Distributed in-memory SPARQL Processing

Haziiev Eskender
MN: 2994738

Rheinische Friedrich-Wilhelms-Universität Bonn
Institute of Computer Science III

In partial fulfillment of the requirements for the degree of
Master of Science

April 2019

Distributed in-memory SPARQL Processing

Master Thesis

Submitted by Eskender Haziiev

First examiner/Erstgutachter: Prof. Dr. Jens Lehmann

Second examiner/Zweitgutachter: Dr. Damien Graux

Supervisor/Betreuer: Dr. Hajira Jabeen

Rheinische Friedrich-Wilhelms-Universität Bonn

Institute of Computer Science III

Smart Data Analytics

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Bonn

Eskender Haziiev

Abstract

The goal of this thesis was to develop a novel approach for distributed in-memory SPARQL processing based on SANSA, Apache Spark and Apache Jena. In this work the concepts of the RDF tensor and the degree of freedom were used. The degree of freedom was considered as a measure of triple pattern's explicit constraints. The algorithms proposed in the approach were implemented in the framework that used a modification of the vertical partition schema for querying large-scale distributed RDF datasets. We investigated a runtime dependency on the query complexity and proved the P-solvability of the problem. In addition, the effectiveness of the system was experimentally confirmed. Our approach and the developed framework can extend the SANSA functionality and will be built into the Knowledge Representation and Distribution Layer of SANSA.

Zusammenfassung

Das Ziel dieser These war die Entwicklung eines neuartigen Ansatzes, für die verteilte in-memory SPARQL Verarbeitung, basierend auf SANSA, Apache Spark und Apache Jena. In unserer Arbeit wurden die Konzepte des RDF-Tensors und der Freiheitsgrad verwendet. Der Freiheitsgrad wurde als Maß für die expliziten Einschränkungen des Triple-Musters betrachtet. Die Algorithmen, die im Ansatz vorgeschlagenen wurden, wurden in ein Framework implementiert. In diesem Framework wurde die Modifikation des vertikalen Partitionsschemas für die Abfrage großer verteilter RDF-Datasets verwendet. Wir haben die Laufzeitabhängigkeit von der Komplexität der Abfrage untersucht und die P-Lösbarkeit des Problems nachgewiesen. Darüber hinaus wurde die Wirksamkeit des Systems experimentell bestätigt. Unser Ansatz, und das entwickelte Framework, können die SANSA-Funktionalität erweitern, und werden in die "Knowledge Representation and Distribution Layer" von SANSA integriert.

Acknowledgements

I want to thank my supervisor, Dr. Hajira Jabeen, for her guidance and assistance during the preparation of this thesis. Her advice, patience and attention to detail during our discussions helped me to prioritize research goals and systematize knowledge.

I am very grateful to Prof. Dr. Jens Lehmann for introducing me to this research area and increasing my interest in data science through his courses at the university and research.

I also want to thank Dr. Damien Graux and Gëzim Sejdiu for their feedbacks, suggestions and encouragement while working on this thesis.

Last, but not least, I am grateful to my family for their support and encouragement during my studies at Bonn.

Contents

1 Introduction

1.1 Problem Statement

The Internet stores information, the amount of this information noticeably increases significantly every day. This process is complicated by the characteristics of large-scale data.

Knowledge extraction and information retrieval from the vast amount of data related to the financial, medical, engineering, scientific, social, military fields etc. is a complex problem that requires modern approaches and special techniques to be developed. These technologies must meet basic requirements such as distributed in-memory data processing in acceptable execution time.

For this reason, many researchers, who focused on the problem, proposed innovative approaches and obtained significant results in this area (see, for example, [1], [2], [8], [10], [11], [16], [17], [20], [21], [29], [36], [37], [38], [46]).

Among the currently most used and effective technologies we can name Semantic Web, Resource Description Framework, SPARQL, Apache Spark, Scala programming language etc. One of such projects is the Semantic Analytics Stack (SANSA) platform, an open-source structured data processing engine for performing distributed computation over large-scale RDF datasets[16]. It includes several libraries for creating applications and provides the facilities for Semantic data representation, Querying, Inference, and Analytics. Having the property of extensibility, SANSA can be supplemented with additional modules that meet the principles of its architecture.

Given the above, we find the problem of distributed in-memory SPARQL query processing is of interest for analysis and implementation.

1.2 Proposed Solution

Many authors suggest to solve the problem of the efficient processing of large-scale RDF data in several ways, in particular:

1. using parallel processing of data distributed over a cluster; data partitioning can be implemented by one of the components of RDF triple;

2. using different tensors to represent RDF datasets;

3. using evaluation metrics to measure process efficiency, such as response time and amount of memory used;

4. using different estimates of the complexity of SPARQL query to optimize query processing;

5. using of low-level programming languages in the implementation of frameworks.

From our point of view, extending the SANSA functionality makes the approach based on the following principles more preferable:

1. process RDF datasets in parallel along with vertical partitioning based on the triple predicate; it is also advisable to utilize RDF tensor that represents a dataset;

2. introduce a concept of the complexity of a query based on the number of variables, the number of triples, and the number of common variables; this can optimize the execution of a complex query and reduce its response time; it is natural to apply the concept of the degree of freedom to evaluation of the query triple pattern;

3. evaluate processing efficiency both qualitatively and quantitatively.

1.3 Contribution

The main contribution of this thesis is a new approach that can extend the functionality of the SANSA platform and expand its scope.

For this purpose, the following problems were solved:

- A framework to process SPARQL queries using the concepts of the RDF tensor, DOF and vertical partitioning of the datasets was developed;

- Query complexity parameters were introduced; the parameters are the number of variables, the number of triples, and the number of common variables;

- Propositions on the P-solvability of the considered problem were proved;

- Qualitative and quantitative estimates of the framework were obtained;

- Effectiveness and efficiency of the presented approach were confirmed experimentally.

This thesis is organized as follows.

Chapter 2 contains background information concerning this work. The focus is put on widespread modern concepts and technologies used to extract and process semantic information from the Internet and various datasets. Namely, Semantic Web, Resource Description Framework, Tensors, SPARQL, Apache Jena, Apach Spark and Scala are described.

Chapter 3 provides an overview of the related work and clarifies the thesis scope in the general architecture of the SANSA platform. Also the concept of the degree of freedom of a query triple pattern, that allows us to identify the different types of triples contained in complex SPARQL queries, is considered.

Chapter 4 describes the architecture of our approach along with the description of its key features and developed algorithms. We discuss types of SPARQL queries covered, introduce parameters of the query complexity and methods for qualitative and quantitative evaluation of the effectiveness of the proposed approach.

Chapter 5 is devoted to the implementation of the approach and provides information about the characteristics of the cluster, the different types of datasets and the SPARQL queries used for testing. The chapter presents the results of the comprehensive evaluation using the comparative analysis of data in the form of tables and charts in accordance with various parameters. At the same time, we focus on the scalability of the proposed approach, the accuracy and correctness of the obtained results.

Chapter 6 summarizes the main results of this work and contains the perspectives for further development of the approach and ways to improve its effectiveness.

2 Fundamentals

2.1 Semantic Web

The term "Semantic Web" was introduced by Tim Berners-Lee and Mark Fischetti in [43]. They consider Semantic Web as "the Web [in which computers] become capable of analyzing all the data on the Web – the content, links, and transactions between people and computers". In [44] T. Berners-Lee, J. Hendler, and O. Lassila described "an expected evolution of the existing Web to a Semantic Web". In particularly, the authors write that "The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation". And also they mention that "The Semantic Web will bring structure to the meaningful content of Web pages, creating an environment where software agents roaming from page to page can readily carry out sophisticated tasks for users".

The Semantic Web is an extension of the World Wide Web through standards by the World Wide Web Consortium (W3C) [30] that support common data formats and exchange protocols on the Web. In [31] the authors write that "the basic idea of the Semantic Web is to describe the meaning (i.e. the semantics) of Web content in a way that can be interpreted by computers".

According to the standards of the W3C "Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries ... It is based on the Resource Description Framework (RDF)" [49].

Thus, Semantic Web is based on the World Wide Web by standardizing the presentation of information in a form suitable for machine processing.

There is a significant difference between the World Wide Web and the Semantic Web.

The World Wide Web consists of HTML pages with textual information intended for reading and understanding by humans. Searching for information on the World Wide Web is carried out through keywords and URI (Uniform Resource Identifier), which is the unique address of a particular page.

The Semantic Web is based on machine-readable elements or nodes that are vertices of the Knowledge Graph. Main technologies, including URI, Resource Description Frame-

work (RDF) and ontology, are used to transform such kind of information into a machine-readable form and are described in detail in [31, 33, 40]. Using ontologies created in a specific language, such as OWL (Web Ontology Language), a computer interprets RDF statements and derives logical inferences from data.

As a result, client programs are able to directly receive the statements in the form of "some subject — type of relation — another subject or object" (also known as triple) and make logical conclusions from data.

2.2 Resource Description Framework

The first official specification of the Resource Description Framework (RDF) was published by W3C in 1999 and was extended in 2004 to represent semantic information in general.

RDF is based on a simple graph-oriented data schema, where an RDF document describes a directed graph with a set of nodes linked by directed edges. Both nodes and edges are labeled with the identifiers to distinguish them. It should be noted that according to [31] RDF is defined as a formal language for describing structured information. It can characterize general relationships between objects, known as "resources", in the form of the graphs but not as hierarchical tree structures. The graphs can include multiple unconnected sub-graphs and therefore such data model is more suitable for the composition of distributed information sources. Moreover, since sub-graphs may not have uniform identifiers, RDF uses so-called URIs (Uniform Resource Identifiers) as names to distinguish resources from each another.

It is possible to use data values and blank nodes for this. Data values in RDF are represented by literals, which are the reserved names of the RDF resources of a particular data type, but they can not be used as sources or labels of the edges. However, the same URI can be used for labeling both nodes and edges in the RDF graph.

RDF statements can be defined in the form of subject–predicate–object expressions, also known as RDF triples. In RDF, the subject is a resource and the object is the triple target value, whereas a predicate denotes a relationship (i.e., an edge) between them. Each part of the triple can be a URI, but an object can also be a literal or a blank node. A set of such triples is called an RDF graph and, therefore, RDF graphs can be considered as a collection of triples in an arbitrary order ([35, 51, 52]).

2.3 Tensors

Tamara G. Kolda and Brett W. Bader define tensors as multi-dimensional arrays that can also be viewed as an ordered collection of slices or columns ([41, 42]). The order of a tensor is a number of its dimensions. Hence, a slice is a two-dimensional tensor, a column is a one-dimensional tensor. Furthermore, N-dimensional tensor can be represented as a product of one-dimensional tensors.

In [5] the author writes that an order-d tensor $\mathcal{A}^{i_1 \times i_2 \times \ldots \times n_d}$ is defined as a real d-dimensional array $\mathcal{A}(1 : i_1, 1 : i_2, \ldots, 1 : n_d)$ where an index range in the k-th mode is varied from 1 to n_k. For instance, a scalar is an order-0 tensor, a vector is an order-1 tensor, a matrix is an order-2 tensor etc. On this base, Charles F. Van Loan stated that any tensor can be decomposed into several simpler tensors.

Also in [45] a tensor $\mathcal{A}$ is considered as a sum of rank-1 tensors and the following definition of the rank-1 tensor is given:

Definition 2.3.1. *Let $f \in \mathbb{R}^{n_1}$, $g \in \mathbb{R}^{n_2}$, $h \in \mathbb{R}^{n_3}$, then $\mathcal{B} = f \circ g \circ h \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ is defined as*

$$\mathcal{B}(i_1, i_2, i_3) = f(i_1)g(i_2)h(i_3). \tag{2.3.1}$$

□

Then a rank of tensor $\mathcal{A}$ is defined as the smallest number of rank-1 tensors that sum to $\mathcal{A}$.

A tensor approach to learning Semantic Web and Knowledge Graphs is used in [45] where the authors consider the tensor-based semantic graph representation for RDF knowledge bases and its decomposition for effective Semantic Web search.

In the context of RDF knowledge bases, data can be seen as a graph with nodes representing RDF resources and edges corresponding to RDF predicates, i.e., relations between resources. Therefore, in [45] Semantic Web is modeled by a three-dimensional tensor $\mathcal{T}$ where each of its slices represents an adjacency matrix for one RDF property.

Since the Semantic Web graph can be described by an adjacency matrix $M \in \mathbb{R}^{k \times l}$, by applying the Singular Value Decomposition (SVD) M one can obtain three matrices $U \in \mathbb{R}^{k \times m}$, $S \in \mathbb{R}^{m \times m}$, $V \in \mathbb{R}^{l \times m}$, where U and V represent the outlinks and the inlinks with respect to the principal factor contained in S. Formally, tensor $\mathcal{T} \in \mathbb{R}^{k \times l \times m}$ is decomposed by n-rank-PARAFAC into components matrices $U_1 \in \mathbb{R}^{k \times n}$, $U_2 \in \mathbb{R}^{l \times n}$, $U_3 \in \mathbb{R}^{m \times n}$ and n principal factors in the descending order. Via these $\mathcal{T}$ can be written as Kruskal tensor, i.e. a sum of outer products of k-th column of U_i ([45]).

In [24] Makoto Nakatsuji et al. improve the tensor approach using semantic sensitive tensor factorization (SSTF). SSTF has two components: semantic grounding and tensor factorization with semantic augmentation, which resolves problems of semantic ambiguity and sparsity. Semantic grounding resolves semantic ambiguities by linking the objects to the vocabularies (i.e. taxonomies) and constructs a tensor with objects linked to classes. Tensor factorization with semantic augmentation solves the sparsity problem by incorporating semantic biases based on vocabulary classes into tensor factorization. To do this, it determines multiple sets of sparse objects according to the degree of sparsity to create multiple augmented tensors. It then factorizes the original tensor and the augmented tensors simultaneously to compute the feature vectors for objects and for classes. By factorizing multiple augmented tensors, it creates feature vectors for classes according to the degree of the sparsity. As a result, SSTF achieves much higher prediction accuracy than other methods ([24]).

It should be noted, that [24] deals with a third-order tensor $\mathcal{R}$ and assumes that there exists at most one observation for each (m, n, k)-th element of $\mathcal{R}$. Tensor factorization assigns a D-dimensional feature vector to each user u_m, item v_n and tag t_k, which have lengths M, N and K respectively. Accordingly, each element $r_{m,n,k}$ in tensor $\mathcal{R}$ can be approximated as the inner product of these vectors, i.e. of rank-1 tensors:

$$r_{m,n,k} \approx \langle u_m, v_n, t_k \rangle \equiv \sum_{d=1}^{D} u_{m,d} \cdot v_{n,d} \cdot t_{k,d}. \tag{2.3.2}$$

The tensor approach used by Roberto De Virgilio in [36] is provided below. The author considers RDF data represented by triples of the form $\langle s, p, o \rangle$, where the subject $s \in \mathcal{S} = \mathcal{I} \cup \mathcal{B}$, the predicate $p \in \mathcal{P} = \mathcal{I}$ and the object $o \in \mathcal{O} = \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$. Here $\mathcal{I}$, $\mathcal{B}$ and $\mathcal{L}$ are disjoints finite sets of IRIs, blank nodes and literals, respectively.

Definition 2.3.2. (Ontology tensor) *Let $\mathcal{S}$ be the finite set of subjects, $\mathcal{O}$ be the finite set of objects, $\mathcal{P}$ the finite set of predicates and $\mathbb{B}$. Then the ontology tensor is a rank-3 tensor $\mathcal{T} : \mathcal{S} \times \mathcal{P} \times \mathcal{O} \longrightarrow \mathbb{B}$, being $\mathbb{B}$ a boolean ring.* □

Definition 2.3.3. (RDF tensor)

Let G be an RDF graph. The RDF tensor $\mathcal{R}(G) =: \mathcal{R}$ on G is an ontology tensor such that

$$\mathcal{R} = r_{i,j,k} := \begin{cases} 1 & \langle \mathbb{S}^{-1}(i), \mathbb{P}^{-1}(j), \mathbb{O}^{-1}(k) \rangle \in G, \\ 0, & otherwise \end{cases}, \tag{2.3.3}$$

where $\mathbb{S} : \mathcal{S} \longrightarrow \mathbb{N}$, $\mathbb{P} : \mathcal{P} \longrightarrow \mathbb{N}$, and $\mathbb{O} : \mathcal{O} \longrightarrow \mathbb{N}$ are the indexing functions of subjects,

predicates, and objects. □

Thus, the RDF tensor $\mathcal{R}$ is considered by Roberto De Virgilio as a three-dimensional tensor. The results of the query triple pattern on $\mathcal{R}$ is written via Kronecker delta, for instance, $\mathcal{R}_{i,j,k}\delta_j^{\mathcal{P}(type)}\delta_k^{\mathcal{O}(person)}$.

2.4 SPARQL

According to the W3C Recommendation [53], SPARQL (acronym SPARQL Protocol and RDF Query Language) is a technology developed for extraction and modification of data stored in RDF graphs. It allows to perform many complex operations on data such as information retrieval, searching, filtration based on a given criteria, transformation etc.

A SPARQL query contains a query triple pattern with a structure similar to RDF triples. However, any subject, predicate and object in the query triple pattern can be a variable starting with a question mark.

There are different types of SPARQL queries (see [53, 22, 23]):

1. SELECT query returns variables and their bindings directly;

2. CONSTRUCT form returns a single RDF graph specified by a graph template;

3. ASK query tests whether or not a query pattern has a solution and generates a result equal to True or False;

4. DESCRIBE form returns a single result RDF graph containing RDF data about resources.

In general, SPARQL query composition scheme is as follows:

PREFIX describes prefix declarations for abbreviated URIs, it reduces the URIs used in the query;

FROM (optional) determines the sources of the requested RDF graphs and defines the *RDF dataset* which is being queried;

SELECT composes the resulting clause;

WHERE specifies the basic graph pattern to match and defines the constraints of the query;

ORDER BY, LIMIT and OFFSET can be used as solution output modifiers.

2.5 Apache Jena

Apache Jena is an open source Semantic Web framework created in the Java programming language. This framework provides an API for reading and writing data in RDF graphs. The graphs in the framework are represented by an abstract "model" that can be created from various data sources such as files, databases, etc. One can work with the model using SPARQL and SPARUL.

In Java terms, Jena uses the *Model* class as the main container of RDF information, represented in the form of a graph. At the same time, *Graph* is a more common interface for low-level RDF repositories.

In general, there are several different concepts of RDF containers in Apache Jena:

- *Model* has a rich Java API with a large number of convenient methods for application developers.
- *Graph* has a simple Java API designed to extend the functionality of Jena.

Model / Statement / RDFNode (API) is well suited as an application interface, but Graph / Triple / Node (SPI) works better for abstraction from storage, where "regularity, independence" is considered as a more valuable factor.

To work with triplets Apache Jena implies the following concepts (suggested in [3]):

- *Triple* – Statement
- *Subject* – Resource
- *Predicate* – Property
- *Object* – RDFNode

Statement declares a fact about the resource. Sometimes the statement is also called a *Triple*, since it contains three parts:

- Resource (subject) is somehow the object id.
- Property (predicate) can be considered as a parameter of the object.
- RDFNode (object) is the parameter value. The value can be either the object id (Resource) or some value (Literal).

The product includes several repositories, including triples storage (Jena TDB)

In addition, the framework includes its own triples storage (Jena TDB), an interface to relational storage (Jena SDB), storage in memory (In-Memory), as well as tools to support user storage.

2.6 Apache Spark

Apache Spark is an open source platform for distributed data processing that is part of the ecosystem of Hadoop projects. Unlike the classic handler from the Hadoop core, which implements the two-level concept of MapReduce with disk storage, Spark utilizes specialized primitives for recurrent processing in RAM, which provides significant performance for some classes of tasks ([34]). In particular, the possibility of multiple access to the data stored in memory makes the framework attractive for machine learning algorithms ([26]).

Spark applications consist of a single *driver process* and several *executor processes* ([27]).

The driver is the heart of an application and performs the following functions:

1. It keeps and processes information about the application status.
2. The driver responds to user program requests.
3. It analyzes and distributes tasks between executors and the order of their execution.

Executors perform tasks and report the result to the driver.

Since the driver and executors are common processes, Spark can work in the pseudo-distributed *local mode*, where all of them run on the same computer with one processor core per executor, i.e., the cluster's work is only emulated. It is useful for development and testing, when distributed storage is not needed and the local file system is used instead.

To deploy an application Apache Spark employs the *cluster manager* that controls physical machines and allocates resources and distributed storage. For this purpose, Spark can utilize the built-in manager (for standalone mode) or YARN or Apache Mesos ([7]).

Spark supports several distributed storage systems, such as HDFS, OpenStack Swift, NoSQL-DBMS Cassandra, Amazon S3. The framework provides APIs in the following programming languages: Scala, Python, Java, SQL and R.

The framework consists of a kernel and several extensions, such as Spark SQL (API to perfrom SQL queries on data), Spark Streaming (add-on for streaming data processing), Spark MLlib (a set of machine learning libraries), GraphX (distributed graph processing).

The Apache Spark API is centered around the data structure called *resilient distributed dataset* (RDD), that is a fault-tolerant multi-set of read-only data elements distributed over a cluster. RDD supports the implementation of both iterative algorithms that access the data many times in a loop, and interactive intelligence analysis, i.e., repeated data requests in the database. The class of iterative algorithms includes training algorithms for machine learning systems, that was the main reason for the development of Apache Spark ([9]).

2.7 Scala

Scala is a programming language that combines static typing, an object-oriented and functional approach. Martin Odersky began developing it in the early 2000s at the EPFL Programming Methods Lab. The creators of Scala decided that the language should work on the Java virtual machine and provide access to Java libraries.

Although the Scala programming language is a "symbiosis" of *Java* and *C#*, it was influenced by other languages and technologies ([25]). The property expression is borrowed from *Sather*, and the concept of a unified object model is migrated from *Smalltalk*. *Beta* shared the idea of nesting, including classes. Abstract types in Scala resemble abstract signature types in *SML* and *OCaml*. In addition, Scala has adopted the features of such functional languages as *Scheme*, *Standard ML* and *Haskell*.

However, Scala has several advantages over Java:

- More compact syntax
- Less stereotypical code
- More saturated type system
- Smaller language grammar

Another important advantage of Scala is the synthesis of object-oriented (OOP) and functional programming (FP). In OOP, communication with objects is organized using "messages" or method calls, while FP is programming using functions. Some features of OOP are useful in a purely functional approach, in particular, first class modules (obtained by composition of objects), point syntax (calling methods in objects), and type classes / instances as first class constructs. All these elements of a successful combination of two paradigms are presented in Scala.

An example of a data science-oriented abstraction that hides a complex computational model is Apache Spark, which is the leading platform for distributed data analysis. Here Scala is used not only to implement Spark itself, but also to process data ([50]).

3 Related Work

3.1 Related Work

During the last years, an increasing amount of RDF data enhances the role of clustering in the context of RDF database systems. According to recent statistics, 150 billions of RDF triples and almost 10, 000 linked datasets are available in Linked Open Data (LOD) Cloud ([4]). According to [14], the current freely available Semantic Web data is approximately 150 billion triples in about 3,000 datasets, many of which are accessible via SPARQL query servers called SPARQL endpoints.

Naturally, many researchers focus on issues related to RDF data representation in query processing, methods for reducing computational costs in terms of space–time, as well as how to achieve efficient implementation and correct results.

Representation methods for large distributed in-memory RDF data include vertex partitioning, edge partitioning and horizontal partitioning (see [2], [18], [37]).

Introduced in [2] as a SPARQL query processor for Hadoop, S2X implements basic graph pattern matching of SPARQL as a graph-parallel task. All other SPARQL operators are working in a data-parallel manner. First, the authors define a mapping from RDF to the property graph model of GraphX, that is a graph abstraction on top of Spark. Based on this model, S2X is defined as a SPARQL implementation on top of GraphX and Spark.

Horizontal partitioning of RDF data is also used in TRiAD engine for distributed SPARQL processing (see [37]).

Another splitting approach for RDF data is vertical partitioning method used in [1], [8], [10], [36], [38].

Abadi et al. utilize vertical partitioning, which can be considered as a special case of property tables, where each table holds information about a single predicate [10]. A table stores tuples in the form of $\langle subject, object \rangle$ and contains the pairs associated with a given predicate. Each table is sorted by the subject column, so individual subjects can be located quickly and fast merge joins can be used to restore information about multiple properties for subsets of subjects.

S2RDF, a Hadoop-based SPARQL query processor for large-scale distributed RDF data,

is presented in [1]. It is implemented on top of Spark and includes a new relational partitioning schema for RDF data called ExtVP. To execute SPARQL queries through ExtVP, S2RDF uses Spark and Vertical Partitioned schema as the base data layout for RDF. The results for a query triple pattern with bound predicate can be retrieved by only accessing the corresponding vertical partitioned table, which leads to a large reduction of the input size. S2RDF evaluation schema includes measurements of query execution time, average runtime and the best possible runtime. Note that the tested queries contain only conjunctions and do not include other options.

Another approach to evaluate SPARQL queries on distributed RDF datasets is implemented in [8]. It is based on Apach Spark and translates conjunctive queries into Spark executable code. Damien Graux et al. focus on the problem of evaluating the Basic Graph Pattern fragment (BGP) over an RDF dataset. Such fragments are composed of conjunctions of triple patterns (TPs), where each TP expresses conditions that must be matched by rdf triple parts for it to be selected. The authors indicate that there are often relatively few distinct predicates compared to the number of distinct subjects or objects. Thus, they use the vertically partitioned architecture introduced in [10]. To process a conjunction of query triple patterns, the triple patterns are joined using their common variables as a key. If there are no common variables, the cross product is returned. Experimental results are based on three metrics: query execution times, preprocessing times and disk footprints.

Sandra Álvarez-Garcá et al. use a predicate to vertically partition a dataset into disjoint subsets of pairs (subject, object), one per predicate (see [38]). These subsets are represented as binary matrices of subjects×objects in which 1-bits mean that the corresponding triple exists in the dataset. This model results in very sparse matrices, which are efficiently compressed using k^2-trees and two compact indexes. The indexes are listing the predicates related to each different subject and object, in order to address the specific weaknesses of vertically partitioned representations. Thus, the most compressed representations and the best overall performance for RDF retrieval are achieved. The experimental results show that the method overcomes traditional vertical partitioning for join resolution.

To fast query execution on RDF datasets in distributed environment some papers use a hybrid approach ([48]), that have two steps. First, so-called hybrid partitioning is executed and then RDF data is distributed using so-called reachability matrix.

In [28] the authors focus on the advantage of using the common variables in query

processing: the common subject-object identifier facilitates bitwise operations in joins, where the subject position in one query triple pattern is joined over the object position in another query triple pattern.

A number of papers deals with the problem of query complexity, since the complexity of a query directly determines the runtime and the amount of memory required for its execution.

Indeed, in [21] experimental evaluation of SPARQL queries for DBpedia datasets with 109,750,000 triples shows that 58% of SELECT queries contain a single pattern matching RDF subject, predicate, and object triple, FILTER and similar constructs are not included; 6% contain two patterns, 35% have three patterns and SELECT queries with four patterns are not used.

The results of analysis of the SPARQL queries complexity are presented in [17] (see also [11]). It is shown that operators AND and UNION are commutative and associative, but AND, OPT, and FILTER are distributive with respect to the UNION operator. Query evaluation can be solved in polynomial time for graph pattern expressions constructed by using only AND and FILTER operators; evaluation is NP-complete for graph patterns built only with AND, FILTER and UNION operators; for graph patterns that include the OPT operator, the evaluation problem is PSPACE-complete.

M. Schmidt et al. in [29] pay attention to the fact that SPARQL provides advanced operators (such as SELECT, AND, FILTER, OPTIONAL and UNION) which can be used to compose more expressive queries. The authors map the OPTIONAL operator to a left outer join, AND is mapped to a join operation, UNION to an algebraic union, FILTER to a selection, and SELECT is a projection.

The following results of [29] should be especially noted:

- evaluation of queries containing OPTIONAL or AND&OPTIONAL is PSPACE-hard;
- evaluation of queries containing UNION or FILTER&UNION is PTIME-complete;
- evaluation of queries containing AND&UNION is NP-complete.

An equally significant aspect of query processing is validation of the obtained results. The problems of accuracy, correctness and effectiveness of the developed approaches are discussed, for example, in [1], [20], [36], [46].

In [46] T. Neumann et al. evaluate query-execution performance for SELECT...WHERE queries containing DISTINCT and conjunction operators. The authors conduct a series of

ten query evaluations and measure the average execution time together with the standard deviation for each query, i.e., the mean plus or minus the standard deviation.

The metrics used to evaluate knowledge base queries, include hits at 10, i.e., percentage of the correct answers ranked in the top 10, and mean rank. K. Guu et al. use a specific evaluation metric to measure accuracy and correctness in the problem of traversing knowledge graphs in vector spaces (see [20]). They evaluate on hits at 10, as well as a normalized version of mean rank, *mean quantile*, which accounts for the total number of candidates. For a query q, the quantile of a correct answer t is the fraction of incorrect answers ranked after t.

An abstract algebraic framework for the efficient and effective analysis of RDF data is presented in [36]. Implemented in C++, the framework creates an RDF tensor and utilizes the notion of the degree of freedom (DOF) of a triple pattern to improve the efficiency of distributed RDF data processing. The author provides experimental evaluation of query memory footprints as well as an average execution time for a series of ten query evaluations. Using the RDF tensor, a significant reduction in memory is achieved, while DOF helps to select a triple pattern with the highest probability of decreasing the search space.

3.2 Degree of Freedom

The concept of the degree of freedom (DOF) of a query triple pattern t is introduced in [36], where DOF is defined as "a measure of triple pattern's explicit constraints".

Definition 3.2.1. *Let v and k be the number of variables and constants in triple t, respectively. The degree of freedom of a triple t is the function defined as $dof(t) := v - k$, $dof \in \{+3, +1, -1, -3\}$.* □

From the definition 3.2.1 it follows that:

1. A triple with no constraints has the highest DOF value, equal to +3, and is associated to variables only.

2. For a triple with two variables and only one constant we have $dof(t) = +1$.

3. $dof(t) = -1$ means that a triple is bounded to one constant and contains two unbounded variables.

4. The lowest DOF value corresponds to a triple, which is bound to three constants, i.e. $dof(t) = -3$.

3.3 Scope

SANSA (i.e., Semantic Analytics Stack) is a powerful system that includes a set of libraries for distributed processing of large-scale RDF data. Library members are grouped together with respect to the problems they solve and are united by one goal (one of the data processing steps). Therefore, the architecture of SANSA can be considered as a high-rise building, each floor of which (i.e., a group of methods and technologies) is used as a basis of the next floors. As a layer superstructure, such a notion of SANSA architecture is presented in [16, 39]. Figures 3.3.1 – 3.3.3 partially use diagrams from these papers to illustrate the scope of the thesis in the architecture of SANSA.

Our goal is to implement new features inside of Knowledge Distribution and Representation Layer and Querying Layer, but we do not add new functionality to other layers, such as Distributed In-Memory Processing, Inference and Machine Learning.

Figure 3.3.1 illustrates the scope of the thesis in the general architecture of SANSA.

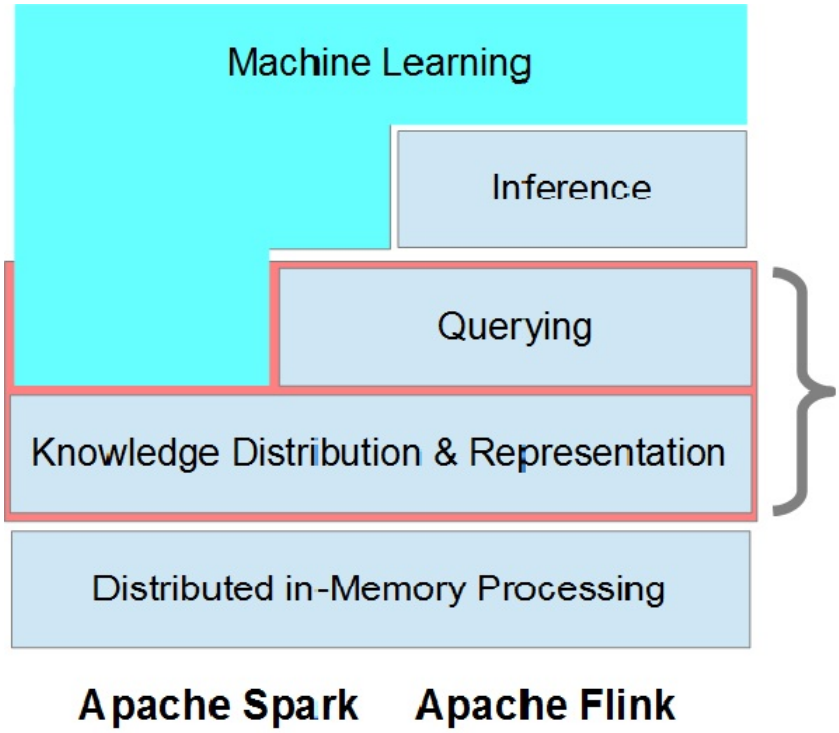


Figure 3.3.1: Thesis scope w.r.t. SANSA architecture (based on [16])

It should be noted that we use the methods of Knowledge Distribution and Representation Layer for reading RDF data to transform it into an RDF tensor (see Figure 3.3.2).

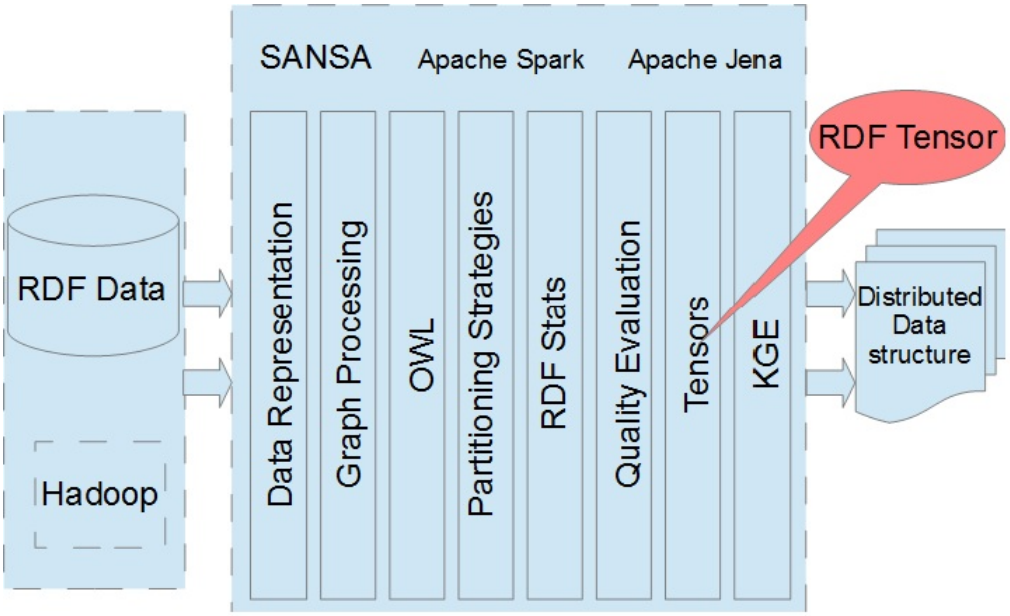


Figure 3.3.2: Thesis scope w.r.t. SANSA Knowledge Distribution and Representation Layer (based on [39])

At Querying Layer (see Figure 3.3.3) we evaluate queries with different options and calculate the *degree of freedom* of a query triple pattern (suggested in [36]) in order to achieve a significant reduction in the size of the data being processed.

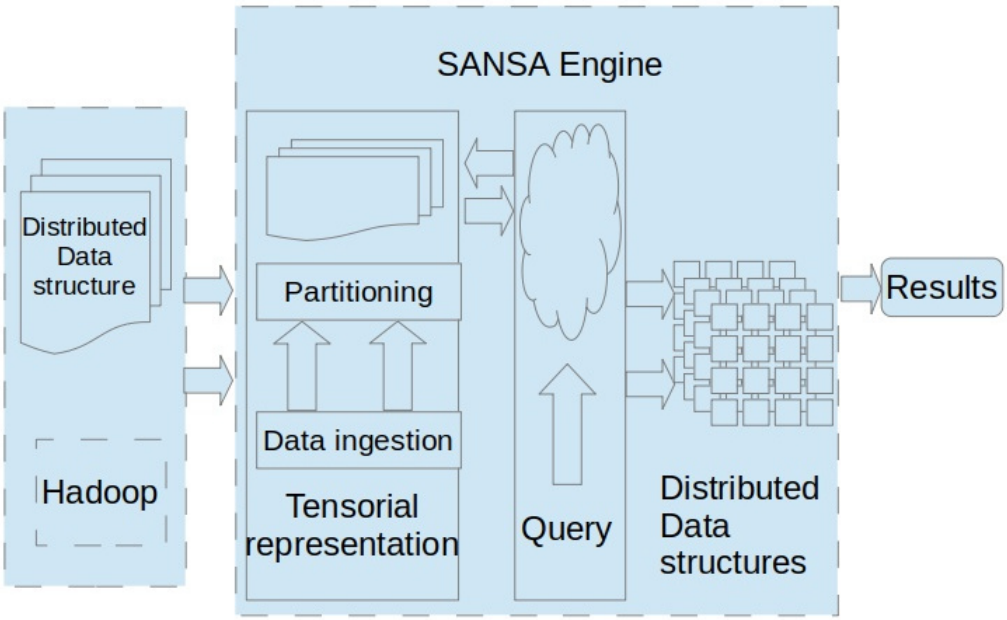


Figure 3.3.3: Thesis scope w.r.t. SANSA Querying Layer ([39] is partially used)

4 Proposed Approach

4.1 Architecture

It is necessary to develop effective methods to solve such a complex problem as knowledge extraction and information retrieval from distributed large-scale data. Built on the top of Spark, Flink, Jena etc., SANSA implements the idea of combining several libraries into a scalable system that solves the problems of data analysis and machine learning.

Adding a new feature to the system is a complex process that should follow a unified approach to system design; it is also necessary to fulfill the requirements of maintaining the efficiency of the methods and technologies already included into the system. In order to achieve our goals we used the existing functionality of SANSA, Spark, Jena, and also developed several new methods and classes (RDFTensor, Builder etc.) that meet the requirements of the unified approach. These methods work in the SANSA environment, and Figure 4.1.1 illustrates their interaction in a block form.

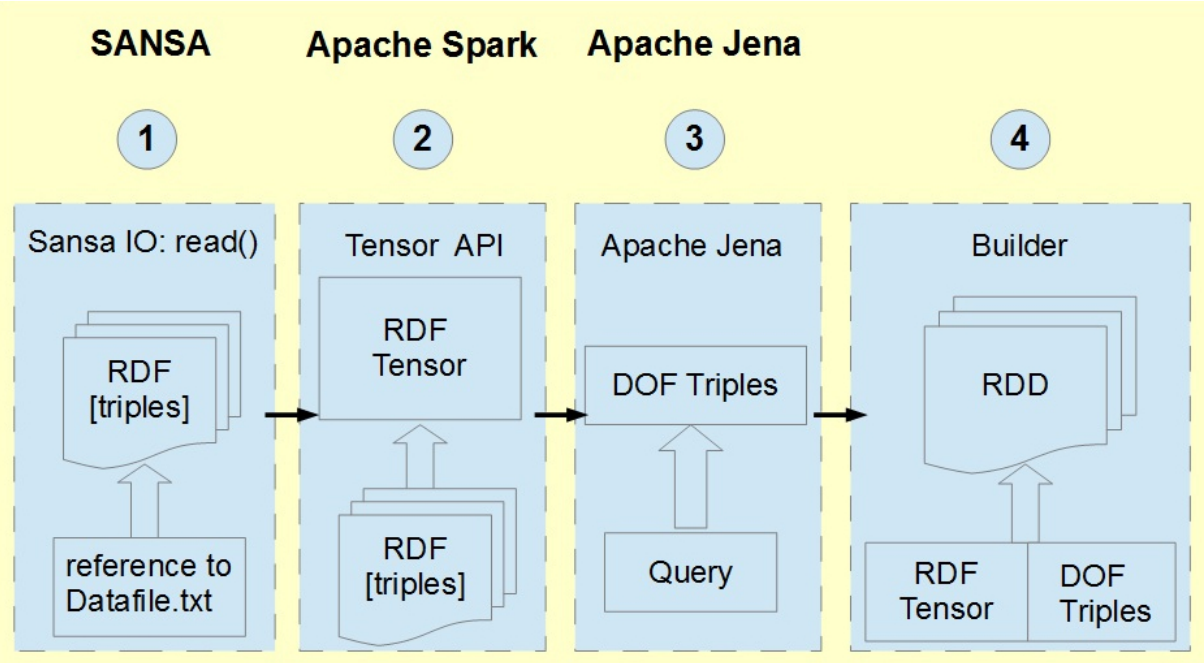


Figure 4.1.1: Approach architecture

Block 1 imports the *net.sansa_stack.rdf.spark.io* package methods, which read the input text file and convert it into an RDD of triples for further data processing.

Block 2 is an interface with an API that calculates a tensor from the RDD of triples obtained as a result of processing in Block 1. This is the contribution of the thesis.

Block 3 receives a set of query strings as input and converts them into Apache Jena Query objects. It also recursively scans the query triple pattern to calculate the *degree of freedom* of each triple pattern.

Builder in Block 4 uses the results of two previous steps to get the final answer, which can be obtained in one of two forms: as RDD or as text (by calling *collect()* method on RDD). It should be noted that the type of the final dataset is not the RDD of triples but the RDD of values of the query result variables. This is also a contribution that was developed in the thesis for the first time.

4.2 Key Features

Distributed data querying can vary over time and depends on the size of a dataset. Such datasets, as well as SPARQL query processing, can be processed in parallel on multiple nodes via Apache Spark.

SPARQL parallel query processing is widely developed (see, for example, [1], [2], [8]). The main scenario can be described as follows. An RDF dataset with a large amount of triples is partitioned and distributed across a cluster, so that query statements can be executed locally on each piece of data in a certain way (for example, through vertical partitioning, indexes, low-dimensional vector spaces [20]). Subsequently, local results are merged in order to produce the final outcome.

Our approach is able to handle SPARQL queries for massive RDF datasets in a distributed way. It is based on the concept of the *degree of freedom* (DOF) of a query triple pattern and is capable to analyze RDF graphs without any *prior knowledge* or *initial data statistics*. Since DOF measures the constraints of a query triple pattern, we start with the query triple pattern that has the lowest DOF value, and consistently execute our processing by selecting the triple pattern with the highest probability of decreasing the search space.

Figure 4.2.1 shows the schema of distributed SPARQL query processing. Here we do not employ any schema or indexing definition over the RDF graph. Our primary model is based on the *RDF tensor* and avoids storing RDF data in triple form for better performance. The main idea is to transform an original dataset into the RDF tensor, split

the tensor, and process a query in parallel, starting with a query triple pattern with the lowest degree of freedom.

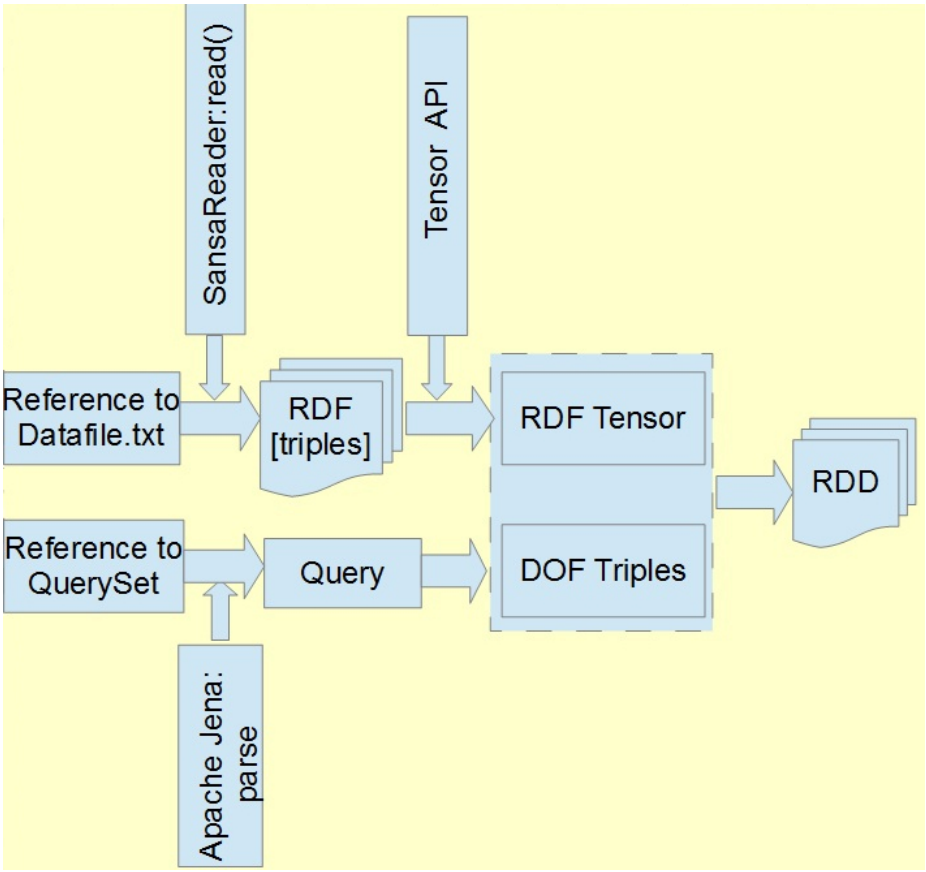


Figure 4.2.1: Data processing schema

4.3 Approach

In this section we present framework methods that are used in parallel query processing.

Algorithm 1 describes distributed SPARQL query processing that takes an original dataset and SPARQL query string as input, creates the RDF tensor, for each triple it computes the degree of freedom (DOF) of the query triple pattern. After that it calculates the resulting set of values of the query variables with respect to the tensorial notation and DOF of the current query triple pattern.

Line 1 of the algorithm calls *rdf* method of SANSA *RDFReader* class and loads RDF data from the physical storage, i.e., from a local file system or the Hadoop Distributed File System (HDFS). This method returns an RDD of triples.

Next, we have to convert the dataset of triples into RDD of numbers, which will represent tensor in further processing. First step in tensor creation is to determine a domain,

Algorithm 1 Distributed SPARQL query processing via DOF analysis

Input: $RDF\,rdf$: RDF dataset
Input: $org.apache.jena.query.Query\,query$: executing query
Output: RDD: RDD of values associated to query result variables

1: $RDD : dataset \leftarrow rdf.toRDD < Triple > ()$

2: **global** $Subjects, Predicates, Objects \leftarrow zipWithIndex(dataset)$
3: **global** $Tensor \leftarrow dataset.join(Objects).join(Predicates).join(Subjects)$
4: $Tensor \leftarrow Tensor.map(Predicates, Subjects, Objects)$

5: $dofs \leftarrow RecursiveElementVisitor.visit(query)$

6: **while** $dofs \neq \emptyset$ **do**
7: $dof, triple \leftarrow dofs.dequeue()$
8: $subTotal \leftarrow Tensor.process(dof, triple)$
9: $dofs.recalculate()$
10: $bindResults(Result, subTotal)$

where the tensor is defined. Apache Spark *zipWithIndex* method helps us to map elements of RDF sets into the corresponding (finite) subsets of the natural numbers (Line 2). It zips the elements of the source entity with their indexes. Hence, we identify all unique items of the dataset, i.e., subjects, predicates and objects, and store them into the RDD. Then we assign an unique ID to each element in a distributed way.

By calling *join* method several times in a row, we transform the dataset of triples into the RDD of all subjects, predicates and objects, joined together with their indexes (see Figure 4.3.1).

For large-scale RDF datasets $\overline{p}$ — the number of predicates is significantly smaller than $\overline{s}$ — the number of subjects and $\overline{o}$ — the number of objects (see [2], [8]). Using the definition of RDF triple (see [36]) and the definition 2.3.3, we get the following important inequality:

$$\overline{p} \ll \overline{s} \leq \overline{o}, \qquad \overline{p} := \max(\mathbb{P}),\ \overline{s} := \max(\mathbb{S}),\ \overline{o} := \max(\mathbb{O}). \tag{4.3.1}$$

Therefore, we can apply our modification of the *vertical partitioning schema* (see, for example, [18]). To use it further, Line 4 converts the RDF tensor, placing the predicate column in the first position.

$RecursiveElementVisitor$ traverses the query triple pattern and calculates the initial values of the *degree of freedom* for each triple (Line 5). As a result, it returns priority queue of pairs (i.e. high priority corresponds to low DOF associated to a constraint t). Lines 6–10 loop all over such pairs to obtain intermediate tensor computations for t (Line

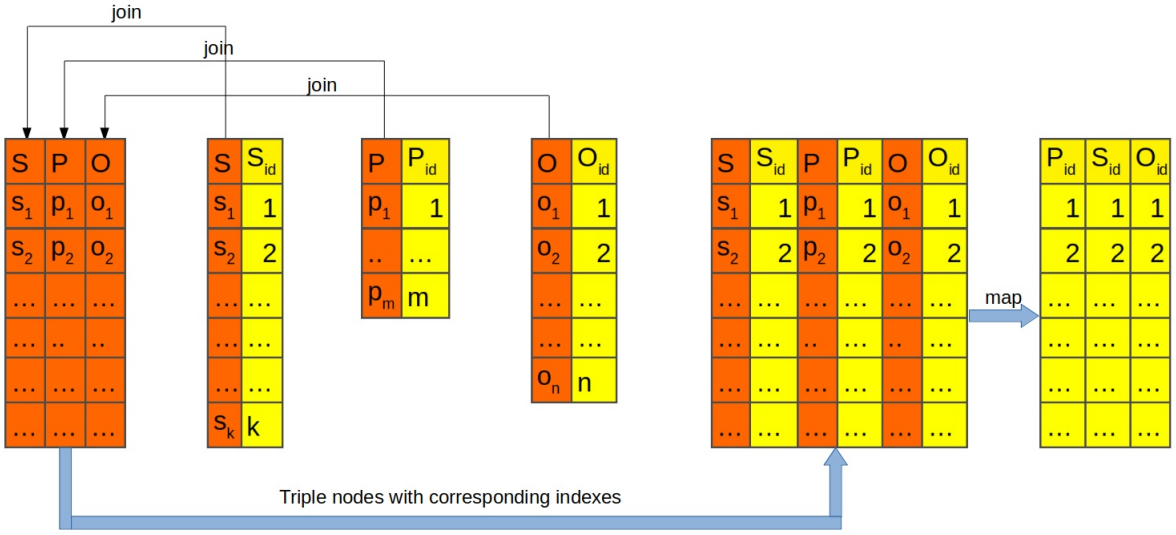


Figure 4.3.1: RDF–to–Tensor transformation

8). Algorithm 2 hereinafter describes this step in more detail. Each iteration returns a set of values associated with the triple variables. We have to *recompute* the degree of freedom of the remaining constraints, since the variable can be promoted to the role of the constant (Line 9). *bindResults* connects the intermediate results with the results of previous iterations.

Constraint processing is performed by *Tensor.process* method. As shown in Algorithm 2, it takes as input a constraint t and a map M, which keys are the variables from all query triple patterns. At the beginning we associate an empty RDD to each key, i.e. M is empty. As we traverse the query triple pattern recursively, M will contain results of the previous computations.

Line 1 of the procedure employs the tensor and the indexed sets of subjects, predicates and objects, obtained in Lines 2–4 of the Algorithm 1. Line 2 makes a local copy of the tensor that will be used and modified in the further processing. *getSubject*, *getPredicate*, *getObject* methods of Apache Jena *Triple* class return the corresponding field of a triple (Line 3). Since we represent the tensor as a resilient distributed dataset with three columns of the numbers, Line 4 introduces a variable *index* that refers to a particular tensor column. Lines 5–15 process the triple part and the corresponding indexed set cyclically. We always start with the *predicate* and *Predicates* set in order to apply our modification of the vertical partition schema. If *isVariable* method of the class *Triple* returns **true** (Line 6), and the variable is bounded, i.e. the map M already contains values for it (Line 7), the indexed set and the existing map values will be joined together (Line 9), producing a subset of the existing indexed set. If the variable is unbounded,

the whole indexed set will be used. In case of the field being a *constant*, $filter$ method removes the indexed set elements, which are not equal to the current triple field, i.e. Line 13 obtains an index value of the corresponding field.

Line 14 performs tensor *slicing* is performed, calling *join* method for the tensor and result variables and using *index* as the key column.

At the end, we return the map M, updated with an iteratively sliced tensor (Lines 16–17).

Algorithm 2 Tensor processing

Input: $Triple\,t$: Query constraint
Input: $Map[Var, RDD[Node]]\ M$: Map with the results of previous computations
Output: $Map[Var, RDD[Node]]$: Map with RDD values associated to each variable

1: **global** $Tensor, Subjects, Predicates, Objects$

2: $temp \leftarrow Tensor$
3: s, p, $o \leftarrow t.getSubject()$, $t.getPredicate()$, $t.getObject()$
4: $index \leftarrow 0$

5: **for** $(node, IndexedSet) \in ((p, Predicates)$, $(s, Subjects)$, $(o, Objects))$ **do**
6: **if** $node.isVariable$ **then**
7: $isBounded \leftarrow M.hasKey(node)$
8: **if** $isBounded$ **then**
9: $rdd \leftarrow IndexedSet.nodes.join(M.get(node))$
10: **else**
11: $rdd \leftarrow IndexedSet$
12: **else**
13: $rdd \leftarrow IndexedSet.nodes.filter(current == node)$
14: $temp \leftarrow temp.map(row \rightarrow (row(index), row)).join(rdd.indexes)$
15: $index++$

16: $M.update(temp, t)$
17: **return** M

4.4 Query Set

According to the proposed approach, we use a set of queries of `SELECT...WHERE` type with different options. To cover the main features provided by SPARQL, we considered the options `Distinct, Optional, Union, Filter` and the conjunction operation. Thus, SPARQL key features for RDF querying can be implemented.

In order to study the validity of the proposed approach, it is necessary to compare the results with ones obtained by different authors in similar conditions. For this purpose,

we use a query set from [36] as the basis for test queries. Table 4.4.1 demonstrates the complexity of the query triple pattern for the query set members.

Table 4.4.1: Query complexity

Query	Result Variables	Number of Variables	Number of Triples	Common Variables	Union	Conjunction	Optional
Q1	1	2	1	0			
Q2	3	3	1	0			
Q3	3	4	4	1		+	+
Q4	4	4	2	0	+		
Q5	3	4	5	3	+	+	
Q6	3	3	3	1		+	
Q7	8	10	11	1		+	+
Q8	2	3	3	2	+	+	
Q9	5	5	7	3	+	+	+
Q10	1	9	8	3		+	+
Q11	3	4	9	4	+		+
Q12	1	3	3	2		+	
Q13	2	3	2	1		+	
Q14	1	4	3	1	+		
Q15	4	5	4	1		+	+
Q16	3	5	3	2		+	+
Q17	2	2	3	1	+		
Q18	3	5	2	1	+		
Q19	1	3	2	1	+		
Q20	8	8	8	1	+	+	+
Q21	2	2	1	0			
Q22	1	3	2	1		+	
Q23	1	2	2	1		+	
Q24	5	5	4	2		+	
Q25	4	5	4	1		+	

4.5 Qualitative and quantitative evaluation of query processing

Our goal is to enrich SANSA layers with new methods and interfaces that can enhance the effectiveness of SANSA and expand its scope. To do this, we need to find gaps in the existing functionality, which can be supplemented by new features.

To achieve the goal, three steps have to be performed:

1. We need to find out the scope of the Master thesis in the general scheme of SANSA in order to develop new methods and interfaces. Thus, we contribute to the expansion of the functionality implemented in SANSA.

2. We have to provide the methods for qualitative and quantitative evaluation of the

effectiveness of the proposed components.

3. It is necessary to conduct a series of numerical experiments and evaluate the reliability and validity of the proposed approach.

The first step was discussed above, and here we present the second.

The main criteria for the effectiveness of a computer method or technology are the size of the memory used and the program execution time.

In our case, for large-scale distributed in memory knowledge graphs, we estimate the query execution time, which is also determined by the representation of data within the system.

Usually, two methods are used to estimate the execution time of a computer method (module, interface, etc.):

- Qualitative evaluation proves the estimated time of the investigated method. Hence, it allows classifying the implementation of the method as being P- or NP-hard.
- For quantitative evaluation a series of numerical experiments is conducted on data of various sizes M and the execution time $T(M)$ is analyzed, i.e. if

$$T(M) = \mathcal{O}(M^k), \quad 1 \leq k, \tag{4.5.1}$$

then the investigated method is quantified positively.

Sometimes it is possible to compare quantitative evaluation of runtime $T(M)$ with similar results $\widetilde{T}(M)$ by another researchers who use different methods, but with the same functionality and with the same input data. If, as a result of the comparison, we get the estimate:

$$T(M) = [1 + o(1)]\,\widetilde{T}(M), \tag{4.5.2}$$

then $T(M)$ and $\widetilde{T}(M)$ have the same order of growth, and this fact indicates that the proposed approach is not inferior in terms of quantitative evaluation compared with the existing methods.

The best-case scenario is when quantitative evaluation confirms qualitative.

Note that in [36] the general asymptotic complexity of the method is given in terms of $\mathcal{O}(nnz(M))$ and $\mathcal{O}(nnz(v))$, where the notation $nnz(M)$ with M being the rank-3 RDF tensor, denotes the number of its non-zero values; $nnz(v)$ is equal to the number of non-zero elements of the vector v. It should also be noted that the estimates in [36]

are given separately for the sub-operations Insertion, Deletion and Update, Hadamard Product, Tensor Application, Mapping.

Below we present a qualitative estimate in terms of the size of the RDF dataset. We show that its theoretical complexity is polynomial; the obtained right-hand estimate shows the asymptotic behavior over time, which depends on the size of the RDD dataset.

Proposition 4.5.1. *Getting the resulting RDD from the RDF tensor is a P-solvable problem, i.e. it requires polynomial time:*

$$T_{RDD} = \mathcal{O}(N^2) \ll T_{RDF} = \mathcal{O}(N^3), \tag{4.5.3}$$

where N is a number of all unique objects of the RDF tensor.

Proof. Note that from the triple definition it follows that in the Cartesian coordinate system the RDF tensor can be represented as a three-dimensional cube with the dimension along the axes equal to $number(S) := \overline{s}$, $number(P) := \overline{p}$, $number(O) := \overline{o}$ (see, also (4.3.1)). Here $number(X)$ denotes the cardinality of the corresponding set of subjects, predicates and objects. Therefore, the maximum size of the dataset does not exceed $number(S) \times number(P) \times number(O) = \overline{s} \times \overline{p} \times \overline{o}$. From the right-hand of (4.3.1) (in fact from the definitions of the sets S and O) it follows that

$$\overline{s} \leq \overline{o}. \tag{4.5.4}$$

As mentioned above in (4.3.1)

$$\overline{p} =: c \ll \overline{o} := N \in \mathbb{N}, \quad c = \text{const} \in \mathbb{N}.$$

This implies that the size of the RDD does not exceed cN^2 and any greedy algorithm will search at most cN^2 points of the three-dimensional cube, therefore the problem has a polynomial time complexity

$$T_{RDD} = \mathcal{O}(N^2). \tag{4.5.5}$$

□

Remark 4.5.1. *Since query execution is based on a database search algorithm, and estimate* (4.5.3) *is satisfied for any greedy search algorithm (i.e., in the worst case), it holds for all efficient search algorithms used to execute the query.* □

Remark 4.5.2. *Calculated once, the RDF tensor can be applied for querying any number of times , which significantly reduces the overall execution time.* □

Proposition 4.5.1 holds for a simple `SELECT... WHERE` query with a number of variables less than or equal to three. However, so-called *conjunctive* queries are used more often (see additionally about conjunctive query patterns and their complexity in [36]).

Let several simple sub-queries (i.e. the query triple patterns) be conjugated and, the search is performed over k variables of the query triple patterns, m query triple patterns and n common variables of the query triple pattern, $k \geq 1$, $m \geq 1$, $n \geq 0$. Note, if all the query pattern variables are different, then $n = 0$; if $m = 1$, then the query contains only one query triple pattern (i.e., the simplest case).

The following statement about the complexity of the *conjunctive* query holds.

Proposition 4.5.2. *Let $Q_{k,m,n}$ conjunctive query have k query variables, m query triples and n query common variables, $k \geq 1$, $m \geq 1$, $n \geq 0$. Then*

1. *the complexity of $Q_{k,m,n}$ is determined by the values of k, m and n;*
2. *the response time $T(Q_{k,m,n})$ can be evaluated as*

$$T(Q_{k,m,n}) = f(m), \tag{4.5.6}$$

   where $f(m)$ is an increasing function dependent on the variable m for any fixed pairs of k and n;

3. *the response time $T(Q_{k,m,n})$ can be evaluated as*

$$T(Q_{k,m,n}) = g(k), \tag{4.5.7}$$

   where $g(k)$ is an increasing function dependent on the variable k for any fixed pairs of m and n;

4. *the response time $T(Q_{k,m,n})$ can be evaluated as*

$$T(Q_{k,m,n}) = h(n), \tag{4.5.8}$$

   where $h(n)$ is a decreasing function dependent on the variable n for any fixed pairs of m and k.

Proof. We prove Statement 2 by induction. For $m = 1$ this follows from the fact that a simple query triple pattern has at most three variables, and its response time is bounded from above by N^k for k variables, $k \leq 3$, and $n \equiv 0$ (see also the proof in 4.5.1). Suppose that (4.5.6) holds for any $m_1 > 1$ and we show that (4.5.6) is satisfied for $m = m_1 + 1$. The query is represented by a conjunction of $m = m_1 + 1$ simple sub-queries (i.e. the

query triple patterns) with k variables and n common variables. The response time of the conjunction of m_1 simple queries with k_1 variables and n_1 common variables is $f(m_1)$.

Hence, we have that the conjunction of $m = m_1 + 1$ simple queries has no more than $k \leq k_1 + 3$ variables (and no more than $n \leq n_1 + 3$ common variables). And it follows that the response time of the conjunction of $m = m_1 + 1$ simple sub-queries is bounded from above by $N^3 f(m_1)$, which is increasing function. Note that increasing the number of common variables can only reduce the power of N^3, since an increase in the number of common variables may include those already presented in m_1, but in any case such increase cannot be greater than 3.

Statement 3 is proved in the similar way.

Statement 1 follows from Statements 2 and 3.

Statement 4 is proved in the same way as Statement 2, with the only difference that instead of the function $h(n_1)$ we have $n > n_1$ and $\Delta := n - n_1 > 0$, and then we get the function $N^{-\Delta} h(n_1) = \dfrac{h(n_1)}{N^{\Delta}} < h(n_1)$.

□

Remark 4.5.3. *Propositions 4.5.1 and 4.5.2 contain the upper estimates, that can be improved by considering more specific cases and using the degree of freedom of the query triple pattern.* □

Remark 4.5.4. *Proposition 4.5.2 can be verified experimentally (see more information in the following chapter).* □

5 Implementation and Evaluation

5.1 Datasets and Queries

The effectiveness and efficiency of the presented approach are tested on Dbpedia datasets of different sizes:

D1: http://downloads.dbpedia.org/3.9/simple/ 2,7 GB (17M triples);

D2: http://downloads.dbpedia.org/3.9/ro/ 6,6 GB (41M triples);

D3: http://downloads.dbpedia.org/3.9/uk/ 20,8 GB (79M triples);

D4: http://downloads.dbpedia.org/3.9/de/ 48,6 GB (336M triples).

In order to evaluate the effectiveness of the algorithm, we use a set of 25 queries similar to those used in [36] and described in Appendix A (see also, https://www.dropbox.com/sh/pz-0i67s9ohbpb9t/oEGo-J8yui). To cover the key features of SPARQL, each query has a common type of SELECT ... WHERE, but differs in the combination of options (Distinct, Optional, Union, Concatenation, Filter).

5.2 Cluster Environment

The performance of the developed system havs been tested on a cluster with 2 executors having a total of 128 cores, each server has Xeon Intel CPUs at 2.3GHz, 256GB of RAM and 400GB of disk space, running Ubuntu 16.04.3 LTS (Xenial) and connected via a Gigabit Ethernet2 network. We have used Spark Standalone mode with Spark version 2.2.1 and Scala with version 2.11.11. Each Spark executor is assigned a memory of 250GB.

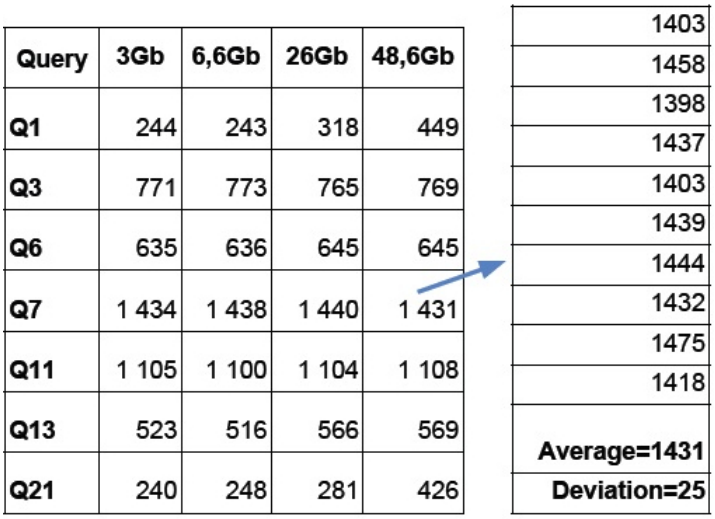
5.3 Results analysis

Taking into account Remark 4.5.2, this section presents the measurement results of the query execution time. As in [36], we conducted a series of ten cold runs and measured the average response time to process the query.

Generally speaking, it is actually advisable to analyze the execution time of any query in the form of $AverageResponseTime \pm StandartDeviation$. This way of presenting results allows to explore the actual corridor, which includes the expected processing time of the

query. Thus, it is possible to reduce the influence of run-time fluctuations to determine the dependence of the runtime on the size of the data set.

An example of calculating the average response time with the standard deviation from the average value is shown in Figure 5.3.1.



Query	3Gb	6,6Gb	26Gb	48,6Gb
Q1	244	243	318	449
Q3	771	773	765	769
Q6	635	636	645	645
Q7	1 434	1 438	1 440	1 431
Q11	1 105	1 100	1 104	1 108
Q13	523	516	566	569
Q21	240	248	281	426

1403
1458
1398
1437
1403
1439
1444
1432
1475
1418
Average=1431
Deviation=25

Figure 5.3.1: Average response time and standard deviation from the average value

We obtained experimental confirmation of the effect of query complexity on the execution time, which was specified in Proposition 4.5.1 and Proposition 4.5.2. The results of the experiments are presented in Table 5.3.1 and Table 5.3.4 and show the validity of the approach to determining the complexity of the queries.

In fact, the complexity of the query and, accordingly, the execution time of the query are determined primarily by the number of the query triple patterns and the number of the query common variables.

From Table 5.3.1 it can be seen that Q23 with the same number of the query triple patterns has a shorter response time than Q4, since the first one has a common variable and Conjunction operation, and the second one has no common variables and contains Union operation (see Appendix A and Table 4.4.1).

The results of Table 5.3.2, based on Table 5.3.1, confirm Statement 2 of Proposition 4.5.2 and more clearly demonstrate, that response time of a query is an increasing function dependent on the number of the query triple patterns for any fixed pairs of the number of variables and common variables.

Table 5.3.1: Response time w.r.t. the number of the query triple patterns

Query	Number of Result Variables	Number of Variables	Number of Query Triple Patterns	Number of Common Variables	Response Time on D4 (in ms)
Q21	2	2	1	0	426.2
Q2	3	3	1	0	432.6
Q1	1	2	1	0	448.7
Q23	1	2	2	1	547.2
Q4	4	4	2	0	550.7
Q22	1	3	2	1	553.5
Q19	1	3	2	1	553.7
Q13	2	3	2	1	568.8
Q18	3	5	2	1	627
Q17	2	2	3	1	604.8
Q6	3	3	3	1	644.7
Q14	1	4	3	1	654.9
Q12	1	3	3	2	669.2
Q8	2	3	3	2	677.3
Q16	3	5	3	2	707.5
Q3	3	4	4	1	768.5
Q24	5	5	4	2	782.5
Q15	4	5	4	1	791.3
Q25	4	5	4	1	793.1
Q5	3	4	5	3	885
Q9	5	5	7	3	1036.1
Q20	8	8	8	1	1116.6
Q10	1	9	8	3	1167.7
Q11	3	4	9	4	1108
Q7	8	10	11	1	1430.7

Table 5.3.2: Response time w.r.t. the increasing number of the query triple patterns

Query	Response Time on D4 (in ms)	Number of Query Triple Patterns	Number of Variables	Number of Common Variables
Q1	449	1	2	0
Q23	547	2	2	1
Q6	645	3	3	1
Q25	793	4	5	1
Q20	1117	8	8	1
Q7	1431	11	10	1

The analysis of the results of Table 5.3.3 is also based on Table 5.3.1 and confirms Statement 3 of Proposition 4.5.2. It demonstrates that the query response time is an increasing function dependent on the number of variables for any fixed pairs of the number of query triple patterns and common variables.

Table 5.3.3: Response time w.r.t. the increasing number of variables

Query	Response Time on D4 (in ms)	Number of Variables	Number of Query Triple Patterns	Number of Common Variables
Q21	426	2	1	0
Q13	569	3	2	1
Q3	769	4	4	1
Q24	783	5	4	2
Q20	1117	8	8	1
Q10	1168	9	8	3
Q7	1431	11	10	1

Table 5.3.4: Response time w.r.t. the number of common variables

Query	Number of Result Variables	Number of Variables	Number of Query Triple Patterns	Number of Common Variables	Response Time on D4 (in ms)
Q21	2	2	1	0	426.2
Q2	3	3	1	0	432.6
Q1	1	2	1	0	448.7
Q4	4	4	2	0	550.7
Q23	1	2	2	1	547.2
Q22	1	3	2	1	553.5
Q19	1	3	2	1	553.7
Q13	2	3	2	1	568.8
Q17	2	2	3	1	604.8
Q18	3	5	2	1	627
Q6	3	3	3	1	644.7
Q14	1	4	3	1	654.9
Q3	3	4	4	1	768.5
Q15	4	5	4	1	791.3
Q25	4	5	4	1	793.1
Q20	8	8	8	1	1116.6
Q7	8	10	11	1	1430.7
Q12	1	3	3	2	669.2
Q8	2	3	3	2	677.3
Q16	3	5	3	2	707.5
Q24	5	5	4	2	782.5
Q5	3	4	5	3	885
Q9	5	5	7	3	1036.1
Q10	1	9	8	3	1167.7
Q11	3	4	9	4	1108

The influence of the number of common variables on the complexity and, therefore, on the response time of the query can be traced by the experimental results given in

Table 5.3.4. For convenience and clarity of analysis of Table 5.3.4, we selected and grouped in Table 5.3.5 the results that most clearly characterize the property under investigation.

Table 5.3.5: Decreasing response time w.r.t. the number of common variables

Query	Response Time on D4 (in ms)	Number of Common Variables	Number of Query Triple Patterns	Number of Variables
Q15	791	1	5	4
Q24	783	2	5	4
Q4	551	0	2	4
Q23	547	1	2	2
Q20	1117	1	8	8
Q11	1108	4	9	4

The analysis of the data from Table 5.3.5 shows that the response time is a decreasing function dependent on the number of common Variables for any fixed pairs of the number of triples and variables (see first and second rows of Table 5.3.5 and taking into account Statement 4 of Proposition 4.5.2). One can also consider a decrease in the response time when the number of the query triple patterns is fixed, and the number of variables is reduced. In addition, if the number of common variable increases, the response time can be reduced even if there are an increase in the number of the query triple patterns, an increase in the number of triples and a decrease in the number of variables (see fifth and sixth rows of Table 5.3.5).

It should be noted that the longer Q18 runtime with a smaller number of the query pattern triples compared to Q17 in Table 5.3.4 is explained by the fact, that the predicates used in Q17 and Q18 are constants and a variable respectively (see Appendix A and Table 4.4.1).

Now we will discuss the performance of the proposed framework based on the results obtained. Basically it can be estimated in terms of availability, response time [54], scalability and accuracy.

The availability of a computer system is typically measured as a factor of its reliability. Since the proposed framework is considered as a part of SANSA, this property was not the subject of our study. However, it can be the subject of future research.

Also note that scalability is usually understood either as system extensibility, or as acceptability of the results generated by the system when the size of the processed dataset changes. Below we analyze the behavior of the response time of the queries as the size of the dataset being processed increases. We remind that we consider the average query

execution time, and all queries have different complexity and various combinations of options.

Table 5.3.6 shows the results of the query response time (in ms) for four RDF datasets.

Table 5.3.6: Response time (in ms) w.r.t. the dataset size

Query	D1	D2	D3	D4
Q1	244,3	243	317,5	448.7
Q2	278	281,5	441,3	432.6
Q3	771	773,3	764,7	798.5
Q4	563,8	572,9	577,5	550.7
Q5	902,5	904	909,9	885
Q6	634,8	636	644,8	644.7
Q7	1433,6	1437,5	1440	1430.7
Q8	676,2	697,9	699,1	677.3
Q9	1045,4	1032,6	1052,6	1036.1
Q10	1164,1	1182,3	1200,7	1167.7
Q11	1105,2	1100,3	1103,7	1108
Q12	644	663,5	634,6	669.2
Q13	522,8	516,2	566,2	568.8
Q14	685,9	680,4	690,9	654.9
Q15	805,2	782,9	813	791.3
Q16	741,2	715,6	702,6	707.5
Q17	630,3	599	584,8	604.8
Q18	651,3	658,7	661,8	627
Q19	568,8	575,1	569,2	553.7
Q20	1110,9	1097,8	1117,6	1116.6
Q21	240,2	248,3	280,7	426.2
Q22	494,5	550,7	566,8	553.5
Q23	398,2	522,5	533,4	547.2
Q24	754,2	739,7	766,6	782.5
Q25	786,9	792,4	778	793.1

Analysis of the values presented in Table 5.3.6, showed that the query execution time (in ms) increases depending on the size of the dataset. In particular, it can be seen that when the number of triples in the datasets D1 and D3 goes from 17 to 79 million respectively (i.e., 4.57 times), the query execution time of Q1 increases 1.3 times.

Constructed for Q1 and Q21 respectively, Figure 5.3.2 and Figure 5.3.3 show that the query execution time is in direct ratio to the size of the dataset.

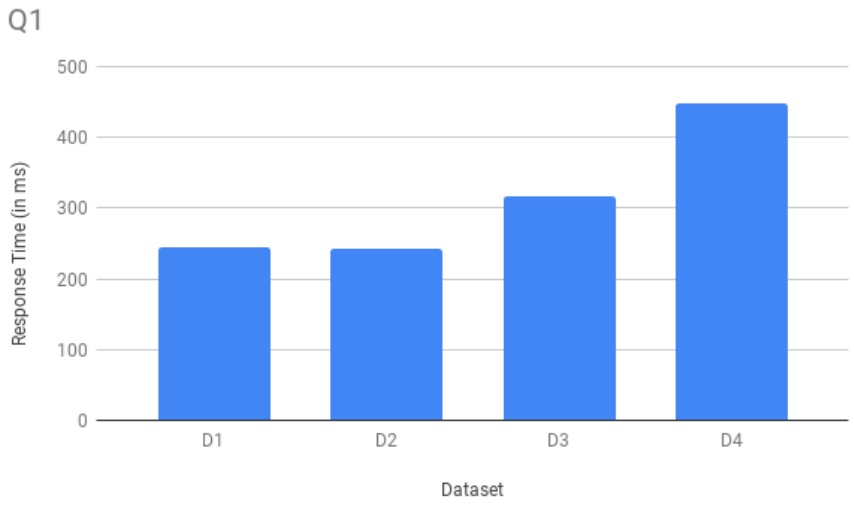


Figure 5.3.2: Q1 response time

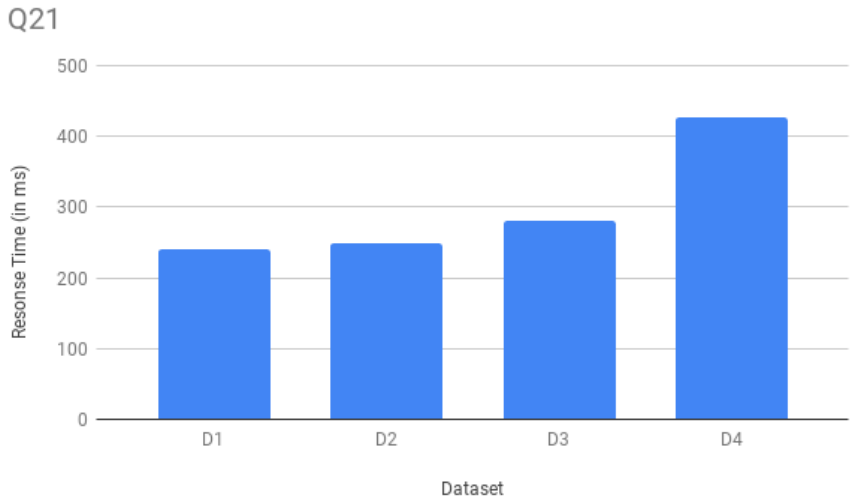


Figure 5.3.3: Q21 response time

It should be noted that the presence of Conjunction, Union and other additional operators and options in the processed query significantly affects the execution time (as well as in [36]).

For example, when the number of processed triples in D1 and D3 increased by 4.57 times, then the execution time of queries Q22 and Q23, including Conjunction operator, increases by 1.15 and 1.34 times respectively. At the same time, the scalability factor, as the ratio of the increase in the average response time of Q22 and Q23 to the increase in the size of D1 and D4, grows and falls within the range of 6.2%-7.6% (see Figure 5.3.4).

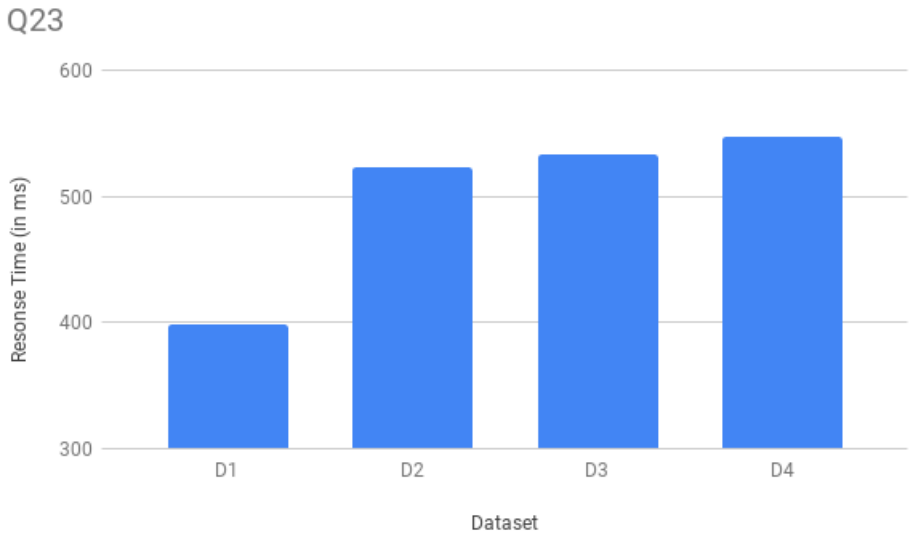


Figure 5.3.4: Q23 response time

While Union results in a scalability factor in the range of 5.8%-6.0%, using Conjunction and Filter increases the query execution time, maintaining a tendency to grow as the dataset size increases (see Figure 5.3.5, the scalability factor equals to 5,6%).

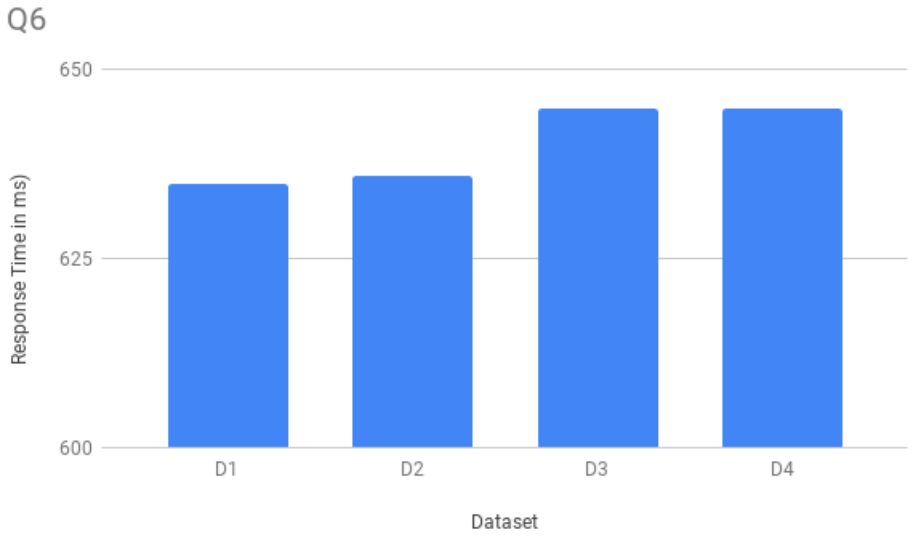


Figure 5.3.5: Q6 response time

However, the use of Union in query Q11 can lead to a significant increase in the execution time of query processing (see, for example, Figure 5.3.6).

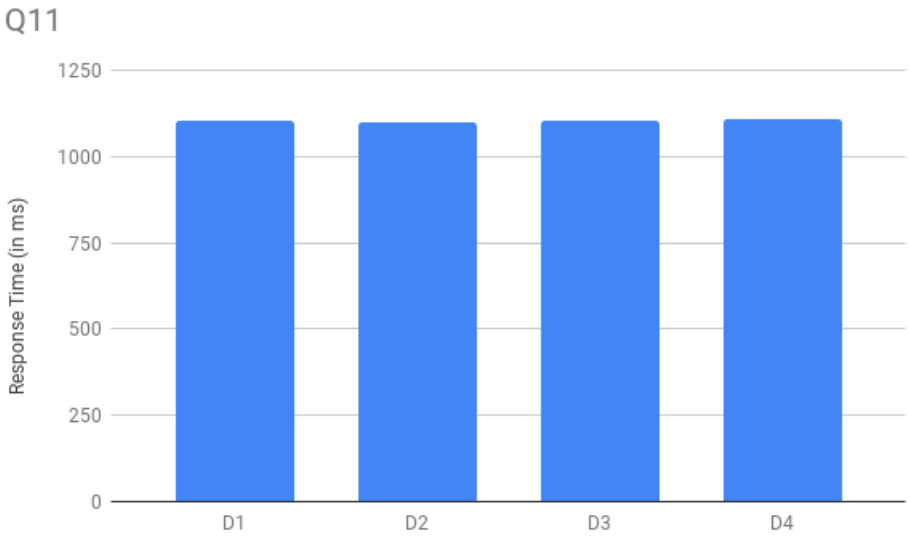


Figure 5.3.6: Q11 response time

Naturally, a longer response time corresponds to more complex queries with many query variables and having Union, Optional, Conjunction operators, for example, Q7, Q10, Q20. In addition, an increase in the execution time may depend on the use of the Distinct, Optional and Filter operators. For considered queries, when the size of the dataset increases from minimum to maximum within the considered values, the use of the Distinct operator leads to an increase in Q1 execution time by 1.3 times; Optional increases Q15 response time by 1.04 times. However, the scalability factor remains within the acceptable limits of 5.6%–9.9%.

In general, the execution time of any processed query increased by no more than 1.84 times, when the size of the datasets used was increased from the minimum to the maximum (i.e., the D1 and D4, having a size of 2.7 GB and 48.6 GB, differed in size by 18 times).

Therefore, the ratio of the change in the average response time (in ms) of any considered query to an increase in the size (in GB) of the data sets used is in the range of 5.6%–10.3%.

Figure 5.3.7 and Figure 5.3.8 demonstrate this property. The quadruple of each query outlined in Figure 5.3.7 differs in growth in the above-indicated range and indicates the scalability of data processing.

Parallel graphs at Figure 5.3.8 depict the scalability property using Standart Stacked Area Chart; slight variation is in the same range.

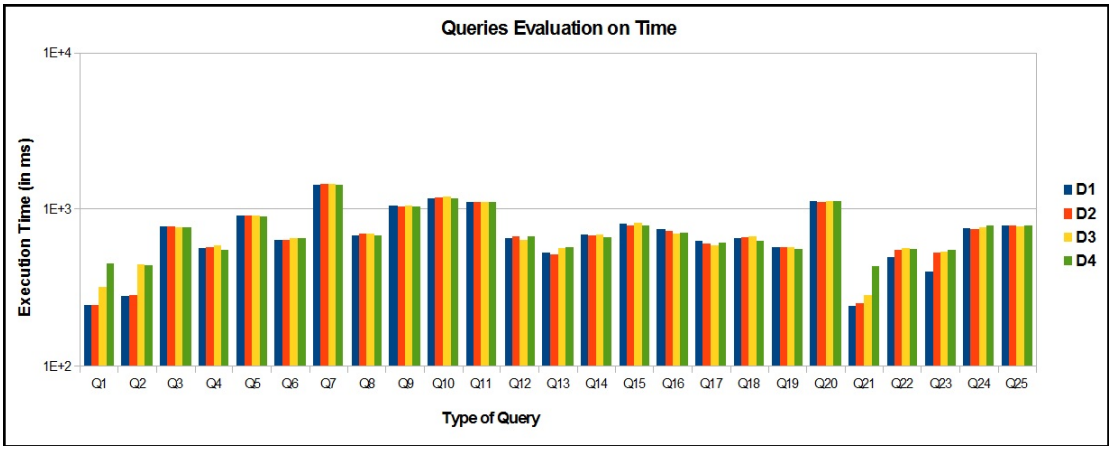


Figure 5.3.7: Relation between the query execution time and the size of the dataset

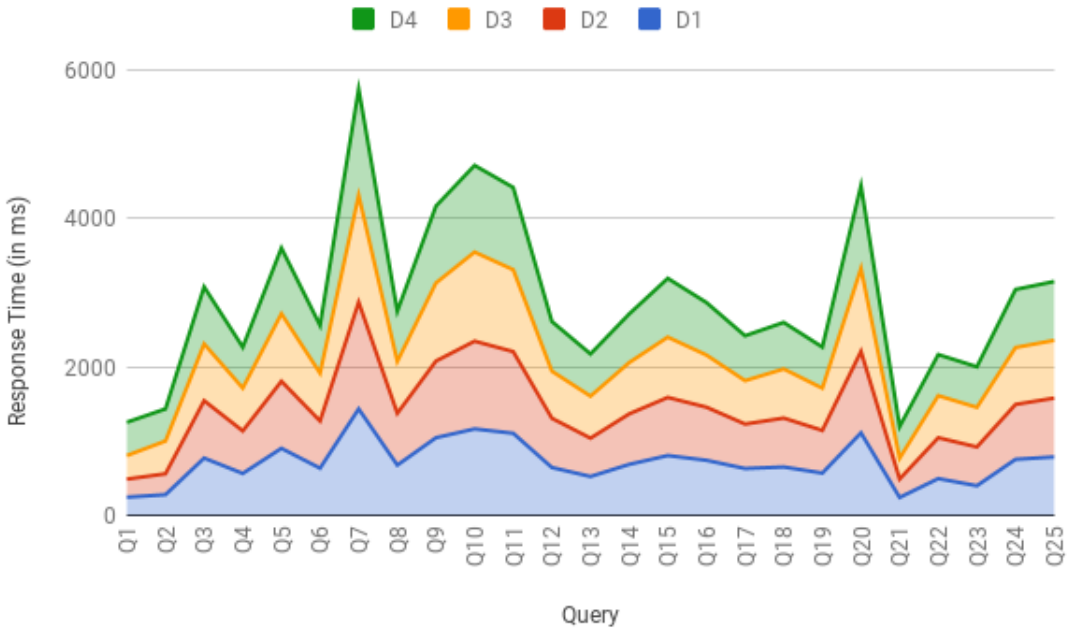


Figure 5.3.8: Scalability Diagram

In [36], the RDF tensor was implemented for a similar query set using C ++ and for comparison with other approaches the author presented the response time on DBPEDIA v3.6 (200M triples loaded into the official SPARQL endpoint) on a single machine. For Q1, Q2, Q6, Q16, Q18, Q19, Q21, Q22 and Q25, which are similar to our queries, de Virgilio's results are less than 1 ms; for Q7, Q9, Q11 and Q20, which are also similar, it exceeds 100 ms and does not exceed 1000 ms; for the rest, the response time ranges from 1 ms to 100 ms.

In order to properly compare the query execution time, all queries were divided into two groups. First group members had a runtime less than 10 ms, and the second group with the querying time between 10 ms and 1000 ms. It was found that the runtimes obtained in this thesis differ in the first group by 2 orders of magnitude, and in the second group only by 1 order from the similar results in [36]. This is explained by the fact that the low-level operations used in [36], while increasing the size of the dataset, do not significantly increase the response time during processing.

The response time obtained on dataset D3 for queries Q9, Q7 and Q20 varies from 1052.6 ms to 1440 ms; for the rest – from 317.5 ms to 1200.7 ms.

Thus, our results differ from [36] by no more than 300 times. Such difference may be due to the content of triples of the datasets. A smaller number of triples satisfying the selection conditions corresponds to a shorter response time of a query. Because of this, the response time of Q16 and Q17 in Table 5.3.6 decreases by about 5%-7% with an increase in the size of the dataset by 4.57 times.

Now consider the question of the accuracy and correctness of the results.

Initially, the validation of the proposed approach, the developed algorithms and the implemented framework was performed manually for small datasets; for example, the graph presented in [36] was used in query processing and its validation. The obtained results confirmed the correctness and accuracy of the the query processing algorithms, the proposed approach and the framework.

We also launched several queries on the Sparql endpoint for Ukrainian Dbpedia, which is available at http://uk.dbpedia.org/sparql. To prove the consistency of the approach, the size of the query results was compared with the data, obtained by querying D3 (dataset for Ukrainian Dbpedia) of 20.8 GB in size and with more than 79M triples. Information on the number of triplets obtained for some queries is given in Table 5.3.7.

Table 5.3.7: Accuracy and correctness of query processing

Query	Number of Response Triples
Q1	422
Q5	274
Q7	1837
Q8	411
Q9	42203
Q12	876
Q14	136174
Q22	640578
Q23	17933

6 Conclusion and Perspectives

This thesis presents an approach for distributed in-memory SPARQL processing using the functionality of SANSA, Apache Spark, Apache Jena etc.

Our approach was based on the concepts of the RDF tensor, the degree of freedom of the query triple pattern, the complexity of the query, vertical partitioning of RDF datasets. We also investigated the effect of complexity of query parameters on the execution time, as well as the P-solvability of the problem in general. A framework developed for large distributed in-memory can be considered as part of SANSA.

Summarizing, with the help of the RDF tensor, we extended the functionality of SANSA Knowledge representation and Distribution layer. Since the RDF tensor can be considered as a three-dimensional cube, the predicate set has a sparseness property, and the number of predicates is usually smaller than the number of objects and the number of subjects in large RDF datasets, therefore vertical partitioning was applied to reduce the query execution time.

We proved and confirmed experimentally the effectiveness of the developed framework, the P-solvability of the investigated problem, and the direct dependence of the query execution time on the the number of query triples and query variables. These results were obtained for the first time.

As a measure of triple pattern's explicit constraints the degree of freedom of the query triple pattern was used to optimize the query execution in accordance with its complexity. We identified several key parameters of the query complexity, such as the number of query triples, the number of query variables and the number of query common variables.

For future work, we would like to include in the developed framework other types of SPARQL queries, in addition to the implemented "Select" queries.

It would be interesting to investigate the question of how to improve the execution time by changing the order in which query triples are selected for processing.

Finally, this thesis addressed to the problem of distributed in-memory SPARQL processing and justified the need to develop a framework for implementing SPARQL queries for this purpose.

We hope the presented approach and our framework, along with the proposed steps for its further development, can help in extending the functionality of SANSA and could be useful for distributed data processing and knowledge extraction.

List of Figures

List of Tables

Appendix A. Queries

For query processing on Dbpedia data sets we have a set of 25 SPARQL queries, similar to those defined in [36]. Common query prefixes are listed below:

```
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX dbp-owl: <http://dbpedia.org/ontology/>
PREFIX dbp-prop: <http://dbpedia.org/property/>
PREFIX dbp-yago: <http://dbpedia.org/class/yago/>
PREFIX dbp-cat: <http://dbpedia.org/resource/Category/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX georss: <http://www.georss.org/georss/>
```

The description of all query patterns is given below:

Q1: Find all the distinct object values of the *type* property.

```
SELECT DISTINCT ?v1
WHERE { ?x rdf:type ?v1. }
```

Q2: Return a complete data set.

```
SELECT *
WHERE { ?x ?v2 ?v1. }
```

Q3: For every *Person* return values of the properties *thumbnail* and *page* and the optional *homepage* property.

```
SELECT ?v4 ?v8 ?v10
WHERE { ?v5 dbp-owl:thumbnail ?v4.
?v5 rdf:type dbp-owl:Person.
```

```
?v5 foaf:page ?v8.
OPTIONAL {?v5 foaf:homepage ?v10.}}
```

Q4: Return double set of all subjects and objects with the *name* property.

```
SELECT ?v6 ?v9 ?v8 ?v4
WHERE { { ?v6 foaf:name ?v8. }
UNION {?v9 foaf:name ?v4.}}
```

Q5: Union result sets for subjects and properties *name* and *comment* with values for the same constraints but the additional *series* property.

```
SELECT DISTINCT ?v3 ?v4 ?v5
WHERE { { ?v3 foaf:name ?v4; rdfs:comment ?v5. }
UNION
{ ?v3 dbp-prop:series ?v8. ?v3 foaf:name ?v4; rdfs:comment ?v5. } }
```

Q6: For the *Company108058098* object select a resource with a *homepage* and the *numEmployees* greater than or equal to 10.

```
SELECT DISTINCT ?v3 ?v5 ?v7
WHERE { ?v3 rdf:type dbp-yago:Company108058098.
?v3 dbp-prop:numEmployees ?v5.
?v3 foaf:homepage ?v7.
FILTER ( xsd:integer(?v5)>=10)}
```

Q7: Find required resource and node values of *comment* property. Optional values for the following properties can be present: *subject, industry, location, locationCountry, locationCity, manufacturer, products, model, point, type.*

```
SELECT distinct ?v0 ?v1 ?v2 ?v3 ?v5 ?v6 ?v7 ?v10
WHERE { ?v0 rdfs:comment ?v1.
OPTIONAL {?v0 skos:subject ?v6}
OPTIONAL {?v0 dbp-prop:industry ?v5}
OPTIONAL {?v0 dbp-prop:location ?v2}
OPTIONAL {?v0 dbp-prop:locationCountry ?v3}
OPTIONAL {?v0 dbp-prop:locationCity ?v9; dbp-prop:manufacturer ?v0}
OPTIONAL {?v0 dbp-prop:products ?v11; dbp-prop:model ?v0}
OPTIONAL {?v0 georss:point ?v10 }
OPTIONAL {?v0 rdf:type ?v7 }}
```

Q8: Join subjects and objects for the *populationUrban* and *type* properties with values of the *population* property.

```
SELECT ?v2 ?v4
WHERE { { ?v2 dbp-prop:population ?v4. }
UNION
{ ?v2 rdf:type ?v1. ?v2 dbp-prop:populationUrban ?v4. } }
```

Q9: For the resource with the *type* of Settlement find subjects with the *city* property. Union the results with the values of the properties *location, iata, iataLocationIdentifier*. Optional values for the properties *homepage* and *nativename* can be present.

```
SELECT *
WHERE { {?v2 a dbp-owl:Settlement. ?v6 dbp-owl:city ?v2.}
UNION {?v6 dbp-owl:location ?v2. ?v6 dbp-prop:iata ?v5.}
UNION {?v6 dbp-owl:iataLocationIdentifier ?v5.}
OPTIONAL { ?v6 foaf:homepage ?v7. }
OPTIONAL { ?v6 dbp-prop:nativename ?v8.}}
```

Q10: For the resource with a *type* of SoccerPlayer and the *page* property find the following property values: *position, clubs, capacity, birthPlace* and the optional *number* property. Return all the distinct values of the *page*.

```
SELECT DISTINCT ?v0
WHERE { ?v3 foaf:page ?v0.
?v3 rdf:type dbp-owl:SoccerPlayer.
?v3 dbp-prop:position ?v6.
?v3 dbp-prop:clubs ?v8.
?v8 dbp-owl:capacity ?v1.
?v3 dbp-owl:birthPlace ?v5.
?v5 ?v4 ?v2.
OPTIONAL { ?v3 dbp-owl:number ?v9 .}}
```

Q11:

```
SELECT distinct ?v3 ?v4 ?v2
WHERE { {?x dbp-prop:subsid ?v3
OPTIONAL {?v2 rdf:type dbp-prop:parent }
OPTIONAL {?x dbp-prop:divisions ?v4 }}
UNION {?v2 rdf:type dbp-prop:parent
```

```
OPTIONAL {?x dbp-prop:subsid ?v3}
OPTIONAL {?x dbp-prop:divisions ?v4 }}
UNION {?x dbp-prop:divisions ?v4
OPTIONAL {?x dbp-prop:subsid ?v3}
OPTIONAL {?v2 rdf:type dbp-prop:parent }} }
```

Q12: For a subject with type *Person* find all the *nationality* values and return its *label* property objects.

```
SELECT DISTINCT ?v5
WHERE { ?v2 rdf:type dbp-owl:Person .
?v2 dbp-owl:nationality ?v4 .
?v4 rdfs:label ?v5.}
```

Q13: What are the distinct resource values subject with the *type* property , combined together with the distinct objects of the *label* predicate for the same subject.

```
SELECT DISTINCT ?v2 ?v3
WHERE { ?v2 rdf:type ?x; rdfs:label ?v3 . }
```

Q14: Get an object set of the *comment* property for subjects, that might also have the properties *depiction* and *homepage*.

```
SELECT ?v0
WHERE {{ ?x rdfs:comment ?v0.}
UNION {?x foaf:depiction ?v1}
UNION {?x foaf:homepage ?v2 }}
```

Q15: Find a resource with the required properties *subject* and *name* and the optional *comment* property.

```
SELECT ?v6 ?v8 ?v10 ?v4
WHERE { ?v4 skos:subject ?v5 . ?v4 foaf:name ?v6 .
OPTIONAL { ?v4 rdfs:comment ?v8 .}
OPTIONAL { ?v4 rdfs:comment ?v10 . } }
```

Q16: For a set of all the distinct values of the available resources find the optional *label* property of the corresponding predicates and objects.

```
SELECT DISTINCT ?x ?v6 ?v7
WHERE { ?x ?v4 ?v5.
OPTIONAL {?v5 rdfs:label ?v6} .
```

```
OPTIONAL {?v4 rdfs:label ?v7 }}
```

Q17: What are the values of the *subject* property. In addition, the resource might have the predefined *subject*.

```
SELECT DISTINCT ?v2 ?x
WHERE{ {?v2 skos:subject ?x}
UNION
{?v2 skos:subject dbp-cat:Prefectures_in_France.}
UNION
{?v2 skos:subject dbp-cat:German_state_capitals .} }
```

Q18: For the same property value, first output the object and then the subject.

```
SELECT ?v3 ?v4 ?v5
WHERE { { ?x ?v3 ?v4. } UNION { ?v5 ?v3 ?y } }
```

Q19: Create a union of two equal sets of resources with the *label* property.

```
SELECT ?v1
WHERE { { ?v1 rdfs:label ?x } UNION { ?v1 rdfs:label ?y } }
```

Q20: Select all the values of the available query variables, where resources must have the required properties *PopulatedPlace, abstract, label* and the optional properties *populationTotal, thumbnail*. In addition, such resources can be placed as values of the *redirect* property nodes.

```
SELECT *
WHERE { {?v6 a dbp-owl:PopulatedPlace;
dbp-owl:abstract ?v1;
rdfs:label ?v2.}
UNION { ?v5 dbp-prop:redirect ?v6. }
OPTIONAL { ?v6 foaf:depiction ?v8 }
OPTIONAL { ?v6 foaf:homepage ?v10 }
OPTIONAL { ?v6 dbp-owl:populationTotal ?v12 }
OPTIONAL { ?v6 dbp-owl:thumbnail ?v14 } }
```

Q21: What are the values of the *redirect* property.

```
SELECT * WHERE { ?x dbp-prop:redirect ?v0 . }
```

Q22: What kind of objects have subjects with the properties *homepage* and *type*.

```
SELECT ?v2
WHERE { ?v3 foaf:homepage ?v2. ?v3 rdf:type ?x }
```

Q23: For a subject with type *Person* and property *page* find all objects.

```
SELECT ?v4
WHERE { ?v2 rdf:type dbp-owl:Person. ?v2 foaf:page ?v4 . }
```

Q24: Find the values of the following properties: *foundationPlace* and *type*. An object *Organisation* must exist.

```
SELECT *
WHERE { ?v1 a dbp-owl:Organisation .
?v2 dbp-owl:foundationPlace ?y .
?v4 rdf:type ?x . }
```

Q25: Find the values of the properties *name, pages, isbn, author* for the same resource.

```
SELECT ?v0 ?v1 ?v2 ?v3
WHERE { ?v6 dbp-prop:name ?v0 .
?v6 dbp-prop:pages ?v1 .
?v6 dbp-prop:isbn ?v2 .
?v6 dbp-prop:author ?v3 .}
```

Bibliography

[1] Alexander Schätzle, Martin Zablocki, Simon Skilevic, Georg Lausen. *S2RDF: RDF querying with SPARQL on Spark*. Proceedings of the VLDB Endowment, 9(12), 2015, DOI: 10.14778/2977797.2977806. See also, ArXiv:1512.07021v3 [cs.DB] 27 January 2016.

[2] Alexander Schätzle, Martin Zablocki, Thorsten Berberich, Georg Lausen. *S2X: Graph-Parallel Querying of RDF with GraphX*. Proceedings of the VLDB Endowment, 9579(6), 155-168, 2016, DOI: 10.1007/978-3-319-41576-5 12. ISBN 978-0-9815316-8-7. See also, ArXiv:1512.07021v3 [cs.DB] 27 January 2016.

[3] *An Introduction to RDF and the Jena RDF API.* https://jena.apache.org/tutorials/rdf_api.html

[4] Cataldo Musto, Giovanni Semeraro, Marco de Gemmis and Pasquale Lops. *Tuning Personalized PageRank for SemanticsAware Recommendations Based on Linked Open Data*. In: Eva Blomqvist et al. (eds) The Semantic Web: 14th International Conference, ESWC 2017, Slovenia, 2017, Proceedings, Part 1, 169-184. DOI: 10.1007/978-3-219-58068-5 11

[5] Charles F. Van Loan. *Lecture 1. Lecture 1. Introduction to Tensor Computations*. The Gene Golub SIAM Summer School 2010, Selva di Fasano, Brindisi, Italy. http://www.cs.cornell.edu/cv/SummerSchool/Introduction.pdf

[6] Charles F. Van Loan. *Lecture 5. The CP Representation and Tensor Rank*. The Gene Golub SIAM Summer School, 2010, Selva di Fasano, Brindisi, Italy. http://www.cs.cornell.edu/cv/SummerSchool/DecompositionsI.pdf

[7] *Cluster Mode Overview - Spark 1.2.0 Documentation - Cluster Manager Types*. https://spark.apache.org/docs/1.2.0/cluster-overview.html#cluster-manager-types

[8] Damien Graux, Louis Jachiet, Pierre Genevès, Nabil Layaïda. *SPARQLGX: Efficient Distributed Evaluation of SPARQL with Apache Spark*. The 15th Interna-

tional Semantic Web Conference, October 2016, Kobe, Japan. The 15th International Semantic Web Conference, 10, 80-87, 2016. DOI: 10.1007/978-3-319-46547-0 9. https://hal.inria.fr/hal-01344915

[9] Derrick Harris. *4 reasons why Spark could jolt Hadoop into hyperdrive*. https://gigaom.com/2014/06/28/4-reasons-why-spark-could-jolt-hadoop-into-hyperdrive

[10] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. *Scalable semantic web data management using vertical partitioning*. VLDB, Vienna, Austria, 2007, 411-422.

[11] Francois Picalausa, Stijn Vansummeren. What are real SPARQL queries like? Proceedings of the International Workshop on Semantic Web Information Management, Athens, Greece, June 12-16, 2011. doi:10.1145/1999299.1999306

[12] Gezim Sejdiu, Ivan Ermilov, Jens Lehmann and Mohamed Nadjib-Mami. *DistLODStats: Distributed Computation of RDF Dataset Statistics*. Proceedings of 17th International Semantic Web Conference, 2018.

[13] 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.

[14] *International Workshop on Semantic Big Data (SBD 2018)*, January 2019, https://www.ifis.uni-luebeck.de/ groppe/sbd/2018/aims-scope

[15] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van Kleef, Soren Auer, et al. *Dbpedia — a large-scale, multilingual knowledge base extracted from wikipedia*. Semantic Web, 6(2), 167–195, 2015.

[16] Jens Lehmann, Gezim Sejdiu, Lorenz Bühmann, Patrick Westphal, Claus Stadler, Ivan Ermilov, Simon Bin, Muhammad Saleem, Axel-Cyrille Ngonga Ngomo and Hajira Jabeen. *Distributed Semantic Analytics using the SANSA Stack*. Proceedings of 16th International Semantic Web Conference — Resources Track (ISWC 2017), 2017.

[17] J. Perez, M. Arenas, and C. Gutierrez. *Semantics and complexity of SPARQL*. Trans. Database Syst., 34(3), 2009, 45p.DOI = 10.1145/1567274.1567278 http://doi.acm.org/10.1145/1567274.1567278.

[18] Jiewen Huang, Daniel J. Abadi and Kun Ren. *Scalable SPARQL Querying of Large RDF Graphs*. PVLDB, Vol. 4, No 11, August 2011, pp. 1123-1134, http://www.vldb.org/pvldb/vol4/p1123-huang.pdf

[19] K. Wilkinson, C. Sayers, H. Kuno and D. Reynolds. *Efficient RDF Storage and Retrieval in Jena2*, SWDB, 2003, 131–150.

[20] Kelvin Guu, John Miller and Percy Liang. *Traversing Knowledge Graphs in Vector Space*. Conference on Empirical Methods on Natural Language Processing (EMNLP), 2015.

[21] Knud Möller and Michael Hausenblas and Richard Cyganiak and Siegfried Handschuh. Learning from Linked Open Data Usage: Patterns & Metrics, Web Science Conference 2010, 2010.

[22] Lee Feigenbaum. *SPARQL By Example: The Cheat Sheet*, VP Technology and Standards, Cambridge Semantics, December 2018, http://www.iro.umontreal.ca/lapalme/ift6281/sparql-1-1-cheat-sheet.pdf

[23] Lee Feigenbaum, Eric Prud'hommeaux. *SPARQL By Example: A Tutorial*, Cambridge Semantics, December 2018, https://www.cambridgesemantics.com/blog/semantic-university/learn-sparql/sparql-by-example/

[24] Makoto Nakatsuji, Hiroyuki Toda, Hiroshi Sawada, Jin Guang Zheng, James A. Hendler. *Semantic sensitive tensor factorization*. Artificial Intelligence, 230, 224–245, 2016.

[25] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide* (3rd ed.). Artima Inc., 837/859. ISBN 978-0-9815316-8-7.

[26] Matei Zaharia. *Spark: In-Memory Cluster Computing for Iterative and Interactive Applications*. https://www.youtube.com/watch?v=qLvLg-sqxKc

[27] Matei Zaharia, Bill Chambers. *Spark: The Definitive Guide*. O'Reilly Media, February 2017. ISBN 978-1-4919-1221-8.

[28] Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. Matrix "Bit"loaded: A Scalable Lightweight Join Query Processor for RDF Data. In Proc. of 19th International World Wide Web Conference, 2010, Raleigh, North Carolina, USA, pp. 41-50. DOI: 10.1145/1772690.1772696

[29] Michael Schmidt, Michael Meier, Georg Lausen. Foundations of SPARQL query optimization. In Proceedings of Database Theory ICDT 2010, 13th International Conference, Lausanne, Switzerland, March 23-25, 2010, pp. 4–33. https://doi.org/10.1145/1804669.1804675.

[30] N. Bikakis, C. Tsinaraki, N. Gioldasis, I. Stavrakantonakis, S. Christodoulakis. *The XML and Semantic Web Worlds: Technologies, Interoperability and Integration. A survey of the State of the Art*. In Semantic Hyper/Multi-media Adaptation: Schemes and Applications, Springer, 2013.

[31] Pascal Hitzler, Markus Krötzsch and Sebastian Rudolph. *Foundations of Semantic Web Technologies*. Chapman and Hall/CRC, Taylor and Francis Group, Boca Raton, London, New York, 2010. ISBN 978-1-4200-9050-5.

[32] Patrick Westphal, Javier Fernández, Sabrina Kirrane and Jens Lehmann. *SPIRIT: A Semantic Transparency and Compliance Stack*. 14th International Conference on Semantic Systems, Poster and Demos, 2018.

[33] *Publications Of The W3C Semantic Web Activity*, December 2018, https://www.w3.org/2001/sw/Specs

[34] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, Ion Stoica. *Shark: SQL and Rich Analytics at Scale*. https://amplab.cs.berkeley.edu/wp-content/uploads/2013/02/shark_sigmod2013.pdf

[35] *Resource Description Framework (RDF)*, December 2018, https://www.w3.org/RDF/

[36] Roberto De Virgilio. *Distributed in-memory SPARQL Processing via DOF Analysis*. In Proc. 20th International Conference on Extending Database Technology (EDBT), March 21-24, 2017. — Venice, Italy: ISBN 978-3-89318-073-8.

[37] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, Martin Theobald. TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing. SIGMOD '14: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, June 22–27, 2014, Snowbird, UT, USA. http://dx.doi.org/10.1145/2588555.2610511.

[38] Sandra Álvarez-Garcá, Nieves Brisaboa, Javier D. Fernández, Miguel A. Martánez-Prieto and Gonzalo Navarro. *Compressed Vertical Partitioning for Full-In-Memory*

RDF Management. Knowledge and Information Systems, No 1, 2014, DOI: 10.1007/s10115-014-0770-y. See also, ArXiv:1310.4954v2 [cs.DB] 21 October 2013.

[39] *Semantic Analytics Stack (SANSA). Open Source Algorithms for Distributed Data Processing for Large-scale RDF Knowledge Graphs*. http://sansa-stack.net/libraries/. 21.11.2018.

[40] *Semantic Web Development Tools*. December 2018, https://www.w3.org/2001/sw/wiki/Tools

[41] Tamara G. Kolda and Brett W. Bader. *Tensor Decompositions and Applications*, SIAM Review, 51(3), 455-500, 2009. http://doi.org/10.1137/07070111X

[42] Tamara G. Kolda and Brett W. Bader. *SANDIA Report*, SAND2007-6702, 2007, https://public.ca.sandia.gov/ tgkolda/pubs/bibtgkfiles/SAND2007-6702.pdf

[43] Tim Berners-Lee, Mark Fischetti. *Weaving the Web*. HarperCollins Publishers L.L.C.SanFrancisco, 1999. ISBN 978-0-06-251587-2.

[44] Tim Berners-Lee, James Hendler and O. Lassila. *The Semantic Web. A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities*. Scientific American, 2841(5):34, May 2001, pp. 29-37. doi:10.1038/scientificamerican0501-34

[45] Thomas Franz, Antje Schultz, Sergej Sizov and Steffen Staab. *TripleRank: Ranking SemanticWeb Data By Tensor Decomposition*. In: Bernstein A. et al. (eds) The Semantic Web – ISWC 2009. ISWC 2009. Lecture Notes in Computer Science, 5823. Springer, Berlin, Heidelberg. DOI https://doi.org/10.1007/978-3-642-04930-9 14.

[46] Thomas Neumann, Gerhard Weikum. In Scalable join processing on very large RDF graphs. In Proceedings of SIGMOD International Conference on Management of data, pp. 627-640, Providence, Rhode Island, USA, June 29-July 02, 2009. doi: 10.1145/1559845.1559911

[47] Tomás Kliegr. *Linked hypernyms: Enriching dbpedia with targeted hypernym discovery*. Web Semantics: Science, Services and Agents on the World Wide Web, 31, 59–69, 2015.

[48] Trupti Padiya and Minal Bhise. *DWAHP: Workload Aware Hybrid Partitioning and Distribution of RDF Data*. In Proceedings of IDEAS'17, Bristol, United Kingdom, July 2017, 235-241, DOI: 10.1145/3105831.3105864

[49] *W3C Semantic Web Activity.* December 2018, https://www.w3.org/2001/sw/

[50] *Spark Programming Guide.* https://spark.apache.org/docs/-2.1.1/programming-guide.html

[51] *W3C. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation 10 February 2004*, December 2018, https://www.w3.org/TR/rdf-concepts/

[52] *W3C. RDF Semantics. W3C Recommendation 10 February 2004*, December 2018, https://www.w3.org/TR/rdf-mt/

[53] *W3C. SPARQL Query Language for RDF. W3C Recommendation 15 January 2008*, December 2018, /https://www.w3.org/TR/rdf-sparql-query/

[54] Wescott Bob. *The Every Computer Performance Book, Chapter 3: Useful laws.* CreateSpace, 2013.— 222 p. ISBN 1482657759.