Chawla *et al. J Big Data* (2021) 8:130
https://doi.org/10.1186/s40537-021-00519-6

Journal of Big Data

RESEARCH Open Access

MuSe: a multi-level storage scheme for big RDF data using MapReduce

Tanvi Chawla, Girdhari Singh and Emmanuel S. Pilli*

*Correspondence:
espilli.cse@mnit.ac.in
Department of Computer Science and Engineering, Malaviya National Institute of Technology, Jaipur, India

Abstract

Resource Description Framework (RDF) model owing to its flexible structure is increasingly being used to represent Linked data. The rise in amount of Linked data and Knowledge graphs has resulted in an increase in the volume of RDF data. RDF is used to model metadata especially for social media domains where the data is linked. With the plethora of RDF data sources available on the Web, scalable RDF data management becomes a tedious task. In this paper, we present MuSe—an efficient distributed RDF storage scheme for storing and querying RDF data with Hadoop MapReduce. In MuSe, the Big RDF data is stored at two levels for answering the common triple patterns in SPARQL queries. MuSe considers the type of frequently occuring triple patterns and optimizes RDF storage to answer such triple patterns in minimum time. It accesses only the tables that are sufficient for answering a triple pattern instead of scanning the whole RDF dataset. The extensive experiments on two synthetic RDF datasets i.e. LUBM and WatDiv, show that MuSe outperforms the compared state-of-the art frameworks in terms of query execution time and scalability.

Keywords: RDF, SPARQL, Hadoop, HDFS, MapReduce, Storage

Introduction

Semantic Web is an outcome of the vision of W3C of a 'Web of Linked data' [1]. The Linked data can be easily understood and accessed and; is represented by the Semantic Web [2]. Semantic Web provides ease of access to all information available on the World Wide Web (WWW) and represents it in a format that is understandable to both humans and machines. The Semantic Web is being put to good use for information retrieval [3]. The semantic web technologies are being used by many people in building vocabularies, writing data handling rules, and creating data stores on the web. The technologies like Web Ontology Language (OWL), RDF, and SPARQL Protocol and RDF Query Language (SPARQL) empower Linked data [4, 5]. The foundation to publish and link data is provided by the RDF while, SPARQL is the commonly accepted query language used for the Semantic Web. The Semantic Web data is structured and interoperable so it is easily shared and reused by heterogeneous applications across the Web [6]. Semantic Web has established RDF as the standard model for data interchange. The flexible nature of RDF is a result of its underlying graph-based model that makes it a popular and standard choice for data interchange on the Semantic Web. RDF is a key data representation

format that helps to maintain data for the Semantic Web in a structured form [7–9]. So, RDF serves as a common format for integrating and sharing structured and semi-structured data across various applications. The RDF format was earlier just used for representing data for the Semantic Web but, presently it is used to represent any high-quality data connected by links. The RDF format provides a standard way to represent such data. With RDF any kind of information can be expressed with a uniform structure in a simple manner [10].

The Linked Open Data (LOD) movement has increased the amount of RDF data being published on the web. The representation of data in such a linked form enables knowledge discovery [11]. Some of the popular search engines that provide support for RDF data are Google, Bing, etc. Linked data is being used to integrate data from different domains. This distributed data linked from different domains form a knowledge graph. Some of the popular knowledge graphs are Dbpedia, Freebase, YAGO, etc. [12]. RDF can be used to represent the data for social media platforms such as Twitter, LinkedIn, Facebook etc. [13]. RDF is also being widely used to represent and process data for spatiotemporal data domains [14]. The Semantic Web approaches are increasingly being used to represent academic and research data. RDF is being used to represent metadata on the web such as for the web pages and the search engines. Many big companies such as Yahoo, Google, Microsoft, and Facebook are maintaining their metadata in a structured form using RDF. One of the websites that mostly uses Semantic Web technologies is the media company; British Broadcasting Corporation (i.e., the BBC). Thus, many real-world applications use RDF to represent their data collected from different sources in a standard format and for data integration. Some of the applications of LOD are BBC World Cup 2010 and 2012, Dbpedia, etc. Dbpedia is a linked data version of Wikipedia. RDF format is also being used in the field of life sciences to represent, integrate and analyze large biomedical datasets. Some of the RDF datasets are listed on the DataSetRDF-Dumps page of the W3C. One such data collected from sensors and sensor observations available on this page is the Linked Sensor data containing 1.7 billion triples.

RDF data storage and processing on a single machine is becoming more challenging with the continuously increasing dataset size [15]. The "Big Semantic Web data" era is a result of the growing success of the Web of data initiatives and the Semantic Web. The very initial approaches for RDF data management are based on centralized architecture and thus have a high scalability overhead. Some examples of centralized RDF systems are RDF-3x, Jena, Hexastore, RDFDB, RDFStore, etc. These systems although are simple but they suffer from the common limitations of the centralized approaches [16].

Later, the focus shifted towards designing Distributed RDF systems for improving the performance of existing systems and to overcome the current limitation of scalability. The research was being diverted towards distributed RDF management to overcome the Big Semantic Web data challenge. The term scalability means the ability of a system to handle a large amount of data without any compromise in its performance. Some well-known Distributed RDF systems are Virtuoso, Clustered TDB, Yars2, BigOWLIM, RDF-Peers, etc. These distributed RDF databases have been an efficient solution to overcome the challenges of centralized RDF systems.

With advancements being made in the field of Distributed RDF databases and increase in amount of RDF data the drawbacks of Distributed RDF databases became more

evident. So researchers were working towards the improvement of existent Distributed RDF systems or building these Distributed systems from scratch. The major limitation of these systems is their prerequisite of a dedicated infrastructure. These systems are not cost-effective to constitute a scalable framework owing to their requirement for a large number of resources. Hence, efforts were being put towards leveraging commodity-grade machines on the generic cloud platforms for large-scale RDF data processing. Because of these developments, the MapReduce framework was gaining popularity for building scalable RDF management systems for managing the large-scale RDF data and to overcome the existing challenges such as proposed by Mazumdar et al. [17].

Motivation

The existing works on Big RDF Data storage generally store the RDF data on a single level in Hadoop. Most of the frameworks also do not take into account the type of triple patterns in a SPARQL query. Some of these frameworks are discussed in the "Related Work" section of this paper. Hence, such frameworks suffer from the huge costs in terms of time incurred during SPARQL Query Processing. These frameworks spend most of the time in scanning large two column VP tables created during Big RDF Storage. For example, in LUBM dataset the VP table for rdf:type predicate contains large number of records (and its size is $\sim 2GB$). This motivated us to design a storage scheme that minimizes the size of these large tables and thus, reduces the scan space for query processing. The idea behind MuSe is to create another level of storage keeping into consideration the structure of triple patterns in a SPARQL query. This second level contains subtables that are directly scanned to generate intermediate results for predicate-object bound triple patterns. Hence, the minimum scan space requires less data scan time thereby, reducing the query execution time in MuSe.

Key contributions

Our work presents a multi-level RDF storage scheme for storing and processing Big RDF Data (or loarge scale RDF data) on Hadoop. The main contributions of this work are as follows:

- propose an efficient and scalable method, called MuSe; for storing Big RDF data on the Hadoop Distributed File System (HDFS).
- SPARQL queries are converted into a suitable format to be run as MapReduce jobs on the Big RDF data stored by MuSe on Hadoop.
- extensive experiments on two RDF datasets have been conducted to verify the query performance and scalability of our Big RDF data storage method.

From results we observe that on an average, MuSe outperforms the compared state-of-the-art methods by an order of magnitude. The rest of this paper is organized as follows. Section 2 presents the frameworks designed for storage and querying of Big RDF Data. Section 3 describes the proposed storage and query processing architectures implemented in MuSe for Big RDF data. In Section 4 we show our extensive experimental results. We finally conclude and discuss the future work in Section 5. The SPARQL queries used for LUBM datasets in our experiments are given in Secion 6.

Related work

This section discusses some of the existing distributed frameworks for storing and querying large scale RDF data.

Graux et al. [18] proposed SPARQLGX, a distributed RDF framework based on Apache Spark to evaluate SPARQL queries. With SPARQLGX the SPARQL queries can be executed on large scale data stored across multiple nodes in a cluster. SPARQLGX translates the SPARQL queries into Scala code that can directly be executed with the Spark API. This framework uses the vertical partitioned architecture proposed by Abadi et al. [19] for storing large scale RDF data. Thus, the RDF data is stored in two column tables having entries for only subject and object i.e. (s,o) for a triple (s,p,o) in the original RDF dataset. These tables are assigned the predicate name 'p' corresponding to which the s and o entries are stored. The advantage with this storage architecture is that it reduces the evaluation time of triple patterns in SPARQL queries where the predicate 'p' is constant for eg. triple pattern (?s p ?o). And in practice, it has been observed that most SPARQL queries have triple patterns with a constant predicate. Thus, vertical partitioning storage strategy is quite suitable for RDF data, it reduces the dataset size and provides an indexation for the RDF data. SPARQLGX suffers from the drawback of reading the entire dataset in order to compute statistics over the dataset. It does not take the query structure into account and reads only data related to that query. Thus, it also has the additional overhead of computing statistics prior to query processing.

Hassan et al. [20] proposed two distributed data stores i.e. 3CStore and VPExp for storing large scale RDF data. These storage approaches are based on the vertical partitioning strategy. 3CStore scheme uses a three-column layout for RDF data storage. The four possible correlations that can occur between two triple patterns based on the position of their join variable based on their corresponding subject and object positions. These correlations are subject-subject (SS), subject-object (SO), object-subject (OS) and object-object (OO). On the basis of these correlations a subset of VP table with all the other VP tables using inner join is pre-computed. Then three-column stores are created for all the four correlations. The main objective of 3CStore storage model is to minimize the number of join operations and the input data size during SPARQL query evaluation. In VPExp approach, the input data size is minimized only for the *rdf:type* predicate. This predicate is quite common in the RDF datasets and it contains the most number of rows while vertical partitioning the RDF dataset. VPExp splits this predicate into the number of distinct objects that this predicate had. Thus, the goal of VPExp is to minimize the input data for the *rdf:type* predicate table when the triple pattern in a SPARQL query has *rdf:type* at the predicate position and the corresponding object is not a variable. It is observed that the proposed storage approach requires more storage space and a longer data loading time than the other architectures.

Chawla et al. [9] proposed HyPSo, a hybrid partitioning strategy for processing SPARQL queries on Big RDF data. HyPSo combines vertical partitioning strategy for RDF storage with hash partitioning by subject. The proposed strategy improves SPARQL query performance by avoiding scanning of whole vertical partition to compute results. Here, all objects belonging to a subject are stored together thus, eliminating the overhead of reading entire partition. The SPARQL queries are then executed according to this hybrid partitioning schema. The data is stored as flat files on HDFS. It converts the

SPARQL queries into Pig Latin queries to be executed on the stored RDF data. These Pig queries are internally executed as MapReduce jobs on HyPSo stored RDF data. This architecture has a drawback that RDF data belonging to a particular subject may be stored in different vertical partitioned tables on different nodes. Thus, the inter-node communication time may increase during query processing resulting in a degrade in the query performance. So, it is not efficient for all types of SPARQL queries

Schätzle et al. [21] proposed Sempala, a scalable RDF framework for processing SPARQL queries on Hadoop. The Big RDF data is stored using Parquet, a columnar layout on Hadoop and Impala, a SQL query engine is used for query execution. Sempala stores RDF data in a single unified property table and thus, efficiently answers star-shaped pattern queries. Thus, with this storage schema all RDF properties used in the dataset are stored together thereby, reducing number of joins for a query. All queries can be answered using this single table. The drawback with this framework is that it is efficient for star queries only and not for other query types.

Punnoose et al. [23] propose Rya, a scalable RDF management system that provides efficient support for SPARQL queries. The proposed system introduces storage, indexing and query processing methods for Big RDF data that support processing of SPARQL queries on large scale RDF data. The proposed system Rya stores RDF data in triples form in Accumulo, these triples are indexed across three tables that can satisfy all the triple patterns. These three tables are SPO, POS and OSP. These are named according to the order of the triples components stored. Such as the OSP table stores a triple in the Row ID as (object, subject, predicate). These three table indexes are scanned to answer a SPARQL query and it is observed to work fast in many situations. Rya suffers from the limitation of replicating data multiple times for exploiting its index data storage architecture over all the possible elements. Thus, each of the three table indexes will be stored multiple times.

Rohloff et al. [24] proposed SHARD, a high-performance massively scalable distributed system that uses the MapReduce framework. This information system, persists the graph data as RDF triples and the SPARQL query language is used to respond to queries over the system. The SHARD system is evaluated using the LUBM benchmark dataset that is used for benchmarking RDF triple-stores. The design motivation for SHARD is to provide the ability for persisting and rapidly querying large data graphs. In flat files of SHARD, each line represents all the triples associated with a different subject. Thus, SHARD stores RDF data in a simple and easy to read format. SHARD has the limitation that it stores RDF data in plain files on HDFS. Thus, it needs to scan entire dataset during SPARQL query processing. Also, the dataset may need to be scanned multiple times in case a query contains multiple clauses.

Cossu et al. [25] proposed PRoST, Partitioned RDF on Spark Tables that stores Big RDF data using a hybrid scheme by combining two popular RDF storage techniques i.e. Binary and Property tables (Vertical Partitioning+Property Table). For processing the query is first split up into several sub-parts and then each subquery is executed using the most suitable storage approach. Then, next the results from the subqueries are joined together to obtain the final result. As PRoST, stores the Big RDF data using two RDF storage schemes (i.e. PT and VP) it occupies more storage space. And it is observed that PRoST occupies double space than SPARQLGX [18].

Schätzle et al. [26] proposed PigSPARQL, a system to process complex SPARQL queries on a MapReduce cluster. PigSPARQL uses the Vertical Partitioning data storage model. This framework makes use of the observation that a typical SPARQL query has a bounded predicate in the triple pattern while, the subject and object are generally variables. It uses vertical partitioning for storing large scale RDF data as this approach reduces amount of RDF data required to be loaded for query execution. With this strategy all RDF triples belonging to a predicate are stored in the same file and each predicate has its own file. Also, it just requires a single mapreduce job in advance and it doesn't consume much disk space. From results, it was seen that this approach is a quite effective solution for improving SPARQL query performance so the authors suggested that it can be extended for further improvement.

Multi-Level Big RDF Storage Scheme (MuSe): Architecture

In this section, we first introduce the storage strategy used in our proposed approach, MuSe. Then, we present the method of processing SPARQL queries on Big RDF data stored using MuSe on HDFS. We also describe in detail how SPARQL queries are implemented using Hadoop MapReduce on the HDFS.

Figure 1 depicts the architecture of MuSe. As depicted in the figure, the large scale RDF data input by the user is stored on HDFS in Hadoop. MuSe stores this input data at two levels i.e. *Level I* and *Level II* for improving SPARQL query performance. MuSe is based on the vertical partitioning (VP) storage scheme for storing Big RDF data. At Level I, triple patterns with bound predicates are processed in a SPARQL query. Level II,

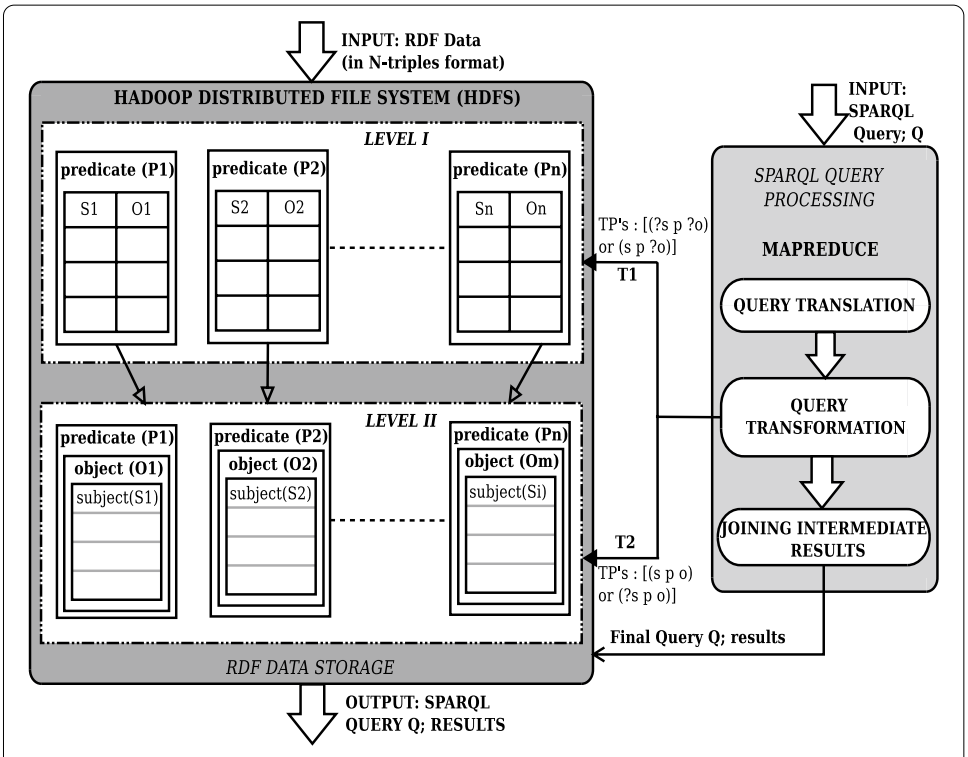


Fig. 1 MuSe Architecture

is used for processing triple patterns where the predicate and object are bound and the subject is a variable. So, second level processes the triple patterns of the form (?s p o). MuSe further simplifies the vertical partitioning scheme to reduce the search space of predicate bound triple patterns in the SPARQL queries.

MuSe RDF Storage

The architecture of MuSe is based on the general observation that in SPARQL queries generally the predicate is bound while the subject and object may be bound or unbound. The different types of triple patterns in a SPARQL query are: (?s p o), (s ?p o), (s p ?o), (?s ?p o), (s ?p ?o) and (?s p ?o) [27]. A triple pattern in the SPARQL query is implemented either on Level I or Level II according to this observation. In MuSe, the data is pre-processed by the user and this pre-processed data in N-Triples format is stored on the HDFS. Further, MuSe stores this data at two levels for implementing individual triple patterns during SPARQL query processing. On Level I, the simple vertical partitioning (VP) storage scheme is used. So, here RDF data is stored in multiple two column tables. The subject and object of our input RDF triples are stored in each column. Each table is vertical partition table that is named by the predicate corresponding to which subjects and objects are stored in that table. So Level I, stores table of format p(s,o) where table name is represented by 'p' while (s,o) are stored in the two columns of these tables. Thus, triple pattern of a SPARQL query in the form (?s p ?o), (s p ?o) and (?s p o) are implemented at Level I of MuSe. The advantage with VP scheme is that it reduces the scan space for processing SPARQL queries thereby improving query performance.

The need to further reduce scan space for improving SPARQL query performance motivated us to propose MuSe storage model. Level II, in MuSe storage model fulfils the requirement of reducing scan space for SPARQL queries having triples patterns where both predicate and object are bound while subject is unbound. A SPARQL query, with triple patterns of form (?s p o) can be easily implemented at Level II in lesser time. This additional level reduces the scan space for such triple patterns in a SPARQL query. Here, we further split a vertical partitioned table into smaller tables. Each predicate table is divided into tables named by distinct objects in the predicate table. Thus, tables at level II are of the form p[o(s)]. The smaller tables at this level are named by the object and contain a single column with entries of all subjects belonging to that predicate-object pair. This storage model of MuSe helps in minimizing the amount of scan space required for processing SPARQL queries. Thus, triple patterns with a bound predicate-object pair can be implemented at this level. The corresponding predicate table that matches with the predicate of a triple pattern in SPARQL query will be accessed. The triple pattern will be matched to its corresponding table by firstly matching the bound predicate in the first pass and then matching the object in second pass.

In this way, only the entries matching to predicate-object pair in a triple pattern of our SPARQL query will be retrieved. So only a small table will be scanned instead of scanning an entire predicate table with many entries. For example, in (?s p o) triple pattern firstly the predicate table 'p' will be retrieved in first pass. In second pass, the table 'o' will be searched amongst the list of subtables present in table p. After, accessing the subtable 'o' all entries of this table will be retrieved as the intermediate results for this triple pattern. Similarly, this process will be repeated for all such triple patterns in the SPARQL

query. If there exists some other triple patterns where only predicate is bound in such cases, intermediate results for these patterns will be retrieved by matching that predicate with its corresponding VP table at Level I. This storage scheme is beneficial for different types of SPARQL queries as the queries have multiple kinds of triple patterns. These two levels in MuSe enable answering of SPARQL queries in minimum execution time by selecting best possible storage model for different types of triple patterns. The RDF data from level I is taken as input and is transformed to form subtables at level II. The RDF data for both levels is stored on the HDFS. As the number of predicates in RDF data are limited so, we choose vertical partitioning strategy for storage at Level I to store input RDF data in multiple tables. Also, for storage at level II we choose to store subtables named by object as triple patterns with bound predicate-object pair are more common than those with bound subject-predicate pair.

As stated earlier, all other triple patterns where predicate is bound are implemented at Level I as this strategy supports fast data access compared to executing triple patterns on RDF data stored in a single file. MuSe is especially beneficial for large scale RDF data. The different types of RDF data in N-Triples format can be well handled by MuSe. The storage structure of MuSe requires that prior to running any SPARQL query the bound terms for each triple pattern in the query must be analyzed for loading its corresponding table or subtable during query execution. Accordingly, the table or subtable will be loaded for a bound predicate in each triple pattern of the query. If an object is bound in the triple pattern then its corresponding subtable will be loaded. Firstly, the main table will be loaded after matching the predicate of triple pattern with our stored VP tables. Then second, a subtable will be loaded after object corresponding to that predicate in triple pattern is matched with available subtables under that main predicate table on HDFS. This process is repeated for all triple patterns in the SPARQL queries. With the addition of Level II, in MuSe storage scheme we try to solve the issue of scanning large predicate tables for eg. predicate *rdf:type* in RDF datasets forms a large table. Thus, division of such large tables into subtables provides ease of access to data especially in cases where the data size is large and scan space needs to be reduced for minimizing query response time. The advantage of MuSe storage scheme is that it is easy to implement and is suitable for different types of SPARQL queries.

The subject entries are retrieved as intermediate results on implementing a triple pattern with bound predicate-object pair in a SPARQL query. Similarly, for each triple pattern in the SPARQL query these intermediate results are retrieved and are joined to form final results of the query. The star-shaped SPARQL queries will especially be benefitted from this storage scheme as in such queries the subject is unbound while both predicate and object are bound. These star queries contain subject–subject (ss) joins.

The number of P tables at Level I is equal to the number of distinct predicates in the input RDF dataset; I. The number of P-O subtables is equal to the number of distinct objects matching each distinct predicate in the input dataset. The number of distinct predicates may vary for different datasets like WatDiv dataset contains more distinct predicates than the LUBM dataset. For example, 'n' P tables will be built at Level I for a dataset; I containing 'n' distinct predicates. And, suppose there are 'm' distinct objects for each distinct predicate. Then, 'nm' number of P-O subtables will be built for each P table. So, in all 'n' tables (P tables) will be stored at Level-I and 'nm' tables (P-O subtables)

will be stored at Level-II. The space complexity of Algorithm 1 is O[(n-size)+(nm-size)]. The space occupied by MuSe depends on the size of P tables and P-O subtables. Thus, it can be said that the amount of tables generated at levels I and II depends on the number of predicates in the input RDF dataset. In worst case, there might be no bound predicate in the query for which entire RDF dataset file needs to be loaded for query processing. Then, this algorithm requires space equivalent to the size of input RDF data. Thus, the space complexity of Algorithm 1 in worst case will be O(I-size). Time complexity of MuSe storage depends on Table generation and Table loading time. Firstly, it will consume some time to generate P and P-O subtables at both levels. Then, it will load tables and subtables respective to the triple patterns in the input query. So, if there are 'i' distinct predicates and 'j' distinct predicate-object pairs in the query. Then, time complexity of Algorithm 1 in average case is O[(n+nm)+(i+ij)] where, (n+nm) is table generation time and (i+ij) is table loading time. The best case is when only 'ij' subtables need to be loaded so best case time complexity is O[(n+nm)+(ij)]. The worst case happens when entire RDF dataset file needs to be loaded as there is no bound predicate in the query. Thus, worst case time complexity will be O(I). Algorithm 1, depicts the two level storage of MuSe.

Algorithm 1: MuSe two-level Storage

Input: RDF dataset, I and SPARQL Query, Q
Output: P tables, p[(s,o)] and P-O subtables, p[o(s)]
1: Extract 'n' distinct predicates in the input dataset, I.
2: Generate 'n' P tables corresponding to the number of distinct predicates in dataset, I.
3: Generate 'nm' P-O subtables corresponding to the number of P tables (generated in previous step) for all possible predicate-object pairs in the dataset, I.
4: Retrieve triple patterns (1....n), t_p from the input query, Q.
5: **if** *(predicate 'p' is bound in each* t_p*)* **then**
  Load P Tables named by 'p'for each distinct predicate in all triple patterns.
  /* This is the level I storage of MuSe where tables are named by the predicate, p and subject-object pairs, s-o corresponding to this predicate are stored as entries in the VP Tables. These are in the form p[(s,o)] and we name them as P Tables. */
6: **else**
  Load the RDF data directly stored in a flat file on HDFS.
  /* In case, a predicate is not known for a triple pattern in the query Q then, this file will be accessed to generate intermediate results for that pattern. */
7: **for** (Triple pattern t_p, extract the bounded predicate-object, p-o pairs) **do**
8: Match the predicate extracted in previous step to each P Table.
9: In case, a match is found load predicate-object (P-O) subtables for the corresponding P Table.
  /* This is the level II storage of MuSe where tables are named by predicate, p and the subtable is named by object, o from each bounded predicate-object pair of the retrieved triple patterns. These are in the form p[o(s)] and we named them as P-O subtables. */
10: **end for**

MuSe SPARQL query processing

MuSe is implemented using the Hadoop MapReduce framework for SPARQL query processing. The input RDF data transformed to MuSe storage model is stored on the HDFS and SPARQL queries in the form of MapReduce jobs are run on this RDF data. As shown in Fig. 1, the SPARQL query is given as input by the user to the SPARQL query processing module. In this module, the SPARQL query given as input is transformed into a format suitable to be run on HDFS as the SPARQL queries cannot be directly run on Hadoop. So, we transform the input SPARQL queries into Apache Pig

format. The queries in this Pig Latin format are internally executed as a sequence of MapReduce jobs. In these Pig queries, the data for both level I and level II is scanned according to the type of triple pattern in the SPARQL query. For example, for a triple pattern [?X rdf:type ub:GraduateStudent] the subtable will be scanned by a writing a statement rdf:type = LOAD '/usr/local/hadoop/input/lubm100VP/rdf_type/ub_GraduateStuden' USING PigStorage('\t') AS (s); in the Pig query. Here, all subject entries in the subtable named ub_GraduateStudent will be retrieved as the subject is variable or unbound denoted by ("?X"). And, for scanning a VP table for a triple pattern in form [?X rdf:type ?Y] the statement rdf:type = LOAD '/usr/local/hadoop/input/lubm100VP/rdf_type' USING PigStorage('\t') AS (s,o); will be written in the Pig query. Here, all subject-object pair entries in the VP table named rdf_type will be retrieved because both subject and object are unbound as denoted by ("X" and "Y").

The SPARQL queries with bound objects in triple patterns are directly executed by scanning the corresponding subtables for that objects thus reducing the data scan space and hence access time. Hence, MuSe helps in improving the SPARQL query performance by providing apt storage model according to the different types of triple patterns in the query. For each subtable scanned corresponding to the predicate the subject entries in the subtable are retrieved as intermediate results for the triple pattern. Depending upon the triple patterns in a SPARQL query, the number of subtables and VP tables required to be accessed may vary.

The intermediate results of each triple pattern are joined internally by the Pig query and stored on HDFS to be later retrieved by the user. The purpose of each triple pattern in a SPARQL query is to find the value of variables denoted by ("?") in the triple patterns. MuSe executes each triple pattern in the SPARQL query as a sub-query so these subqueries are executed as a series of triple pattern matching operations. Thus, variables in each triple pattern are retrieved by matching bindings with the input or stored RDF dataset. The triple pattern matching algorithm used for SPARQL query processing in MuSe is shown in Algorithm 2.

In case, the predicate is a variable (that is an uncommon scenario) then the entire input file (in N-Triples format stored on HDFS) will be scanned to generate intermediate results for the triple pattern in the query. For taking the original input file as the source we only need to modify the translation process in the way the triple patterns are treated in MuSe storage model. This is quite a time consuming process and degrades SPARQL query performance. But according to our assumption the triple pattern in SPARQL queries have a bounded predicate so this scenario is unlikely to happen. The object subtable will be retrieved after accessing the predicate table by matching predicate of a triple pattern with the name of each VP table on the HDFS. After this, the object in that triple pattern will be matched with the subtables in the previously matched VP table. The reason to choose object-based subtables in Level II of MuSe is that in a RDF dataset the number of distinct objects is less than the number of distinct subjects. So, if we choose a subject-based subtable scheme then these subtables would be quite large in number and would be diffcult to handle. Also, the bounded predicate-object pair is more common in the SPARQL queries than subject–object pair. For example. the star-shaped queries have variable subjects in each triple pattern.

The space complexity of Algorithm 2, depends on intermediate results (IR) generated by each triple pattern on the input query. Thus, if there are 'i' triple patterns in the query then its space complexity is O(IR-size$_1$+IR-size$_2$+....+IR-size$_i$). Time complexity of this algorithm depends on the time required for scanning P tables and P-O subtables for processing each triple pattern of the query. In best case, MuSe triple pattern matching algorithm needs to scan 'ij' subtables so best case time complexity is O(ij). While, in average case it needs to scan 'i' tables as well as 'ij' subtables so average case time complexity is O(i+ij). In worst case, entire RDF dataset file needs to be scanned. Thus, worst case time complexity is O(I). The complexities of Algorithms 1 and 2 are summarized in Table 1.

Algorithm 2: MuSe Triple Pattern Matching Algorithm

Input: $t_p \leftarrow$ (s,p,o)

Output: Intermediate Results; IR

```
1 if p is not variable then
2     if o is not variable then
3         ti = scanTable p[o(s)]; /* Now, subtables containing subject entries
              will be scanned as input ti, for the triple pattern t_p, if the
              predicate-object pair is bound.                                  */
4         else
5             ti = scanTable p[(s,o)]; /* Now, VP tables containing subject-object
                  pair entries will be scanned as input ti, for the triple pattern
                  t_p, if only the predicate is bound.                         */
6 else
7     ti = scanFile (I.nt) /* The entire input RDF dataset stored on HDFS in
          N-Triples file named I, will be scanned in case if predicate is variable
          in triple pattern t_p, of the SPARQL Query Q.                        */
8 IR = ti; /* The data scanned for each triple pattern t_p, is retrieved as its
      intermediate result, IR.                                                 */
```

Table 1 Time and Space complexities of Algorithm 1 and 2

Algorithm	Time Complexity	Space Complexity
Algo 1	O[(n+nm)+(ij)] (Best Case) O[(n+nm)+(i+ij)] (Average Case) O(I) (Worst Case)	O[(n-size)+(nm-size)] (Best and Average Case) O(I-size) (Worst Case)
Algo 2	O(ij) (Best Case) O(i+ij) (Average Case) O(I) (Worst Case)	O(IR-size$_1$+IR-size$_2$ +....+IR-size$_i$) (Best, Average and Worst Case)

```
Q1:
SELECT ?X ?Y ?Z
WHERE {
        ?X rdf:type ub:GraduateStudent .
        ?Y rdf:type ub:University .
        ?Z rdf:type ub:Department .
        ?X ub:memberOf ?Z .
        ?Z ub:subOrganizationOf ?Y .
        ?X ub:undergraduateDegreeFrom ?Y .}
```

Results and discussion

In this section, we discuss the experiments conducted on different large scale RDF datasets to test the performance of MuSe. We have carried out extensive experiments on two popular RDF benchmark datasets i.e. LUBM and WatDiv to verify the efficiency and scalability of MuSe and compared it with the state-of-the-art SHARD and PigSPARQL frameworks. MuSe is also compared with a hybrid partitioning technique; HyPSo. HyPSo partitions the Big RDF data and stores this partitioned Big RDF Data on HDFS. All large scale RDF data input from the user and transformed with MuSe is stored on the Hadoop Distributed File System (HDFS).

Experimental setup and datasets

Our experiments are conducted on a cluster of 4 machines each equipped with 16GB RAM, 2TB of disk space and Intel® Xeon(R) E3-1220 v6 processor. The cluster runs Hadoop 2.9.0 with Pig 0.17.0 on Ubuntu 16.04 LTS operating system. The Lehigh University Benchmark (LUBM) dataset and Waterloo SPARQL Diversity Test Suite (WatDiv) v0.6 are used in our experiments. The WatDiv dataset is generated from WatDiv binary given in the test suite. WatDiv dataset generated from its model file is in the N-Triples format. We can specify the scale factor for setting the number of triples that need to be generated for eg. a scale factor of 1 approximately generates 100K triples. In our experiments we generate four WatDiv datasets containing approximately 10, 100, 200 and 400 million triples respectively as given in Table 2. The LUBM data generator class is used for generating datasets of 10, 50, 100 and 1000 universities and the random seed value is taken as 0 for data generation. In our evaluation, LUBM (n) means a LUBM dataset with n number of universities. The statistics of these datasets used in our evaluation are given in Table 2. The dataset generated from LUBM data generator is in OWL

Table 2 Dataset statistics

Dataset	No. of triples	Raw size	Size on HDFS
LUBM (10)	1.31 million	230.8 MB	220.14 MB
LUBM (50)	6.86 million	1.2 GB	1.13 GB
LUBM (100)	13.82 million	2.5 GB	2.28 GB
LUBM (1000)	137.76 million	24.6 GB	22.94 GB
WatDiv (10M)	10.92 million	1.5 GB	1.44 GB
WatDiv (100M)	109.99 million	15.6 GB	14.53 GB
WatDiv (200M)	219.71 million	31.6 GB	29.47 GB
WatDiv (400M)	439.40 million	63.5 GB	59.13 GB

format. This data is converted into N-Triples format using the rdfcat toolkit provided by Jena. We have used ten LUBM queries for our evaluation as listed in Appendix. Among these ten tested queries, Q1, Q3, Q4 and Q6 are the complex SPARQL queries. The star-shaped LUBM queries are Q2, Q7, Q8, Q9 and Q10. And Q5, is a simple query having a small input.

The Hadoop framework is an open source framework that supports distributed processing of large datasets on a cluster of computers. The Hadoop MapReduce (MR) programming model [28] provides a software framework for the distributed storage and processing of large scale data or Big Data. With Hadoop framework the massive datasets can be analyzed more quickly in parallel. The distributed file system of Hadoop known as HDFS is used for data storage. HDFS splits the stored data in blocks and sends it to the various nodes in a cluster. The MapReduce applications run on top of the data stored in HDFS. The Hadoop cluster comprises of a master node and many slave nodes depending upon the size of the cluster. In the cluster, the MapReduce processing is done at the slave nodes and the final results are sent to the master node.

We have used Hadoop MapReduce framework for MuSe according to our requirement of storage and query processing of Big RDF Data. Similar, to other applications MuSe uses HDFS for Big RDF Data storage and the MapReduce programming model for retrieval of Big RDF data.

The LUBM datasets contain information related to the academic domain. LUBM is a synthetic dataset. This benchmark dataset has been developed to evaluate the performance of Semantic Web repositories in a systematic and standard manner. This benchmark uses the Univ-Bench ontology. This ontology describes the departments and universities and the activities occuring in them [29]. The WatDiv dataset generator is used to generate the synthetic benchmark WatDiv datasets. By executing this generator different sized WatDiv datasets can be generated by setting different scale factors. The diverse test workloads are generated for WatDiv i.e. Basic Testing, Extension to Basic Testing (Incremental Linear and Mixed Linear Testing) and Stress Testing. The WatDiv dataset for evaluating a system under diverse test workloads. The test queries available with this dataset are used to focus on much wider aspects of query evaluation [30].

The term Big RDF Data or large scale RDF data refers to RDF datasets that are large in size. These large RDF datasets cannot be handled by centralized RDF systems (i.e. triple-stores). The centralized RDF engines are used for processing RDF datasets that are small in size and can be processed on a single machine. The experimental datasets mostly used in our work are large in size as centralized RDF architectures will not be able to handle RDF data of such scale. The centralized RDF systems will either fail to process such large data or they will be very slow in processing this scale RDF datasets. On the other hand, if we process small scale (or as we say less volume) RDF data then centralized RDF engines will prove to be effective. Thus, it can be said that the used datasets belong to Big Data.

Performance evaluation

We have generated datasets for our experiments from the available LUBM and WatDiv Dataset generators. With these data generators we can generate datasets of different sizes. The queries of WatDiv are categorized as Linear queries (L), Star queries (S) and Snowflake queries (F). The existing basic template queries of WatDiv were used in our

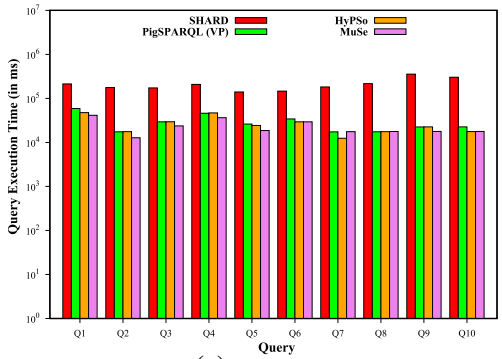
experiments. The efficiency of MuSe is compared on the basis of query execution time. And the scalability is measured by carrying out experiments for MuSe on various scale LUBM and WatDiv datasets.

(a) Efficiency

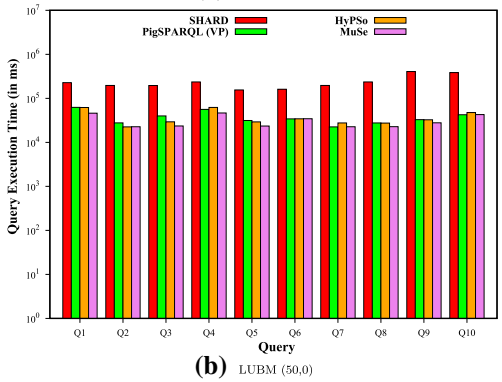
The different sized LUBM and WatDiv datasets input by user after pre-processing are loaded on the HDFS as shown in Fig. 1. The query execution times of SHARD, PigSPARQL (Plain RDF Data), PigSPARQL (VP) and HyPSo are shown in Table 3. We have considered two cases for PigSPARQL framework first, where it stores Big RDF data in Plain text files. And second, where it uses vertical partitioning storage strategy to store Big RDF data in two column tables. The query response times on LUBM datasets for 10, 50, 100 and 1000 universities are recorded in Table 3a–d respectively. In Table 4, we depict the query response times on WatDiv datasets, with Table 4a–d containing times for 10, 100, 200 and 400 million triples respectively. Figures 2 and 3 shows the query execution time comparison of MuSe with SHARD, PigSPARQL (Plain RDF Data), PigSPARQL (VP) and HyPSo architectures for LUBM and WatDiv datasets respectively. Figure 2a–d illustrate the execution time comparison of MuSe on LUBM datasets of 10, 50, 100 and 1000 universities respectively. Also, a similar comparison of MuSe on WatDiv datasets having 10M, 100M, 200M and 400M triples is shown in Fig. 3a–d respectively.

We observe that the average query execution time of MuSe is less than the compared state-of-the-art frameworks for both tested datasets. MuSe executes with an average time gain of 86.2%, 78.8%, 20.9% and 14.9% over the SHARD, PigSPARQL (Plain RDF Data), PigSPARQL (VP) and HyPSo architectures respectively for LUBM datasets. While for WatDiv datasets, it shows an average time gain of 95.6%, 10.2% and 6.7% over the PigSPARQL (Plain RDF Data), PigSPARQL (VP) and HyPSo architectures respectively. The parser of SHARD framework does not support Watdiv queries so, we have compared MuSe only with PigSPARQL and written "n/a" for the case of comparison with SHARD. From the figures, we see that MuSe performs better than the other two frameworks for all 10 queries tested on LUBM datasets. While, for WatDiv datasets its performance degrades only in case of query S6 over PigSPARQL (VP). MuSe performs better than PigSPARQL PigSPARQL (VP) for all other 7 queries tested on WatDiv datasets. We have tested different shaped SPARQL queries i.e. snowflake (F2 and F4), Linear (L4) and star-shaped (S2, S3, S4, S5 and S6) of WatDiv dataset on MuSe. It is observed that the query time of S6 increases with increasing data size in case of WatDiv dataset. As the overhead increases, with increasing dataset size so the response time for this query rises. S6 is a star query with 3 triple patterns. But all three triple patterns are predicate-bound. So, this query cannot leverage much benefits of the second level storage in MuSe. For S6, three VP tables at Level I in MuSe need to be accessed to generate intermediate results and answer this query. But overall the average query response time of MuSe is less than (Plain RDF Data), PigSPARQL (VP) and HyPSo for the all the tested WatDiv datasets.

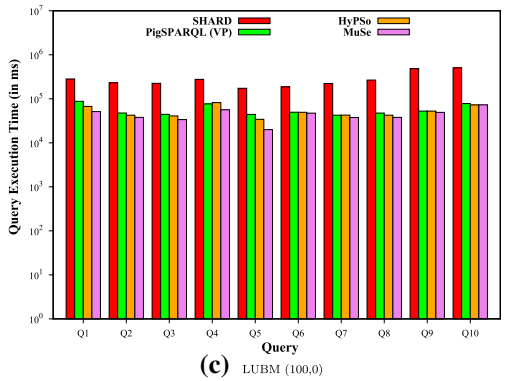
The comparatively poor performance of the compared frameworks can be attributed to their use of simple standard storage methods for all types of queries. These architectures do not leverage the advantage of known elements in triple patterns of a SPARQL query. While MuSe considers this fact and takes account of this advantage into its architecture. MuSe prunes a part of invalid input data by utilizing the RDF properties embedded in the triple patterns of SPARQL queries at Level I. It takes this a step forward by



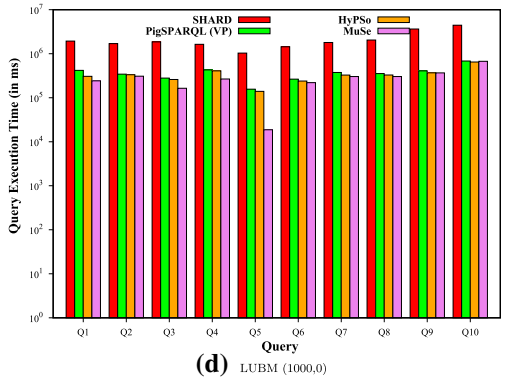
(a) LUBM (10,0)



(b) LUBM (50,0)



(c) LUBM (100,0)



(d) LUBM (1000,0)

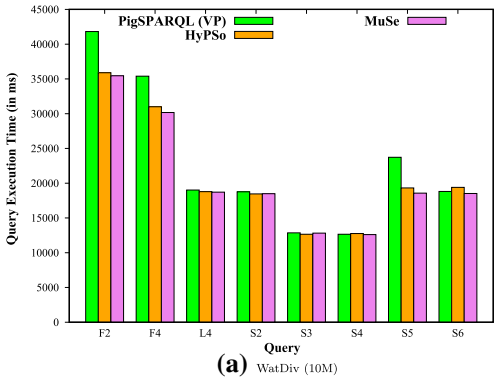
Fig. 2 Query Execution Times for LUBM datasets of 10, 50, 100 and 1000 universities respectively

Table 3 Query execution times (in milliseconds) for LUBM datasets of 10, 50, 100 and 1000 universities respectively with SHARD, PigSPARQL (VP), HyPSo and MuSe

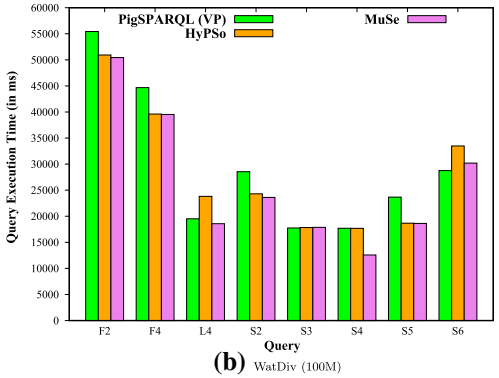
(a) LUBM (10,0)											
Time (in milliseconds)											
LUBM (10,0)											
Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	AM
SHARD	213810	176358	173337	208627	140132	146268	182510	216733	356609	303436	211782
PigSPARQL (Plain RDF Data)	74705	37729	49923	62062	39546	39589	37502	43307	67716	57630	50970.9
PigSPARQL (VP)	58814	17478	29372	46021	26097	34233	17362	17447	22485	22566	29187.5
HyPSo	47564	17553	29482	46956	24300	29389	12451	17551	22530	17666	26544.2
MuSe	41320	12784	23759	36523	18645	29448	17584	17679	17754	17761	**23325.7**
(b) LUBM (50,0)											
Time (in milliseconds)											
LUBM (50,0)											
Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	AM
SHARD	227738	196515	195449	235707	155206	161263	196432	236261	406586	384349	239550.6
PigSPARQL (Plain RDF Data)	197556	147551	134547	176638	114589	122231	142574	177595	298227	243029	175453.7
PigSPARQL (VP)	62359	27678	39814	56161	31401	34202	22447	27569	32781	42515	37692.7
HyPSo	61964	22454	29267	62200	29216	34341	27594	27436	32581	47550	37460.3
MuSe	46091	22679	23564	46604	23472	34445	22641	22772	27853	42808	**31292.9**
(c) LUBM (100,0)											
Time (in milliseconds)											
LUBM (100,0)											
Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	AM
SHARD	282767	230461	224887	276739	173276	186227	222525	266819	484749	504717	285316.7
PigSPARQL (Plain RDF Data)	382814	278190	264924	317275	223825	229918	297936	367654	633311	512774	350862.1
PigSPARQL (VP)	87564	47504	44378	76401	44172	49600	42463	47465	52596	77734	56987.7

Table 3 (continued)

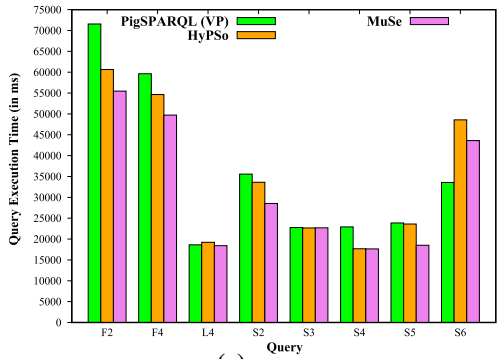
(c) LUBM (100,0)											
Time (in milliseconds)											
LUBM (100,0)											
HyPSo	67129	42446	40728	82040	34125	49426	42577	42448	52615	72537	52607.1
MuSe	51197	37934	33623	56243	19897	47154	37623	37821	49077	72782	*44335.1*
(d) LUBM (1000,0)											
Time (in milliseconds)											
LUBM (1000,0)											
Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	AM
SHARD	1933229	1693506	1861074	1636978	1034258	1436268	1796990	2045578	3647881	4434757	2152051.9
PigSPARQL (Plain RDF Data)	3404064	4075060	2936272	3563580	2249133	2364291	4012030	5172348	6681194	5923127	4038109.9
PigSPARQL (VP)	417814	342850	279480	429338	156659	264406	374463	353211	407984	682930	370913.5
HyPSo	306473	332750	259441	407624	139376	239364	327557	327869	368136	648152	335674.2
MuSe	241669	308034	163877	266795	18699	219726	303092	302926	366352	673249	**286441.9**



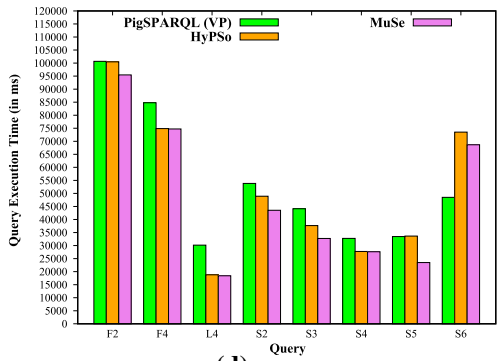
(a) WatDiv (10M)



(b) WatDiv (100M)



(c) WatDiv (200M)



(d) WatDiv (400M)

Fig. 3 Query Execution Times for WatDiv datasets of 10M, 100M, 200M and 400M triples respectively

Table 4 Query execution times (in milliseconds) for WatDiv datasets having 10M, 100M, 200M and 400M triples respectively on PigSPARQL (VP), HyPSo and MuSe

(a) WatDiv (10M)									
Time (in milliseconds)									
WatDiv (10M)									
Query	F2	F4	L4	S2	S3	S4	S5	S6	AM
SHARD	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
PigSPARQL (Plain RDF Data)	246241	262222	73103	143164	143029	142829	143057	108101	157718.25
PigSPARQL (VP)	41814	35394	19015	18764	12864	12656	23725	18811	22880.375
HyPSo	35890	30998	18782	18459	12662	12754	19317	19399	21032.625
MuSe	35454	30170	18714	18487	12817	12603	18575	18526	**20668.25**
(b) WatDiv (100M)									
Time (in milliseconds)									
WatDiv (100M)									
Query	F2	F4	L4	S2	S3	S4	S5	S6	AM
SHARD	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
PigSPARQL (Plain RDF Data)	2187767	2397936	749082	1530010	1729115	1742344	1737574	1185536	1657420.5
PigSPARQL (VP)	55424	44676	19521	28557	17753	17705	23674	28763	29509.125
HyPSo	50932	39609	23813	24304	17840	17686	18674	33488	28293.25
MuSe	50443	39517	18576	23612	17877	12580	18630	30198	**26429.125**
(c) WatDiv (200M)									
Time (in milliseconds)									
WatDiv (200M)									
Query	F2	F4	L4	S2	S3	S4	S5	S6	AM
SHARD	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
PigSPARQL (Plain RDF Data)	4553259	5030129	1617483	3451818	3369584	2830872	3242667	2854154	3368745.75
PigSPARQL (VP)	71569	59640	18611	35576	22760	22920	23855	33565	36062
HyPSo	60645	54645	19233	33618	22662	17673	23610	48578	35083
MuSe	55474	49729	18420	28527	22692	17648	18523	43604	**31827.125**
(d) WatDiv (400M)									
Time (in milliseconds)									
WatDiv (400M)									
Query	F2	F4	L4	S2	S3	S4	S5	S6	AM
SHARD	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
PigSPARQL (Plain RDF Data)	10367268	11318178	2879308	5415441	5480759	5439132	5958496	3835103	6336710.625
PigSPARQL (VP)	100681	84799	30176	53830	44135	32761	33451	48506	53542.375
HyPSo	100479	74878	18792	48922	37674	27738	33659	73528	51958.75
MuSe	95461	74731	18407	43547	32737	27646	23487	68674	*48086.25*

similary pruning the RDF objects bound in SPARQL queries at Level II. Thus, reducing or postponing a large number of Cartesian operations. PigSPARQL also does pruning but, it only prunes the RDF properties and not the other bound elements in a SPARQL query. While, SHARD performs no pruning and simply stores all RDF data in a single file by hashing on the subject. It is also seen that SHARD cannot evaluate multiple triple patterns in a single MapReduce job. HyPSo is designed keeping structure of some particular SPARQL queries into consideration. It only works well for those types of SPARQL queries and thus, its average performance is low as compared to MuSe. The storage and triple pattern matching algorithm used by MuSe is much better than that used by the other two frameworks as it prunes a large amount of invalid RDF data prior to and during query processing.

(b) Scalability

We compared MuSe with PigSPARQL (VP) and HyPSo. We carried out the scalability comparison experiments on various scale LUBM and WatDiv datasets. When the dataset size increases for LUBM and WatDiv the query time of all three methods increases and MuSe was always the best one. Figures 4 and 5 shows the scalability comparison of MuSe on LUBM and WatDiv datasets respectively. Figure 4a–d depict the scalability comparison on LUBM datasets for queries Q1, Q3, Q4 and Q8 respectively. Similarly, Figure 5a–d depict the scalability comparison on WatDiv datasets for queries F2, F4, S2 and S5 respectively. From these figures, we can observe that as the scale of datasets increases, the query time of PigSPARQL (VP) and HyPSo increases dramatically. In contrast, for MuSe the growth rate of query time changes slightly. These extensive experiments were carried out on various sized LUBM datasets i.e. LUBM10, LUBM50, LUBM100 and LUBM1000. Similary, scalability was tested on different sized WatDiv datasets with 10, 100, 200 and 400 million triples. These experiments were conducted on 4 queries of LUBM i.e. Q1, Q3, Q4 and Q8. And for WatDiv the 4 queries tested are F2, F4, S2 and S5. It is seen that SHARD performs worst than the other compared methods. The query performance of MuSe for almost all tested queries maintains to be low in comaprison to the other architectures with increasing size of WatDiv datasets. Thus, confirming that MuSe works well for large scale RDF data and scales well with data size.

Conclusion and future work

In this paper, we proposed MuSe: a Multi-Level Storage Scheme for Big RDF Data Using MapReduce. It is a two level Storage scheme for efficiently answering triple pattern matching SPARQL queries on Big RDF data using MapReduce. The proposed method takes advantage of the fact that mostly the predicate and object are the bound elements in triple patterns of a SPARQL query. MuSe RDF Storage component stores large scale data on Hadoop and its SPARQL query processing component processes the translated SPARQL queries as MapReduce jobs. Our extensive experiments on different sized RDF datasets verify the efficiency and scalability of our method which outperforms SHARD, PigSPARQL and HyPSo. The simple architecture of MuSe can easily be implemented and deployed across a Hadoop cluster. In future, we will investigate the effect of SPARQL query optimization strategies on performance improvement of MuSe. We would include the triple pattern reordering method for optimizing SPARQL queries in MuSe and observe its effect on improvement in SPARQL query performance.

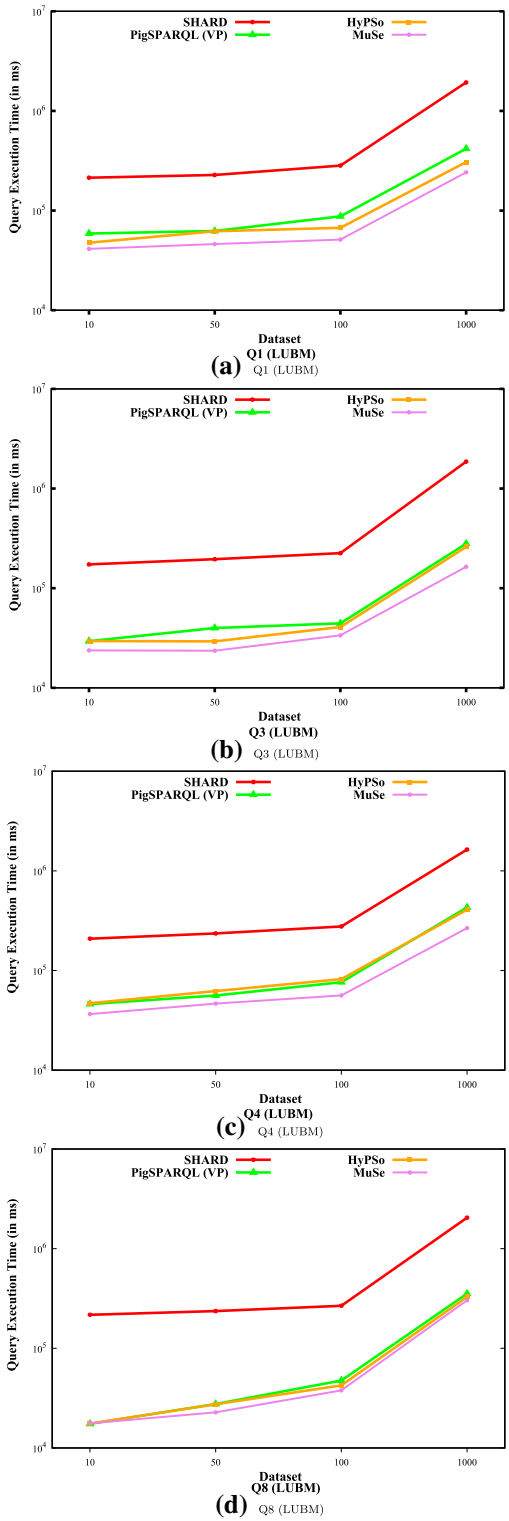


Fig. 4 Data Scalability on LUBM datasets for queries Q1, Q3, Q4 and Q8 respectively

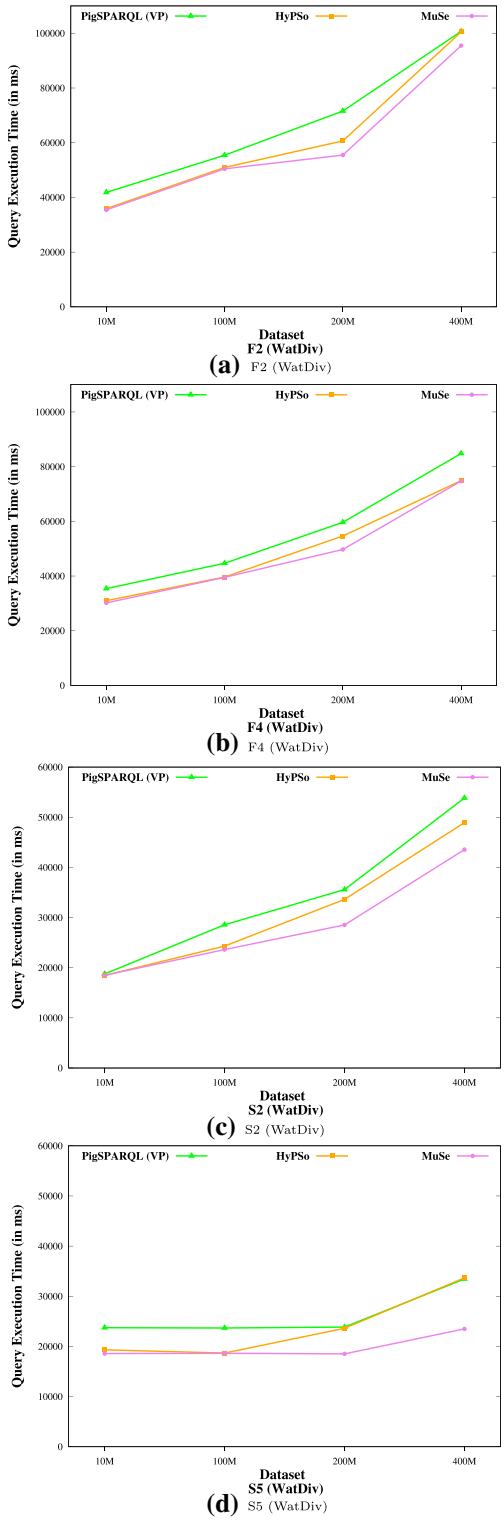


Fig. 5 Data Scalability on WatDiv datasets for queries F2, F4, S2 and S5 respectively

Appendix

LUBM Queries

```
Q2:
SELECT ?X ?Y1 ?Y2 ?Y3
WHERE {
        ?X rdf:type ub:Professor .
        ?X ub:worksFor> <http://www.Department0.
           University0.edu> .
        ?X ub:name> ?Y1 .
        ?X ub:emailAddress ?Y2 .
        ?X ub:telephone ?Y3 .}
```

```
Q3:
SELECT ?X ?Y ?Z
WHERE {
        ?Y ub:subOrganizationOf <http://www.
           University0.edu> .
        ?Y rdf:type ub:Department .
        ?X ub:memberOf ?Y .
        ?X rdf:type ub:Student .
        ?X ub:emailAddress ?Z .}
```

```
Q4:
SELECT ?X ?Y ?Z
WHERE {
         ?X rdf:type ub:Student .
         ?X ub:advisor ?Y .
         ?Y rdf:type ub:Faculty .
         ?X ub:takesCourse ?Z .
         ?Y ub:teacherOf ?Z .
         ?Z rdf:type ub:Course .}
```

```
Q5:
SELECT ?X ?Y
WHERE {
        ?Y ub:subOrganizationOf <http://www.
           University0.edu> .
        ?Y rdf:type ub:Department .
        ?X ub:worksFor ?Y .
        ?X rdf:type ub:Chair .}
```

```
Q6:
SELECT ?tAsst ?teacher ?course
WHERE {
        ?tAsst ub:advisor ?teacher.
        ?teacher ub:teacherOf ?course.
        ?tAsst rdf:type ub:TeachingAssistant.
        ?tAsst ub:teachingAssistantOf ?course.}
```

```
Q7:
SELECT ?X ?Y1 ?Y2 ?Y3
WHERE {
        ?X rdf:type ub:FullProfessor.
        ?X ub:worksFor <http://www.Department0.
           University0.edu>.
        ?X ub:name ?Y1.
        ?X ub:emailAddress ?Y2.
        ?X ub:telephone ?Y3.}
```

```
Q8:
SELECT ?Y ?course ?eMail ?name ?phone
WHERE {
        ?Y ub:telephone ?phone.
        ?Y ub:name  ?name.
        ?Y rdf:type ub:AssistantProfessor.
        ?Y ub:teacherOf ?course.
        ?Y ub:emailAddress ?eMail.
        ?Y ub:worksFor <http://www.Department0.
           University0.edu>.}
```

```
Q9:
SELECT ?Y ?course ?name ?degree ?tel ?unv
WHERE {
        ?Y ub:name ?name.
        ?Y ub:emailAddress ?eMail.
        ?Y ub:telephone ?tel.
        ?Y ub:teacherOf ?course.
        ?Y ub:undergraduateDegreeFrom ?uDegree.
        ?Y ub:mastersDegreeFrom ?mDegree.
        ?Y ub:doctoralDegreeFrom ?dDegree.
        ?Y ub:worksFor ?unv.
        ?Y ub:researchInterest ?interest.
        ?Y rdf:type ub:FullProfessor.}
```

```
Q10:
SELECT ?Y ?course ?name ?degree ?tel ?unv
WHERE {
        ?Y ub:name ?name.
        ?Y ub:emailAddress ?eMail.
        ?Y ub:takesCourse ?course.
        ?Y ub:telephone ?tel.
        ?Y ub:memberOf ?unv.
        ?Y ub:undergraduateDegreeFrom ?degree.
        ?Y ub:advisor ?advisor.
        ?Y rdf:type ub:ResearchAssistant.}
```

Acknowledgements
We sincerely thank the reviewers and the Editor for their valuable suggestions.

Authors' contributions
TC and ESP designed the study. TC performed the experiments in addition to writing the manuscript. All authors reviewed and edited the manuscript. All authors read and approved the final manuscript.

Funding
No funds have been received from any agency for this research.

Availability of data and materials
The data used to support the finding of this study are available from the corresponding author upon request.

Declarations

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Received: 25 May 2021 Accepted: 12 September 2021
Published online: 09 October 2021

References

1. Gandon F. A survey of the first 20 years of research on semantic Web and linked data. Revue des Sciences et Technologies de l'Information-Série ISI: Ingénierie des Systèmes d'Information. 2018.
2. Hassanzadeh O. Introduction to Semantic Web Technologies & Linked Data. University of Toronto. 2011.
3. Shah U, Finin T, Joshi A, Cost RS, Matfield J. Information retrieval on the semantic web. In: Proc. of the Eleventh International Conference on Information and Knowledge Management, McLean Virginia, USA, pp. 461–68. 2002.
4. Prasad JR, Shelke PM, Prasad RS. Semantic Web Technologies. Cham: Springer; 2021. pp. 35–57.
5. Santana LHZ, Mello RDS. Persistence of RDF Data into NoSQL: A Survey and a Unified Reference Architecture. IEEE Transactions on Knowledge and Data Engineering. 2020; pp. 1–20.
6. Cardoso J, Sheth A. The Semantic Web and its applications. In: Semantic Web Services. Processes and Applications. Cham: Springer; 2006. pp. 3–33.
7. Chawla T, Singh G, Pilli ES, Govil M. Storage, partitioning, indexing and retrieval in Big RDF frameworks: a survey. Computer Sc Rev. 2020;38: pp. 1–41.
8. Chawla T, Singh G, Pilli ES. JOTR: Join-Optimistic Triple Reordering Approach for SPARQL Query Optimization on Big RDF Data. In: 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT), Bengaluru, India, pp. 1–7, 2018, IEEE.
9. Chawla T, Singh G, Pilli ES. HyPSo: Hybrid Partitioning for Big RDF Storage and Query Processing. In: Proceedings of the ACM India Joint International Conference on Data Science and Management of Data, Kolkata, India. ACM; 2019. pp. 188–94.
10. Wylot M, Hauswirth M, Cudré-Mauroux P, Sakr S. RDF data storage and query processing schemes: A survey. ACM Computing Surveys (CSUR). 2018;51(4):1–36.
11. Bouchelouche K, Ghomari AR, Zemmouchi-Ghomari L. Open Government Data (OGD) Publication as Linked Open Data (LOD): A Survey. Open Government. 2021;10:1.
12. Ji S, Pan S, Cambria E, Marttinen P, Philip SY. A Survey on Knowledge Graphs: Representation, Acquisition, and Applications. In: IEEE Transactions on Neural Networks and Learning Systems. 2021; pp. 1–27.

13. Kulcu S, Dogdu E, Ozbayoglu AM. A survey on semantic web and big data technologies for social network analysis. In: IEEE International Conference on Big Data (Big Data), Washington DC, USA. 2016. pp. 1768–1777.
14. Zhang F, Lu Q, Du Z, Chen X, Cao C. A comprehensive overview of RDF for spatial and spatiotemporal data management. The Knowledge Engineering Review. 2021. pp. 1–36.
15. Cheng L, Kotoulas S. Scale-out processing of large RDF datasets. IEEE Trans Big Data. 2015;1(4):138–50.
16. Pan Z, Zhu T, Liu H, Ning H. A survey of RDF management technologies and benchmark datasets. J Ambient Intelligence Humanized Computing. 2018;9(5): pp. 1693–704.
17. Mazumdar S, Scionti A. Fast execution of RDF queries using Apache Hadoop, pp. 1–33. Elsevier: Amsterdam. 2020.
18. Graux D, Jachiet L, Geneves P, Layaïda N. SPARQLGX: Efficient distributed evaluation of sparql with apache spark. In: The 15th International Semantic Web Conference (ISWC), Kobe, Japan. Springer; 2016. pp. 80–87.
19. Abadi DJ, Marcus A, Madden SR, Hollenbach K. SW-Store: a vertically partitioned DBMS for Semantic Web data management. VLDB J. 2009;18(2):385–406.
20. Hassan M, Bansal SK. RDF Data Storage Techniques for Efficient SPARQL Query Processing Using Distributed Computation Engines. In: International Conference on Information Reuse and Integration for Data Science (IRI), Salt Lake City, USA, 2018. pp. 323–30.
21. Schätzle A, Przyjaciel-Zablocki M, Neu A, Lausen G. Sempala: interactive SPARQL query processing on hadoop. In: International Semantic Web Conference, Riva del Garda, Italy, Springer; 2014. pp. 164–79.
22. Ranichandra Dharmaraj C, Tripathy B. Adaptive mechanism for distributed query processing and data loading using the RDF data in the cloud. Int J Commun Syst. 2018;31(15):1–12.
23. Punnoose R, Crainiceanu A, Rapp D. SPARQL in the cloud using Rya. Inform Syst. 2015;48: 181–95.
24. Rohloff K, Schantz RE. High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store. In: Programming Support Innovations for Emerging Distributed Applications, Reno, Nevada, ACM; 2010. pp. 1–4.
25. Cossu M, Färber M, Lausen G. Prost: Distributed execution of sparql queries using mixed partitioning strategies. In: 21st International Conference on Extending Database Technology (EDBT), Vienna, Austria, ACM; 2018. pp. 1–5.
26. Schätzle A, Przyjaciel-Zablocki M, Lausen G. PigSPARQL: Mapping SPARQL to pig latin. In: Proc. of the International Workshop on Semantic Web Information Management, Athens, Greece, ACM; 2011. pp. 1–4.
27. Chawla T, Singh G, Pilli ES, Govil M (2016) Research issues in RDF management systems. In: International Conference on Emerging Trends in Communication Technologies (ETCT), Dehradun, India, IEEE, pp. 1–5
28. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Commun ACM. 2008;51(1):107–13.
29. Guo Y, Pan Z, Heflin J. LUBM: A benchmark for OWL knowledge base systems. J Web Semantics. 2005;3(2–3):158–82.
30. Aluç G, Hartig O, Özsu MT, Daudjee K. Diversified stress testing of RDF data management systems. In: International Semantic Web Conference, Riva del Garda, Italy, Springer; 2014. pp. 197–212.

Publisher's Note