SparkRDF: Elastic Discreted RDF Graph Processing Engine With Distributed Memory

Xi Chen
Department of Computer Science
Zhejiang University
Hangzhou, China
xichen@zju.edu.cn

Huajun Chen
Department of Computer Science
Zhejiang University
Hangzhou, China
huajunsir@zju.edu.cn

Ningyu Zhang
Department of Computer Science
Zhejiang University
Hangzhou, China
zxlzr@zju.edu.cn

***Abstract*—With the explosive growth of semantic data on the Web over the past years, many large-scale RDF knowledge bases with billions of facts are generating. This poses significant challenges for the storage and query of big RDF graphs. Current systems still have many limitations in processing big RDF graphs including scalability and real-time. In this paper, we introduce the SparkRDF, an elastic discreted RDF graph processing engine with distributed memory. To reduce the high I/O and communication cost in distributed processing platforms, SparkRDF implements SPARQL query based on Spark, a novel in-memory distributed computing framework for big data processing. All the intermediate results are modeled as Resilient Discreted SubGraph, which are cached in the distributed memory to support fast iterative join operations. To cut down the search space and avoid the overhead of memory, we split the RDF graph into the small Multi-layer Elastic SubGraph based on the relations and classes. For SPARQL query optimization, SparkRDF deploys a serials of optimization strategies, leading to effective reduction on the size of intermediate results, the number of joins and the cost of communication. Our extensive evaluation demonstrates that SparkRDF can efficiently implement non-selective joins faster than both current state-of-the-art distributed and centralized stores, while being able to process other queries in real time, scaling linearly to the amount of data.**

***Keywords*-Large RDF Graph, Big Data, SPARQL, Spark, Distributed memory.**

I. Introduction

With the development of Semantic technologies and Web 3.0, the amount of Semantic Web data represented by the Resource Description Framework (RDF) [1] is increasing rapidly. Commercial search engines including Google and Bing add the semantics of their web contents by RDFa to improve the query accuracy. At the same time, a growing number of organizations and community driven projects are constructing large knowledge bases with billions of facts in many domains to implement more useful applications including bioinformatics, life sciences, social networks and so on. For example, The UniProt Knowledge base contains more than 2.9 billion triples [2]. The DBpedia and Probase also have billions of facts in RDF format [3] [4]. The Linked Open Data Project even announces 52 billion triples were published by 2012 [5].

Challenges Faced with such large RDF data, traditional RDF data management systems are facing two challenges:i)scalability: the ability to process the big RDF graph data. Existing most RDF systems are based on single node [6] [7]. Thus, they are easily vulnerable to the growth of the data size because they usually need to load large indexes into the limited memory. ii) real-time: the capacity to implement SPARQL [8] query over big RDF graph in near real time. For highly iterative SPARQL joins, existing MapReduce-based [9] RDF systems suffer from high I/O cost because of iteratively reading and writing large intermediate results in disk [10].

In fact, a RDF graph can be considered as representing a directed graph, with entities (i.e. subjects and objects) as nodes, and relationships (i.e. predicates) as directed edges. SPARQL, which acts as the standard query language for RDF graph, consists of a conjunctive set of triple patterns(TP). A TP is similar to a RDF triple except that any position in the TP can be a variable. The goal of SPARQL query is to compute the bindings for all variables. As Every TP can be regarded as a subgraph of the big RDF graph, the problem of SPARQL query processing can be transformed to the problem of iterative matching and joining of subgraphs. We can summarize the following stages for querying a big RDF graph: store the RDF data and create indexes, read necessary data inside memory, compute binding values for every TP, implement joins over intermediate results(IR), and build the final results of the query. Hence, a desired query processing system should meet the following requirements as much as possible: (i) reduce the data needed to be loaded in memory (ii)reduce the number of IR (iii)reduce the number of join (iv)reduce the i/o and communication cost of iterative join.

In this paper, we introduce SparkRDF, an elastic discreted RDF graph processing system with distributed memory. It is based on Spark framework, a fast in-memory cluster computing system which is quite suitable for large-scale real-time iterative computing tasks [11]. Unlike existing most systems that use a set of permutations of (S,P,O) indexes to manage RDF, SparkRDF splits the big RDF graph into MESGs(Multi-layer Elastic SubGraph) based on relations

and classes by creating 5 kinds of indexes(C,R,CR,RC,CRC) with different grains to cater for diverse TPs. These index files on demand are modeled as RDSG (Resilient Discreted SubGraph), a collection of in-memory semantic subgraph objects partitioned across machines, which can implement SPARQL query by a series of basic operators. All IRs, which are also regarded as the RDSG, remain in the distributed memory to support further fast joins. Based on the data model and query model, we introduce a new cost model and several corresponding optimization tactics to improve the efficiency of SparkRDF including TR-SPARQL(Type-Restrictive), optimal query plan and on-demand dynamic prepartitioning techniques.

We summarize the main contributions of this work as follows:

(i) We introduce a novel MESG-based storage scheme for managing big RDF graph by constructing relation subgraphs, class subgraphs and their combinations. The method improves the query efficiency by reducing search space and the number of tasks.

(ii) We design a RDSG-based iterative model to process the SPARQL query by implementing iterative join operations with distributed memory, which avoids lots of I/O cost for IRs.

(iii) We devise a cost model and several optimization strategies, including TP-SPARQL, optimal query plan and on demand dynamic prepartitioning techniques. These approaches ensure high performance for SparkRDF.

(iv) We perform an experimental evaluation by comparing SparkRDF and current state-of-the-art distributed and centralized stores. The results prove the effectiveness of SparkRDF.

II. RELATED WORK

There are many researches focusing on the RDF data management. We can summarize these work as the following three categories:

Single-machine Systems: RDF-3X [6] is considered the fastest existing RDF management system today. It builds clustered B+-trees on all six (S,P,O) permutations, thus each RDF graph is stored in six duplicates. Besides, it also employs additional indexes to collect statistical information for pairs and stand-alone entities to eliminate the problem of expensive self-joins and provides great performance improvement. However, storing all the indexes is expensive and query efficiency highly depends on the amount of main memory. The performance penalty can be high as the volume of dataset increases due to the expensive cost of storing, loading, accessing and joining the big indexes at the query evaluation, especially for some unselective queries with large results. BitMat [7] represents an alternative design of the property table approach, in which RDF data is represented as a 3D bit-cube matrix. Each matrix element is a bit denoting the presence or absence of the corresponding triple. TripleBit [12] introduces a bit matrix storage structure and the encoding-based compression method for storing huge RDF graphs more efficiently. It also uses two auxiliary indexing structures to reduce the number and size of indexes to the minimal. As above systems aim at processing RDF data in single machine, they all suffer from the problem of scalability.

Distributed RDF Systems: Currently, most distributed RDF systems are based on popular distributed processing framework Hadoop and MapReduce. HadoopRDF [10] uses HDFS [13] files named after predicate values to partition the input RDF graph to create a pos or pso index. Then the predicate files are splitted into smaller files according to the type of objects. It performs the SPARQL query by a serious of iterative MapReduce jobs, each of which implements a join between two TPs on a variable. Besides, it employs a greedy algorithm to reduce the number of required MapReduce joins at each step. Obviously, the greedy planner ignores the information about join selectivity, inducing a lot of unnecessary intermediate results. Huang et al. [14] deploy single-node RDF-3X systems on multiple machines, and use the MapReduce framework to synchronize query execution. H2RDF+ [15] stores and indexes RDF data in HBase [16]. Then it implements MapReduce-based multi-way Merge and Sort-Merge join algorithms to process SPARQL query. For above systems, it will spend lots of time to iteratively reading and writing IRs in HDFS or HBase as every MapReduce job needs a given input and output.

Distributed in-memory RDF Systems: Trinity.RDF [17] is a distributed memory-based graph engine for big RDF graph. It builds on top of a memory cloud and models the RDF data in its native graph form. Instead of processing SPARQL by expensive join operations, it leverages graph exploration to generate bindings for each of the TP, which greatly reduces the amount of intermediate results and boosts the query performance. But The whole RDF graph must be loaded in main memory for the Trinity.RDF, which is unrealistic in most situations. So this is not a scalable approach, as it needs very strict cluster and memory limitations.

III. SYSTEM ARCHITECTURE

Figure 1 shows the architecture of SparkRDF. In general, SparkRDF consists of five modules: data preprocessing module, graph splitting module, distributed storage module, query parser module and distributed join module.

The data preprocessing module is responsible for unifying the format of input data by converting XML-like RDF data to N-Triples. Then the big RDF graph is splitted into several elastic discreted subgraphs by graph splitter. The distributed storage module subsequently persists the indexes into HDFS. Query parser module takes the SPARQL query from the user, and dynamically generates the query plan including required index information for the query. Based on the index information, corresponding indexes are dynamically loaded

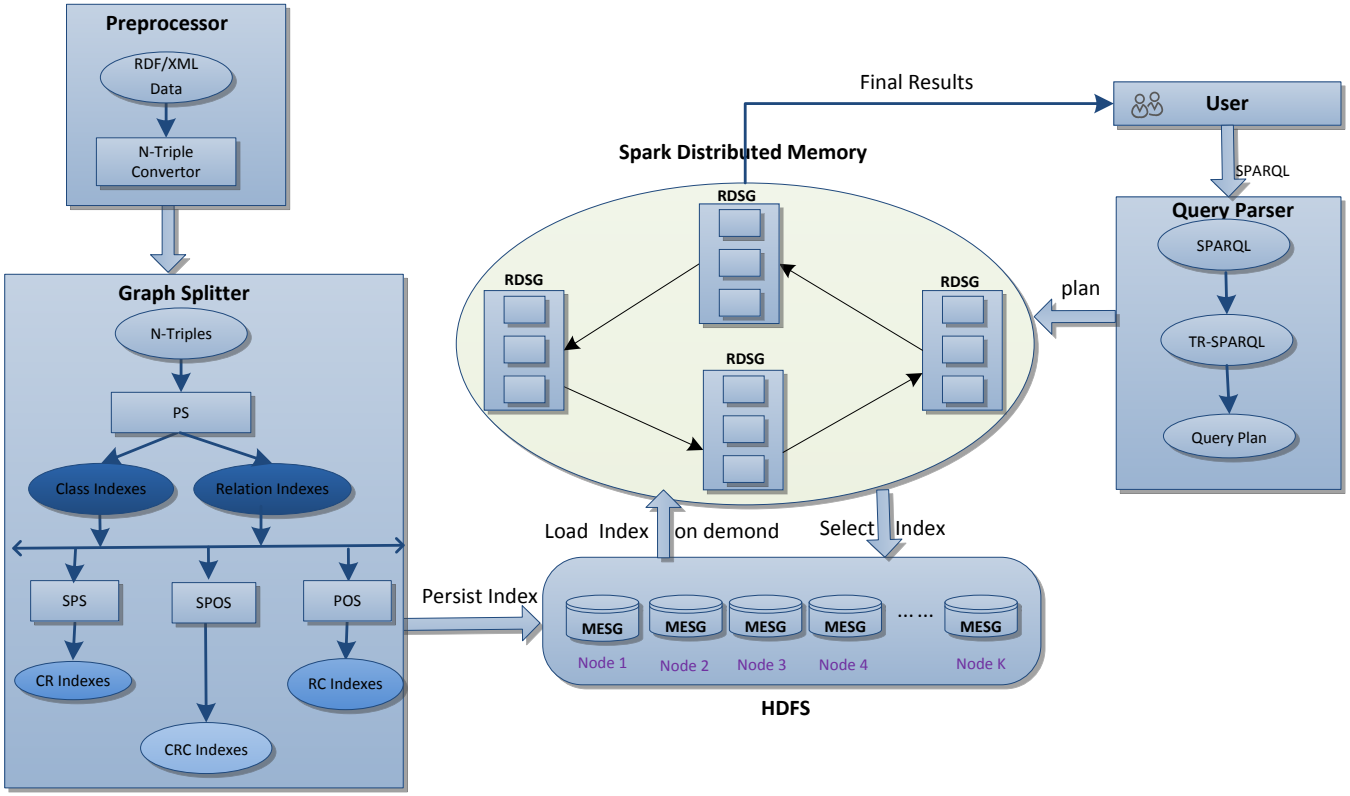


Figure 1. The Architecture of SparkRDF

into the distributed memory and modeled as RDSG for further implementing iterative joins in memory as the query plan requested. At last, the final join results are returned to the user.

IV. Data Modeling

To support SPARQL queries on big RDF graph more effectively, we store the RDF graph in the MESG form and model the index subgraphs as RDSG once they are selected as the query index for the TPs. In this section, we describe the two data models: MESG and RDSG.

A. Index Data Model: Multi-layer Elastic SubGraph

Usually, the size of memory even on large clusters is far from the amount of massive RDF graph data. Conventional index scheme, which uses a set of permutations of (S,P,O) indexes, is not applicable since the size of its index files is still very big which will easily lead to the overhead of memory. The RDF graph should be elaborately splitted into smaller subgraphs to reduce the search space and avoid the overhead of memory.

In this section, we describe the process of splitting the big RDF graph into MESG. It extends traditional vertical partitioning solution by connecting class indexes with relation indexes, whose goal is to construct a smaller index file for every TP in the SPARQL query. At the same time, as it is uncertain that the class information about the entities can are given in the SPARQL query, the SparkRDF needs a multi-layer elastic index scheme to meet the query need for different kinds of TP.

1) Class indexes and Relation indexes: As is shown in the Graph Splitter of Figure 1, in the first step, we construct the class indexes(C) and relation indexes(R) by Predicate Splitter (SP). Specifically, we split the RDF graph by two ways. For the triples, whose predicate is not **rdf:type**, we extract the triples's subject and object into corresponding relation index files(predicate being as the file name). Thus the triples sharing the same predicate are grouped together to accelerate the SPARQL queries that have a binding predicate. For other triples(predicate is **rdf:type**), we divide them into small class files based on the triple's object representing a specific class. The instances belonging to the same class are stored in corresponding class index files(object being as the file name). This further reduces the amount of storage space because the type index files only contain these triples' subject.

2) Class&Relation index: The class indexes and relation indexes enable us to select precise index files based on TP's predicate. But we actually only use the predicate information of a TP, while other information such as its subject and object is missed directly. This will generate some very large relation indexes for those common predicates such as emailAddress, name and so on. A finer-grained partitioning scheme should be designed to further cut down the search space.

In fact, a predicate represents the binary relationship between subject and object, whose type information is often given in a SPARQL query. Thus we can divide the predicate files further according to the type of the subjects and objects. The type information of the subject and object in a triple is available in type indexes. So a set of finer-grained index files(CR,RC,CRC) can be created by joining the two kinds of

Table I
SAMPLE DATA FOR OUR QUERY

(a) The First Layer Index: C and R

Class Index(C)		Relation Index(R)	
Type_GraduateStudent	Type_FullProfessor	hasAdvisor	
GS1	FP1	GS1	FP1
GS2	FP2	GS2	AP1
	FP3	US1	FP2
		RA1	FP3
		US2	AP2

(b) The Second Layer Index: CR and RC

Class&Relation Index(CR)		Relation&Class Index(RC)	
GraduateStudent_hasAdvisor		hasAdvisor_FullProfessor	
GS1	FP1	GS1	FP1
GS2	AP1	US1	FP2
		RA1	FP3

(c) The Third Layer Index: CRC

Class&Relation&Class Index(CRC)	
GraduateStudent_hasAdvisor_FullProfessor	
GS1	FP1

index files(we cache the type information in the first step to support subsequent fast indexing). Take the triple (Student0, advisor, Professor0) for example, as its subject and object belong to class Student and Professor respectively, tuple (Student0, Professor0) will be moved into three different kinds of index files named as Student_advisor(CR), advisor_Professor(RC), Student_advisor_Professor(CRC index).

The flexible index method makes SparkRDF capable of creating a appropriate subgraph for every TP in the SPARQL by selecting the minimum input indexes, which greatly reduces the cost of reading indexes and executing joins. What's more, it can also cut down the number of joins by eliminating unnecessary class TPs(see section V).

3) Example Data: Table I shows some sample data for the query goal: look for the graduate students whose advisor is a full professor. The first layer index constructs two class indexes and one relation index. It is obvious that there exist many undesirable results in the hasAdvisor index including undergraduate students, research assistants and associate professors due to the lack of the limitation for the class of subject and object. The second layer index alleviates the problem by specifying the required class for subject or object. At last, the CRC index provides the best input option.

Another advantage of the data model lies in the data compression because we do not need to store data in triple form. For the class and relation index, we save the 2/3 and 1/3 storage cost respectively. So although we store all the three layer indexes, SparkRDF achieves more than 60% storage reduction based on the statistics for LUBM-10000 data(The raw size is 224G, while our index is only 84G).

B. Memory Data Model: RDSG

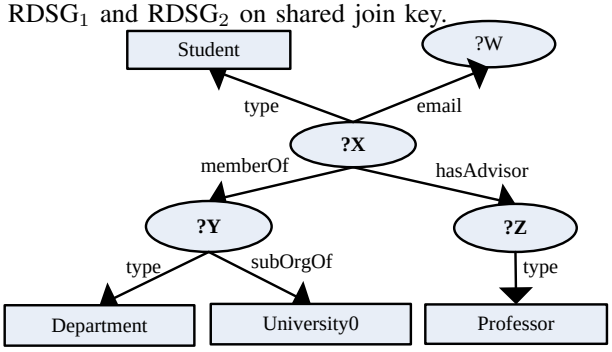
As previously noted, the problem of SPARQL query processing can be transformed to the problem of iterative subgraph matching and joining. We need to compute the matchings for every TP in the SPARQL query by selecting the suitable index subgraphs and implement iterative subgraph join operation on the matchings to get the final result subgraph. To improve the query efficiency, SparkRDF models all the subgraph data using an unified concept called RDSG(Resilient Discreted Semantic subGraph), which is a distributed memory abstraction that lets us perform in-memory query computations on large clusters. Based on the API of Spark, we provide the following basic operators to implement query processing:

(i) **RDSG_Gen:** The operator creates a RDSG given the name of MESG. i.e., RDSG_Gen(GraduateStudent_ hasAdvisor) generates the $RDSG_1$.

(ii) **RDSG_Filter:** The operator filters RDSG based on the subject or object constrains given in a TP i.e., RDSG_Filter($RDSG_1$,"Professor1", flag) filters the graduate students whose advisor is not "Professor1". Flag is a parameter that specifies whether the constrain is subject or object.

(iii) **RDSG_Prepartition:** The operator controls the partitioning of the RDSG. Every RDSG is composed of a list of partition objects, which are distributed in the multiple machines.

(iv) **RDSG_Join:**The operator implements join operation on a set of RDSG and takes in as parameters, the labels of the two types of RDSG to be joined and a join condition e.g. RDSG_Join($RDSG_1$, $RDSG_2$, $RDSG_1$.S=$RDSG_2$.O) returns the results of joining $RDSG_1$ and $RDSG_2$ on shared join key.



- TP1: (?X type Student)
- TP2: (?Y type Department)
- TP3: (?Z type Professor)
- TP4: (?X memberOf ?Y)
- TP5: (?Y subOrgOf University0)
- TP6: (?X email ?W)
- TP7: (?X hasAdvisor ?Z)

Figure 2. The Query Graph of Our Example

V. QUERY PROCESSING

In this section, we first propose our RDSG-based iterative query model. Then we introduce the cost estimation based on the query model. At last, multiple query optimization strategies are presented according to our cost model to improve the query efficiency of the SparkRDF.

A. RDSG-based Iterative Query Model

1) Overview: We represent a SPARQL query by a query graph Q, which contains multiple query variables. Each query variable exists in two or more TPs. Figure 2 shows the query graph of our SPARQL query example.

With Q defined, the problem of SPARQL query processing can be conducted as follows: Firstly decompose Q into an ordered sequence of variables: X, Y, Z. Every query variable is made up of several ordered TPs. TP_1^X refers to the first TP of variable X. To every TP with variable X, we iteratively compute the matches for each TP_i^X, and the matched IRs are used to find the matches for TP_{i+1}^X by implementing joins on the shared variable X. After computing the bindings of a variable, we continue computing the bindings of another variable based on the IRs of X in the same way. At last, the remaining results are the final answer of the query graph.

The query process can be explained by a query plan, which specifies the execution flow of TPs by a serious of jobs. Figure 3 illustrates the query process of SparkRDF. Every job is responsible for computing the bindings for a certain variable. For convenience, we only refer to the TPs by the variable in that pattern(or plus its order subscript if necessary). Thus the query can be simplified as follows:{X,Y_2,Z,XY,Y_5,XW,XZ}. Given the query, several possible query plans are as follows:

Plan 1. job_1={X,XY,XW,XZ}, job_2={Y_2,Y_5},job_3={Z}

Plan 2. job_1={Y_2,XY,Y_5}, job_2={X,XW,XZ},job_3={Z}

Plan 3. job_1={Y_5,Y_2,XY}, job_2={X,XW,XZ},job_3={Z}

Definition 1: **Query Plan(QP).** A sequence of jobs that give the variable join order and TP execution order. QP={Job_1,Job_2,...,Job_n|$\cup_{i=1}^{n}Job_i$=U(TP),$Job_k \cap Job_m=\emptyset$, k$\neq$m,k$\leq$n,m$\leq$n}. Job_k={TP_1^V,TP_2^V,...,$TP_{N_k}^V$}(n is the number of job, N_k is the number of TP corresponding to variable V in the Job_k).

2) Single TP Matching: As is shown in Figure 3, for single TP matching, we use the RDSG-based operators to get the bindings of variables in the TP. Firstly, RDSG_Gen is used to choose appropriate MESG(index files) to create a RDSG. Then the Prepartition operator repartitions the data based on the hash value of the join variable.Then RDSG_Filter further filters lots of unrelated bindings of a variable according to the given subject or object of the TP. We use the Parse(TP) to represent the three phases.

3) Multiple TPs Join: For multiple TPs matching in a job(take $Plan_1$.job_1 for example), the results of parsing the first TP will act as the IR_1^X. Then we compute IR_2^X by implementing RDFG_Join operation between TP_2^X and IR_1^X on the common variable X. Similarly, we get the final results of job_1: IR_4^X. As job_1 also contains other variables, the format of IR is (B(X),(B(Y),B(Z),B(W))). The former denotes the bindings of X, and the latter gets the bindings of other variables.After a job is finished, we need to change the format of the IR to support further join. For $Plan_1$, the IR of the job_1 will become (B(Y),(B(Z),B(W),B(X))). Subsequently, we complete job_2 and job_3 in the similar way. At last, the results of job_3 will return the final answer to user.

B. Cost Estimation

According to the query plan, we run TP-based Spark tasks to answer a SPARQL query. Every task mainly contains two subtasks: parse single TP and join IRs. The cost of parsing single TP_i is composed of the cost of loading, prepartitioning and matching TP_i, which are denoted as |Read(TP_i)|, |Shuffle(TP_i)| and |Match(TP_i)|, respectively. All the three kinds of cost is in positive correlation with the size of MESG. The cost of joining two RDSG is composed of shuffle communication cost in distributed environment and join computation cost in single node: |Shuffle($RDSG_1$,$RDSG_2$)| and |Compute($RDSG_1$,$RDSG_2$)|, both of which are roughly proportional to the size of RDSG. All cost parameters are assigned a different weight based on the bandwidth of disk, network and memory.

$$COST = \sum_{i=1}^{N} Parse(TP_i) + \sum_{j=2}^{N} Join(Result_{j-1}, Match(TP_i)) = Parse(TP_1) + \sum_{i=2}^{N} Task_i \quad (1)$$

$$Task_i = Parse(TP_i) + Join(Result_{i-1}, Match(TP_i)) \quad (2)$$

$$|Parse(TP_i)| = \mu|Read(TP_i)| + \lambda|Shuffle(TP_i)| + \nu|Match(TP_i)| \quad (3)$$

$$|Join(RDSG_1, RDSG_2)| = \lambda|Shuffle(RDSG_1, RDSG_2)| + \omega|Compute(RDSG_1, RDSG_2)| \quad (4)$$

$$Result_i = \begin{cases} Join(Result_{i-1}, Match(TP_i)) & : \ 2 <= \text{i} <= \text{N} \\ Match(TP_i) & : \ \text{i} = 1 \end{cases} \quad (5)$$

Where,

N = the number of TP in the SPARQL query.

Parse(TP_i) = the cost of parsing TP_i.

Join($RDSG_1$,$RDSG_2$) = the cost of joining $RDSG_1$ and $RDSG_2$.

Read(TP_i) = the result of loading the corresponding MESG for TP_i.

Match(TP_i) = the result for single TP matching TP_i.

$Result_i$ = the IRs for TP_1, ... ,TP_i.

Shuffle($RDSG_1$,$RDSG_2$) = the size of the data needed to be moved in distributed environment.

Compute($RDSG_1$,$RDSG_2$)=the size of the data needed to be implemented join operation in single node.

Equation(1) estimates the total cost of processing a query. It includes the two cost of parsing TPs and joining IRs. It can also be seen as the summation of the cost of (N-1) tasks and parsing the first TP. Equation(2) further computes the

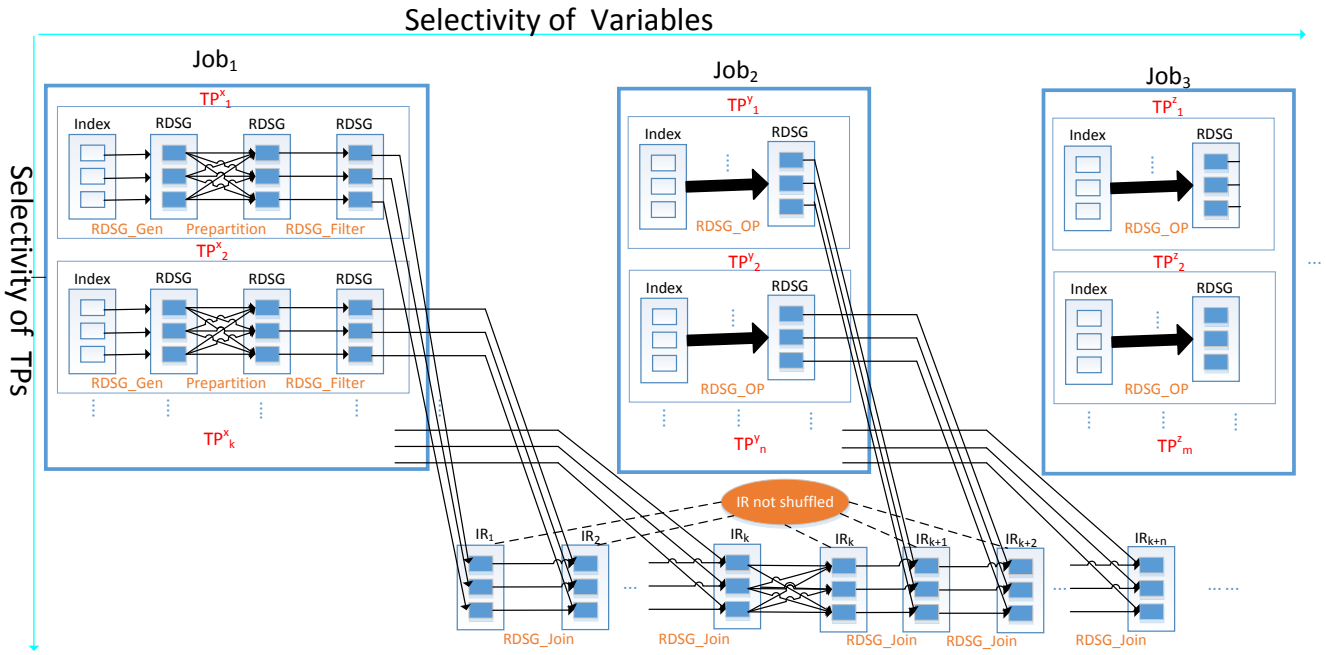


Figure 3. The RDSG_based Iterative Query Model

cost of every task. Equation 3 gives the cost of parsing a TP. Equation (4) shows the cost of implementing join operation in the SparkRDF. Equation(5) calculates the IRs for iterative query. Obviously according to Equation (1), we can reduce the total query cost from two aspects: cut down the number of task and the cost within every task. The former can be achieved by reducing the number of TPs. The latter can be implemented by reducing the number of IRs and size of MESGs(Equation 2).

C. Query Optimization

1) TR-SPARQL: TR-SPARQL refers to a type-restrictive SPARQL by passing variables' class message to corresponding TPs that contains the variable. It can cut down the number of task (remove the TPs whose predicate is rdf:type) and the cost of parsing every TP by forming a more restrictive(smaller) predicate.

PROPERTY 1. *In Semantic Web, every identified entity belongs to a specific class, denoted by predicate "type". The type message can be passed to other TPs that contains the variable, and form a more restrictive TP in a SPARQL query, which will improve the query efficiency greatly.*

PROOF. For TR-SPARQL, the result size of reading the new TP is equal or less than the previous result, that is $|\text{Read}(TP_i^{'})| <= |\text{Read}(TP_i)|$. Similarly, $|\text{Shuffle}(TP_i^{'})| <= |\text{Shuffle}(TP_i)|$, $|\text{Match}(TP_i^{'})| <= |\text{Match}(TP_i)|$. So we can have: $|\text{Parse}(TP_i^{'})| <= |\text{Parse}(TP_i)|$. According to Equation 5, we can get: $|\text{Result}_i^{'}| <= |\text{Result}_i|$. So the cost of join also becomes less: $|\text{Join}^{'}| <= |\text{Join}|$. Moreover, the number of TPs(N') is also not more than N. At last, we prove the property.

To elaborate the use of the property, consider the query example(Figure 4). As the type of X is given, we can sent the information to other TPs that contain X(TP_4, TP_6, TP_7). We apply the same way for Y and Z. Thus, the TPs whose predicate is "rdf:type" will be removed including TP_1, TP_2, TP_3. At the same time, it will help us to form more restrictive MESGs. For example, to $TP_3^{'}$, index file Student_emailAddress is chosen based on its predicate. This will avoid reading many unnecessary data compared to previous index because the predicate(emailAddress) is shared by many other classes such as Professor.

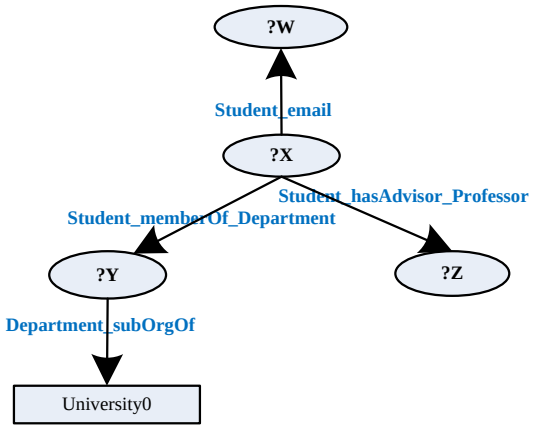


Figure 4. TR-SPARQL for the query example

2) On-demand Dynamic Prepartitioning: As previously discussed, the iterative join process in distributed environment needs two steps: first move the records with the same join key into a node, then implement the join operation within the node. When the size of RDSG is large, the cost of shuffling data will be quite high. Here a on-demand dynamic prepartitioning strategy is deployed to reduce the shuffling cost in the distributed join process. It prepartitions

the MESG only when they are on-demand loaded into the distributed memory, while ignoring the partitioning for other MESGs which are persisted in the HDFS. The prepartitioning scheme guarantees that the records sharing the same variable value will be read into the same partition, thus all the join operation can be finished in local node's memoryand no shuffle cost needs to be paid.

3) Optimal Query Plan: SPARQL usually require multiple joins on different variables. The query order has a significant impact on the performance. A optimal query plan should design a good execution order of jobs and TPs in a job so as to minimize the size of IR. We use a selectivity-based algorithm to create the plan. Specifically speaking, we should first determine the joining order of variables. Then we make the order of TPs in a job. Before describing our query optimization processing, we would like to give some definitions about the join process.

Definition 2: **Selectivity Score (SS)**. A parameter that is used to evaluate the selectivity of a TP or variable. Initially, Table II shows the SS for different formats of TP where symbol _ denotes a binding and symbol ? represents a variable. The more selective TP will have a higher SS. The SS of a variable is the product of its most selective TP's SS and the number of the kind of TPs. For example, SS(Y)= $SS(TP_5)*1=5$.

Definition 3: **Candidate Joining Variable (CV).** A variable that exists in two or more TPs.

Definition 4: **Joining Variable (JV)**. The CJV that has the highest Selectivity Score. Every JV corresponds to a job of the query plan.

Table II
THE SELECTIVITY SCORE FOR COMMON TPS.

Triple Format	Selectivity Score
?S __ __	5
__ __ ?O	5
?S __(CRC) ?O	4
?S __(CR) ?O	3
?S __(RC) ?O	3
?S __ ?O	2
__ ?P __	1
?S ?P __	1
__ ?P ?O	1

We compute the order of variables(jobs) in terms of two factors: (i) the number of IR, (ii) the format of IR. Both factors aim at reducing the size of IR. To the first factor, our priority is the variable having the highest SS since the cost of joining is roughly proportional to the size of their matching results. So choose a more restrictive CV as JV is quite essential. For example, if we first choose Z as JV, many unnecessary results are generated, which greatly adds the burden of further joins. As it is very common multiple variables have the same SS(X=Y), we break the ties by the second factor. The choice of JV directly affects the format of IR. For our example, if the JV is X, the IR of the first job will be (B(X),B(Y),B(W),B(Z)). Instead, the IR will be (B(Y),B(X)) if Y acts as JV. It is obvious that the latter is more inclined to generate the smaller results. So we should choose the variable with less TPs in the situation.

After the order of job is determined, all we need to concern is to decide the executing order of TPs in a job. As the final results of a job are the same no matter how TPs are ordered, we only need to consider how to reduce the number of IR. Similarly, we get the execution order of TP based on their SS in the job. According to the ordered list, we implement the join between the current TP owning highest SS with existing IRs iteratively until the job is finished.

VI. EVALUATION

In the section, we give the performance evaluation of SparkRDF.

Configuration: The experiment is implemented on a cluster with three machines. Each node has 16 GB DDR3 RAM, 8-core Intel Xeon(R) E5606 CPUs at 2.13GHz, 1.5TB disk. The nodes are connected by the network with the bandwith of 1000M/s. All the nodes use CentOS6.4 with the softwares Hadoop-1.0.4, Scala-1.10.1 and Spark-0.9.0.

Compared Systems: We compare SparkRDF with the centralized RDF-3X and distributed HadoopRDF. We run the RDF-3X(v3.0.7) on one of the nodes. HadoopRDF and SparkRDF were executed in the cluster.

Datasets and Queries: In our experiment, we use the widely-used LUBM [18] dataset with the scale of 10000, 20000 and 30000 universities, consisting of 1.3, 2.7 and 4.1 billion triples. To achieve a better performance, we replace all literals in RDF triples by a long id value(64-bit). For the LUBM queries, we chose 7 representative queries that provide a good mixture of both simple and complex structures. The selected set covers all variations of the LUBM test queries. At the same time, they are able to highlight the different decisions and characteristics of SparkRDF and the compared systems. They are roughly classified into 2 categories: selective(Q4,Q5,Q6) and unselective queries(Q1,Q2,Q3,Q7). For the former, there is usually bound subject or object in their TPs to get some specific information about a given subject or object. The latter contains some unselective TPs, aiming at querying the RDF graph by nonrestrictive input conditions. A short description on the chosen queries is provided in the Appendix.

Query Performance: Table III summarizes our comparison with HadoopRDF and RDF-3X(best times are boldfaced). The first observation is that SparkRDF performs much better than HadoopRDF for all queries. This can be attributed to the following three reasons: i)finer granularity of index scheme. The MESG guarantees that SparkRDF can process a TP by inputting a smaller index file than HadoopRDF; ii) efficient query model. SparkRDF uses

Table III
PERFORMANCE COMPARISON WITH HADOOPRDF AND RDF-3X.

	LUBM-10000			LUBM-20000			LUBM-30000		
	cluster systems		centralized system	cluster systems		centralized system	cluster systems		centralized system
	SparkRDF	HadoopRDF	RDF-3X	SparkRDF	HadoopRDF	RDF-3X	SparkRDF	HadoopRDF	RDF-3X
Q1(sec)	**478.5**	8475.4	2131.4	**1123.2**	>3h	4380.3	**1435.4**	>3.5h	failed
Q2(sec)	**11.9**	3425.2	13.8	**25.8**	>2h	28.9	**40.3**	>2.5h	43.5
Q3(sec)	**1.4**	6869.7	24.6	**1.4**	>2.5h	90.7	**1.4**	>3h	failed
Q4(sec)	14.4	11940.3	**0.7**	23.8	>4h	**0.8**	32.5	>8h	**0.8**
Q5(sec)	6.8	2587.5	**0.7**	10.9	>1h	**0.7**	13.0	>3h	**0.7**
Q6(sec)	10.3	7210.5	**0.6**	16.4	>2.5h	**0.7**	20.0	>3h	**0.7**
Q7(sec)	**54.6**	1911.2	101.5	**112.5**	>0.7h	198.5	**201.3**	>1h	853.0

memory-based RDSG model to process iterative join operations, which makes it capable of avoiding many expensive disk-based operations compared to HadoopRDF; iii) optimal query plan. HadoopRDF executes the query plan that greedily reduces the total number of remaining MapReduce joins without considering the joining selectivity, resulting in many unnecessary IRs. However, SparkRDF makes a optimal query plan based on Selectivity Score(SS) to reduce the size of IRs.

Another observation is that HadoopRDF outperformed RDF-3X in Q1,Q2,Q3,Q7, while RDF-3X did better in Q4,Q5,Q6. This result conforms to our initial conjecture: RDF-3X can achieve high performance for queries with high selectivity and bound objects or subjects, while SparkRDF did well for queries with unbound objects or subjects, low selectivity or large intermediate results joins.

Queries of the first class are — Q1,Q2,Q3 and Q7. In particular, Q1 is a complex query having many unselective TPs and needs multiple join operations. SparkRDF achieves almost 5× performance gain compared to RDF-3X, which is mainly thanks to the TR-SPARQL optimization strategy that reduces the number of join and the search space for TPs greatly. At the same time, SparkRDF implements all the join operations based on RDSG which caches large IRs in the distributed memory. It helps SparkRDF work well in processing the query with multiple join operations and large IRs. Meanwhile, it is worth noting that RDF-3X needs quite a lot of time (2131 and 4380 seconds for LUBM-10000, LUBM-20000) or even failed (for LUBM-30000) to implement the query for the three datasets. So RDF-3X might be not applicable in some real-time applications. Q3 is similar to Q1 except that one TP of Q3 does not have corresponding index file(Undergraduate_undergraduateDegreeFrom_University), which means a empty result should return. So SparkRDF finished the query plan without even launching a query job. That is the reason SparkRDF gains has a much performance advantage by factor of more than 20×. Q2 and Q7 represent another kind of typical unselective query containing few simple TPs. The number of TP in Q2 and Q7 are only 2 and 1, respectively. The number of join operations are 1 and 0 for RDF-3X. For the SparkRDF, it does not require any join operations because we can get the final results directly from its MESG index files. Thus all we need to do is just loading the index files and outputting corresponding results. So the efficient index scheme helps SparkRDF outperform the RDF-3X for the simple unselective queries.

In the case of the second kind of queries — Q4, Q5 and Q6, they all contain the TPs with bound objects which make them highly selective. So the size of intermediate and final results is also very small(their final results are 10, 10, 125 on all datasets). Benefiting from the set of permutations of (S,P,O) indexes, RDF-3X can process the queries efficiently by searching a small portion of index data. The index strategy provides RDF-3X a very fast way to look up TPs, similar to a hash table. Hence, RDF-3X completed the queries in less than 1 second. In comparison, SparkRDF needs to read the corresponding relatively large MESG index file and filter unrelated records by bound objects. But for all this, SparkRDF still answers all the queries within 15 seconds(14.4, 6.8, 10,3), which is an entirely acceptable response time. The results show although the performance of SparkRDF for queries with highly selective TPs is lower than RDF-3X, it is still adequate to satisfy the requirement of real-time.

Another important factor for evaluating RDF systems is how the performance scales with the size of data. RDF-3X fails to answer Q1 and Q3 when the data set size is 4.1 billion triples. It returns memory segmentation fault error messages. On the contrary, SparkRDF scales linearly and smoothly (no large variation is observed) when the scale of the datasets increases from 1.3 to 4.1 billion triples. It proves the good scalability of SparkRDF.

To summarize the results, it was evident that: i) for all the queries, SparkRDF outperformed HadoopRDF by a significant margin. ii) for both simple and complex join queries with low-selectivity, SparkRDF prevailed over RDF-3X; iii) Although for queries with highly selectivity , RDF-3X performed better. But SparkRDF still could process the queries within a short time. This confirms with our initial goal of real-time ; iv) for varying size of data, SparkRDF showed good scalability, while RDF-3X is not able to process some queries when the data increases due to the limitation of single machine. This also accords with our

initial goal of scalability.

VII. Conclusion

In the paper, we introduce the SparkRDF, an elastic discreted RDF graph processing engine based on Spark. We present a MESG-based graph splitting scheme to cut down the search space and avoid the overhead of memory. Then a RDSG-based iterative query model is used to implement the query process in distributed memory. Based on the data model and query model, we also introduce a cost estimation method and multiple optimization strategies. The experimental results prove that SparkRDF is a scalable and real-time RDF processing system.

References

[1] G. Klyne and J. J. Carroll, "Resource description framework (RDF): Concepts and abstract syntax," 2006.

[2] U. Consortium *et al.*, "The universal protein resource (UniProt)," *Nucleic acids research*, vol. 36, no. suppl 1, pp. D190–D195, 2008.

[3] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann, "DBpedia-A crystallization point for the Web of Data," *Web Semantics: science, services and agents on the world wide web*, vol. 7, no. 3, pp. 154–165, 2009.

[4] W. Wu, H. Li, H. Wang, and K. Q. Zhu, "Probase: A probabilistic taxonomy for text understanding," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 481–492.

[5] M. Fiorelli, "Linked Open Data," 2013.

[6] T. Neumann and G. Weikum, "The RDF-3X engine for scalable management of RDF data," *The VLDB Journal*, vol. 19, no. 1, pp. 91–113, 2010.

[7] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler, "Matrix Bit loaded: a scalable lightweight join query processor for RDF data," in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 41–50.

[8] E. PrudHommeaux, A. Seaborne *et al.*, "SPARQL query language for RDF," *W3C recommendation*, vol. 15, 2008.

[9] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[10] M. Husain, J. McGlothlin, M. M. Masud, L. Khan, and B. Thuraisingham, "Heuristics-based query processing for large rdf graphs using cloud computing," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 23, no. 9, pp. 1312–1327, 2011.

[11] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, pp. 10–10.

[12] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu, "TripleBit: a fast and compact system for large scale RDF data," *Proceedings of the VLDB Endowment*, vol. 6, no. 7, pp. 517–528, 2013.

[13] D. Borthakur, "HDFS architecture guide," *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, 2008.

[14] J. Huang, D. J. Abadi, and K. Ren, "Scalable SPARQL querying of large RDF graphs," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 1123–1134, 2011.

[15] N. Papailiou, D. Tsoumakos, I. Konstantinou, P. Karras, and N. Koziris, "H2RDF+: An Efficient Data Management System for Big RDF Graphs." SIGMOD, 2014.

[16] L. George, *HBase: the definitive guide*. " O'Reilly Media, Inc.", 2011.

[17] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang, "A distributed graph engine for web scale RDF data," in *Proceedings of the VLDB Endowment*, vol. 6, no. 4. VLDB Endowment, 2013, pp. 265–276.

[18] Y. Guo, Z. Pan, and J. Heflin, "Lubm: A benchmark for owl knowledge base systems," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 3, no. 2, pp. 158–182, 2005.

APPENDIX

We provide the SPARQL queries used in the experimental section: Q1-Q6 are the same as [7], Q7 corresponds to q14 of [18].

Q1: select ?x ?y ?z where { ?z ub:subOrganizationOf ?y . ?y rdf:type ub:University . ?z rdf:type ub:Department. ?x ub:memberOf ?z . ?x rdf:type ub:GraduateStudent . ?x ub:undergraduateDegreeFrom ?y . }
Q2: select ?x where {?x rdf:type ub:Course . ?x ub:name ?y . }
Q3: select ?x ?y ?z where { ?x rdf:type ub:UndergraduateStudent . ?y rdf:type ub:University . ?z rdf:type ub:Department . ?x ub:memberOf ?z . ?z ub:subOrganizationOf ?y . ?x ub:undergraduateDegreeFrom ?y . }
Q4: select ?x where { ?x ub:worksFor <http://www.Department0.University0.edu> . ?x rdf:type ub:FullProfessor. ?x ub:name ?y1 . ?x ub:emailAddress ?y2 . ?x ub:telephone ?y3 . }
Q5: select ?x where { ?x ub:subOrganizationOf <http://www.Department0.University0.edu> . ?x rdf:type ub:ResearchGroup}
Q6: select ?x ?y where { ?y ub:subOrganizationOf <http-://www.University0.edu> . ?y rdf:type ub:Department . ?x ub:worksFor ?y . ?x rdf:type ub:FullProfessor . }
Q7: select ?x where { ?x rdf:type ub:UndergraduateStudent . }