# 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*

Peiqin Gu
*Department of Computer Science*
*Zhejiang University*
*Hangzhou, China*
*peiqingu@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 retrieval of huge RDF graphs. Current systems and methods still can not process web scale RDF data effectively. In this paper, we introduce the SparkRDF, an elastic discreted semantic graph processing engine with distributed memory. To reduce the high I/O and communication costs for distributed processing platforms such as Hadoop, SparkRDF implements SPARQL query based on SPARK, a novel in-memory distributed computing framework for big data processing. All the intermediate join results are cached in the distributed memory to accelerate the process of next join. Instead of building a traditional massive set of permutations of (S,P,O) indexes which causes large space and memory overhead, we store the RDF data in the elastic discreted subgraphs based on the predicates and classes of the ontology. For SPARQL query optimization, SparkRDF's query processor dynamically generates an optimal execution plan for join queries, 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 answers non-selective joins faster than both current state-of-the-art distributed and centralized stores, scaling linearly to the amount of data and machines.***

***Keywords*-Large Semantic Graph, RDF, 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) 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 or Community driven projects are constructing large knowledge bases with billions of facts in many domains including bioinformatics, life sciences, social networks and so on. For example, The UniProt Knowledge base contains . The DBpedia and Probase also have billions of facts in RDF format. The Linked Open Data Project even announces 52 billion triples were published by 2012.

RDF data consists of triples where each triple presents a relationship between its subject (S) and object (O), the name of the relationship is given by the predicate (P), and the triple is represented as (S P O). The collections of triples form a labeled directed graph. SPARQL is the query language for RDF graph by providing subgraph matching queries. How to manage the large-scale RDF graph imposes technical challenges to the conventional storage layout, indexing and query processing. There are many researches focusing on the RDF data management. We can summarize these work as the following three categories:1)single-machine systems: SW-Store, Hexastore, TripleBit, and RDF-3x represent the typical examples. Obviously, the centralized solutions are vulnerable to the growth of the data size. 2)MapReduce-based distributed RDF systems: HadoopRDF, H2RDF and Shard. As the SPARQL graph pattern query needs to be implemented by multiple iterations, it leads to the large amount of disk i/o and communications generated. 3)Distributed in-memory RDF systems: Trinity.RDF.The main drawback of this system is that its performance is bound by the main memory capacity of the cluster, as the whole set of triples needs to be loaded in main memory. This is not a scalable approach, as it needs very strict cluster and memory limitations.

The key of processing large RDF graph is to construct efficient execution plans that can be adapted for the storage scheme and the optimization of SPARQL queries. SPARQL queries are made up of several Basic Graph Patterns(BGP), with the capabilities of filtering RDF triples and binding variables to values. When two triple patterns share common variables, the results of two triple patterns need to be joined. It will contain many iterative joins. Querying these huge graphs involves several stages: store the RDF data and creat indexes, read necessary data inside memory, compute binding values for BGP, implement joins over intermediate results, and build the final results of the query. Hence, a desired query processing system should meet the following requirements(i) reduce the data needed to be loaded in memory (ii)reduce the intermediate results (iii)reduce the number of join(iv)reduce the i/o and communication cost of intermediate results.

In this paper, we introduce SparkRDF, an elastic discreted

RDF graph processing system with distributed memory, which is capable of handling web scale RDF data (billion or enen more). Unlike existing systems that use a set of permutations of (S,P,O) indexes to manage RDF, SparkRDF splits the RDF graph into multiple discreted RDF subgraphs based on the nature of semantic graph: type and relationship. Thus it creates 5 kinds of elastic indexes with different grains (T,P,TP,PT,TPT) to cater for diverse BGP queries. These discreted index files on demand are modeled as a collection of objects partitioned across machines on Spark framework, a fast in-memory cluster computing system which is quite suitable for large-scale iterative computing jobs. Generated intermediate results remain in the distributed memory to support further joins. Based on the data model, several query optimization strategies are made to improve query efficiency including BGP merging, join order making and data prepartitioning.

We summarize the main contributions of this work as follows:

- We design a storage scheme to split RDF graph into elastic discreted subgraphs, which allows to index and query large RDF datasets at a low price to avoid the overhead of memory.
- We devise a cost model and several optimization strategies for RDF query processing with distributed memory based on the characteristics of Spark and Sparql. These approaches ensure high performance on web scale RDF data.
- We perform a experimental evaluation of our system. Results show that SparkRDF can be faster than both current state-of-the-art distributed and centralized stores for some query(complex/non-selective), while being only ... slower in selective onws. Meanwhile, it also can scale linearly to the amount of data and machines.

The remaining of this paper is organized as follows. Section 2 outlines related work and gives a brief introduction of Spark distributed computing platform. In Section 3 , we introduce the architecture of the *SparkRDF* system. Section 4 describes how we model and split big RDF graph. Section 5 gives the distributed SPARQL query processing techniques. In Section 6, we present the results of our experiments. Finally, we conclude and discuss the future work in Section 7.

## II. RELATED WORK AND SPARK

### A. *RDF Processing Systems*

### B. *SPARK Computing Platform*

## III. SYSTEM ARCHITECTURE

Figure 2 shows the architecture of SparkRDF. The SparkRDF consists of five modules: data preprocessing module, graph splitting module, distributed storage module, query parser module and distributed join module in memory.

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 splitting module. The distributed storage module subsequently persists the indexes into HDFS. Query parser module takes the SPARQL query from the user, dynamically generates the query plan and required input list for next step. Based on the input list, corresponding indexes are loaded into the distributed memory for further implementing iterative joins in memory as the query plan requested. At last, the query results are returned to the user.

## IV. DATA MODELING

In Spark, RDD is a distributed memory abstraction that lets programmers perform in-memory computations on large clusters. All the input data needs to be modeled as RDD to support further operations. However, the size of memory even on large clusters is far from the amount of massive RDF graph data. The RDF graph should be elaborately splitted into small parts to fit the cost of memory and query. Conventional systems, which use a set of permutations of (S,P,O) indexes, are not applicable since they need to query different kinds of big index files based on the type of BGP. It will easily lead to the overhead of memory. Vertical partitioning solution splits RDF graph by predicates. But its performance is drastically reduced as the data set size is increased. This is because RDF graph has a severe data skew based on different predicates, especially for some large datasets. For example, (make a statistics based on predicates.)It will also suffer from the overhead of memory and low efficiency when handling big predicate files. The predicate files need a more elastic and finer grained splitting scheme.

In fact, RDF data is created by the specific ontology, which represents knowledge as a hierachy of concepts within a domain using a shared vocabulary to denote the types, properties and interrelationships of those concepts. So in nature, semantic RDF graph data is organised by class and predicate. On the other hand, the SPARQL query, expressing a subgraph of RDF graph with several BGPs, is also closely related to entity type and predicate. Table 1 shows the percentage of type and predicate in common semantic datasets and corresponding SPARQL queries. So the information about class and predicate should be considered as the key to split RDF graph.

In this section, we describe how we model the RDF graph as elastic discreted subgraphs based on RDF Graph Splitter.

Table I
STATISTICS OF QUERIES.

| **Dataset** | **Percentage of Predicate** | **Percentage of Type** |
|---|---|---|
| LUBM | | |
| BTC-10 | | |

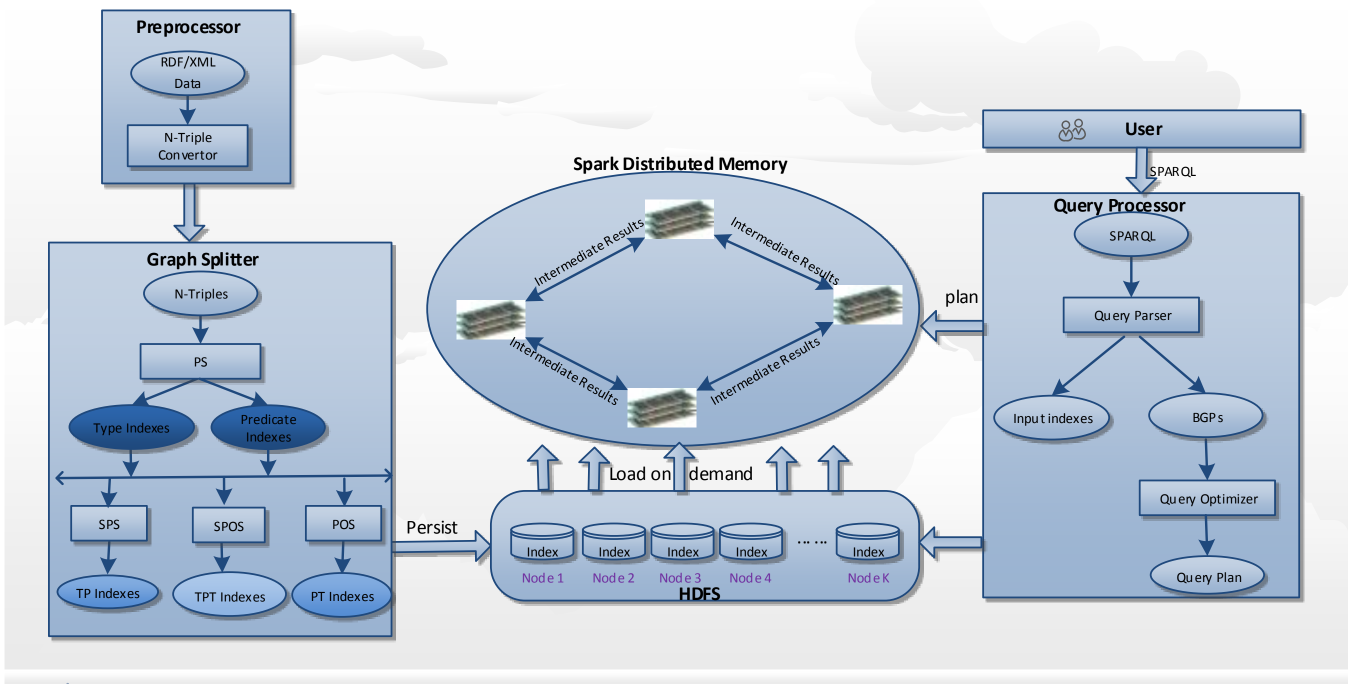

Figure 1. Biological Conceptional Network and Corresponding Reasoning Property Chains

Table II
EVALUATION OF

| Dataset | Predicates | Classes | Triples | Percentage of Type |
|---|---|---|---|---|
| LUBM-10000 | | | | |
| LUBM-20000 | | | | |

## A. *Type indexes and Predicate indexes*

In the first step, we construct the type indexes and predicate indexes 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 predicate 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 type 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.

## B. *Type&Predicate index*

When the RDF graph is splitted into discreted type indexes and predicate indexes, it enables us to select precise index files to reduce search time and space. But we actually only use the predicate information of a BGP, while other information such as its subject and object is missed. What's more, some special index files still remain quite large. So a finer-grained partitioning scheme should be made to further cut down the search space.

A predicate represents the binary relationship between subject and object, whose type information is usually given in a SPARQL query(Statistics). Thus we can divide the predicate files 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 (we cache the type information in last step to support fast indexing). Take the triple (Student0, memberOf, Department0 ) for example, as its subject and object belong to Student and Department respectively, tuple (Student0, Department0) move into three different kinds of index files named as Student_memberOf(TP index), memberOf_Department(PT index), Student_memberOf_Department(TPT index). The flexible index method makes SparkRDF capable of processing different SPARQL queries efficiently by selecting the minimum input based on these elastic discreted index files.

## C. *Example Data*

To achieve a better performance, we replace all literals in RDF triples by a long id value. Table shows the number and size of index files related to LUBM query 8( or 2) for data from 1,000 universities(the compressed total size is 5.659G). From the table, we can see the size of index files vary A typical example is **emailAddress** file, For SparkRDF, it only loads three index files(the total size is ) into memory compared to vertical partitioning(), which achieves great memory space gain and high efficiency. Table 2 gives the some sample data for .

### *D. Resident Distributed Semantic Graph*

SparkRDF models the index files using an unified concept called RDSG(Resilient Discreted Semantic Graph). It is the basic abstraction of index files in the distributed memory. Based on the API of Spark, we provide the following three main operators with which we implement query processing.

(i) **RDSG_Gen:** The operator creates a RDSG given the name of index file i.e., RDSG_Gen(Department) returns all the department instances, RDSG_Gen(Department_subOrganizationOf) returns all departments with corresponding parent organizations(RDSG1), RDSG_Gen(Student_memberOf_Department) returns all students with their department(RDSG2).

(ii) **RDSG_Filter:** The operator filters RDSG based on the subject or object constrains given in a BGP i.e., RDSG_Filter(RDSG1,"University0", flag) filters the departments whose parent organization is not "University0". Flag is a parameter that specifies whether the constrain is subject or object.

(iii) **RDSG_Join:** Generally, a query is comprised of multiple subpatterns that are linked together to form the final graph pattern. 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(RDSG1, RDSG2, RDSG1.S=RDSG2.O) returns the result of the subpattern consisting of BGP3 and BGP4

## V. Query Processing

In this section, we discuss how we process SPARQL queries in SparkRDF.

- BGP$_1$: (?X rdf:type ub:Student)
- BGP$_2$: (?Y rdf:type ub:Department)
- BGP$_3$: (?W rdf:type ub:Professor)
- BGP$_4$: (?X ub:membeOf ?Y)
- BGP$_5$: (?Y ub:subOrganizationOf University0)
- BGP$_6$: (?X ub:emailAddress ?Z)
- BGP$_7$: (?X rdf:hasAdvisor ?W)

### *A. Overview*

We represent a SPARQL query by a query graph Q, which is made up of multiple triple patterns denoted as BGP. Figure shows the query graph corresponding to Example 1.

With Q defined, the problem of SPARQL query processing can be transformed to the problem of subgraph matching. The solution is conducted as follows: Firstly decompose Q into an ordered sequence of BGP: BGP$_1$,...,BGP$_n$. Then we compute the matches for each BGP$_i$ based on RDSG and corresponding operators, and the matched intermediate results are used to find the matches for BGP$_{i+1}$ by implementing joins on the shared variable of the two BGPs. At last, the remaining results are the final answer of the query graph.

### *B. Cost Estimation*

We run BGP-based Spark jobs to answer a SPARQL query. Every job mainly contains three tasks, which are reading, matching and joining. The cost of reading and matching BGP$_i$, which are both in positive correlation with the size of index files, are denoted as |Read(BGP$_i$)| and |Match(BGP$_i$)|. The cost of joining two RDSG is composed of shuffle communication cost in distributed environment and join computation cost in single node: Shuffle(RDSG1,RDSG2), Computation(RDSG1,RDSG2), both of which are roughly proportional to the size of RDSG. The cost parameters are assigned a different weight according to the bandwidth of disk, network and memory.

$$\begin{aligned} COST =& \sum_{i=1}^{n}(\mu|Read(BGP_i)| + \nu|Match(BGP_i)|) + \\ & \sum_{i=2}^{n} Join(Result_{i-1}, Match(BGP_i)) \\ =& \sum_{i=2}^{n} Job_i + \mu|Read(BGP_1)| + \nu|Match(BGP_1)| \end{aligned} \tag{1}$$

$$\begin{aligned} Job_i =& \mu|Read(BGP_i)| + \nu|Match(BGP_i)| + \\ & Join(Result_{i-1}, Match(BGP_i) \end{aligned} \tag{2}$$

$$\begin{aligned} |Join(RDSG_1, RDSG_2)| =& \lambda|Shuffle(RDSG_1, RDSG_2)| + \\ & \omega|Computation(RDSG_1, RDSG_2)| \end{aligned} \tag{3}$$

$$Result_i = \begin{cases} Join(Result_{i-1}, Match(BGP_i)) & : \text{ i>= 2} \\ Match(BGP_i) & : \text{ i= 1} \end{cases} \tag{4}$$

Where,

Read(BGP$_i$)=the cost of loading selected input index files into RDSG.

Match(BGP$_i$)=the cost for single triple pattern matching BGP$_i$.

Result$_i$=the intermediate results for BGP$_1$, ... ,BGP$_i$.

Join(RDSG1,RDSG2)=the cost of joining RDSG1 and RDSG2.

Shuffle(RDSG1,RDSG2)=the cost of moving data in distributed environment.

Computation(RDSG1,RDSG2)=the cost of implementing join operation in single node.

Equation(1) and Equation(2) estimate the total cost of processing a query. It is the summation of the (n-1) individual job costs and the reading and matching cost of the first job(n is the number of BGP in the query). We can reduce the total query cost by two ways: cut down the number of jobs and the cost of every job. The former can be achieved by the optimization strategy called merging BGP(see section ). The latter can be implemented by optimizing join order and data partitioning(Equation 3). Equation (4) shows the cost of implementing join operation in the SparkRDF. Equation(5) calculates the intermediate results for multiple iterative S-parql query.

### C. Query Optimization

*Definition 1:* **Recilient Discreted Semantic Graph (RDSG)**. *Definition 2:* **Selectivity Score (Score)**. *Definition 3:* **Candidate Joining Variable.** A variable that exists in two or more BGPs. *Definition 4:* **Joining Variable**. *Definition 5:* .

PROPERTY 1.

PROPERTY 2.

PROPERTY 3.

### D. Overview

## VI. RESULTS

## VII. DISCUSSION AND CONCLUSION