Journal of Web Engineering, Vol. 12, No.1&2 (2013) 159-180

SCALABLE RDF GRAPH QUERYING USING CLOUD COMPUTING

REN LI

College of Computer Science, Chongqing University, Chongqing, China
renli@cqu.edu.cn

DAN YANG HAIBO HU JUAN XIE LI FU

School of Software Engineering, Chongqing University, Chongqing, China
{dyang, hbhu, xiejuan, fuli}@cqu.edu.cn

Received June 27, 2012
Revised January 6, 2013

With the explosion of the semantic web technologies, conventional SPARQL processing tools do not scale well for large amounts of RDF data because they are designed for use on a single-machine context. Several optimization solutions combined with cloud computing technologies have been proposed to overcome these drawbacks. However, these approaches only consider the SPARQL Basic Graph Pattern processing, and their file system-based schema can barely modify large-scale RDF data randomly. This paper presents a scalable SPARQL Group Graph Pattern (GGP) processing framework for large RDF graphs. We design a novel storage schema on HBase to store RDF data. Furthermore, a query plan generation algorithm is proposed to determine jobs based on a greedy selection strategy. Several query algorithms are also presented to answer SPARQL GGP queries in the MapReduce paradigm. An experiment on a simulation cloud computing environment shows that our framework is more scalable and efficient than traditional approaches when storing and retrieving large volumes of RDF data.

Key words: Semantic Web, RDF, SPARQL, Cloud Computing, MapReduce, HBase
Communicated by: M. Gaedke & O. Pastor

1 Introduction

To achieve the Semantic Web vision [1], several standards were recommended by the World Wide Web Consortium (W3C) to make Web information understandable to humans and machines [2]. Among these standards, the Resource Description Framework (RDF) is the prominent data model for storing and representing information about the Semantic Web [3]. In addition, the W3C recommends SPARQL as the standard RDF query language to extract RDF information [4].

Large volumes of RDF data became available with the development of semantic web technologies. As of September 2011, the number of RDF triples contained in the Link Open Data cloud is over 31 billion [8]. However, most of the existing RDF query tools [9, 10, 11] and optimization solutions [12, 13, 14] have inevitable limitations in performance and scalability when handling large amounts of RDF data because they are designed to run on a centralized environment where the computations are

performed by a single machine. Therefore, using the distributed approach to implement scalable RDF data storage and retrieval is especially important and challenging.

Cloud computing technologies receive comprehensive attention from the IT industry and the academia. As of this writing, MapReduce paradigm [15], which is built on top of the Google File System [16], is the dominant parallel and distributed programming paradigm in the cloud computing community because of its high performance and fault-tolerant capability [17]. Apache also implements MapReduce in the Hadoop open-source framework, which is successfully applied to solve data-intensive problems in various domains [18, 19].

Researchers in the semantic web community are focused on solving the scalability and performance problems of traditional semantic web tools by exploiting cloud computing technologies. For instance, Urbani et al. propose WebPIE [20] as a parallel inference engine to implement scalable RDFS and OWL [21] reasoning based on the MapReduce paradigm. Mutharaju et al. [22] provide an efficient algorithm for the classification of OWL 2 EL ontologies in MapReduce. The two works show that cloud computing technologies can greatly benefit the semantic web area.

Researchers also present a few SPARQL query processing frameworks based on cloud computing technologies to meet the storage and querying requirements of large volumes of RDF data [23, 24]. Results show that their approaches are more scalable and efficient than conventional tools. Nevertheless, all of the previous solutions merely involve the SPARQL Basic Graph Pattern (BGP) processing. Furthermore, the Hadoop distributed file system (HDFS) is used as the RDF storage repository, which can cause a bottleneck in the random modification of RDF data when SPARQL 1.1 [7] queries are answered. Although some researchers attempted to adopt HBase as the RDF repository, which can provide the arbitrary read/write of RDF data in real time [25, 26, 27], their proposed schemas should be improved to find a better trade-off between performance and storage space.

This paper proposes a scalable RDF graph query processing framework based on cloud computing technologies. By running empirical experiments with standard benchmarks and queries, we demonstrate that this framework achieves better performance and scalability than leading state-of-the-art RDF query tools when matched against large amounts of RDF data. This work provides the first solution for complex SPARQL Group Graph Pattern (GGP) processing using MapReduce and HBase.

The main contributions of this paper are threefold. First, to store RDF data on HBase, we design a novel schema that can achieve effective trade-off between storage space and query performance. Second, we present an efficient algorithm based on a greedy selection strategy to determine the query plan. Third, several MapReduce query algorithms are proposed to answer complex SPARQL GGP queries that involve the AND, UNION, and OPTIONAL operators as well as some FILTER restrictions and solution sequence modifiers.

The remainder of this paper is organized as follows. Section 2 presents a brief review of related works. Section 3 describes the architecture of the entire framework, and then defines the storage schema and data models that are used in our framework. Section 4 discusses how we determine the query plan and how we execute query jobs in the MapReduce paradigm. Section 5 presents the analysis of the experimental results. Finally, Section 6 summarizes the conclusions and the future works.

2 Related Works

MapReduce is a framework for the parallel and distributed processing of data in a cluster of commodity machines [16]. In MapReduce, data are converted into key–value pairs and processed in a job that consists of a map phase and a reduce phase. A batch of jobs can also be chained to deal with complex computation tasks. Within one job, the Master node first partitions input data into a number of independent chunks and passes them to the Map nodes. Afterwards, in the map phase, each Map node accepts data chunks and then generates a series of intermediate key–value pairs according to a user-defined map function. Finally, after being notified about the location of intermediate data in the reduce phase, each Reduce node merges the intermediate data with the same key value and generates a series of key–value pairs according to a user-defined reduce function.

As an open-source implementation of Google's Bigtable [28], HBase stores data in tables that can be described as a multidimensional sorted map. Each data row of an HBase table is composed of a unique row key and an arbitrary number of columns, where several columns can be grouped as a column family. A data cell, which is determined by a given row key and a column name, can store multiple versions of data distinguished by a timestamp. In addition, HBase provides a B+ tree-like index on row key, and data retrieval can be implemented by giving a row key or the range of keys.

Pérez et al. [5] describe the algebraic formalization for RDF and SPARQL graph patterns in order to analyze the semantics and complexity of SPARQL queries. They also provide the compositional semantics for binary operators AND, OPTIONAL, UNION, and FILTER. In this paper, we adopt the syntax and computational semantics of SPARQL described in their work.

Several RDF query tools are comprehensively utilized in the semantic web community. Jena [9] is a Java-implemented open source programmatic environment for SPARQL. It includes the ARQ query engine and provides an efficient access to the RDF data set in memory. Furthermore, for persistent RDF triple storage, researchers additionally provide the Jena SDB model, which is built on relational databases. Sesame [10] is a generic architecture for storing and querying large quantities of RDF data, which supports SPARQL and SeRQL queries. Based on its architecture feature, Sesame can be ported to a large variety of different repositories such as relational databases, RDF triple stores, and remote storage services on the Web. RDF-3X [11] is an RISC-style engine for RDF and is considered as the fastest SPARQL query tool. It uses histograms, summary statistics, and query optimization to enable high-performance RDF queries. However, the current version of RDF-3X does not support SPARQL Alternative Graph Pattern processing.

Few studies deal with the scalability and performance problems of centralized RDF query tools by exploiting cloud computing technologies because it is still an emerging research area. Choi et al. [27] propose a query processing system named SPIDER for RDF data based on HBase and MapReduce. They provide an overview of the system architecture and describe several RDF graph query approaches using MapReduce. However, the schema for storing RDF triples in HBase and the detail query algorithms are not presented and no experimental results are reported.

Myung et al. [23] design a universal and efficient MapReduce algorithm for SPARQL BGP processing. They adopt a greedy strategy to select the join key of SPARQL BGP queries and apply the multi-way join method into MapReduce to avoid unnecessary job iterations because running multiple

jobs are computationally expensive in MapReduce. Their approach shows superior performance and scalability in terms of time and data size compared with conventional RDF query tools.

Husain et al. [24] describe a heuristics-based query processing framework for large RDF graphs. They design a novel schema to store RDF data in HDFS as flat files and define several models to represent RDF triples, SPARQL queries, and related terms in their system. A Relaxed-Bestplan algorithm, which has a worst case that is bounded by the log of the total number of variables, is proposed to determine the query plan. The experiments demonstrate that their framework is highly scalable and efficient. However, they only consider the SPARQL BGP processing. The file system-based storage schema cannot meet the random modification demands of RDF data. This paper extends their idea of generating query plan to answer SPARQL GGP queries in MapReduce.

Based on the idea of Hexaxtore [29], Sun et al. [26] propose an RDF storage schema on HBase and present a MapReduce join algorithm for SPARQL BGP processing. They build six HBase tables for RDF data storage to cover all possible combinations of SPARQL triple patterns. Moreover, a greedy strategy is adopted to select the join key. However, RDF data must be replicated six times in their solution, which requires additional storage space and makes data modification difficult.

To compare the performance of cloud computing technologies and traditional relational database cluster technologies for distributed RDF data management, Franke et al. [25] design an RDF data storage schema with two HBase tables and propose query algorithms to evaluate SPARQL queries in MapReduce. Their results show that the HBase solution can deal with a larger RDF data set, and has superior query performance and scalability compared with the MySQL cluster. However, they only consider the SPARQL BGP matching.

Several proposed benchmarks are widely used to test the performance and scalability of semantic web data query tools and reasoners, including the Lehigh University Benchmark (LUBM) [30], the SP2Bench [31], and the Berlin SPARQL Benchmark (BSBM) [32]. These benchmarks contain standard test SPARQL queries and data generators to arbitrarily create an RDF data set. In this paper, we choose SP2Bench as the running example and experimental data set because it is the only benchmark for complex SPARQL GGP testing.

3 The Proposed Framework

First, we give an overview of the architecture of our proposed framework. We then describe a novel RDF data storage schema on HBase and define several data models and terms that will be used.

3.1 Architecture of the proposed framework

As in figure 1, our framework is built on the top of the Hadoop cluster and consists of three modules: Query Plan Generator, Data Adapter, and Query Engine.

The Data Adapter module, which runs in the MapReduce environment, is an interface for accessing HBase. The Data Adapter takes the N-Triple format-based RDF data and stores these into HBase tables in parallel before answering SPARQL queries. The Query Engine calls the Data Adapter to retrieve the RDF data from HBase and then stores the solutions into multiple HDFS files after the query plan is determined.

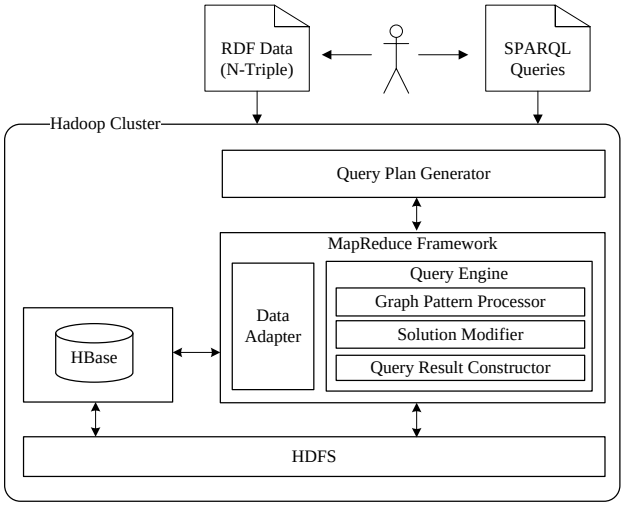


Figure 1 Architecture of the proposed framework.

The Query Plan Generator is responsible for determining the query plan. The SPARQL queries submitted by users within this module are parsed and converted into a Graph Pattern Tree (GPT) model, a Sequence Solution Modifier (SSM) model, and a Query Form (QF) model. The Query Plan Generator generates a series of MapReduce Query Plan models based on the query plan generation algorithm, which will be proposed in Section 4.1. The details of the data models will be discussed in Section 3.3.

The Query Engine Module comprises three sub-modules: Graph Pattern Processor, Solution Modifier, and Query Result Constructor. The Graph Pattern Processor runs a sequence of MapReduce jobs to match the graph patterns and generate a set of solutions according to the query plan. The Solution Modifier creates another ordered sequence solution if there are one or more solution modifiers in the given SPARQL query. The Query Result Constructor generates the final results according to the query form.

In addition to the AND, OPTIONAL, and UNION operators and some commonly used FILTER restrictions like <, >, !=, and so forth, modifiers such as Order, Distinct, and the Select query form can be handled in our framework.

3.2 Storage Schema

As the building block of an RDF graph, each RDF triple (*s*, *p*, *o*) describes a relationship between two resources, where *s* is the subject, *p* is the predicate, and *o* is the object. Each component of an RDF triple can be represented as an IRI, blank node, or literal, where the IRI consists of a prefixed namespace and a local part. The prefixed namespace, for example, *<http://www.w3.org/1999/02/22-rdf-syntax-ns#>*, can also be denoted as *rdf*: for short.

Finding the trade-off between storage space and querying efficiency is a crucial issue for designing data schema in database research. Based on the features of HBase and to cover all possible

combinations of SPARQL triple patterns, we build three HBase tables, namely, T_SP_O, T_PO_S, and T_OS_P, to support the efficient retrieval of RDF data. Furthermore, to reduce the amount of storage space, table T_Pre_N is used to store the prefixed namespaces of IRIs and the corresponding abbreviations. Figure 2 shows the storage schema, where S denotes the subject, P denotes the predicate, and O denotes the object.

RowKey	Column Family				Timestamp
	O(1)	O(2)	...	O(N)	
<S, P>	null	null	...	null	Time T

(a) Table T_SP_O

RowKey	Column Family				Timestamp
	S (1)	S (2)	...	S (N)	
<P, O>	null	null	...	null	Time T

(b) Table T_PO_S

RowKey	Column Family				Timestamp
	P (1)	P (2)	...	P (N)	
<O, S>	null	null	...	null	Time T

(c) Table T_OS_P

RowKey	Column Family	Timestamp
	Abbreviation	
Prefixed Namespace	*Data*	Time T

(d) Table T_Pre_N

Figure 2 Storage schema of the proposed framework.

Table T_SP_O employs the subject–predicate pair as the row key and the corresponding objects as the column names. All columns belong to one column family and all data cells are left empty. Tables T_PO_S and T_OS_P have a similar structure as T_SP_O. The predicate–object and object–subject pair is used as the row key of these two tables. The corresponding subject and predicate values are stored as the column names. Table T_Pre_N contains one column named Abbreviation, the prefixed namespace of IRI is stored as the row key, and the matching abbreviation is stored in data cells.

Although our proposed schema requires that RDF data be replicated twice, storage space is not a critical problem in the cloud computing context. All the possible forms of triple patterns can be directly and efficiently matched through our schema. In addition, all values stored in these three tables are the abbreviations of the namespaces rather than the full IRI string, and no space is wasted for the blank cell in the HBase table. Therefore, our schema can achieve a satisfactory trade-off between space and performance.

Based on the triple structure, there are 2^3 possible combinations of triple patterns. In table 1, we present the querying relationship between HBase tables and the different types of triple patterns.

HBase Table	**Triple Patterns**
T_SP_O	(s, p, ?o), (s, ?p, ?o), (s, p, o), (?s, ?p, ?o)
T_PO_S	(?s, p, o), (?s, p, ?o)
T_OS_P	(s, ?p, o), (?s, ?p, o)

Table 1 The querying relationship between HBase tables and triple patterns.

To handle the triple patterns (?s, p, o), (s, ?p, o), and (s, p, ?o), the two known terms can be set as retrieval conditions to match the row keys of the T_PO_S, T_OS_P, and T_OS_P tables. For the triple pattern (s, p, o), we need to verify whether the triple exists in the T_SP_O table, whereas the triple

pattern (?s, ?p, ?o) requires that the entire RDF data set is loaded. The triple patterns (?s, ?p, o), (?s, p, ?o), and (s, ?p, ?o) can also be handled using the PrefixFilter technique of HBase, which requires a range of row keys for the T_OS_P, T_PO_S, and T_SP_O tables.

Based on the proposed schema, all possible forms of RDF data from different data sources can be transformed and stored in HBase tables by utilizing the Data Adapter. In addition, we implement a transaction mechanism in the Data Adapter which can be considered as an uniform data access layer or middleware. All possible operations, such as querying, inserting, updating or deleting RDF data from proposed HBase tables, are encapsulated and integrated in one transaction procedure. Therefore, although our proposed schema is based on replicating the information in several HBase tables, the data integration problem which is often derived from using this strategy can be addressed.

3.3 Model Definitions

Before discussing the query plan generation algorithm and MapReduce query algorithms in detail, we introduce several model definitions in this section. Query 8 in SP2Bench is used as a running example to illustrate these terms better. We show Query 8 in figure 3 and mark these triple patterns from TP_1 to TP_8 according to their order of appearance. In particular, TP_3 and TP_4 are involved twice in Query 8.

```
SELECT DISTINCT ?name
WHERE {
  ?erdoes rdf:type foaf:Person. (TP1)
  ?erdoes foaf:name "Paul Erdoes"^^ xsd:string. (TP2)
  {
    ?document dc:creator ?erdoes. (TP3)
    ?document dc:creator ?author. (TP4)
    ?document2 dc:creator ?author. (TP5)
    ?document2 dc:creator ?author2. (TP6)
    ?author2 foaf:name ?name (TP7)
    FILTER (?author!=?erdoes &&?document2!=?document
              && ?author2!=?erdoes && ?author2!=?author) (R1)
  } UNION{
    ?document dc:creator ?erdoes. (TP3)
    ?document dc:creator ?author. (TP4)
    ?author foaf:name ?name (TP8)
    FILTER (?author!=?erdoes) (R2)
  }
}
```

Figure 3 Query 8 in SP2Bench.

As the query pattern part of one SPARQL query can be translated into a relational operator tree [6], three models are designed in our proposed framework to represent the query pattern, solution sequence modifier, and query form part of one SPARQL query, respectively.

Definition 1. Graph Pattern Tree (GPT). A GPT is a tree structure model that represents the query pattern in the *where* clause of a given SPARQL query. Each tree node can be the Triple Pattern Node (TPN), the Tuple Node (TN), or the Operator Node (OPN), which are defined as follows:

- TPN is a triple (s, p, o) that corresponds to one triple pattern in the given SPARQL query. Each TPN component can be a variable or concrete value.
- TN is a tuple (t_1, t_2, ..., t_n), $n \geq 1$, that represents the variables in a matching solution, where t_i is a variable name, $1 \leq i \leq n$. Each TN has an attribute *JobN* that is used to record the job number of this TN.
- OPN corresponds to the AND, UNION, or OPTIONAL operators, and it can attach one or more FILTER constraint objects. We say an OPN is a leaf OPN when all child nodes of this OPN are TPNs, and we say the depth of the GPT is the maximum depth number of leaf OPNs.

Definition 2. Solution Sequence Modifier (SSM). An SSM=(Modifier, Parameters) corresponds to the solution sequence modifier part of a SPARQL query. The Modifier can be the operators Distinct, Order By, Projection, Reduced, Offset, or Limit. The Parameters denote the operational objectives of the Modifier.

Definition 3. Query Form (QF). A QF is an operator that uses the solutions from pattern matching to form the result sets or RDF graphs; it can be Select, Construct, Ask, or Describe.

Based on these definitions, each triple pattern contained in Query 8 corresponds to a TPN. An SSM model can also be defined as (Distinct, ?*name*). The QF model for Query 8 is Select. Figure 4 shows the GPT model of Query 8, where the TPNs are represented as cycles and OPNs as rectangles. Therefore, we can denote Query 8 as the tuple [GPT, (Distinct, ?*name*), Select].

The HBase can be easily accessed eight times to match the RDF terms for these triple patterns. However, this approach inevitably leads to unnecessary I/O transfers. This approach is time-consuming because accessing the HBase in multiple MapReduce jobs is computationally expensive. Hence, a more efficient data retrieval solution is in demand.

Definition 4. Query Pattern (QP). For one GPT model, if there are n TPNs, $TPN_1=(s_1, p_1, o_1)$, ..., $TPN_n=(s_n, p_n, o_n)$, $n \geq 2$, satisfying the subjects, predicates and objects are variables or share common concrete values of V_s, V_p, and V_o. A QP is a triple (s, p, o) used to denote the common RDF retrieval information of TPN_1, ..., TPN_n, where s, p, and o are assigned to *Var* when the subjects s_1, ..., s_n, predicates p_1,..., p_n and objects o_1, ..., o_n are variables; otherwise, s, p, and o are assigned to V_s, V_p, and V_o, respectively.

As the subjects and objects of TPN_3, TPN_4, TPN_5, and TPN_6 are variables in Query 8, the predicates share same the concrete value *dc:creator*. Therefore, a QP model is defined as QP_1=(*Var*, *dc*:*creator*, *Var*). Likewise, for TPN_7 and TPN_8, another QP model can be defined as QP_2=(*Var*, *foaf*:*name*, *Var*). By using QP models, we can set the concrete values as retrieval conditions to match the RDF triples from HBase. For example, we can use QP_1 to retrieve all RDF terms for TPN_3, TPN_4, TPN_5, and TPN_6 in one job, rather than access HBase four times. However, as no QP model can be defined for TPN_1 and TPN_2, the RDF data must still be retrieved separately.

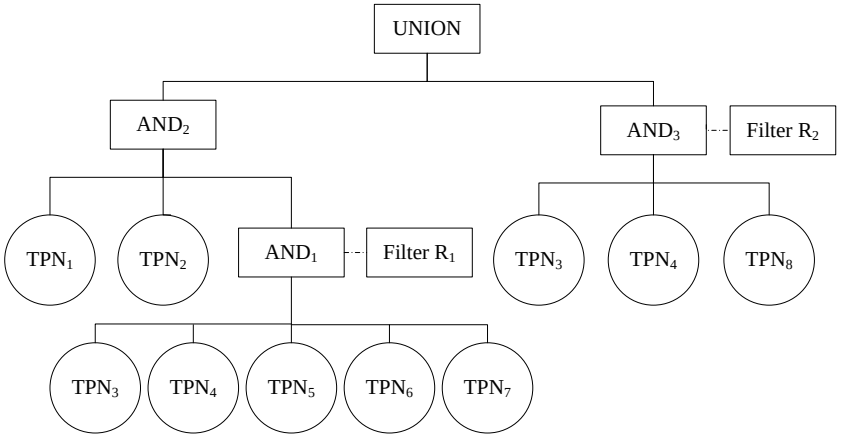


Figure 4 GPT model of Query 8.

Definition 5. Shared Variable (SV). A shared variable is the common variable in two or more TPNs or TNs belonging to one OPN.

Definition 6. MapReduce Query Plan (MRQP). An MRQP is a tuple (*JobID*, *Opt*, *SV*, *Tag*, *Flt*), where *JobID* denotes the job sequence number; *Opt* can be the AND, OPTIONAL, or UNION operators; *Tag* is a collection of target TPNs or TNs; *SV* is the shared variable in *Tag*; and *Flt* is the FILTER constraint attached to *Opt*.

In Query 8, variable ?*document* is the shared variable for TPN_3 and TPN_4 while variable ?*author* is the shared variable for TPN_4 and TPN_5. On the contrary, the variable ?*name* existing in TPN_7 and TPN_8 is not the shared variable because these two TPNs belong to different OPNs. Moreover, suppose the TPN_4 and TPN_5 are determined to be handled in the first job, we can define a MRQP model as (1, *AND*, ?*author*, {TPN_4,TPN_5}, R_1). After passing this MRQP model to the Query Engine, the Map nodes take the RDF data matched to TPN_4 and TPN_5 as input and generate a series of intermediate key–value pairs in which the binding of ?*author* is set as key. The intermediate data with the same key values are subsequently passed to one Reduce node, and the AND and FILTER operations are executed in reduce function.

4 MapReduce-based Query Processing

In this section, we first describe how to handle the AND, OPTIONAL, and UNION operators in MapReduce paradigm and how we determine the query plan. Subsequently, we propose several algorithmic solutions to answer SPARQL GGP queries in MapReduce.

4.1 Query Plan Generation Algorithm

Two or more shared variables contained in one TPN or TN cannot be a key at the same time because each MapReduce job processes chunks of data in the key–value pair format. Additionally, a batch of MapReduce jobs can barely be avoided to answer complex SPARQL GGP queries. The number of jobs should also be reduced to achieve higher performance because running multiple MapReduce jobs is

computationally expensive. In this section, we propose our query plan generation algorithm, which is extended from the Relaxed-Bestplan [24].

Based on the syntax and semantics of SPARQL, the UNION operator combines two or more GGPs so that one of several alternative graph patterns may match; if more than one of the alternatives match, then all of the possible pattern solutions are found. In MapReduce, one UNION operator can be handled in one job. For example, to answer the expression {(?*a*, ?*b*, ?*c*)} UNION {(?*a*, ?*b*, ?*d*)}, the map function takes the RDF data that match two triple patterns and generates a series of solutions for the tuple (?*a*, ?*b*, ?*c*, ?*d*). In this case, even the reduce phase is unnecessary.

Similarly, the OPTIONAL operator, which adds optional solutions for one graph pattern, can be handled in one MapReduce job. For instance, suppose there are two GGPs, P_1={(?*a*, b, ?*c*)} and P_2={(?*a*, d, ?*d*)}. To answer the optional pattern expression P_1 OPTIONAL P_2, the map function primarily takes the RDF terms that match either P_1 or P_2 and generates a series of key–value pairs. For P_1 and P_2, the key part of each intermediate data consists of the shared variable ?*a* and its matching term. The variables ?*c* and ?*d* with their matching values are assigned to the value part of P_1 and P_2, respectively. Reduce nodes accept the intermediate data with the same key and generate several optional solutions for (?*a*, ?*c*, ?*d*) in the reduce phase.

Husain et al. [24] propose the Relaxed-Bestplan algorithm to determine the query plan for processing SPARQL BGP in MapReduce. They prove that an exponential number of join cases have to be faced since more than one job is needed to answer complex SPARQL BGP queries, and gathering summary statistics for a large number of cases would be very much time and space consuming. In addition, they observe that several disk I/O and network transfers are expensive operation for running a job in MapReduce. Therefore, they follow the idea of greedy selection strategy and design the Relaxed-Bestplan algorithm to find the job plan that has the minimum number of jobs.

The idea of the Relaxed-Bestplan algorithm is followed to handle the AND operator for GGPs because handling it for multiple GGPs requires the same computational semantics with BGP. Therefore, for one AND operator that contains N triple patterns or tuples and K shared variables, the generated query plan contains at most J jobs, where

$$J = \begin{cases} 0 & N = 0, \\ 1 & N = 1 \text{ or } K = 1, \\ \min([1.71\log_2 N], K), & N, K > 1. \end{cases} \tag{1}$$

In addition, the FILTER operators restrict the solutions of a graph pattern match according to a given expression, and the sequence solution modifiers create another sequence for an unordered collection of solutions. Based on the key–value pair model of MapReduce, FILTER operators and sequence solution modifiers for GGPs can be handled in the map phase or reduce phase after the solutions are generated. Therefore, we can summarize that for handling one OPN, SJ jobs are required, where

$$SJ = \begin{cases} 1, \text{ OPN is OPTIONAL or UNION}, \\ J, \text{ OPN is AND}. \end{cases} \tag{2}$$

The OPNs can be easily and sequentially processed. Taking Query 8 as an example, based on Formula (1), handling the operator AND_1 for TPN_3, TPN_4, TPN_5, TPN_6, and TPN_7 requires three jobs; AND_2 for TPN_3, TPN_4, and TPN_8 needs two jobs; AND_3 for TPN_1, TPN_2, and the results of AND_1 can be answered in one job. The root UNION operator needs one job to obtain the final results. Therefore, we need to run seven jobs to get the results if we handle these four OPNs sequentially. However, this querying strategy is clearly inefficient.

As an alternative, a greedy selection strategy can be adopted to perform as many operations as possible in each job since the AND_1, AND_2, and AND_3 process the triple patterns independently. For example, in the first three jobs, in addition to processing AND_1, we can execute the joining operation for AND_2 simultaneously. Then, the joining operation for AND_3 and the final UNION operation can also be handled. Therefore, the entire query plan needs to perform MapReduce jobs five times. Based on this greedy selection strategy, we propose a novel query plan generation algorithm for SPARQL GGP processing as outlined in figure 5.

Input: a GPT model *T*. Output: a collection of MRQP models *QPlan*.
1. Initialization *QPlan* ←∅, *L*←Depth of *T*
2. For *a*=*L* to 1
3. *N*={OPN_1, ..., OPN_n}←All OPNs in *a*-th level
4. For (*i*=1 to *n*)
5. *CN*={$Node_1$, ..., $Node_m$}←All child nodes of OPN_i
6. If OPN_i is a leaf OPN, do *t*←1, Else *t*←Max_job_Nums(CN)+1
7. *S*={sv_1,...,sv_s}←All SVs in CN sorted in non-decreasing order of E-count
8. If OPN_i is OPTIONAL
9. MRQP←(*t*, OPTIONAL, *S*, CN, FILTER), *QPlan* ← *QPlan* ∪{MRQP}
10. Else If OPN_i is UNION
11. MRQP←(*t*, UNION, null, CN, FILTER), *QPlan* ← *QPlan* ∪{ MRQP }
12. Else If OPN_i is AND
13. While(*CN* ≠ ∅)
14. For *k*=1 to *s* and if Can-Eliminate(*CN*, sv_k)
15. TP(*CN*, sv_k) ← All TPNs or TNs in *CN* containing variable sv_k
16. MRQP←(*t*, AND, sv_k, TP (*CN*, sv_k), FILTER), *QPlan*←*QPlan* ∪{MRQP}
17. Temp←Temp ∪ Join_Result (TP(*CN*, sv_k)), CN←CN−TP(*CN*, sv_k)
18. End For
19. *t*←*t*+1, *CN*←*CN*∪Temp
20. End While
21. End If
22. Replace OPN_i with new TN(*CN*, *t*)
23. End For
24. End For
25. Return *QPlan*

Figure 5 Query plan generation algorithm for SPARQL GGP processing.

The proposed algorithm takes one GPT model and generates a collection of MRQP models as the query plan. The algorithm starts by initializing an empty collection *QPlan* and assigning the depth of GPT to *L*. All of the OPNs are then evaluated iteratively from the bottom up. For each OPN in one level, if it is a leaf OPN, the current job number denoted as the parameter *t* is assigned to 1; otherwise, *t* equals 1 + the maximum job number of child nodes belonging to the current OPN. Subsequently, all child nodes of the current OPN are collected into *CN*. All the shared variables contained in the *CN* are sorted in a non-decreasing order of the E-count [24] and stored in collection *S*. Afterward, the query plan is generated according to the operators. As shown in Line 9, if the current OPN is OPTIONAL, then one MRQP model is built to denote the operations to be executed in a MapReduce job, where *t* is the job ID, *CN* is the target TPNs or TNs, and *S* is the shared variables in the *CN*. Similarly, if the OPN is UNION, one MRQP model is built and inserted into the *QPlan*, as shown in Line 11. If the OPN is AND, the Relaxed-Bestplan algorithm is modified to build the MRQP models iteratively. In the while loop starting from Line 13, if *CN* is not empty and the shared variable can be completely or partially eliminated, all corresponding TPNs or TNs in *CN* are picked up and stored in the collection *TP*. One MRQP model is then defined as shown in Line 16. Before iterating the next shared variable, the TPNs or TNs that have been handled are removed from *CN* and the intermediate computational results are stored in *Temp*. Until all shared variables are evaluated, both *t* and *CN* are updated before running the next iteration. After the query plan for such OPN is determined, as shown in Line 22, we replace this OPN with a new TN node built with all of the variables in *CN*. The *JobN* attribute is assigned to *t*. Finally, after all OPNs in every level are handled, the query plan *QPlan* is returned from the algorithm.

According to the MRQP models generated by our proposed algorithm, the querying procedure starts from each leaf OPN and terminates at the root OPN. Hence, taking each leaf OPN as a starting point, one corresponding path to the root exists. If two paths P_1 and P_2 intersect at one OPN *O*, and *O* needs *SJ* jobs, then PJ_1 and PJ_2 jobs are required for P_1 and P_2, respectively. The computation for *O* needs to wait until P_1 and P_2 complete the query tasks. Hence, the total number of jobs for handling *O* is *SJ* + maximum job number of P_1 and P_2.

Formally, giving one GPT model that contains *n* leaf OPNs, *n* paths P_1,..., P_n exist. Each path takes one leaf OPN as the starting point and one root OPN as the end point. Our proposed algorithm generates *SumJ* jobs to answer this GPT:

$$SumJ = \max(PJ_1,\ldots,PJ_n), \tag{3}$$

where PJ_i is the total job number for path P_i, $1 \le i \le n$. Moreover, if the depth of GPT is *L* and the leaf OPN of one path *P* is located in the *D*-th level of GPT, $D \le L$, then this path will take *PJ* jobs:

$$PJ = \sum_{j=D}^{1} SJ_j, \tag{4}$$

where *SJ* is the number of jobs for one OPN in the path defined as Formula (2).

Taking Query 8 as an example, two leaf OPNs exist: AND_1 and AND_2. Thus, two paths, P_1 = <AND_1, AND_3, UNION> and P_2 = <AND_2, UNION> can be handled at the same time. In addition, AND_1 needs three jobs, AND_2 requires two jobs, and AND_3 and the root UNION operator take one job, respectively. Thus, PJ_1=5 and PJ_2=3. The proposed algorithm requires five jobs to answer Query 8 as

SumJ=max(5, 3). Compared with the sequence querying approach that requires seven jobs, our query plan generation algorithm achieves better performance with fewer jobs.

In our proposed algorithm, the outer for loop in Line 2 runs at most *L* times, the for loop in Line 4 runs at most *N* times, where *L* is the depth of the input GPT model and *N* is the maximum number of OPNs for all levels. The while loop in Line 13 runs at most *J* times and the inner for loop runs at most *K* times, where *J* is the maximum number of jobs for AND operators and *K* is the maximum number of shared variables of OPNs. Therefore, the overall complexity of our query plan generation algorithm is $O(LNK(J + \log K))$.

4.2. MapReduce Query Algorithms

After the query plan is determined, a batch of MapReduce jobs is executed in the Query Engine module. In the first job, the Data Adapter matches all TPNs over RDF triples in the HBase and stores the solutions in HDFS files. The RDF data retrieval procedure is outlined in figure 6.

1. Build a collection of QP models *Q* for TPNs, $Q=\{QP_1, ..., QP_n\}$, $n\geq 0$.
2. Build a collection *T* to store remainder TPNs in GPT, $T=\{TPN_1,..., TPN_s\}$, $s\geq 0$.
3. Initial one multi-table supported MapReduce job.
4. Assign the query parameters for the job based on *Q* and *T*.
5. Execute the job to retrieve RDF data from HBase.
6. Store the matching solutions in HDFS files $QP_1, ..., QP_n, TPN_1,..., TPN_s$.

Figure 6 RDF data retrieval algorithm.

For example, as indicated in Section 3.3, two QP models QP_1 and QP_2 can be built for Query 8, whereas TPN_1 and TPN_2 must be handled separately. Therefore, the Data Adapter sets the querying parameters with the bounded terms in $Q=\{QP_1, QP_2\}$ and $T=\{TPN_1, TPN_2\}$, and runs one job to retrieve all matching RDF terms from the HBase. Finally, all corresponding solutions are stored into four HDFS files, namely, QP_1, QP_2, TPN_1, and TPN_2. The Query Engine then executes the Central Control algorithm in the Master node to schedule the querying jobs and monitor the execution, as shown in figure 7.

Input: a collection of MRQP models *QPlan*.
1. Initialization $J\leftarrow 1$, $InputFiles \leftarrow \varnothing$, $SumJ\leftarrow$total job numbers of *QPlan*
2. While ($J \leq SumJ$)
3. For each MRQP in *QPlan* whose *JobID* equals *J*
4. $InputFiles \leftarrow InputFiles \cup \{MRQP.Tag\}$, $CurrentOPN \leftarrow CurrentOPN \cup \{MRQP\}$
5. EndForeach
6. Job.Location $\leftarrow$ *InputFiles*, DictributedCache $\leftarrow$ *CurrentOPN*
7. If *J* equals *SumJ*, do Distributed Cache $\leftarrow$ { *SSM*, *QF* }
8. Execute Job and do $InputFiles \leftarrow \varnothing$, $J\leftarrow J+1$
9. End While

Figure 7 Central Control algorithm.

The Central Control algorithm takes a collection of MRQP models *QPlan* and iteratively executes a batch of MapReduce jobs until no MRQP model is left. First, two parameters *J* and *InputFiles* are initialized to represent the current job number and target RDF files, respectively. Subsequently, in the while loop, several MapReduce jobs are initialized and executed according to their sequence numbers. Starting from Line 3, for each MRQP model with a *JobID*=current job number *J*, the RDF files stored in HDFS are selected as the data source for the map function, and the query tasks in the corresponding MRQP models are collected into the collection *CurrentOPN*. The data source locations of the current job are then assigned to the *InputFiles* while the *CurrentOPN* is stored into the Distributed Cache object. In particular, if the current job is the last one, the *SSM* and *QF* models are stored into the Distributed Cache object. Finally, the job is executed and parameters *InputFiles* and *J* are updated.

After the query tasks are assigned, the map function executes the algorithm depicted in figure 8 to generate intermediate key–value pairs or to execute the UNION operation. At the beginning, the corresponding RDF solutions in the HDFS files and a collection of MRQP models in the Distributed Cache are transferred to the Map nodes. For each solution that matches a TPN or TN in one MRQP model *M*, if the OPT attribute of *M* is UNION, a new solution *S* is constructed with all of the variables in *M* and the corresponding RDF term values; otherwise, the key–value pair is constructed. As shown in Line 7, the key part is assigned to the shared variable and its matching value, and the value part is assigned to the remaining variables and their matching values in the solution. Finally, the key–value pairs are emitted to the Reduce nodes.

```
Input: a collection of RDF solutions. Output: key–value pairs
1. OPTs←CurrentOPTs in Distributed Cache
2. If the solution accepted is matched to the TPN or TN in OPTs
3. If M.OPT equals UNION
4.   Construct a new solution S that consists of all the variable names in M.
5.   S←the corresponding RDF terms in solution
6.   Output S to one HDFS file
7. Else key←(M.var, RDF term of M.var in solution), value←solution
8. Emit(key, value)
```

Figure 8 Query algorithm in Map function.

In the reduce phase, all intermediate key–value pair data with the same key values are transferred to one reduce node. Several operations are then executed as depicted in figure 9.

First, each reduce node obtains a list of MRQP model for the current job from the Distributed Cache and stores them in the *Opt*. The *n* collections U_1,..., U_n are then initialized, where *n* is the number of MRQP models. For each intermediate key–value pair received, one collection U_i is used to store the corresponding value part. Subsequently, we use the computational semantics of the AND and OPTIONAL operators defined by [5] to iteratively compute U_1,..., U_n according to the Opt and Filter attribute in the MRQP. Finally, *SSM* and *QP* are executed for U_R, if required, and the computational results are output to the HDFS.

Input: a collection of key–value pairs with the same key. Output: matching solutions.
1. *Opt* ← *CurrentOPTs* in Distributed Cache
2. *SSM* ← Sequence Solution Modifier models in Distributed Cache
3. *QP* ← Query Form models in Distributed Cache
4. For each MRQP model *M* in *Opt*
5. Initial collections U_1,..., U_n, where *n* is the TPN or TN numbers of the MRQP model
6. For each key–value pair *P* that corresponds to *M*, do $U_i \leftarrow P$.value, $1 \le i \le n$
7. Execute *M*.Opt for U_1,..., U_n and store the computational results in U_R
8. Execute *M*.Filter for U_R if required.
9. Execute *SSM* for U_R if SSM ≠ ∅
10. Output solutions U_R to HDFS according to *QP* if *QP* ≠ ∅

Figure 9 Query algorithm in Reduce function.

5 Experiments

5.1. Experiment Setup

The experiment is performed based on the synthetic data set SP2Bench, which is widely used to test the performance and scalability of the semantic web tools. Different from LUBM and BSBM, which contain simple SPARQL BGP test queries, SP2Bench provides 16 standard queries to evaluate complex graph patterns.

Six queries are chosen from SP2Bench: Queries 2, 3a, 4, 6, 8, and 9. The others are ignored because Queries 1, 5a, 5b, 10, and 11 are simple SPARQL BGP queries which do not contain UNION or OPTIONAL operator. Queries 3b and 3c have same structure and variables with Query 3a, only the conditions of Filter restriction part are different. Moreover, since current version of our framework does not support key words *bound* and *ASK*, we do not choose Queries 7, 12a, 12b and 12c.

To simulate the cloud environment, we use an Ethernet connected 9-node Hadoop cluster for our framework. Each node has the following configuration: Pentium IV 3.00 GHz CPU, 1.5 GB main memory, and 80 GB disk space. We run Jena, Sesame, and RDF-3X on a powerful single machine with Intel i5 2.50 GHz dual core processor, 8 GB main memory, and 4 TB disk space.

We use Hadoop-0.20.2 and Hbase-0.20.6 for our framework and compare it with the Jena-2.6.4 In-memory and SDB models, Sesame 2.6.3 main-memory model, and RDF-3X 0.3.7. MySQL version 5.0 database management system is used for the Jena SDB model.

5.1. Evaluation

For evaluation purposes, we create SP2Bench data sets with 4, 8, 12, 15, 20, and 30 million RDF triples to compare our proposed framework with that of Jena In-memory, SDB, Sesame, and RDF-3X. To evaluate the scalability of our framework, four SP2Bench data sets with 10, 20, 30, and 40 million RDF triples are generated. In this section, we first present the performance comparison and then describe the scalability testing results. In the following subsections, “Cloud” is used to denote our framework.

At first, we use SP2Bench Queries 2, 3a, 4, 6, 8, and 9 to compare our framework with those of Jena In-memory and SDB. These queries have simple and complex structures. They include AND, UNION, and OPTIONAL operators, as well as some FILTER and selectivity restrictions. The comparison results are shown in tables 2 and 3, in which the response time is in seconds.

	8 Million Triples		12 Million Triples		15 Million Triples		30 Million Triples	
	In-Mem	Cloud	In-Mem	Cloud	In-Mem	Cloud	In-Mem	Cloud
Query 2	174.75	252.96	286.23	394.53	Failed	458.08	Failed	820.71
Query 3a	95.85	51.50	140.26	83.58	204.17	116.78	Failed	260.78
Query 4	Failed	370.43	Failed	478.40	Failed	579.98	Failed	887.69
Query 6	Failed	375.47	Failed	517.89	Failed	582.79	Failed	1154.99
Query 8	78.75	547.77	136.83	949,52	194.44	1245.74	Failed	2365.54
Query 9	82.36	285.37	139.96	402.95	199.02	502.32	Failed	916.42

Table 2 Comparison between Jena In-memory and our framework.

	4 Million Triples		8 Million Triples		12 Million Triples		15 Million Triples	
	SDB	Cloud	SDB	Cloud	SDB	Cloud	SDB	Cloud
Query 2	192.57	179.05	4090.97	252.96	Failed	394.53	Failed	458.08
Query 3a	172.93	44.54	1287.33	51.50	Failed	83.58	Failed	116.78
Query 4	Failed	243.88	Failed	370.43	Failed	478.40	Failed	579.98
Query 6	Failed	257.26	Failed	375.47	Failed	517.89	Failed	582.79
Query 8	Failed	376.54	Failed	547.77	Failed	949,52	Failed	1245.74
Query 9	223.78	171.80	1881.46	285.37	Failed	402.95	Failed	502.32

Table 3 Comparison between Jena SDB and our framework.

Query 2 includes 10 triple patterns. The first nine patterns execute the AND operation before running the OPTIONAL operator for the last pattern. Two bound terms exist in the first triple pattern and all triple patterns share one variable ?*inproc*. Hence, Query 2 is highly selective, resulting in a small set size. For this query, Jena In-memory is faster than our framework when the data set size is 8 and 12 million triples, but it fails to answer the query when we input 15 and 30 million triples. For all four data set sizes, our framework outperforms Jena SDB. In particular, SDB fails to answer the query when we input 12 and 15 million triples.

Query 3a is a simple query that selects all articles with the property *swrc*:*pages*. It involves only one AND operator for two triple patterns and one FILTER restriction. All the components of the second triple pattern are variables, and thus this query is only lowly selective and produces large results. For all data set sizes, our framework outperforms Jena In-memory and SDB. When the data set size increases, Jena In-memory and SDB fails to answer this query.

Query 4 requires returning all the distinct pairs of article author names for authors who published in the same journal. This result set is the largest. For all four data set sizes, our framework beats both Jena In-memory and SDB.

Query 6 is more complex than Query 4 and the returned result set is large. Jena In-memory and SDB fail to answer this query for all data set sizes, but our framework generates the output in a reasonable amount of time.

Query 8 is a composite query because it includes the AND, UNION, and Filter operators for multiple GGPs. Jena In-memory beats our framework for 8, 12, and 15 million RDF triples. However, when the size of the input RDF data set increases to 30 million triples, Jena In-memory shows an out-of-memory exception. Jena SDB cannot answer this query for all input data.

Query 9 returns the incoming and out-coming properties of persons, and the size of the result data set is small because of the Distinct restriction. Jena In-memory beats our framework for this query when the data set size is 8, 12, and 15 million triples, but it fails to answer the query when we input 30 million triples. For all four data set sizes, our framework outperforms Jena SDB.

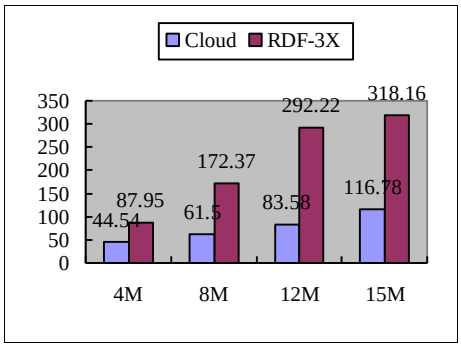
Jena In-memory achieves better performance than our framework for queries with high selectivity and bound objects when processing small RDF data sets. However, because of the memory limitation, Jena In-memory cannot finish the query tasks when the data set size increases. For queries with unbound objects, low selectivity, and large result sets, our framework outperforms Jena In-memory. Our framework also achieves better performance and scalability than Jena SDB because the latter cannot execute queries with large input data set sizes and result sets.

The current version of RDF-3X does not support OPTIONAL querying. Thus, we choose Queries 3a, 4, 8, and 9, and create four SP2Bench data sets at 4, 8, 12, and 15 million triples, to compare with our framework. Before answering SPARQL queries, RDF-3X requires the target RDF data set to be loaded into the memory. Thus, the response time of RDF-3X in our experiment consists of the initial data load time and the query time. Figure 10 shows the experimental results, where the X-axis represents the data set sizes and the Y-axis denotes the response time in seconds.

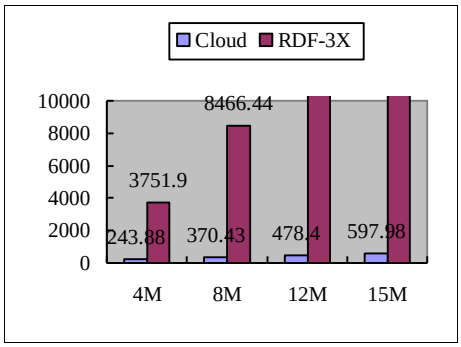
For Query 3a, as shown in figure 10(a), our framework beats RDF-3X for all four data set sizes. Query 4 produces a large result set. Thus, RDF-3X takes a long time to generate the final output. Our implementation takes less time. When the data set contains 4 million triples, RDF-3X takes 3751.9 seconds to get the result, whereas our framework takes only 243.88 seconds. When the RDF data set size increases to 12 and 15 million triples, RDF-3X runs for more than five and eight hours, respectively, at which point the process is discontinued. Our framework takes only 478.4 and 597.98 seconds for the same amount of triples, respectively. For Queries 8 and 9, RDF-3X is faster because both queries are highly selective and have small result sets.

The experimental results show that RDF-3X achieves better performance for queries with high selectivity. However, our framework is much faster than RDF-3X for queries with low selectivity and large result sets.

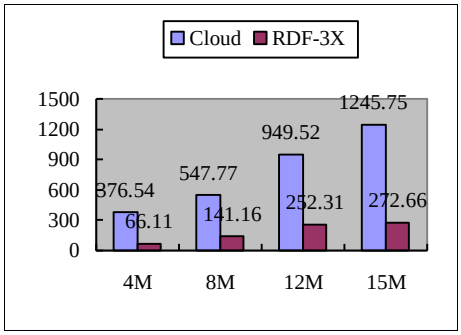
The third experiment compares our framework with the Sesame main-memory and database models. We create SP2Bench data sets at 8, 12, 15, and 30 million triples, and Queries 2, 3a, 4, 6, 8, and 9 for the test. However, because of the complex storage schema, the Sesame database model takes more than 12 hours to store 8 million triples into MySQL. Therefore, we discontinue the running comparison with the Sesame database model, and summarize that it is not suitable for storing and querying large amounts of RDF data. Table 4 shows the comparison between Sesame main-memory and our framework.



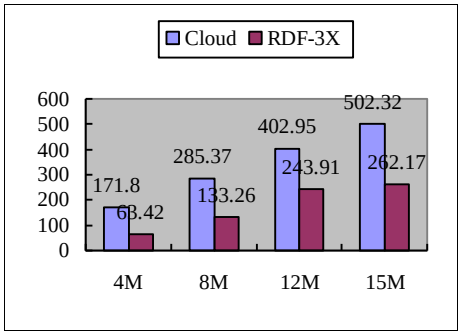
(a) Response time for SP2Bech Query 3a



(b) Response time for SP2Bech Query 4



(c) Response time for SP2Bech Query 8



(d) Response time for SP2Bech Query 9

Figure 10 Comparison between RDF-3X and our framework.

For Queries 2, 8, and 9, the Sesame main-memory is faster than our framework for small input data set because these queries are highly selective and have small result sets. For Query 3a, Sesame takes 145.28, 236.80, and 313.43 seconds to generate the results for the first three data sets, whereas our framework outperforms it with 51.50, 83.58, and 116.78 seconds, respectively. Furthermore, as the size of result sets increase, Sesame main-memory can barely finish executing Queries 4 and 6 all four data set sizes. By contrast, our framework answers Queries 4 and 6 in a reasonable amount of time. Similar to the situation using Jena In-memory, when the number of RDF triples increases to 30 million, Sesame main-memory fails to answer all queries due to memory limitation.

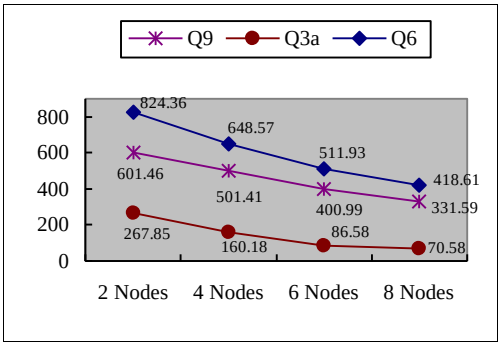
Based on the results, the Sesame main-memory works well for queries with high selectivity and small result sets when the size of the input data set is small, but does not scale well when the size increases. For queries with low selectivity and large result sets, our framework performs better than the Sesame main-memory model.

Scalability tests are also performed for the experiments. First, we repeat the same six queries for 10 million triples by increasing the number of nodes from two to eight to evaluate the scalability of our framework. As shown in figure 11, the time to answer these queries decreases, as expected, when the number of nodes increases. For example, Query 6 takes 824.36 seconds with two nodes, 648.57 seconds with four nodes, 511.93 seconds with six nodes, and 418.61 seconds with eight nodes. Hence,

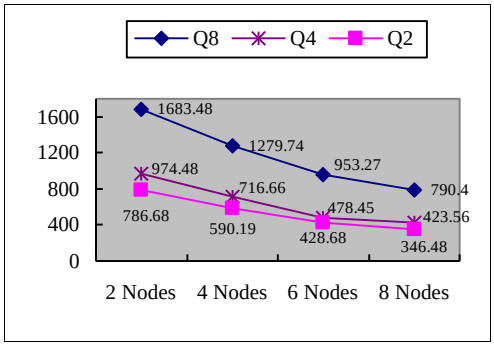
increasing the number of computing nodes in the Hadoop cluster can improve the performance of our framework.

	8 Million Triples		12 Million Triples		15 Million Triples		30 Million Triples	
	Sesame	Cloud	Sesame	Cloud	Sesame	Cloud	Sesame	Cloud
Query 2	143.99	252.96	240.79	394.53	294.97	458.08	Failed	82071
Query 3a	145.28	51.50	236.80	83.58	313.43	116.78	Failed	260.78
Query 4	Failed	370.43	Failed	478.40	Failed	579.98	Failed	887.69
Query 6	Failed	375.47	Failed	517.89	Failed	582.79	Failed	1154.99
Query 8	144.95	547.77	235.03	949,52	285.78	1245.74	Failed	2365.54
Query 9	143.81	285.37	229.26	402.9	283.91	502.32	Failed	916.42

Table 4 Comparison results between Sesame main-memory and our framework.



(a) Response time for Queries 9, 3a, and 6



(b) Response time for Queries 8, 4, and 2

Figure 11 Experimental results for increasing the number of computing nodes.

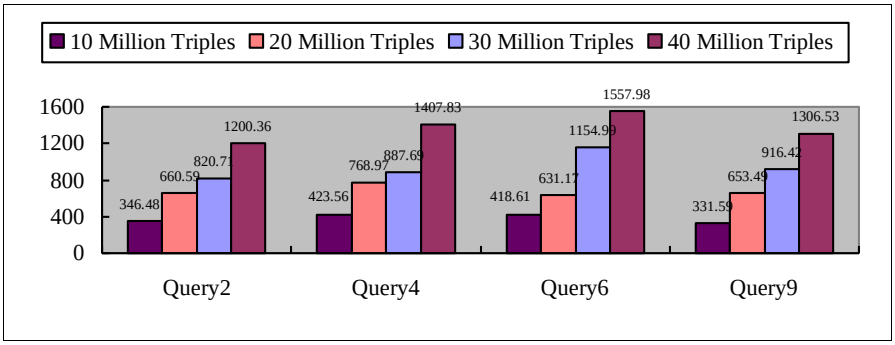


Figure 12 Experimental results for increasing the number of RDF triples.

Finally, we use Queries 2, 4, 6, and 9 for 10, 20, 30, and 40 million triples to test the scalability of our framework. The experimental results are shown in figure 12, where the response time is in seconds. As the size of the data increases, the time to answer the queries increases sublinearly. For example,

Query 2 takes 346.88 seconds for 10 million triples and 1200.36 seconds for 40 million triples. Therefore, the data size increases four times and the time to answer increases only 3.46 times. The same conclusion is observed for Queries 4, 6, and 9.

Overall, based on the reported experimental results, our proposed framework is more efficient and scalable than the conventional RDF query tools when processing large amounts of RDF triples.

6 Conclusions and Future Works

In this paper, we propose a scalable SPARQL GGP querying framework for large amount of RDF data by exploiting the cloud computing technologies. A novel RDF data storage schema is designed on the HBase based on the syntax and semantics of RDF and SPARQL. To decrease the number of MapReduce jobs, a greedy-strategy-based algorithm is proposed to determine the query plan. Several MapReduce query algorithms are also described. Through comparisons with comprehensively used RDF query tools, our proposed framework performs better with queries that have low selectivity and large result sizes when processing large-scale RDF data. Additionally, our framework is more scalable than the conventional RDF query tools.

We will integrate all existing FILTER restriction operators, query forms, and the novel SPARQL 1.1 operators into our proposed framework in our future work. In addition, because the RDF data has to be distributed to the computing nodes via networks, our solution dose not perform well for the small size of RDF triples. In the future, we will also continue optimizing the performance of our framework in two aspects. First, we will parallelize the map and reduce function of MapReduce paradigm to improve the performance of each computing node. Second, we plan to integrate data compression technologies into MapReduce to reduce the network transfers and provide an efficient and scalable cloud service for future RDF data management.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (91118005, 61103114), the Natural Science Foundation of Chongqing City in China (2011BA2022), and the Fundamental Research Funds for the Central Universities in China (CDJXS11181162).

We also wish to thank the reviewers for their valuable comments and suggestions.

References

1. Berners-Lee, T., Hendler, J. and Lassila, O., The Semantic Web: A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. Scientific American, 2001.
2. Mishra, R.B. and Kumar, S., Semantic Web Reasoners and Languages. Artificial Intelligence Review, vol. 35, no. 4, pp. 339–368, 2011.
3. W3C, Resource Description Framework (RDF): concepts and abstract syntax, 2004, http://www.w3.org/TR/rdf-concepts/.
4. W3C, SPARQL query language for RDF, 2008, http://www.w3.org/TR/rdf-sparql-query/.
5. Pérez, J., Arenas, M. and Gutierrez, C., Semantics and Complexity of SPARQL. In Proceedings of the 5th International Semantic Web Conference, pp. 30–43, 2006.
6. Cyganiak, R., A relational algebra for SPARQL, HP-Labs Technical Report, HPL-2005-170. http://www.hpl.hp.com/techreports/2005/HPL-2005-170.html.

7. W3C, SPARQL 1.1 Query Language, 2012, http://www.w3.org/TR/sparql11-query/
8. Bizer, C., Jentzsch, A. and Cyganiak, R., State of the LOD Cloud, http://www4. wiwiss.fuberlin. de/lodcloud/state/
9. Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A. and Wilkinson, K., Jena: Implementing the Semantic Web Recommendations, In Proceedings of the 13th International World Wide Web Conference, 2004, pp. 806–815.
10. Broekstra, J. and Kampman, A., Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema, In Proceedings of the 1st International Semantic Web Conference, 2002.
11. Neumann, T. and Weikum, G., The RDF-3X Engine for Scalable Management of RDF Data, VLDB Journal, vol. 19, pp. 91–113, 2010.
12. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C. and Reynolds, D., SPARQL Basic Graph Pattern Optimization using Selectivity Estimation, In Proc. 17th International Conference on World Wide Web 2008, WWW '08, pp. 595–604, 2008.
13. Vidal, M. E., Ruckhaus, E., Lampo, T., Martinez, A., Sierra, J. and Polleres, A., Efficiently Joining Group Patterns in SPARQL Queries, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), The Semantic Web: Research and Applications –7th Extended Semantic Web Conference, ESWC 2010, Proceedings, pp. 228–242, 2010.
14. Groppe, J. and Groppe, S., Parallelizing Join Computations of SPARQL Queries for Large Semantic Web Databases, In Proceedings of the 26th Annual ACM Symposium on Applied Computing, pp. 1681–1686, 2011.
15. Dean, J. and Ghemawat, S., MapReduce: Simplified Data Processing on Large Clusters, Communications of the ACM, vol. 51, no. 1, pp. 107–113, 2008.
16. Ghemawat, S., Gobioff, H., Leung, S-T., The Google File System, In Proc. 19th ACM Symposium on Operating Systems Principles, pp. 29–43, 2003.
17. Mika, P. and Tummarello, G., Web Semantics in the Clouds, IEEE Intelligent Systems, vol. 23, no. 5, pp. 82–87, 2008.
18. Alham, N. K., Li, M.Z, Liu. Y and Hammoud, S., A MapReduce-based Distributed SVM Algorithm for automatic image annotation, Computers & Mathematics with Applications, vol. 62, no. 7, pp. 2801–2811, 2011.
19. Xue, W., Shi, J. W. and Yang, B., X-RIME: Cloud-Based Large Scale Social Network Analysis, In Proceedings of 2010 IEEE International Conference on Services Computing, pp. 506–513, 2010.
20. Urbani, J., Kotoulas, S., Maassen, J., Harmelen, F.V. and Bal, H., WebPIE: A Web-scale Parallel Inference Engine using MapReduce, Journal of Web Semantics, vol. 10, pp. 59–75, 2012.
21. Grau, B.C., Horrocks, I., Motik, B., Parsia, B., Patel-Scheider, P. and Sattler, U., OWL 2: The Next Step for OWL, Journal of Web Semantics, vol. 6, no. 4, pp. 309–322, 2008.
22. Mutharaju, R., Maier, F. and Hitzler, P., A MapReduce Algorithm for EL+, In Proceedings of the 23rd International Workshop on Description Logics, pp. 464–474, 2010.
23. Myung, J., Yeon, J. and Lee, S., SPARQL Basic Graph Pattern Processing with Iterative MapReduce, In Proceedings of 2010 Workshop on Massive Data Analytics on the Cloud, MDAC 2010, in Association with the 19th Annual World Wide Web Conference, WWW 2010, 2010.
24. Husain, M. F., McGlothlin, J., Masud, M. M., Khan, L. R. and Thuraisingham, B., Heuristics-based Query Processing for Large RDF Graphs using Cloud Computing, IEEE Transactions on Knowledge and Data Engineering, vol. 23, no. 9, pp. 1312–1327, 2011.
25. Franke, C., Morin, S., Chebotko, A., Abraham, J. and Brazier, P., Distributed Semantic Web Data Management in HBase and MySQL Cluster, In Proceedings of 2011 IEEE 4th International Conference on Cloud Computing, pp. 105–112, 2011.

26. Sun, J. and Jin, Q., Scalable RDF Store based on HBase and MapReduce, In Proceedings of the 3rd International Conference on Advanced Computer Theory and Engineering, vol. 1, pp. V1633–V1636, 2010.
27. Choi, H., Son, J., Cho, Y., Sung, M. K. and Chung, Y. D., SPIDER: A System for Scalable, Parallel / Distributed Evaluation of large-scale RDF Data, In Proceedings of International Conference on Information and Knowledge Management, pp. 2087–2088, 2009.
28. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A. and Gruber, R. E., Bigtable: A Distributed Storage System for Structured Data, ACM Transactions on Computer Systems, vol. 26, no. 2, 2008.
29. Weiss, C., Karras, P. and Bernstein, A., Hexastore: Sextuple Indexing for Semantic Web Data Management, In Proceedings of VLDB Endowment, vol.1, no.1, pp. 1008–1019, 2008.
30. Guo, Y., Pan, Z. and Heflin, J., LUBM: A benchmark for OWL knowledge base systems, Journal of Web Semantics, vol. 3, no. 2–3, pp. 158–182, 2005.
31. Schmidt, M., Hornung, T., Lausen, G. and Pinkel, C., SP2Bech: A SPARQL performance benchmark, In Proceedings of the 25th IEEE International Conference on Data Engineering, pp. 222–233, 2009.
32. Bizer, C. and Schultz, A., The Berlin SPARQL benchmark, International Journal on Semantic Web and Information Systems, vol. 5, no. 2, pp. 1–24, 2009.