DISE: A Distributed in-Memory SPARQL Processing Engine over Tensor Data

Hajira Jabeen
jabeen@cs.uni-bonn.de
University of Bonn
Bonn, Germany

Eskender Haziiev
haziiev@uni-bonn.de
University of Bonn
Bonn, Germany

Gezim Sejdiu
sejdiu@cs.uni-bonn.de
University of Bonn
Bonn, Germany

Jens Lehmann
jens.lehmannn@cs.uni-bonn.de
University of Bonn
Bonn, Germany

Type	Software Framework
Website	http://sansa-stack.net/dise/
URL	https://doi.org/10.6084/m9.figshare.10055387

Abstract—**SPARQL is a W3C standard for querying the data stored as Resource Description Framework (RDF). The SPARQL queries are represented using triple-patterns, and the querying process searches for these patterns in given RDF. Most of the existing SPARQL evaluators provide centralized, DBMS inspired solutions consuming high resources and offering limited flexibility. To deal with the increasing size of RDF data, it is important to develop scalable and efficient solutions for distributed SPARQL query evaluation. In this paper, we present DISE – an open-source implementation of distributed in-memory SPARQL engine that can scale out to a cluster of machines. DISE represents the RDF graph as a three-way distributed tensor for querying large-scale RDF datasets. This distributed tensor representation offers opportunities for novel distributed applications. DISE translates the SPARQL queries into Spark-tensor operations by exploiting the information about the query complexity and creating a dynamic execution plan. We have tested the scalability and efficiency of DISE on different datasets. The results for this new representation based querying have been found scalable, efficient and comparable to a similar approach.**

I. Introduction

Knowledge Graphs (KG)s have gained lots of traction in recent years for their ability to ingest heterogeneous data in a machine-readable form. Semantic Web presents a well-established format to represent KGs using Resource Description Framework (RDF)[1]. RDF represents a KG as a set of triples, where each triple represents two vertices and an edge of the graph. The W3C standard querying language for RDF data is SPARQL Protocol and RDF Query Language (SPARQL)[2]. Owing to wider acceptance of RDF among multiple fields like bioinformatics, life sciences, business intelligence, social networks, and many others, we count more than 10,000 RDF datasets available online[3]. The size and number of these datasets are continuously increasing and as a result, efficient processing and analysis of these large RDF datasets have become necessary. Scalable RDF querying requires distributed storage and efficient searching strategies for query-processing engines. Indeed, there is a strong need to develop fundamentally new approaches that leverage the existing (big data) processing engines (e.g. Apache Spark[4]) and develop approaches that are both efficient and scalable.

In this paper, we present DISE, that not only exploits in-memory distributed processing to achieve scalability, but also utilizes a distinct RDF representation; as tensors. DISE can deal with the volatile data and it does not use statistics of data to achieve fast performance, instead, it executes SPARQL queries as tensor operations by using Apache Spark as the scalable RDF processing engine. We have integrated DISE as a new SPARQL engine in SANSA [1], the scalable semantic analytics stack built to allow scalable RDF processing using Apache Spark.

In summary, the major contributions of this paper are:

- A distributed tensor representation of RDF data.
- An efficient SPARQL to the tensor-operation translator in Spark-Scala compliant code.
- Query decomposition for dynamic query execution planning
- Empirical evaluation for scalability.
- Comparative evaluation with the semantic-query-engine in SANSA.
- Integration into the SANSA framework.

The remaining of the paper is organized as follows: Section II discusses the supplementary information for this work. Section III provides an overview of the related work. Section IV describes the architecture of DISE. Section V is devoted to the implementation details of DISE, experimental setup, the datasets, and SPARQL queries used for evaluation. Section VI summarizes the main results of this work and discusses the prospects for further development of DISE.

[1] https://www.w3.org/TR/rdf11-concepts/
[2] https://www.w3.org/TR/rdf-sparql-query/
[3] http://lodstats.aksw.org/
[4] http://spark.apache.org/

II. Preliminaries

A. Resource Description Framework and Tensors

1) Resource Description Framework (RDF): RDF is defined as a formal language for describing structured information [2]. The RDF statements are represented in the form of triples (see, [3], [4], [5]) as $\langle s, p, o\rangle$, where s, p and o are called subject, predicate and object respectively. These triples in RDF are built from three disjoint sets $\mathcal{I}$, $\mathcal{B}$, and $\mathcal{L}$ containing IRIs, blank nodes and literals. For validity it is required that $s \in \mathcal{I} \cup \mathcal{B}$, $p \in \mathcal{I}$, and $o \in \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$. It is important to note that sets $\mathcal{I}$, $\mathcal{B}$ and $\mathcal{L}$ are countable and finite. This property enables indexing or ordering of elements in these sets.

2) Tensor representation of RDF: RDF triples $\langle s, p, o\rangle$, can be modeled [6] as a 3D tensor $\mathcal{R}$, where each of its slices represents an adjacency matrix of "subject-and-objects" $s \cup o$, and predicates p (property/edge). Bader et al. [7] define tensors as multidimensional arrays that can be viewed as an ordered collection of slices or columns. Where a slice is a two-dimensional tensor, a column is a one-dimensional tensor. Furthermore, an N-dimensional tensor can be represented as a product of one-dimensional tensors.

Definition II.1 shows that RDF tensor $\mathcal{R}$ is a 3D tensor.

Definition II.1 (RDF Tensor)**.** *Let G be RDF graph. The RDF tensor* $\mathcal{R} := \mathcal{R}(G)$ *on G is a matrix such that* $\mathcal{R} = r_{i,j,k} := 1$, *if* $\langle\ \mathbf{S}^{-1}(i),\ \mathbf{P}^{-1}(j),\ \mathbf{O}^{-1}(k)\rangle\ \in G$ *and* $\mathcal{R} = r_{i,j,k} := 0$ *otherwise. Here* $\mathbf{S} : \mathcal{S} \longrightarrow \mathbf{N}$, $\mathbf{P} : \mathcal{P} \longrightarrow \mathbf{N}$, $\mathbf{O} : \mathcal{O} \longrightarrow \mathbf{N}$ *are the indexing functions for distinct subjects, predicates and objects.*

B. SPARQL and Degree of Freedom

1) Structure of a SPARQL query: According to W3C recommendation [8], SPARQL is a technology developed for extraction and modification of data stored in RDF graphs. It allows to perform complex operations on data such as information retrieval, searching, filtering based on given criteria, or transformation etc. SPARQL query contains a Basic Graph Pattern (BGP) with a structure similar to RDF triples, each triple in BGP is the query triple pattern (TP). However, any subject, predicate or object in the query TP can be a variable, represented with a preceding question mark. The variables in TPs of a BGP can be overlapping. There are different types of SPARQL queries, but we focus on the SELECT query that returns variables and their bindings directly. A SPARQL query includes five sections:

1) Optional headers for a human-readable query e.g. `PREFIX` that describes prefix declarations for abbreviated URIs, it reduces the URIs used in the query.
2) Standard allowed query forms e.g. `SELECT` that composes the resulting clause.
3) Optional clauses that specify the dataset e.g. `FROM` that determines the source i.e. defines the *RDF dataset* being queried.
4) `WHERE` clause that specifies the basic graph pattern (BGP) to match and defines the constraints of the query. The basic BGPs combine several TPs using conjunction.
5) Optional solution modifiers operating over the result set e.g. `ORDER BY`, `LIMIT` and `OFFSET` can be used as solution output modifiers.

2) Degree of Freedom of a SPARQL Query: The concept of the degree of freedom (DOF) of a query Q is introduced in [9], where DOF is defined as "a measure of a triple pattern's explicit constraints".

Definition II.2 (Degree of Freedom)**.** *Let v and k be the numbers of variables and constants in a triple t, respectively. The degree of freedom of the triple t is the function defined as* $dof(t) := v - k$, $dof\ \in\ \{+3, +1, -1, -3\}$.

From Definition II.2 we have: TP with no constraints has the highest DOF , equal to $+3$, and is associated to variables only. TP having two variables and one constant has $dof(t) = +1$; $dof(t) = -1$ means that TP is bounded to one constant and contains two unbounded variables. The lowest DOF corresponds to TP, bounded to three constants i.e. $dof(t) = -3$.

C. Apache Spark

Apache Spark [10] is an open source platform for distributed data processing. Spark is a general-purpose processing engine that is suitable for use in a wide range of practical applications for big data. Spark uses in-memory distributed processing to achieve efficient performance The Apache Spark API [11] is centred around the distributed 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 iterative algorithms that access the data iteratively, and for interactive intelligence analysis, i.e. repeated data requests in the database. On top of the core data processing engine employing RDDs, Spark provides specialised libraries for SQL, machine learning, graph computation, stream processing and APIs for Python, R, Java and Scala (the native Spark language [12]).

D. SANSA

Scalable Semantic Analytics Stack (SANSA)[5] [1] is a framework that offers a set of libraries for distributed processing of large-scale RDF data. SANSA provides libraries for the layers corresponding to Semantic Web Layer cake: RDF-representation layer, SPARQL, Inference and Analytics layer.

III. Related work

Recent statistics indicate that currently, freely available RDF data contains approximately 150 billion triples in about 3,000 datasets, many of which are accessible via SPARQL query servers called SPARQL endpoints[6]. Many researchers have focused on issues related to RDF data representation in query processing, methods for reducing computational costs in terms

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

[6] https://www.ifis.uni-luebeck.de/~groppe/sbd/2018/aims-scope

of space and time, as well as efficient methods to achieve accurate results for querying large scale RDF data.

Representation for large distributed in-memory RDF data includes various types of data partitioning. Horizontal partitioning of RDF data is used in TRiAD [13] engine for distributed SPARQL processing. Another splitting approach for RDF data is vertical partitioning (VP) used in [14], [15], [16], [17], [9]. In [16] VP is used as so-called property tables. Each table stores tuples $\langle subject, object\rangle$ associated with a given predicate sorted by the subject, so individual subjects can be located quickly.

S2RDF [17], a Hadoop-based SPARQL query processor for large-scale distributed RDF data is implemented for Spark and includes a new relational partitioning schema for RDF data called ExtVP. To execute SPARQL queries through ExtVP, S2RDF uses VP Extension as the basic data layout for RDF. The results for a query TP with bound predicate can be retrieved by accessing the corresponding VP table which leads to a large reduction in the input size. However, it supports only conjunction queries.

SPARQLGX [15] is based on Apache Spark and translates conjunctive queries into Spark executable code. The authors focus on the problem of evaluating the Basic Graph Pattern fragment over an RDF dataset. Such fragments are composed only of conjunctions of TP s, where each TP expresses conditions that must be matched by RDF triple for selection. The authors indicate that there are often relatively few distinct predicates compared to the number of distinct subjects or objects. To process a conjunction of query TP s, the TP s are joined using their common variables as a key.

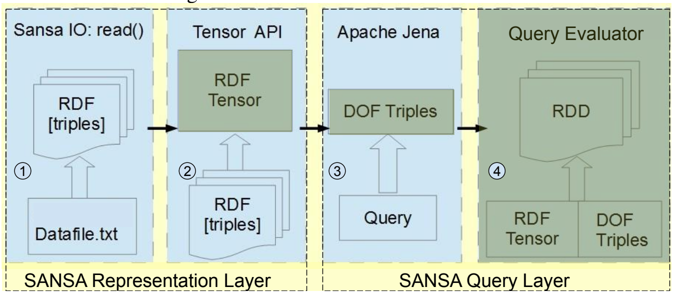
VP is used in [14] to create disjoint subsets of pairs (subject, object), per predicate. These subsets are represented as binary matrices of subjects$\times$objects in which 1 means that the corresponding triple exists in the dataset. This model results in very sparse matrices, which are compressed using k^2-trees and two compact indexes. The indexes list the predicates related to different subject and object.The experiments show that this method can overcome the shortcomings of traditional VP for join resolution.

In [9] the authors propose to create an RDF tensor and use DOF of a TP to improve the efficiency of distributed RDF data processing. Using the RDF tensor, a significant reduction in memory usage is achieved, while DOF helps to select TP with the highest probability of decreasing the search space.

While being efficient, [14] and [9], are implemented using sequential processing in C on a single machine. Therefore, these approaches are not suitable for very large data and do not offer scalable performance.

An analysis of the complexity of the SPARQL queries is presented in [18], [19]. It is shown that operators AND and UNION are commutative and associative, but AND, OPTIONAL and FILTER are distributive with respect to UNION operator. Query evaluation can be solved in polynomial time for Basic Graph Pattern(BGP) expressions constructed by using AND and FILTER operators only; the evaluation is NP-complete for BGPs built with AND, FILTER and UNION operators; for BGPs that include the OPT operator, the evaluation problem is PSPACE-complete.

Fig. 1. DISE Architecture Overview.



M. Schmidt et al. in [20] discussed the SELECT, AND, FILTER, OPTIONAL and UNION operators, that can be used to construct expressive queries. The authors map the OPTIONAL operator to a left outer join; AND, UNION, FILTER are mapped to join, algebraic union, and selection respectively. The SELECT is mapped as a projection operation. The authors conclude that the 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 [20].

The problems of accuracy, correctness and effectiveness of the developed approaches are discussed in [21], [22], [17], [9].

In summary, most of the literature covered above is either not scalable, or the source has not been made public to help the community. DISE addresses this gap and presents a scalable, easy to use, and an open source research contribution.

IV. DISE

A. Architecture

The architecture of our "Distributed in-Memory SPARQL Processing Engine" DISE is presented in Figure 1. The figure depicts the integration of DISE within SANSA. The DISE represented by darker (green) shade. DISE transforms the input RDF into a tensor representation that has been integrated into the SANSA representation layer, whereas the tensor-based querying is integrated into the SANSA query layer. The first block (1) to the left makes use of the SANSA RDF reader. It reads the input text file and converts it into an RDD of triples for further data processing. The second block (2) creates a tensor from the RDD of triples from the previous block. The next block (3) receives the set of query strings as input and converts them into Apache Jena[7] Query objects. It also recursively scans the query TP s to calculate the DOF of each TP for query planning. Builder in the next block (4) uses the results of two previous steps (2 and 3) to translate the query into tensor operations and to calculate the query result.

DISE processes SPARQL queries for large RDF datasets in a distributed manner. It computes DOF for query TP to

[7] https://jena.apache.org/

estimate the query complexity. DISE does not require any pre-computation of *prior knowledge* or *initial data statistics* for the efficient querying. Since DOF measures the constraints of a query TP, we start with the query TP that has the lowest DOF value, and iteratively sexecute the results and select the next TP with the highest probability of decreasing the search space.

Our primary model based on the *RDF tensor* avoids storing RDF data in triple form for better performance. The main idea is to transform an original dataset into RDF tensor, split the tensor, and process the query in parallel, starting with a query TP with the lowest DOF . We have developed two methods being used in our framework for parallel query processing.

B. Working

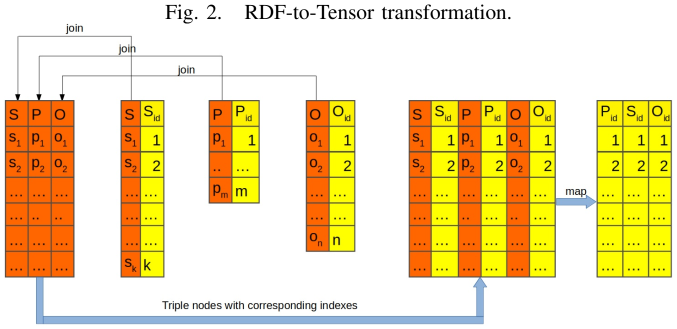
Algorithm 1 describes the distributed SPARQL query processing in DISE. It takes the RDF data and a SPARQL query string as an input. The algorithm creates an RDF tensor, computes DOF for each query TP and calculates resulting set of the query variables values concerning the DOF of the query TP, which is detailed in Algorithm 2. The output of the algorithm 1 is a key value pair $resultingMap$ representing the values of the variables from the input SPARQL query.

Algorithm 1: Distributed SPARQL query processing via DOF

```
input  : rdf : RDF: RDF dataset, query: SPARQL query
output: resultingMap : [Var, RDD[Node]]: Map with RDD of nodes associated to the query result variables
1  $RDD : dataset \leftarrow rdf.toRDD < Triple > ()$
2  $subjects, predicates, objects \leftarrow zipWithIndex(dataset)$
3  $tensor \leftarrow dataset.join(objects).join(predicates).join(subjects)$
4  $tensor \leftarrow tensor.map(predicates, subjects, objects)$
5  $dofs \leftarrow RecursiveElementVisitor.visit(query)$
6  $resultingMap \leftarrow Map[Var, RDD[Node]]()$
7  while $dofs \neq \emptyset$ do
8      $dof, triple \leftarrow dofs.dequeue()$
9      $tempMap \leftarrow tensor.process(triple, ResultingMap)$
10     $dofs.recalculate()$
11     $resultingMap \leftarrow bindResults(tempMap)$
12 return $resultingMap$
```

Line 1 of the Algorithm 1 loads RDF data from the physical storage and returns an RDD of triples using SANSA API. Line 2 identifies all unique items of the dataset, i.e. subjects, predicates and objects, and store them into the RDD of indexes. Line 3 creates an RDD of indexes at their respective positions as in triples RDD. Line 4 represents the final tensor with indexes at the representative positions. This process from line 1 – line 4 is illustrated in Figure 2.

Fig. 2. RDF-to-Tensor transformation.



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, also, [15], [23]). Using the definition of RDF triple (see [9]) and Definition II.1, we get the inequality:

$$\overline{p} \ll \overline{s} \ll \overline{o}, \ \overline{p} := \max(\mathbf{P}), \ \ \overline{s} := \max(\mathbf{S}), \ \ \overline{o} := \max(\mathbf{O}). \quad (1)$$

Thus, in order to optimize the performance, we modify the RDF tensor at Line 4 by placing the predicate column in the first position.

$RecursiveElementVisitor$ (Line 5) reads the input query, traverses each TP and calculates the initial values of the DOF for each TP. As a result, it returns a priority queue of pairs of the query TPs along with their corresponding DOF values (i.e. high priority corresponds to the low DOF). Lines 7 – 11 iterate over these pairs to obtain intermediate computations in the form of the map object for TP (Line 9). Algorithm 2 (described later) covers the detail. Each iteration of the $while$ loop calculates a map with an RDD of values associated to the current query variables. We must *recompute* the DOF of the remaining constraints, since a variable can be promoted to the role of the constant (Line 10).

Query TP processing is performed by $Tensor.process$ method (Line 9), as shown in Algorithm 2, it takes as input a query triple pattern *tp* and a map M, in which the keys are the variables from the query TPs. At the beginning we associate an empty RDD to each key, i.e. M is empty. As we traverse the query TP s recursively, M will contain results from the previous computations.

The Algorithm 2 processes the tensor using the information from individual triple pattern TP of the query. It takes as an input a key value Map, with variables as keys and their bindings as values of the Map. This map is updated with respect to the triple pattern under processing. Line 2 processes the predicate, subject and object node of the TP iteratively. Line 3 searches the variable, in the existing Map, if the value already exists from previous computations, the value is added to the resultSet. if not, the value of the particular node is extracted from the original tensor and added to the resultSet. If the node is not a variable, its value is extracted from the tensor and stored in resultSet. Same process is repeated with the three nodes of the triple pattern and the resultSet is passed

Algorithm 2: Tensor processing

```
input : $tp : TriplePattern$,
        $m : Map[Var, Values[Node]]$: Map with the
        results of previous computations
output: $result : Map[Var, Values[Node]]$: Map with
        values associated to each variable
1  $index \leftarrow 0$
2  while $tp = getNode(TP[index])$ do
3      if $tp.isVariable$ then
4          $isBounded \leftarrow m.hasKey(tp)$
5          if $isBounded$ then
6              $resultSet \leftarrow tensor.nodes.join(M.get(tp))$
7          else
8              $resultSet \leftarrow tensor.nodes(index)$
9      else
10         $resultSet \leftarrow tensor.nodes.filter(current == triplePart)$
11     $index++$
12 $M.update(resultSet)$
13 return $m$
```

to update the Map value with corresponding variables and their values in Line 12.

bindResults at Line 11 in Algorithm 1 merges the map returned by the Algorithm 2 for one TP with the resulting map, containing the query pattern variables for all query TPs. Finally, the $resultingMap$ returns the variable results for the SPARQL query.

C. DISE as a resource

The RDF-to-Tensor scalable representation offered in DISE has its use in a multitude of machine learning applications e.g. embedding models for link prediction [24] like TransE, TransH, or RESCAL, clustering algorithms, or anomaly detection. DISE can play a vital role to improve the state of the art in scalable machine learning models exploiting the RDF-to-Tensor representation. In order to target reproducability, the DISE-code is open source [8] and the results have been generated on generic and openly available data. DISE demonstrates a SPARQL query engine as one of the practical use of this tensor representation. DISE is easy to use and its documentation[9] is the part of SANSA documentation.

D. Complexity Evaluation

To evaluate the effectiveness of DISE, we have applied two methods: theoretical evaluation for classifying the method as being *P*-hard or *NP*-hard and empirical evaluation of experimental results for execution time $T(M)$ on a data-set having size M.

Below we present an estimate in terms of a size of RDF dataset. We show that the theoretical complexity of DISE is polynomial; obtained right-hand estimate shows the asymptotic behavior over time, which depends on the size of data. Table I represents the notations used in this and the following sections.

TABLE I
USED NOTATIONS

Notation	Description
RV	Number of Result Variables
NV	Number of Variables
NT	Number of Triples
CV	Number of Common Variables
U	Union
C	Conjunction
O	Optional

Lemma IV.1. *Let N be a number of all unique objects of the RDF tensor. Then in proposed approach the calculation of resulting RDD from RDF tensor is a P-solvable problem:*

$$T_{RDD} = \mathcal{O}(N^2) \ll T_{RDF} = \mathcal{O}(N^3). \qquad (2)$$

Proof: From the triple definition it follows that in Cartesian coordinate system RDF tensor can be represented as 3D cube with the dimensions along the axes equal to $\overline{s}$, $\overline{p}$, $\overline{o}$ (see (1)). So, the maximum size of the dataset does not exceed $\overline{s} \times \overline{p} \times \overline{o}$. From (1) it follows that $\overline{s} \ll \overline{o}$ and $\overline{p} =: c \ll \overline{o} := N \in \mathbf{N}$, $c = \text{const} \in \mathbf{N}$. Thus, RDD size does not exceed cN^2, and any greedy algorithm handles at most cN^2 points of 3D cube. Therefore, the problem is P-solvable and $T_{RDD} = \mathcal{O}(N^2)$. The right side of (2) is trivial. ■

Since (2) is satisfied for any greedy search algorithm (i.e. in the worst case), it holds for all efficient ones. Calculated once RDF tensor can be applied for querying any number of times and may significantly reduce the overall execution time.

Lemma IV.1 holds for a simple `SELECT` query with $NV \ll 3$. In *conjunctive* queries $NV > 3$ and $NT > 1$, so we consider this case. Let several simple sub-queries (i.e. the query TP s) be conjugated and, the search is performed over m query TP s, k variables of all query TP s, and n CV of all query TP s, $k \geq 1$, $m \geq 1$, $n \geq 0$. Note, if all the query TP variables are different, then $n = 0$; if $m = 1$, then the query contains only one query TP (i.e. the simplest case).

Theorem IV.1. *Let conjunctive query $Q_{k,m,n}$ has k variables, m triples and n common variables, $k \geq 1$, $m \geq 1$, $n \geq 0$. Then*

i) *Complexity of $Q_{k,m,n}$ is defined by the values k, m, n.*
ii) *Response time $T(Q_{k,m,n}) = f(m)$, where $f(m)$ is an increasing function on a variable m for any fixed k, n.*
iii) *$T(Q_{k,m,n}) = g(k)$, where $g(k)$ is an increasing function on a variable k for any fixed m and n.*
iv) *$T(Q_{k,m,n}) = h(n)$, where $h(n)$ is an increasing function on a variable n for any fixed m and k.*

Proof: We prove (ii) by induction. For $m = 1$ it follows from the fact that a simple query TP has at most three

[8] https://github.com/SANSA-Stack/SANSA-Query/tree/develop/sansa-query-spark/src/main/scala/net/sansa_stack/query/spark/dof

[9] http://sansa-stack.net/dise/

variables, and its response time is bounded from above by N^k for k variables, $k \ll 3$, and $n \equiv 0$. Suppose that the formula in (ii) holds for any $m_1 > 1$ and let show that it will be satisfied for $m = m_1 + 1$. The query used conjunction of $m = m_1 + 1$ query TP s with k variables and n CV. The response time of conjunction of m_1 simple queries with k_1 variables and n_1 CV is $f(m_1)$. Hence, we have that conjunction of $m = m_1 + 1$ simple queries has no more than $k \ll k_1 + 3$ variables (and no more than $n \ll n_1 + 3$ CV). 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 an increasing function. Note that the increasing of the number of CV can only reduce the power of N^3, since an increase in the number of CV may include those, already presented in m_1, but in any case such an increase cannot be greater than 3. (iii) is proved in the similar way. (i) follows from (ii) and (iii). (iv) is proved in the same way as (ii), with the only difference that instead of the function $h(n_1)$ we have $n > n_1$ and $\Delta := n - n_1 > 0$; from this we get $N^{-\Delta} h(n_1) = \dfrac{h(n_1)}{N^{\Delta}} < h(n_1)$. ■

V. Experiment and Evaluation

A. Datasets, Queries and Cluster Environment

We have evaluated the performance of DISE over several DBpedia [25] datasets with varying sizes detailed in Table II.

TABLE II
Dataset characteristics (NT format)

Dataset	Number of Triples	Size (in Gb)
D1[4]	17M	2.7
D2[5]	41M	6.6
D3[6]	79M	20.8

The evaluation was done on a cluster with 2 executors having a total of 128 cores and 150 Gb memory. Each server has Xeon Intel CPUs with 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 version 2.11.11. For testing the query performance, we have used the set of 25 queries [9] containing a combination of operators Distinct, Optional, Union, Filter, Conjunction. We will mostly discuss the results of nine queries selected in order of varying complexity. Table III shows an overview of the complexity of selected queries w.r.t. different parameters shown in Table I.

B. Performance evaluation of DISE

The results reported in this section are average response time (in ms) of ten cold runs for each query (as in [22]). As anticipated, we have observed that the run-time of the queries depends on their complexity, which is a combination of the NV, TP and CV. Figure 3 shows that the runtime increases

[4]http://downloads.dbpedia.org/3.9/simple/
[5]http://downloads.dbpedia.org/3.9/ro/
[6]http://downloads.dbpedia.org/3.9/uk/

TABLE III
Query Complexity

Query	RV	NV	TP	CV	U	C	O
Q1	1	2	1	0			
Q2	3	4	4	1		+	+
Q3	3	4	5	3	+	+	
Q4	8	10	11	1		+	+
Q5	3	5	3	2		+	+
Q6	1	3	2	1	+		
Q7	8	8	8	1	+	+	+
Q8	5	5	4	2		+	
Q9	3	5	2	1	+		

as a function of the number of variables in the query. Figure 4 demonstrates that the runtime increases with the increase in number of triple patterns in the queries, and Figure 5 indicates that the execution time increases with the increase in number of common variables (resulting in joins) in the queries.

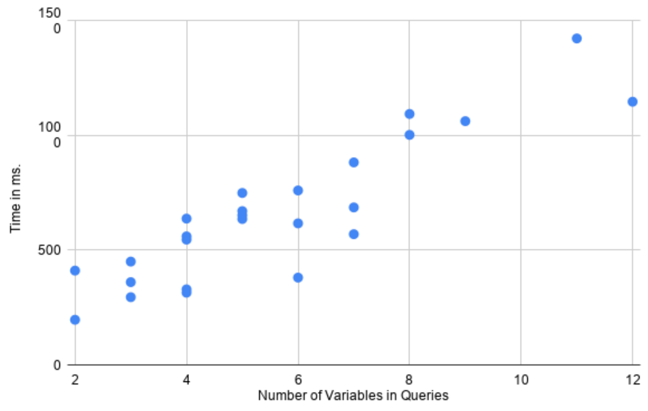


Fig. 3. Effect of number of variables NV on query execution time

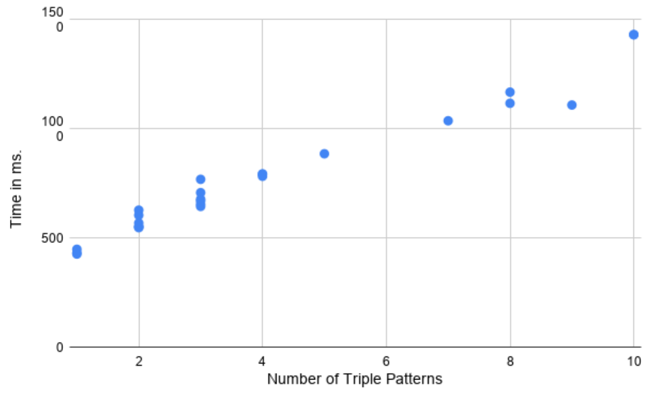


Fig. 4. Effect of Number of Triple Patterns TP on query execution time

Table IV presents the runtime of the selected queries. Looking at the complexity from Table III we can see that Q8 having the same number of query TP s as Q2 (although one more variable) has a shorter runtime than Q2 due to the fact that Q2 has an `OPTIONAL`, and it needs to perform joins between large amount of triples that influences the performance. The

TABLE IV
QUERY RUNTIME OF DISE (IN MS)

Query	(D1)	(D2)	(D3)
Q1	242	243	239
Q2	580	573	630
Q3	722	804	929
Q4	1,482	1,467	1,472
Q5	503	503	509
Q6	360	379	369
Q7	987	1,106	1,129
Q8	581	567	575
Q9	425	426	424

comparison of runtime of Q5 and Q8 (Q5 has one more query TP , and same NV and CV) confirms (ii) from Theorem IV.1 and shows that the query runtime is an increasing function on the number of the query TP s for any fixed values of NV and CV. The analysis of runtime of Q6 and Q9, with Q9 having two more NV, same TP and CV, shows that the runtime increases with the number of variables for any fixed values of TP and CV (see (iv) in Theorem IV.1). This can be observed that DISE supports a range of complex and simple queries. The execution time shown in the evaluations increases with the query complexity. However, the simpler queries are more common in practice [26], resulting in relatively less execution time for real life usecases.

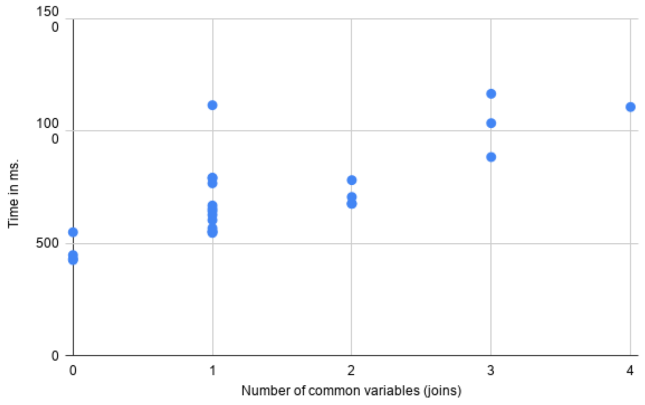


Fig. 5. Effect of common variables CV on query execution time

C. Scalability of DISE

In order to test the scalability of DISE, we have evaluated the runtime performance of 25 queries on datasets of varying sizes. The results reveal that the DISE has the ability to scale up with the increasing size of data. Figure 6 shows that the runtime of queries increased 1.2 times, from 0.2ms to 1.5s, when datasize increased by 3.2 times; thus, the runtime increased proportionally to the dataset size, that has increased from 2.7 Gb to 20.8 Gb, i.e. from 17M triples to 79M triples, see Table IV. This is an anticipated behaviour, as the larger data size requires larger resources in turn, effecting the query execution time. With this observation we demonstrate that DISE can scale-up to larger data.

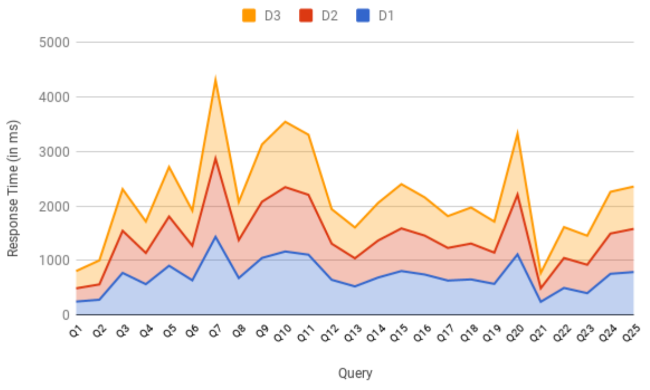


Fig. 6. DISE Scalability Diagram.

TABLE V
QUERY RUNTIME OF SANSA (IN MS)

Query	(D1)	(D2)	(D3)
Q1	112	192	194
Q2	679	732	702
Q6	285	264	371
Q8	419	469	444
Q9	313	324	360

D. Comparison with SANSA query engine

In order to compare SANSA and DISE, the queries are executed on datasets D1, D2, D3 using SANSA semantic-based [27] query engine. The semantic-based query engine in SANSA is a scalable approach for efficient evaluation of SPARQL queries over distributed RDF datasets. It uses a semantic-based partitioning strategy by grouping into a subject-based facts (e.g. all entities which are associated with a unique subject) as the data distribution and converts SPARQL to Spark executable code. It is important to note the fundamental difference in ”Semantic Partition based query engine in SANSA” and DISE. Both approaches follow a completely different representation and are therefore not directly comparable. However, we present this comparison only to show that performance of DISE is comparable with another scalable query engine. The response times for SANA are shown in Table V.

The Figure 7 shows the performance comparison of DISE and SANSA engine corresponding to a few selected queries over the three datasets, the three lines (depicted with red-orange shades) demonstrate the response time of SANSA on datasets D1, D2, D3, respectively), and the second group of lines (depicted in shades of green) show the response time obtained using DISE on the same datasets. Here, this can be observed that the performance of DISE is almost comparable to SANSA, in addition to being scalable.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we present DISE – a scalable distributed in-memory SPARQL engine using Apache Spark. DISE is based on the representation of RDF as tensor and utilises the

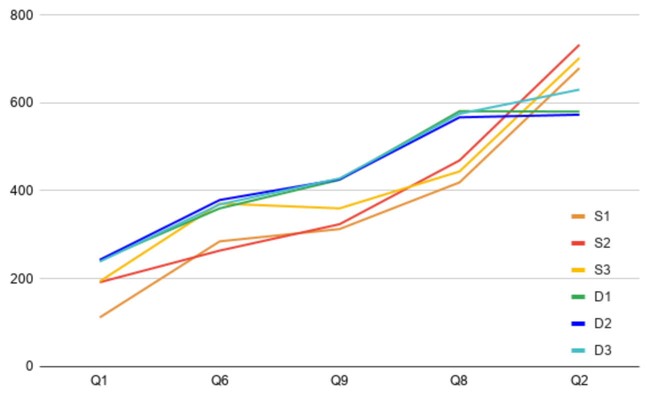


Fig. 7. Comparison Scalability DISE and SANSA (lines with shades of blue and green corresponded to DISE and red corresponde to SANSA relatively).

DOF of the query TP to determine the query complexity and creating a dynamic execution plan. DISE in integrated into SANSA. It extends the functionality of SANSA Knowledge Representation Layer and Query Layer. The empirical evaluation of DISE shows its performance over a range of different queries with varying complexity, demonstrating that DISE is a generic SPARQL processing engine. We also showed that the runtime of query evaluation is proportional to the query complexity. The scalability of DISE was shown by evaluating the performance over a range of dataset with varying sizes. In addition, the results of DISE are comparable with an existing SANSA query engine. In future, we plan to include other types of SPARQL queries. We want to optimize the tensor operations, in order to obtain faster joins and more efficient runtime performance.

Acknowledgment

This work was partly supported by the EU Horizon2020 projects Boost4.0 (GA no. 780732), QROWD (GA no. 723088), LAMBDA (GA No. 809965), and CLEOPATRA (GA no. 812997).

References

[1] J. Lehmann, G. Sejdiu, L. Bühmann, P. Westphal, C. Stadler, I. Ermilov, S. Bin, N. Chakraborty, M. Saleem, A.-C. N. Ngonga, and H. Jabeen, "Distributed Semantic Analytics using the SANSA Stack," in *Proceedings of 16th International Semantic Web Conference - Resources Track (ISWC'2017)*, 2017.

[2] P. Hitzler, M. Krötzsch, and S. Rudolph, *Foundations of Semantic Web Technologies*. Chapman and Hall/CRC, 2010.

[3] Resource Description Framework (RDF). [Online]. Available: http://www.w3.org/2001/sw/

[4] W3C. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation 10 February 2004. [Online]. Available: http://www.w3.org/TR/rdf-concepts/

[5] W3C. RDF Semantics. W3C Recommendation 10 February 2004. [Online]. Available: http://www.w3.org/TR/rdf-mt/

[6] T. Franz, A. Schultz, S. Sizov, and S. Staab, "Triplerank: Ranking semantic web data by tensor decomposition," in *International Semantic Web conference*. Springer, 2009, pp. 213–228.

[7] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM review*, vol. 51, no. 3, pp. 455–500, 2009.

[8] W3C. SPARQL Query Language for RDF. W3C Recommendation 15 january 2008. [Online]. Available: http://www.w3.org/TR/rdf-sparql-query/

[9] R. D. Virgilio, "Distributed in-memory SPARQL Processing via DOF Analysis," in *Proceedings of the 20th International Conference on Extending Database Technology (EDBT)*, 03 2017, pp. 155–168.

[10] B. Chambers and M. Zaharia, *Spark: The Definitive Guide: Big Data Processing Made Simple*. O'Reilly Media, 2018.

[11] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: SQL and rich analytics at scale," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*. ACM, 2013, pp. 13–24.

[12] Spark Programming Guide. [Online]. Available: http://spark.apache.org/docs/-2.1.1/programming-guide.html

[13] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald, "TriAD: A distributed shared-nothing RDF engine based on asynchronous message passing," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 06 2014.

[14] S. Álvarez Garcá, N. Brisaboa, J. Fernández, M. A. Martánez-Prieto, and G. Navarro, "Compressed Vertical Partitioning for Efficient RDF Management," *Knowledge and Information Systems*, 01 2014.

[15] D. Graux, L. Jachiet, P. Genevs, and N. Layada, "Sparqlgx: Efficient distributed evaluation of sparql with apache spark." in *International Semantic Web Conference (2)*, ser. Lecture Notes in Computer Science, vol. 9982, 2016, pp. 80–87.

[16] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "Scalable semantic web data management using vertical partitioning," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07. VLDB Endowment, 2007, pp. 411–422.

[17] A. Schätzle, M. Zablocki, S. Skilevic, and G. Lausen, "S2RDF: RDF querying with SPARQL on Spark," *Proceedings of the VLDB Endowment*, vol. 9, 12 2015.

[18] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and complexity of sparql," *ACM Trans. Database Syst.*, vol. 34, no. 3, pp. 16:1–16:45, Sep. 2009.

[19] F. Picalausa and S. Vansummeren, "What are real SPARQL queries like?" *Proceedings of the International Workshop on Semantic Web Information Management, SWIM 2011*, 06 2011.

[20] M. Schmidt, M. Meier, and G. Lausen, "Foundations of SPARQL Query Optimization," in *Proceedings of the 13th International Conference on Database Theory*, ser. ICDT '10, 03 2010, pp. 4–33.

[21] K. Guu, J. Miller, and P. Liang, "Traversing knowledge graphs in vector space," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational Linguistics, Sep. 2015, pp. 318–327.

[22] T. Neumann and G. Weikum, "Scalable Join Processing on Very Large RDF Graphs," *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD 2009), ACM, 627-640 (2009)*, 01 2009.

[23] A. Schätzle, M. Przyjaciel-Zablocki, T. Berberich, and G. Lausen, "S2x: Graph-parallel querying of rdf with graphx," in *Biomedical Data Management and Graph Online Querying*. Springer International Publishing, 2016, pp. 155–168.

[24] Q. Wang, Z. Mao, B. Wang, and L. Guo, "Knowledge graph embedding: A survey of approaches and applications," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 12, pp. 2724–2743, 2017.

[25] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer, "DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia," *Semantic Web Journal*, vol. 6, no. 2, pp. 167–195, 2015.

[26] K. Mller, M. Hausenblas, R. Cyganiak, and G. A. Grimnes, "Learning from linked open data usage: Patterns & metrics," in *Proceedings of the WebSci10: Extending the Frontiers of Society On-Line. Web Science Conference (WebSci), April 26-27, Raleigh, North Carolina, United States*, 2010.

[27] G. Sejdiu, D. Graux, I. Khan, I. Lytra, H. Jabeen, and J. Lehmann, "Towards A Scalable Semantic-based Distributed Approach for SPARQL query evaluation," in *15th International Conference on Semantic Systems (SEMANTiCS)*, 2019.