p. 26).

[67] E. Ruckhaus, O. Baldizán, and M.-E. Vidal, "Analyzing Linked Data Quality with LiQuate," *On the Move to Meaningful Internet Systems: OTM 2013 Workshops*, ed. by Y. T. Demey and H. Panetto, Springer Berlin Heidelberg, 2013 629, ISBN: 978-3-642-41033-8 (cit. on p. 26).

[68] C. Bizer and R. Cyganiak,
Quality-Driven Information Filtering Using the WIQA Policy Framework,
Web Semant. **7** (2009) 1, ISSN: 1570-8268,
URL: https://doi.org/10.1016/j.websem.2008.02.005 (cit. on p. 26).

[69] C. Guéret, P. Groth, C. Stadler, and J. Lehmann, "Assessing Linked Data Mappings Using Network Measures," *The Semantic Web: Research and Applications*, ed. by E. Simperl, P. Cimiano, A. Polleres, O. Corcho, and V. Presutti, Springer Berlin Heidelberg, 2012 87, ISBN: 978-3-642-30284-8 (cit. on p. 26).

[70] D. Kontokostas, P. Westphal, S. Auer, S. Hellmann, J. Lehmann, R. Cornelissen, and A. Zaveri, "Test-Driven Evaluation of Linked Data Quality," *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, Association for Computing Machinery, 2014 747, ISBN: 9781450327442, URL: https://doi.org/10.1145/2566486.2568002 (cit. on p. 26).

[71] J. Broekstra, A. Kampman, and F. van Harmelen, "Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema," *The Semantic Web - ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002, Proceedings*, 2002 54 (cit. on p. 27).

[72] K. Wilkinson, "Jena Property Table Implementation," *SSWS*, 2006 35 (cit. on p. 27).

[73] D. J. Abadi, A. Marcus, S. Madden, and K. Hollenbach, *SW-Store: a vertically partitioned DBMS for Semantic Web data management*, VLDB J. **18** (2009) 385 (cit. on p. 27).

[74] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao, *gStore: Answering SPARQL Queries via Subgraph Matching*, PVLDB **4** (2011) 482 (cit. on p. 27).

[75] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach, "Scalable Semantic Web Data Management Using Vertical Partitioning," *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, 2007 411 (cit. on p. 27).

[76] C. Weiss, P. Karras, and A. Bernstein, *Hexastore: sextuple indexing for semantic web data management*, PVLDB **1** (2008) 1008 (cit. on p. 27).

[77] T. Neumann and G. Weikum, *The RDF-3X engine for scalable management of RDF data*, VLDB J. **19** (2010) 91 (cit. on p. 27).

[78] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald, "TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing," *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, 2014 289 (cit. on p. 27).

[79] K. Lee and L. Liu, *Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning*, Proc. VLDB Endow. **6** (2013) 1894, ISSN: 2150-8097 (cit. on p. 27).

[80] M. Wylot and P. Cudré-Mauroux, *DiploCloud: Efficient and Scalable Management of RDF Data in the Cloud*, IEEE Trans. Knowl. Data Eng. **28** (2016) 659 (cit. on p. 27).

[81] R. Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, and M. Sahli, *Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning*, The VLDB Journal—The International Journal on Very Large Data Bases **25** (2016) 355 (cit. on p. 27).

[82] A. Schätzle, M. Przyjaciel-Zablocki, A. Neu, and G. Lausen, "Sempala: Interactive SPARQL Query Processing on Hadoop," *The Semantic Web – ISWC 2014*, ed. by P. Mika, T. Tudorache, A. Bernstein, C. Welty, C. Knoblock, D. Vrandečić, P. Groth, N. Noy, K. Janowicz, and C. Goble, Springer International Publishing, 2014 164, ISBN: 978-3-319-11964-9 (cit. on p. 27).

[83] A. Schätzle, M. Przyjaciel-Zablocki, and G. Lausen, "PigSPARQL: Mapping SPARQL to Pig Latin," *Proceedings of the International Workshop on Semantic Web Information Management*, SWIM '11, ACM, 2011 4:1, ISBN: 978-1-4503-0651-5, URL: http://doi.acm.org/10.1145/1999299.1999303 (cit. on p. 27).

[84] R. Punnoose, A. Crainiceanu, and D. Rapp, "Rya: A Scalable RDF Triple Store for the Clouds," *Proceedings of the 1st International Workshop on Cloud Intelligence*, Cloud-I '12, ACM, 2012 4:1, ISBN: 978-1-4503-1596-8, URL: http://doi.acm.org/10.1145/2347673.2347677 (cit. on p. 28).

[85] V. Khadilkar, M. Kantarcioglu, B. Thuraisingham, and P. Castagna, "Jena-HBase: A Distributed, Scalable and Efficient RDF Triple Store," *Proceedings of the 2012th International Conference on Posters &; Demonstrations Track - Volume 914*, ISWC-PD'12, 2012 85 (cit. on p. 28).

[86] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris, "H2RDF+: High-performance distributed joins over large-scale RDF graphs," *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*, 2013 255 (cit. on p. 28).

[87] K. Rohloff and R. E. Schantz, "High-performance, Massively Scalable Distributed Systems Using the MapReduce Software Framework: The SHARD Triple-store," *Programming Support Innovations for Emerging Distributed Applications*, PSI EtA '10, ACM, 2010 4:1, ISBN: 978-1-4503-0544-0, URL: http://doi.acm.org/10.1145/1940747.1940751 (cit. on pp. 28, 72, 76).

[88] D. Graux, L. Jachiet, P. Genevès, and N. Layaïda, "A Multi-Criteria Experimental Ranking of Distributed SPARQL Evaluators," *Big Data 2018 - IEEE International Conference on Big Data*, IEEE, 2018 1, URL: https://hal.inria.fr/hal-01381781 (cit. on p. 29).

[89] A.-C. Ngonga Ngomo, S. Auer, J. Lehmann, and A. Zaveri, "Introduction to Linked Data and Its Lifecycle on the Web," *Reasoning Web*, 2014 (cit. on pp. 31, 49).

[90] C. Stadler, J. Lehmann, K. Höffner, and S. Auer, *LinkedGeoData: A Core for a Web of Spatial Open Data*, Semantic Web Journal **3** (2012) 333, URL: http://jens-lehmann.org/files/2012/linkedgeodata2.pdf (cit. on pp. 39, 55).

[91] C. Bizer and A. Schultz, *The Berlin SPARQL Benchmark*, Int. J. Semantic Web Inf. Syst. **5** (2009) 1 (cit. on pp. 39, 55).

[92] C. Batini and M. Scannapieco, *Data and Information Quality - Dimensions, Principles and Techniques*, Data-Centric Systems and Applications, Springer, 2016, ISBN: 978-3-319-24104-3, URL: https://doi.org/10.1007/978-3-319-24106-7 (cit. on p. 50).

[93] A. Isaac and R. Albertoni, *Data on the Web Best Practices: Data Quality Vocabulary*, W3C Note, https://www.w3.org/TR/2016/NOTE-vocab-dqv-20161215/: W3C, 2016 (cit. on p. 54).

[94] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "Scalable Semantic Web Data Management Using Vertical Partitioning," *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, VLDB Endowment, 2007 411, ISBN: 9781595936493 (cit. on p. 63).

[95] C. Stadler, J. Unbehauen, P. Westphal, M. A. Sherif, and J. Lehmann, "Simplified RDB2RDF Mapping," *Proceedings of the 8th Workshop on Linked Data on the Web (LDOW2015), Florence, Italy*, 2015, URL: http://svn.aksw.org/papers/2015/LDOW_SML/paper-camery-ready_public.pdf (cit. on p. 66).

[96] Y. Guo, Z. Pan, and J. Heflin, *LUBM: A benchmark for OWL knowledge base systems*, J. Web Semant. **3** (2005) 158 (cit. on pp. 66, 75, 117).

[97] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee, "Diversified Stress Testing of RDF Data Management Systems," *International Semantic Web Conference*, 2014 (cit. on pp. 66, 75, 117).

[98] G. Wood, *Ethereum: A secure decentralised generalised transaction ledger*, Ethereum project yellow paper **151** (2014) 1 (cit. on p. 88).

[99] D. Vrandečić and M. Krötzsch, *Wikidata: A Free Collaborative Knowledgebase*, Commun. ACM **57** (2014) 78, ISSN: 0001-0782, URL: http://doi.acm.org/10.1145/2629489 (cit. on p. 95).

[100] S. Athanasiou, G. Giannopoulos, D. Graux, N. Karagiannakis, J. Lehmann, A.-C. N. Ngomo, K. Patroumpas, M. A. Sherif, and D. Skoutas, "Big POI data integration with Linked Data technologies," *22nd International Conference on Extending Database Technology (EDBT), Lisbon, Portugal*, 2019 477 (cit. on p. 97).

[101] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, 2013 3111 (cit. on p. 97).

[102] J. MacQueen, "Some methods for classification and analysis of multivariate observations," *5th Berkeley Symposium on Mathematical Statistics and Probability*, 1967 281 (cit. on p. 97).

[103] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al., *A density-based algorithm for discovering clusters in large spatial databases with noise.*, Kdd **96** (1996) 226 (cit. on p. 97).

[104] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias, *Binary RDF Representation for Publication and Exchange (HDT)*, Web Semantics: Science, Services and Agents on the World Wide Web **19** (2013) 22, URL: `http://www.websemanticsjournal.org/index.php/ps/article/view/328` (cit. on p. 102).

[105] D. Calvanese, B. Cogrel, S. Komla-Ebri, R. Kontchakov, D. Lanti, M. Rezk, M. Rodriguez-Muro, and G. Xiao, *Ontop: Answering SPARQL queries over relational databases*, Semantic Web **8** (2017) 471 (cit. on p. 103).

[106] H. Thakkar, M. Dubey, G. Sejdiu, A.-C. Ngonga Ngomo, J. Debattista, C. Lange, J. Lehmann, S. Auer, and M.-E. Vidal, *LITMUS: An Open Extensible Framework for Benchmarking RDF Data Management Solutions*, 2016, arXiv: `1608.02800 [cs.PF]` (cit. on p. 103).

[107] M. Röder, D. Kuchelev, and A.-C. Ngonga Ngomo, *HOBBIT: A platform for benchmarking Big Linked Data*, Data Science **Preprint** (2019) 1, ISSN: 2451-8492 (cit. on p. 103).

APPENDIX A

SANSA Framework Release History

In the course of the thesis, the following releases of software components were produced (under the *Apache Licence 2.0*).

- SANSA[1] software component related releases[2]:
  - **SANSA v0.7** *2020-01*: SPARQL query engine over compressed RDF data. Refactoring quality assessment metrics. Further alignment and development of the functionality on the Flink module. Support for quality assessment over streaming RDF data. Bug fixes.
  - **SANSA v0.6** *2019-06*: Support for RDF compression techniques. Refactoring semantic-based query engine. Support for ingestion of additional RDF formats. Align and perfrom evaluations on the Sparklify query engine (see Section 6.1 of Chapter 6 for more details). Bug fixes.
  - **SANSA v0.5** *2018-12*: Further improvements for Scalable RDF Quality Assessment. Refactoring semantic-based partitioning and RDF Dataset Statistics. Bug fixes.
  - **SANSA v0.4** *2018-06*: Further support for Scalable RDF Quality Assessment. Support for ingestion of additional RDF formats. Further improvements for semantic-based partitioning. Support for graph-based partitioning. Query engine on top of semantic-based partitioning. Bug fixes.
  - **SANSA v0.3** *2017-12*: Support for Scalable RDF Quality Assessment(see Chapter 5 for more details). Support for ingestion of additional RDF formats. Support for semantic-based partitioning (see Section 6.2 of Chapter 6 for more details).
  - **SANSA v0.2** *2017-06*: Support for Scalable RDF Dataset Statistics (see Chapter 4 for more details).
  - **SANSA v0.1** *2016-12*: Initial SANSA release. SANSA website and guidelines. Support for reading and writing RDF files in N-Triples format.
- Big Data Europe Platform[3]

[1] https://github.com/SANSA-Stack

[2] This list only includes components that are part of the thesis. The features of SANSA itself are more than those mentioned.

[3] https://github.com/big-data-europe

  - **BDE v1** *2017-11*: Integrate SANSA framework with the Big Data Europe Platform. Build the stack for the *SC4: Smart Green and Integrated Transport* on the Big Data Europe Integrator Platform.
  - **BDE v1** *2016-11*: Dockerized Big Data Europe Components (e.g Apache Spark, Apache Flink).

- DL-Learner Framework[4]
  - **DL-Learner v1.4.0** *2019-09*: Docker image for DL-Learner.

[4] https://github.com/SmartDataAnalytics/DL-Learner

APPENDIX B

SPARQL Benchmark Queries

We make use of two well-known SPARQL benchmarks for our query engine evaluations (cf. Chapter 6): the *Waterloo SPARQL Diversity Test Suite (WatDiv)* v0.6 [97] and *Lehigh University Benchmark (LUBM)* v3.1 [96].

In this section, we list the queries used during our benchmarks.

B.1 LUBM SPARQL Queries

```
==Q1==
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE {
    ?X rdf:type ub:GraduateStudent .
    ?X ub:takesCourse <http://www.Department0.University0.edu/GraduateCourse0> .
}

==Q2==
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X ?Y ?Z
WHERE {
    ?X rdf:type ub:GraduateStudent .
    ?Y rdf:type ub:University .
    ?Z rdf:type ub:Department .
    ?X ub:memberOf ?Z .
    ?Z ub:subOrganizationOf ?Y .
    ?X ub:undergraduateDegreeFrom ?Y
}

==Q3==
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE {
    ?X rdf:type ub:Publication .
    ?X ub:publicationAuthor <http://www.Department0.University0.edu/AssistantProfessor0>
}

==Q4==
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
```

```
SELECT ?X ?Y1 ?Y2 ?Y3
WHERE {
    ?X rdf:type ub:Professor .
    ?X ub:worksFor <http://www.Department0.University0.edu> .
    ?X ub:name ?Y1 .
    ?X ub:emailAddress ?Y2 .
    ?X ub:telephone ?Y3
}

==Q5==
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE {
    ?X rdf:type ub:Person .
    ?X ub:memberOf <http://www.Department0.University0.edu>
}

==Q6==
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE {
    ?X rdf:type ub:Student
}

==Q7==
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X ?Y
WHERE {
    ?X rdf:type ub:Student .
    ?Y rdf:type ub:Course .
    ?X ub:takesCourse ?Y .
    <http://www.Department0.University0.edu/AssociateProfessor0> ub:teacherOf ?Y
}

==Q8==
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X ?Y ?Z
WHERE {
    ?X rdf:type ub:Student .
    ?Y rdf:type ub:Department .
    ?X ub:memberOf ?Y .
    ?Y ub:subOrganizationOf <http://www.University0.edu> .
    ?X ub:emailAddress ?Z
}

==Q9==
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X ?Y ?Z
WHERE {
    ?X rdf:type ub:Student .
    ?Y rdf:type ub:Faculty .
    ?Z rdf:type ub:Course .
    ?X ub:advisor ?Y .
    ?Y ub:teacherOf ?Z .
    ?X ub:takesCourse ?Z
}

==Q10==
```

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE {
    ?X rdf:type ub:Student .
    ?X ub:takesCourse <http://www.Department0.University0.edu/GraduateCourse0>
}

==Q11==
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE {
    ?X rdf:type ub:ResearchGroup .
    ?X ub:subOrganizationOf <http://www.University0.edu>
}

==Q12==
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X ?Y
WHERE {
    ?X rdf:type ub:Chair .
    ?Y rdf:type ub:Department .
    ?X ub:worksFor ?Y .
    ?Y ub:subOrganizationOf <http://www.University0.edu>
}

==Q13==
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE {
    ?X rdf:type ub:Person .
    <http://www.University0.edu> ub:hasAlumnus ?X
}

==Q14==
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE {
    ?X rdf:type ub:UndergraduateStudent
}
```

B.2 WatDiv SPARQL Queries

```
==C1==
SELECT ?v0 ?v4 ?v6 ?v7
WHERE {
    ?v0 <http://schema.org/caption> ?v1 .
    ?v0 <http://schema.org/text> ?v2 .
    ?v0 <http://schema.org/contentRating> ?v3 .
    ?v0 <http://purl.org/stuff/rev#hasReview> ?v4 .
    ?v4 <http://purl.org/stuff/rev#title> ?v5 .
    ?v4 <http://purl.org/stuff/rev#reviewer> ?v6 .
    ?v7 <http://schema.org/actor> ?v6 .
    ?v7 <http://schema.org/language> ?v8 .
}

==C2==
SELECT ?v0 ?v3 ?v4 ?v8
WHERE {
```

```
    ?v0 <http://schema.org/legalName> ?v1 .
    ?v0 <http://purl.org/goodrelations/offers> ?v2 .
    ?v2 <http://schema.org/eligibleRegion> <http://db.uwaterloo.ca/~galuc/wsdbm/Country5> .
    ?v2 <http://purl.org/goodrelations/includes> ?v3 .
    ?v4 <http://schema.org/jobTitle> ?v5 .
    ?v4 <http://xmlns.com/foaf/homepage> ?v6 .
    ?v4 <http://db.uwaterloo.ca/~galuc/wsdbm/makesPurchase> ?v7 .
    ?v7 <http://db.uwaterloo.ca/~galuc/wsdbm/purchaseFor> ?v3 .
    ?v3 <http://purl.org/stuff/rev#hasReview> ?v8 .
    ?v8 <http://purl.org/stuff/rev#totalVotes> ?v9 .
}

==C3==
SELECT ?v0
WHERE {
    ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/likes> ?v1 .
    ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/friendOf> ?v2 .
    ?v0 <http://purl.org/dc/terms/Location> ?v3 .
    ?v0 <http://xmlns.com/foaf/age> ?v4 .
    ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/gender> ?v5 .
    ?v0 <http://xmlns.com/foaf/givenName> ?v6 .
}

==F1==
SELECT ?v0 ?v2 ?v3 ?v4 ?v5
WHERE {
    ?v0 <http://ogp.me/ns#tag> <http://db.uwaterloo.ca/~galuc/wsdbm/Topic13> .
    ?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?v2 .
    ?v3 <http://schema.org/trailer> ?v4 .
    ?v3 <http://schema.org/keywords> ?v5 .
    ?v3 <http://db.uwaterloo.ca/~galuc/wsdbm/hasGenre> ?v0 .
    ?v3 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://db.uwaterloo.ca/~galuc/wsdbm/
        ProductCategory2> .
}

==F2==
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7
WHERE {
    ?v0 <http://xmlns.com/foaf/homepage> ?v1 .
    ?v0 <http://ogp.me/ns#title> ?v2 .
    ?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?v3 .
    ?v0 <http://schema.org/caption> ?v4 .
    ?v0 <http://schema.org/description> ?v5 .
    ?v1 <http://schema.org/url> ?v6 .
    ?v1 <http://db.uwaterloo.ca/~galuc/wsdbm/hits> ?v7 .
    ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/hasGenre> <http://db.uwaterloo.ca/~galuc/wsdbm/SubGenre22> .
}

==F3==
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6
WHERE {
    ?v0 <http://schema.org/contentRating> ?v1 .
    ?v0 <http://schema.org/contentSize> ?v2 .
    ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/hasGenre> <http://db.uwaterloo.ca/~galuc/wsdbm/SubGenre60> .
    ?v4 <http://db.uwaterloo.ca/~galuc/wsdbm/makesPurchase> ?v5 .
    ?v5 <http://db.uwaterloo.ca/~galuc/wsdbm/purchaseDate> ?v6 .
    ?v5 <http://db.uwaterloo.ca/~galuc/wsdbm/purchaseFor> ?v0 .
}

==F4==
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7 ?v8
WHERE {
    ?v0 <http://xmlns.com/foaf/homepage> ?v1 .
```

```
    ?v2 <http://purl.org/goodrelations/includes> ?v0 .
    ?v0 <http://ogp.me/ns#tag> <http://db.uwaterloo.ca/~galuc/wsdbm/Topic13> .
    ?v0 <http://schema.org/description> ?v4 .
    ?v0 <http://schema.org/contentSize> ?v8 .
    ?v1 <http://schema.org/url> ?v5 .
    ?v1 <http://db.uwaterloo.ca/~galuc/wsdbm/hits> ?v6 .
    ?v1 <http://schema.org/language> <http://db.uwaterloo.ca/~galuc/wsdbm/Language0> .
    ?v7 <http://db.uwaterloo.ca/~galuc/wsdbm/likes> ?v0 .
}

==F5==
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6
WHERE {
    ?v0 <http://purl.org/goodrelations/includes> ?v1 .
    <http://db.uwaterloo.ca/~galuc/wsdbm/Retailer842> <http://purl.org/goodrelations/offers> ?v0 .
    ?v0 <http://purl.org/goodrelations/price> ?v3 .
    ?v0 <http://purl.org/goodrelations/validThrough> ?v4 .
    ?v1 <http://ogp.me/ns#title> ?v5 .
    ?v1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?v6 .
}

==L1==
SELECT ?v0 ?v2 ?v3
WHERE {
    ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/subscribes> <http://db.uwaterloo.ca/~galuc/wsdbm/Website16661> .
    ?v2 <http://schema.org/caption> ?v3 .
    ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/likes> ?v2 .
}

==L2==
SELECT ?v1 ?v2
WHERE {
    <http://db.uwaterloo.ca/~galuc/wsdbm/City13> <http://www.geonames.org/ontology#parentCountry> ?v1 .
    ?v2 <http://db.uwaterloo.ca/~galuc/wsdbm/likes> <http://db.uwaterloo.ca/~galuc/wsdbm/Product0> .
    ?v2 <http://schema.org/nationality> ?v1 .
}

==L3==
SELECT ?v0 ?v1
WHERE {
    ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/likes> ?v1 .
    ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/subscribes> <http://db.uwaterloo.ca/~galuc/wsdbm/Website2633> .
}

==L4==
SELECT ?v0 ?v2
WHERE {
    ?v0 <http://ogp.me/ns#tag> <http://db.uwaterloo.ca/~galuc/wsdbm/Topic96> .
    ?v0 <http://schema.org/caption> ?v2 .
}

==L5==
SELECT ?v0 ?v1 ?v3
WHERE {
    ?v0 <http://schema.org/jobTitle> ?v1 .
    <http://db.uwaterloo.ca/~galuc/wsdbm/City13> <http://www.geonames.org/ontology#parentCountry> ?v3 .
    ?v0 <http://schema.org/nationality> ?v3 .
}

==S1==
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8 ?v9
WHERE {
    ?v0 <http://purl.org/goodrelations/includes> ?v1 .
```

```
    <http://db.uwaterloo.ca/~galuc/wsdbm/Retailer633> <http://purl.org/goodrelations/offers> ?v0 .
    ?v0 <http://purl.org/goodrelations/price> ?v3 .
    ?v0 <http://purl.org/goodrelations/serialNumber> ?v4 .
    ?v0 <http://purl.org/goodrelations/validFrom> ?v5 .
    ?v0 <http://purl.org/goodrelations/validThrough> ?v6 .
    ?v0 <http://schema.org/eligibleQuantity> ?v7 .
    ?v0 <http://schema.org/eligibleRegion> ?v8 .
    ?v0 <http://schema.org/priceValidUntil> ?v9 .
}

==S2==
SELECT ?v0 ?v1 ?v3
WHERE {
    ?v0 <http://purl.org/dc/terms/Location> ?v1 .
    ?v0 <http://schema.org/nationality> <http://db.uwaterloo.ca/~galuc/wsdbm/Country8> .
    ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/gender> ?v3 .
    ?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://db.uwaterloo.ca/~galuc/wsdbm/Role2> .
}

==S3==
SELECT ?v0 ?v2 ?v3 ?v4
WHERE {
    ?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://db.uwaterloo.ca/~galuc/wsdbm/
        ProductCategory9> .
    ?v0 <http://schema.org/caption> ?v2 .
    ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/hasGenre> ?v3 .
    ?v0 <http://schema.org/publisher> ?v4 .
}

==S4==
SELECT ?v0 ?v2 ?v3
WHERE {
    ?v0 <http://xmlns.com/foaf/age> <http://db.uwaterloo.ca/~galuc/wsdbm/AgeGroup0> .
    ?v0 <http://xmlns.com/foaf/familyName> ?v2 .
    ?v3 <http://purl.org/ontology/mo/artist> ?v0 .
    ?v0 <http://schema.org/nationality> <http://db.uwaterloo.ca/~galuc/wsdbm/Country1> .
}

==S5==
SELECT ?v0 ?v2 ?v3
WHERE {
    ?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://db.uwaterloo.ca/~galuc/wsdbm/
        ProductCategory3> .
    ?v0 <http://schema.org/description> ?v2 .
    ?v0 <http://schema.org/keywords> ?v3 .
    ?v0 <http://schema.org/language> <http://db.uwaterloo.ca/~galuc/wsdbm/Language0> .
}

==S6==
SELECT ?v0 ?v1 ?v2
WHERE {
    ?v0 <http://purl.org/ontology/mo/conductor> ?v1 .
    ?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?v2 .
    ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/hasGenre> <http://db.uwaterloo.ca/~galuc/wsdbm/SubGenre90> .
}

==S7==
SELECT ?v0 ?v1 ?v2
WHERE {
    ?v0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?v1 .
    ?v0 <http://schema.org/text> ?v2 .
    <http://db.uwaterloo.ca/~galuc/wsdbm/User52828> <http://db.uwaterloo.ca/~galuc/wsdbm/likes> ?v0 .
}
```

APPENDIX C

List of Publications

- *Conference Papers (peer reviewed)*

1. **Gezim Sejdiu**; Anisa Rula; Jens Lehmann; and Hajira Jabeen, "A Scalable Framework for Quality Assessment of RDF Datasets," in Proceedings of 18th International Semantic Web Conference (ISWC), 2019. URL: http://jens-lehmann.org/files/2019/iswc_dist_quality_assessment.pdf
2. Claus Stadler; **Gezim Sejdiu**; Damien Graux; and Jens Lehmann, "Sparklify: A Scalable Software Component for Efficient evaluation of SPARQL queries over distributed RDF datasets," in Proceedings of 18th International Semantic Web Conference (ISWC), 2019. URL: http://jens-lehmann.org/files/2019/iswc_sparklify.pdf
3. **Gezim Sejdiu**; Damien Graux; Imran Khan; Ioanna Lytra; Hajira Jabeen; and Jens Lehmann, "Towards A Scalable Semantic-based Distributed Approach for SPARQL query evaluation," 15th International Conference on Semantic Systems (SEMANTiCS), Research & Innovation , 2019. URL: https://gezimsejdiu.github.io/publications/semantic_based_query_paper_SEMANTICS2019.pdf
4. Hajira Jabeen; Eskender Haziiev; **Gezim Sejdiu**; and Jens Lehmann. "DISE: A Distributed in-Memory SPARQL Processing Engine over Tensor Data". In 14th IEEE International Conference on Semantic Computing (ICSC'20), 2020. URL: http://jens-lehmann.org/files/2020/icsc_dise.pdf
5. Natanael Arndt; Sebastian Zänker; **Gezim Sejdiu**; and Sebastian Tramp. "Jekyll RDF: Template-Based Linked Data Publication with Minimized Effort and Maximum Scalability". In 19th International Conference on Web Engineering (ICWE 2019), of ICWE 2019, Daejeon, Korea, June 2019. URL: https://svn.aksw.org/papers/2019/ICWE_JekyllRDF/public.pdf
6. **Gezim Sejdiu**; Ivan Ermilov; Jens Lehmann; and Mohamed Nadjib-Mami, "DistLODStats: Distributed Computation of RDF Dataset Statistics," in Proceedings of 17th International Semantic Web Conference (ISWC), 2018. URL: http://jens-lehmann.org/files/2018/iswc_distlodstats.pdf
7. Hajira Jabeen; Rajjat Dadwal; **Gezim Sejdiu**; and Jens Lehmann, "Divided we stand out! Forging Cohorts fOr Numeric Outlier Detection in large scale knowledge graphs

(CONOD)," in 21st International Conference on Knowledge Engineering and Knowledge Management (EKAW'2018), 2018. URL: http://jens-lehmann.org/files/2018/ekaw_conod.pdf

8. Jens Lehmann; **Gezim Sejdiu**; Lorenz Bühmann; Patrick Westphal; Claus Stadler; Ivan Ermilov; Simon Bin; Nilesh Chakraborty; Muhammad Saleem; Axel-Cyrille Ngomo Ngonga; and Hajira Jabeen, "Distributed Semantic Analytics using the SANSA Stack,"; in Proceedings of 16th International Semantic Web Conference - Resources Track (ISWC'2017), 2017. URL: http://svn.aksw.org/papers/2017/ISWC_SANSA_SoftwareFramework/public.pdf
9. Ivan Ermilov; Axel-Cyrille Ngonga Ngomo; Aad Versteden; Hajira Jabeen; **Gezim Sejdiu**; Giorgos Argyriou; Luigi Selmi; Jürgen Jakobitsch; and Jens Lehmann, "Managing Lifecycle of Big Data Applications,"; in KESW, 2017. URL: https://svn.aksw.org/papers/2017/KESW_BDE_Workflow/public.pdf
10. Sören Auer; Simon Scerri; Aad Versteden; Erika Pauwels; Angelos Charalambidis; Stasinos Konstantopoulos; Jens Lehmann; Hajira Jabeen; Ivan Ermilov; **Gezim Sejdiu**; Andreas Ikonomopoulos; Spyros Andronopoulos; Mandy Vlachogiannis; Charalambos Pappas; Athanasios Davettas; Iraklis A. Klampanos; Efstathios Grigoropoulos; Vangelis Karkaletsis; Victor Boer; Ronald Siebes; Mohamed Nadjib Mami; Sergio Albani; Michele Lazzarini; Paulo Nunes; Emanuele Angiuli; Nikiforos Pittaras; George Giannakopoulos; Giorgos Argyriou; George Stamoulis; George Papadakis; Manolis Koubarakis; Pythagoras Karampiperis; Axel-Cyrille Ngonga Ngomo; and Maria-Esther Vidal, "The BigDataEurope Platform – Supporting the Variety Dimension of Big Data," in 17th International Conference on Web Engineering (ICWE2017), 2017. URL: http://jens-lehmann.org/files/2017/icwe_bde.pdf

- *Demo & Poster Papers (peer reviewed)*

11. Claus Stadler; **Gezim Sejdiu**; Damien Graux; and Jens Lehmann. "Querying large-scale RDF datasets using the SANSA framework". In Proceedings of 18th International Semantic Web Conference (ISWC), Poster & Demos, 2019. URL: https://gezimsejdiu.github.io/publications/sansa-sparklify-ISWC-demo.pdf
12. Danning Sui; **Gezim Sejdiu**; Damien Graux; and Jens Lehmann. "The Hubs and Authorities Transaction NetworkAnalysis using the SANSA framework". In 15th International Conference on Semantic Systems (SEMANTiCS), Poster & Demos, 2019. URL: http://tiny.cc/4ukxcz
13. Rajjat Dadwal; Damien Graux; **Gezim Sejdiu**; Hajira Jabeen; and Jens Lehmann. "Clustering Pipelines of large RDF POI Data" in Proceedings of 16th Extended Semantic Web Conference (ESWC), Poster & Demos, 2019. URL: https://gezimsejdiu.github.io/publications/piping-clustering-eswc19-poster.pdf
14. **Gezim Sejdiu**; Ivan Ermilov; Jens Lehmann; and Mohamed-Nadjib Mami, "STATisfy Me: What are my Stats?," in Proceedings of 17th International Semantic Web Conference (ISWC), Poster & Demos, 2018. URL: http://jens-lehmann.org/files/2018/iswc_statisfy_pd.pdf

15. Damien Graux; **Gezim Sejdiu**; Hajira Jabeen; Jens Lehmann; Danning Sui; Dominik Muhs; and Johannes Pfeffer, "Profiting from Kitties on Ethereum: Leveraging Blockchain RDF with SANSA," in 14th International Conference on Semantic Systems, Poster & Demos, 2018. URL: http://jens-lehmann.org/files/2018/semantics_ethereum_pd.pdf

16. Ivan Ermilov; Jens Lehmann; **Gezim Sejdiu**; Lorenz Bühmann; Patrick Westphal; Claus Stadler; Simon Bin; Nilesh Chakraborty; Henning Petzka; Muhammad Saleem; Axel-Cyrille Ngomo Ngonga; and Hajira Jabeen, "The Tale of Sansa Spark," in Proceedings of 16th International Semantic Web Conference, Poster & Demos, 2017 (**Best Demo Award**). URL: http://jens-lehmann.org/files/2017/iswc_pd_sansa.pdf

- *Miscellaneous Papers*

17. Damien Graux; **Gezim Sejdiu**; Claus Stadler; Giulio Napolitano; and Jens Lehmann. "MINDS: a translator to embed mathematical expressions inside SPARQL queries". Technical Report University of Bonn, Smart Data Analytics, 2018. URL: https://smartdataanalytics.github.io/minds/MINDS_v0.1_report.pdf

18. Pieter Heyvaert; David Chaves-Fraga; Freddy Priyatna; Anastasia Dimou; Juan Sequeda; Hajira Jabeen; Damien Graux; **Gezim Sejdiu**; Mohammed; Saleem; and Jens Lehmann. "Preface for the Knowledge Graph Building and Large Scale RDF Analytics Workshops". In Joint Proceedings of the 1st International Workshop on Knowledge Graph Building and 1st International Workshop on Large Scale RDF Analytics co-located with 16th Extended Semantic Web Conference (ESWC 2019), 2019. URL: http://ceur-ws.org/Vol-2489/xpreface.pdf

19. Jens Lehmann; **Gezim Sejdiu**; and Hajira Jabeen. "Distributed Knowledge Graph Processing in SANSA". HPI Future SOC Lab: Proceedings 2017, 130, 21. URL: https://scholar.google.com/scholar?oi=bibs&cluster=6701703243740603519&btnI=1

20. Harsh Thakkar; Mohnish Dubey; **Gezim Sejdiu**; Axel-Cyrille Ngonga Ngomo; Jeremy Debattista; Christoph Lange; Jens Lehmann; Sören Auer; and Maria-Esther Vidal, "LITMUS: An Open Extensible Framework for Benchmarking RDF Data Management Solutions," , 2016. URL: http://arxiv.org/pdf/1608.02800

List of Figures

1.1 **Thesis Contributions**. Four are the main contributions of this thesis: (1) a scalable distributed approach for evaluation of RDF dataset statistics; (2) a scalable framework for quality assessment of RDF datasets; (3) a scalable framework for SPARQL evaluation of large RDF data; (4) a comprehensive, open-source RDF processing and analytics stack for distributed in-memory computing with the real use cases where the thesis results are applicable. 5

2.1 **Semantic Web Stack**. The Semantic Web Stack, also known as Semantic Web Cake or Semantic Web Layer Cake, illustrates the architecture of the Semantic Web, according to W3C. 12
2.2 **Sample RDF Graph representation**. Small knowledge base about 'Gezim Sejdiu' represented as a graph. 13
2.3 **MapReduce dataflow**. A MapReduce dataflow illustrated with the "Character Count" example. 19
2.4 **Spark Architecture Diagram**. A Spark Cluster Mode Overview. 20

4.1 **RDD lineage of a Criterion execution**. It consists of three steps: (1) saving RDF data into a scalable storage, (2) parsing and mapping RDF into the main dataset (RDD of triples), and (3) performing statistical criteria evaluation on the main dataset. . . . 34
4.2 **Overview of DistLODStats's abstract architecture**. It is composed of three steps: First, it reads RDF data from HDFS and converts them into RDD of triples. Second, this latter undergoes a Filtering operation applying the Rule's Filter and producing a new filtered RDD. Third, 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. As a result, a statistical representation is generated. 38
4.3 **Speedup performance evaluation of DistLODStats**. Reports speedup performance analysis 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 geometric mean of the speedup is 7.4x. 41
4.4 **Sizeup performance evaluation of DistLODStats**. The analysis keeps the number of nodes in a cluster constant (5 worker nodes) and grows the size of datasets (BSBM) to measure whether our approach can deal with larger datasets. We see that the execution time cost grows linearly and is near-constant when the size of the dataset increases. It stays near-constant as long as the data fits in memory which demonstrates one of the advantages of utilizing an in-memory approach in performing the statistics computation. 42

4.5 **Scalability performance evaluation on DistLODStats**. The analysis keeps the size of the dataset constant ($BSBM_{50GB}$) and varies the number of workers on the cluster. The number of workers varies from 1, 2, 3, and 4 to 5. We can see that as the number of workers increases, the execution time cost is super-linear on $BSBM_{50GB}$ dataset. . 43

4.6 **Speedup Ratio and Efficiency of DistLODStats**. The speedup performance trend is consistent as the number of workers increases. Efficiency increased only up to the 4th worker for $BSBM_{50GB}$ dataset. The results imply that DistLODStats can achieve near-linear or even superlinear scalability in performance. 44

4.7 **Overall Breakdown by Criterion Analysis (log scale)**. The execution time is longer when there is data movement in the cluster compared to when data is processed without movement. 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. 45

4.8 **STATisfy overview architecture**. Main services of STATisfy: *Client* – will create a remote Spark cluster for initialization, and submit jobs through REST APIs. *Livy REST Server* – it will then discover this job and sent through Remote Procedure Call (RCP) to SparkSession, where the code will be initialized and executed using DistLODStats. 46

5.1 **Overview of distributed quality assessment's abstract architecture**. Main components of DistQualityAssessment: 1) Definitions – defining quality metrics parameters, 2) Retrieving the RDF data, 3) Parsing and mapping RDF data into the main dataset (RDD of triples), and 4) Quality metric evaluation. 53

5.2 **Sizeup performance evaluation of DistQualityAssessment**. The analysis fixes the number of nodes to 6 and grows the size of datasets to measure whether DistQualityAssessment can deal with larger datasets. We see that the execution time increases 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. 58

5.3 **Node scalability performance evaluation of DistQualityAssessment**. The analysis keeps the size of the dataset constant ($BSBM_{200GB}$) and varies the number of workers on the cluster. The number of workers varies from 1, 2, 3, 4 and 5 to 6. We can see that as the number of workers increases, the execution time cost-decrease is almost linear. It decreases about 14 times (from 433.31 minutes down to 28.8 minutes) as cluster nodes increase from one to six worker nodes. The results shown here imply that our approach can achieve near-linear scalability in performance in the context of speedup. 59

5.4 **Effectiveness of DistQualityAssessment**. The speedup performance trend shows that it achieves almost linear speedup and even superlinear in some cases. The speedup grows faster than the number of worker nodes due to the computation task for the metric being computationally intensive, and the data does not fit in the cache when executed on a single node but fits into several machines when the workload is divided amongst the cluster for parallel evaluation. 60

5.5 **Overall analysis of by metric in the cluster mode (log scale)**. It shows that the execution is sometimes a little longer when there is a shuffling involved in the cluster compared to when data is processed without movement e.g. Metric L2 and L1. Metric SV3 and CN2 are the most expensive ones in terms of runtime. This is due to the extra overhead caused by extracting the literals for objects and checking the lexical form of its datatype. 61

6.1 **Sparklify Architecture Overview**. It consists of four main components: Data modeling – data ingestion and data partitioning (using the extensible VP), Mappings/Views – the relational-to-RDF mapping, Query Translator – SQL query generator from the SPARQL query, and Query Evaluator - SQL query evaluated directly into the Spark SQL engine. 65

6.2 **Sizeup analysis (on Watdiv dataset)**. The analysis keeps the number of nodes constant i.e. 6 worker nodes and grow the size of the dataset (Watdiv) in order to measure whether the approaches chosen for evaluation can deal with larger datasets. As depicted, the execution time for Sparklify grows linearly as compared with SPARQLGX-SDE, and keep staying near-linear when the size of the dataset increases. 69

6.3 **Node scalability (on Watdiv-100M)**. The analysis varies the number of worker nodes e.g. from 1, 3, to 6 worker nodes and keeps the size of the dataset constant i.e. *Watdiv-100M*. It shows that as the number of nodes increases, the runtime cost for Sparklify decreases linearly. It decreases about 0.6 times (from 2547.26 seconds down to 1588.4 seconds) as worker nodes increase from one to three nodes. 70

6.4 **Overall analysis of queries on the Watdiv-100M dataset (cluster mode)**. This analysis gives more insights about running Watdiv queries on *Watdiv-100M* dataset in a cluster mode on both approaches, Sparklify and SPARQLGX-SDE. The findings show that SPARQLGX-SDE performance decreases as the number of triple patterns involved in the query increase. In contrast to SPARQLGX-SDE, Sparklify seems to perform well when there are more triple patterns involved (i.e. *QC*, *QF* and *QS*) but slightly worst when there are linear queries (see *QL*) evaluated. 71

6.5 **Semantic-based System Architecture Overview**. It consists of three main facets: Data Storage Model – model and partition the data using the semantic-based approach, SPARQL Query Fragments Translator – the process of generating the Scala code in the format of Spark RDD operations, and Query Evaluator – the SPARQL evaluation using the Spark RDD executable code (generated from the previous step). 72

6.6 **Sizeup analysis (on LUBM dataset)**. The analysis keeps the number of nodes constant i.e. 5 worker nodes and increases the size of the datasets to measure whether a semantic-based approach deals with larger datasets. The query execution time for our approach grows linearly when the size of the datasets increases. This shows the scalability of our approach as compared to SHARD, in the context of the sizeup. SHARD suffers from the expensive overhead of MapReduce joins which impacts its performance, as a result, it is significantly worse than other systems. 78

6.7 **Node scalability (on LUBM-1K)**. The analysis increases the number of worker nodes and keeps the size of the dataset constant. We vary them from 1, 3 to 5 worker nodes. As the number of nodes increases, the runtime cost of our query engine decreases linearly as compared with the SHARD, which keeps staying constant. SHARD performance stays constant (high) even when more worker nodes are added. This trend is due to the communication overhead SHARD needs to perform between map and reduce steps. The execution time of our approach decreases about 1.7 times (from 1,821.75 seconds down to 656.85 seconds) as the worker nodes increase from one to five nodes. 79
6.8 **Overall analysis of queries on the LUBM-1K dataset (cluster mode)**. This analysis depicts some of LUBM queries (Q1, Q5, Q14) run on a *LUBM-1K* dataset in a cluster mode on all the systems. Overall, our approach performs better compared to the Hadoop-based system, SHARD due to the use of the Spark framework which leverages the in-memory computation for faster performance. However, the performance declines as compared to other approaches that use vertical partitioning (e.g., SPARQLGX-SDE on RDD and Sparklify on Spark SQL). This is due to the fact that our approach performs de-duplication of triples that involves shuffling and incurs network overhead. 80

7.1 **Overview of the SANSA stack**. The SANSA framework combines distributed analytics and semantic technologies into a scalable semantic analytics stack. 85
7.2 **SANSA-Notebooks architecture**. An interactive toolkit on top of dockerized Hadoop-Spark-Workbench with Apache Zeppelin. 87
7.3 **SANSA Notebooks example**. RDF-Stats Spark application running in SANSA-Notebooks with statistics visualization. 87
7.4 **Hubs and Authorities analysis workflow**. The architecture overview for gaining insight about Hubs and Authorities using the SANSA framework. 89
7.5 **PageRank Score Distribution of Top-50 Accounts**. 90
7.6 **Category Distribution of Top-50 Accounts**. 90
7.7 **Transaction Network of Top Hubs and Authorities**. 91
7.8 **Leveraging Blockchain RDF Data with SANSA: CryptoKitties as a Use Case**. . . 92
7.9 **Transport pilot initialization workflow**. The SC4 initialization pipeline including different BDE dockerized components. 94
7.10 **A Semantic-Geo Clustering flow**. It consists of five main components: data pre-processing, SPARQL filtering, word embedding, semantic clustering, and geo-clustering. 96
7.11 **Visualizations (on a map) of the Semantic-Geo clustering pipeline steps**. Visualizations of a zoom over a particular Austrian region with K-means results of POIs (left) and geographical clustering with relevant AOIs (right). 97

List of Tables

4.1 **Definition of Spark rules (using Scala notation) per criterion**. A list of statistical criteria following the Rule (Filter->Action) -> Postproc paradigm using the Spark/Scala notation. 36

4.2 **Complexity and data shuffling breakdown by statistical criterion**. Notation conventions: n = number of triples; V = number of vertices; E = number of edges. 37

4.3 **Dataset summary information (nt format)**. Lists dataset characteristics used on the evaluation of the DistLODStats. The size (in GB) and the number of triples are given. 40

4.4 **Distributed Processing on Large-Scale Datasets**. Reports the performance analysis of the *speedup* gained by DistLODStats as compared with the original centralized version. The experiments were run on four datasets ($DBpedia_{en}$, $DBpedia_{de}$, $DBpedia_{fr}$, and $LinkedGeoData$) in 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. 40

5.1 **Quality Assessment Pattern**. A reusable template for quality metric implementation composed of transformations and actions. 51

5.2 **Definition of selected metrics following $\mathcal{QAP}$**. List of few selected quality metrics defined against proposed QAP. 52

5.3 **Dataset summary information (nt format)**. Lists dataset information used on the evaluation of the DistQualityAssessment. The size (in GB) and the number of triples are given. 56

5.4 **Performance evaluation on large-scale RDF datasets**. A *speedup* analysis gained by DistQualityAssessment as compared with Luzzu. The experiments were run on five datasets ($DBpedia_{en}$, $DBpedia_{de}$, $DBpedia_{fr}$, $LinkedGeoData$ and $BSBM_{200GB}$). Luzzu was run in a local environment on a single machine with two strategies: (1) streaming the data for each metric separately, and (2) one stream/load – all metrics evaluated just once. 57

6.1 **Summary information of used datasets (nt format)**. Lists dataset characteristics used on the evaluation. The size (in GB) and the number of triples are given. 67

6.2 **Performance analysis on large-scale RDF datasets**. Comparison analysis of Sparklify as compared with *SPARQLGX*'s direct evaluator named SDE. The loading time for partitioning and query execution time is reported. 68

6.3 **Dataset characteristics (nt format)**. Lists dataset information used on the evaluation. The size (in GB) and the number of triples are given. 76

6.4 **Performance analysis on large-scale RDF datasets**. A comparison of our approach with SHARD – the original approach implemented on Hadoop MapReduce, SPARQLGX's direct evaluator SDE, and Sparklify w.r.t query execution time. . . . 77

Acronyms

AET Algebra Expression Tree. 66

AOI Areas of Interest. 7, 95, 97, 98

API Application Programming Interface. 21, 47, 53, 81, 84, 85, 87, 95, 96

BGP Basic Graph Pattern. 17, 73, 74, 103

BSP Bulk Synchronous Parallel. 2

CLI Command Line Interface. 45

DAG Directed Acyclic Graph. 33

DSL Domain Specific Language. 26

ETH Ether. 89, 91

ETL Extract, Transform, Load. 21

GFS Google File System. 18

HDFS Hadoop Distributed File-System. 11, 18, 21, 27, 29, 34, 39, 53, 55, 65, 67, 72, 75, 85–87, 94

HDT Header, Dictionary, Triples. 102

IRI International Resource Identifiers. 66

LOD Linked Open Data. 2, 26

LQML Quality Metric Language. 26

OWL Web Ontology Language. 26, 63, 85, 86, 88

POI Points Of Interests. 7, 95–98

RCP Remote Procedure Call. 47

RDD Resilient Distributed Dataset. 4, 20, 21, 29, 31–35, 37, 38, 47, 49, 52–55, 61, 65, 72–74, 76, 79, 87–89

RDF Resource Description Framework. 1–16, 23–29, 31, 32, 34, 35, 39, 41, 45, 47, 49–57, 61, 63–67, 72, 73, 80, 81, 84–90, 92, 93, 95–104, 115

RDFS Resource Description Framework Schema. 24, 63, 86, 88

RDG Resilient Distributed Graph. 21

SPARQL SPARQL Protocol And RDF Query Language. 1, 3, 4, 6, 7, 9–12, 16, 17, 23–29, 33, 39, 55, 63, 64, 66, 67, 70, 71, 73–76, 78, 80, 81, 85, 86, 89, 96, 98, 99, 101, 103, 104, 115, 117

URI Unique Resource Identifiers. 11–14, 17, 24, 28, 57, 66

VP Vertical Partitioning. 28, 29, 65–67

W3C World Wide Web Consortium. 1, 11, 12, 16, 63, 66, 80, 85

WWW World Wide Web. 11

XML Extensible Markup Language. 15, 16