

Efficient and Customizable Data Partitioning Framework for Distributed Big RDF Data Processing in the Cloud

Kisung Lee, Ling Liu, Yuzhe Tang, Qi Zhang, Yang Zhou
DiSL, College of Computing, Georgia Institute of Technology, Atlanta, USA
{kisung.lee, lingliu}@cc.gatech.edu, {yztang, qzhang90, yzhou86}@gatech.edu

Abstract—Big data business can leverage and benefit from the Clouds, the most optimized, shared, automated, and virtualized computing infrastructures. One of the important challenges in processing big data in the Clouds is how to effectively partition the big data to ensure efficient distributed processing of the data. In this paper we present a Scalable and yet customizable data PARTitioning framework, called SPA, for distributed processing of big RDF graph data. We choose big RDF datasets as our focus of the investigation for two reasons. First, the Linking Open Data cloud has put forwards a good number of big RDF datasets with tens of billions of triples and hundreds of millions of links. Second, such huge RDF graphs can easily overwhelm any single server due to the limited memory and CPU capacity and exceed the processing capacity of many conventional data processing software systems. Our data partitioning framework has two unique features. First, we introduce a suite of vertex-centric data partitioning building blocks to allow efficient and yet customizable partitioning of large heterogeneous RDF graph data. By efficient, we mean that the SPA data partitions can support fast processing of big data of different sizes and complexity. By customizable, we mean that the SPA partitions are adaptive to different query types. Second, we propose a selection of scalable techniques to distribute the building block partitions across a cluster of compute nodes in a manner that minimizes inter-node communication cost by localizing most of the queries on distributed partitions. We evaluate our data partitioning framework and algorithms through extensive experiments using both benchmark and real datasets. Our experimental results show that the SPA data partitioning framework is not only efficient for partitioning and distributing big RDF datasets of diverse sizes and structures but also effective for processing big data queries of different types and complexity.

I. INTRODUCTION

Cloud computing infrastructures are widely recognized as an attractive computing platform for efficient big data processing because it minimizes the upfront ownership cost for the large-scale computing infrastructure demanded by big data analytics. With Linking Open Data community project and World Wide Web Consortium (W3C) advocating RDF (Resource Description Framework) [6] as a standard data model for Web resources, we have witnessed a steady growth of both big RDF datasets and large and growing number of domains and applications capturing their data in RDF and performing big data analytics over big RDF datasets. For example, more than 52 billion RDF triples are published as of March 2012 on Linked Data [12] and about 6.46 billion triples are provided by the Data-gov Wiki [7] as of February 2013. Recently the UK (United Kingdom) government is publishing RDF about its legislation [8] with a SPARQL (a standard query language for RDF) query interface for its data sources [5].

Hadoop MapReduce programming model and Hadoop Distributed File System (HDFS) are one of the most popular distributed computing technologies for distributing big data processing across a large cluster of compute nodes in the

Cloud. However, processing the huge RDF data using Hadoop MapReduce and HDFS poses a number of new technical challenges. First, when viewing a big RDF dataset as an RDF graph, it typically consists of millions of vertices (subjects or objects of RDF triples) connected by millions or billions of edges (predicates of RDF triples). Thus triples are correlated and connected in many different ways. Random partitioning of big data into chunks through either horizontal (by triples) or vertical partitioning (by subject, object or predicate) is no longer a viable solution because data partitions generated by such a simple partitioning method tend to have high correlation with one another. Thus, most of the RDF queries need to be processed through multiple rounds of data shipping across partitions hosted in multiple compute nodes in the Cloud. Second, HDFS (and its attached storage systems) is excellent for managing big table like data where row objects are independent and thus big data can be simply divided into equal-sized chunks which can be stored in a distributed manner and processed in parallel efficiently and reliably. However, HDFS is not optimized for processing big RDF datasets of high correlation. Therefore, even simple retrieval queries can be quite inefficient to run on HDFS. Third but not the least, Hadoop MapReduce programming model is optimized for batch-oriented processing jobs over big data rather than real-time request and respond types of jobs. Thus, without correlation preserving data partitioning, Hadoop MapReduce alone is neither adequate for handling RDF queries nor suitable for structure-based reasoning on RDF graphs.

With these challenges in mind, in this paper, we present a Scalable and yet customizable data PARTitioning framework, called SPA, for distributed processing of big RDF graph data. Our data partitioning framework has two unique features. First, we introduce a suite of vertex-centric data partitioning building blocks, called extended vertex blocks, to allow efficient and yet customizable data partitioning of large heterogeneous RDF graphs by preserving the basic vertex structure. By efficient, we mean that the SPA data partitions can support fast processing of big data of different sizes and complexity. By customizable, we mean that one partitioning technique may not fit all. Thus the SPA partitions are by design adaptive to different data processing demands in terms of structural correlations. Second, we propose a selection of scalable parallel processing techniques to distribute the structured vertex block-based partitions across a cluster of compute nodes in a manner that minimizes inter-node communication cost by localizing most of the queries to independent partitions and by maximizing intra-node processing. By partitioning and distributing big RDF data using structured vertex blocks, we can considerably reduce the inter-node communication cost of complex query processing because most RDF queries can

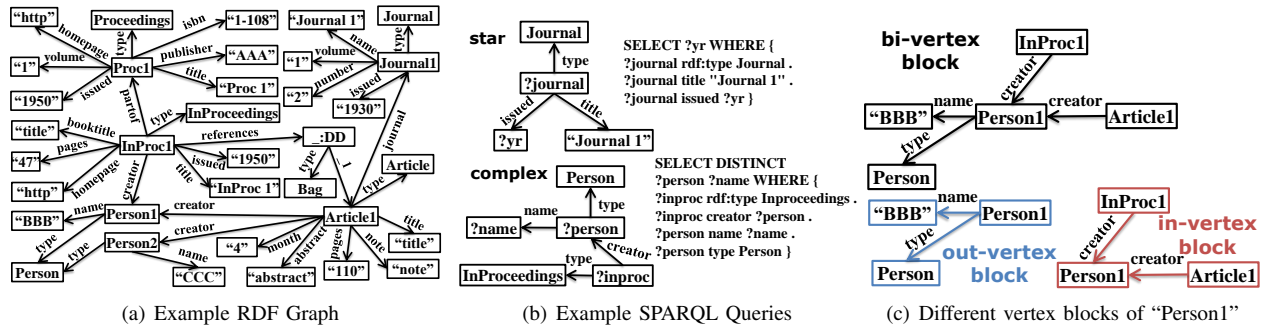


Fig. 1: RDF Examples

be evaluated locally on a partition server without requiring data shipping from other partition nodes. We evaluate our data partitioning framework and algorithms through extensive experiments using both benchmark and real datasets having totally different characteristics. Our experimental results show that the SPA data partitioning framework is efficient and customizable for partitioning and distributing big RDF datasets of diverse sizes and structures, and effective for processing real-time RDF queries of different types and complexity.

II. OVERVIEW

A. RDF and SPARQL

An RDF dataset consists of (subject, predicate, object) triples (or so-called SPO triples) with the predicate representing a relationship between its subject and object. An RDF dataset can be depicted as an RDF graph with subjects and objects as vertices and predicates as labeled edges connecting a pair of subject and object. Each edge is directed, emanating from its subject vertex to its object vertex. Fig. 1(a) shows an example RDF graph based on the structure of SP²Bench (SPARQL Performance Benchmark) [20].

SPARQL is a SQL-like standard query language for RDF recommended by W3C. Most SPARQL queries consist of multiple triple patterns, which are similar to RDF triples except that in each triple pattern, the subject, predicate and object may be a variable. We categorize SPARQL queries into two types: star and complex, based on the join characteristics of triple patterns. *Star* queries consist of subject-subject joins and each join variable is the subject of all the triple patterns involved. We refer to the remaining queries as *complex* queries. Fig. 1(b) shows two example SPARQL query graphs. The first is a star query requesting the year of publication of *Journal 1* and the second is a complex query requesting the names of all persons who are an author of at least one publication of *inproceeding* type. SPARQL query processing can be viewed as finding matching subgraphs in the RDF graph where RDF terms from those subgraphs may be substituted for the query variables.

B. System Architecture

Fig. 2 sketches the architecture of our RDF data partitioning framework SPA. The first prototype of SPA is implemented on top of Hadoop MapReduce and HDFS. The Hadoop SPA framework consists of one *coordinator* and a set of *worker* nodes (VMs) in the SPA cluster. The coordinator serves as the NameNode of HDFS and the JobTracker of Hadoop MapReduce and each worker serves as the DataNode of HDFS

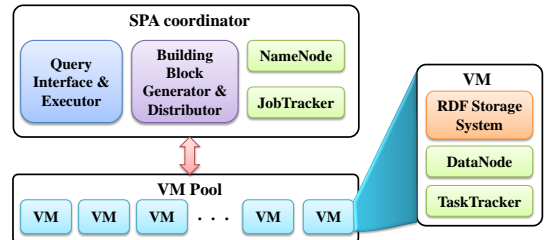


Fig. 2: SPA Architecture

and the TaskTracker of Hadoop MapReduce. To efficiently store the generated partitions, we utilize an RDF-specific storage system on each worker node.

The core components of our RDF data partitioning framework are the partition block generator and distributor. The generator uses a vertex-centric approach to construct partition blocks such that all triples of each partition block are residing in the same worker node. For a big RDF graph with a huge number of vertices, we need to carefully distribute all generated partition blocks across a cluster of worker nodes using a partition distribution mechanism. We provide an efficient distributed implementation of our data partitioning system on top of Hadoop MapReduce and HDFS.

III. SPA: DATA PARTITIONING FRAMEWORK

In this section we describe the SPA data partitioning framework, focusing on the two core components: constructing the partition blocks and distributing them across multiple worker nodes. In order to provide partitioning models that are customizable to different processing needs, we devise three types of partition blocks based on the vertex structure and the vertex access pattern. The goal of constructing partition blocks is to assign all triples of each partition block to the same worker node in order to support efficient query processing.

In terms of efficient query processing, there are two different types of processing: *intra-VM* processing and *inter-VM* processing. By *intra-VM* processing, we mean that a query Q can be fully executed in parallel on each VM by locally searching the subgraphs matching the triple patterns of Q , without any coordination from one VM to another. The coordinator simply sends Q to all worker nodes (VMs), without using Hadoop, and then merges the partial results received from all VMs to generate the final results of Q . By *inter-VM* processing, we mean that a query Q as a whole cannot be executed on any VM, and it needs to be decomposed into a set of subqueries such that each subquery can be evaluated by *intra-VM* processing. Thus, the processing of Q requires multiple rounds of coordination and data transfer across the

cluster of workers using Hadoop. In contrast to intra-VM processing, inter-VM communication cost can be extremely high, especially when the number of subqueries is not small and the size of intermediate results to be transferred across the network of worker nodes is large.

A. Constructing Extended Vertex Blocks

We center our data partitioning on a vertex-centric approach. By vertex-centric, it means that we construct a vertex block for each vertex. By extending a vertex block to an extended vertex block we can assign more triples which are close to a vertex in the same partition for efficient query processing. Before we formally define the concept of vertex block and the concept of extended vertex block, we first define some basic concepts of RDF graphs.

Definition 1. (RDF Graph) An RDF graph is a directed, labeled multigraph, denoted as $G = (V, E, \Sigma_E, l_E)$ where V is a set of vertices and E is a multiset of directed edges (i.e., ordered pairs of vertices). A directed edge $(u, v) \in E$ denotes a triple in the RDF model from subject u to object v . Σ_E is a set of available labels (i.e., predicates) for edges and l_E is a map from an edge to its label ($E \rightarrow \Sigma_E$).

In RDF datasets, multiple triples may have the same subject and object. Thus E is a multiset instead of a set. We consider that two triples in an RDF dataset are correlated if they share the same subject, object or predicate. For simplicity, only vertex-based correlation is considered in this paper. Thus we only consider three types of correlated triples as follows:

Definition 2. (Different types of correlated triples) Let $G = (V, E, \Sigma_E, l_E)$ be an RDF graph. For each vertex $v \in V$, we define a set of edges (triples) whose *subject* vertex is v as the **out-triples** of vertex v , denoted by $E_v^{out} = \{(v, o) | (v, o) \in E\}$. Similarly, we define a set of edges (triples) whose *object* vertex is v as the **in-triples** of a vertex $v \in V$, denoted by $E_v^{in} = \{(s, v) | (s, v) \in E\}$. We define **bi-triples** of a vertex $v \in V$ as the union of its **out-triples** and **in-triples**, denoted by $E_v^{bi} = E_v^{out} \cup E_v^{in}$.

Now we define the concept of *vertex block*, the basic building block for graph partitioning. A vertex block can be represented by a vertex ID and its connected triples. An intuitive way to construct the vertex block of a vertex v is to include all connected triples (i.e., bi-triples) of v regardless of their direction. However, for some RDF datasets, all their queries may request only triples in one direction from a vertex. For example, to evaluate the first SPARQL query in Fig. 1(b), the query processor needs only *out-triples* of a vertex which may be substituted for variable `?journal`. Since one partitioning technique cannot fit all, we need to provide customizable options that can be efficiently used for different RDF datasets and SPARQL query types. In addition to query efficiency, partition blocks should also minimize the triple redundancy by considering only necessary triples and reducing the disk space of generated partitions. This motivates us to introduce three different ways of defining the concept of vertex blocks based on the edge direction of triples.

Definition 3. (Vertex block) Let $G = (V, E, \Sigma_E, l_E)$ be an RDF graph. **Out-vertex block** of a vertex $v \in V$ is a subgraph of G which consists of v and its out-triples, denoted by $V_v^{out} = (V_v^{out}, E_v^{out}, \Sigma_{E_v^{out}}, l_{E_v^{out}})$ such that $V_v^{out} = \{v\} \cup$

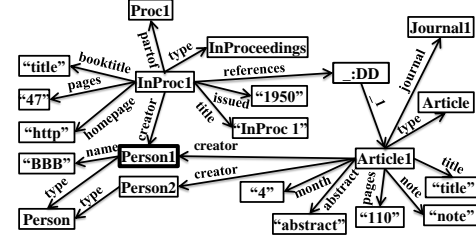


Fig. 3: 2-hop extended bi-vertex block of “Person1”

$\{v^{out} | v^{out} \in V, (v, v^{out}) \in E_v^{out}\}$. Similarly, **in-vertex block** of v is defined as $V_v^{in} = (V_v^{in}, E_v^{in}, \Sigma_{E_v^{in}}, l_{E_v^{in}})$ such that $V_v^{in} = \{v\} \cup \{v^{in} | v^{in} \in V, (v^{in}, v) \in E_v^{in}\}$. Thus we define the **vertex block** of v as the combination of both out-vertex block and in-vertex block, namely the bi-vertex block, and is formally defined as $V_v^{bi} = (V_v^{bi}, E_v^{bi}, \Sigma_{E_v^{bi}}, l_{E_v^{bi}})$ such that $V_v^{bi} = \{v\} \cup \{v^{bi} | v^{bi} \in V, (v, v^{bi}) \in E_v^{out} \text{ or } (v^{bi}, v) \in E_v^{in}\}$. We refer to vertex v as the *anchor* vertex of the vertex block centered at v . Every vertex block has an anchor vertex. We refer to those vertices in a vertex block, which are not the anchor vertex, as the *border* vertices of this vertex block.

In the rest of the paper we will use vertex blocks to refer to all three types of vertex blocks and use bi-vertex block to refer to the combination of in-vertex block and out-vertex block.

Fig. 1(c) shows three different vertex blocks of a vertex `Person1`. By locating all triples of a vertex block in the same partition, we can efficiently process all queries which request only triples directly connected to a vertex because those queries can be processed using intra-VM processing. For example, to process the star query in Fig. 1(b), we can run the query on each VM without any coordination with other VMs because it is guaranteed that all triples connected to a vertex which can be substituted for the variable `?journal` are located in the same VM.

Even though the vertex blocks are beneficial for star queries, which are common in practice, there are some RDF datasets in which chain queries or more complex queries may request more triples that are beyond those directly connected triples. Consider the second query in Fig. 1(b). Clearly there is no guarantee that any triple connecting from `?person` to `?name` is located in the same partition where those connected triples of vertex `?inproc` are located. Thus inter-VM processing may be required. This motivates us to introduce the concept of *k-hop extended vertex block* ($k \geq 1$).

Given a vertex $v \in V$ in an RDF graph G , *k-hop extended vertex block* of v includes not only directly connected triples but also all nearby triples which are indirectly connected to the vertex within *k-hop* radius in G . Thus, a vertex block of v is the same as the *1-hop* extended vertex block of v .

Similar to the vertex block, we define three different ways of defining the extended vertex block based on the edge direction of triples: *k-hop extended out-vertex block*, *k-hop extended in-vertex block* and *k-hop extended bi-vertex block*. Similarly, given a *k-hop* extended vertex block anchored at v , we refer to those vertices that are not the anchor vertex of the extended vertex block as the border vertices. For example, Fig. 3 shows the 2-hop extended bi-vertex block of vertex `Person1`. Due to the space limitation, we omit the formal definitions of the three types of the *k-hop extended vertex block* in this paper.

We below discuss briefly how to set the system-defined

parameter k . For a given RDF dataset, we can define k based on the common types of queries we wish to provide fast evaluation. If the radius of most of such queries is 3 hops or less, we can set $k = 3$. The higher k value is, the higher degree of triple replication each data partition will have and thus more queries can be evaluated using intra-VM processing. In fact, we find that the most frequent queries have 1 hop or 2 hops as their radius.

Once the system-defined k value is set, in order to determine if a given query can be evaluated using intra-VM processing or it has to pay inter-VM processing cost, all we need to do is to check whether every vertex in the corresponding query graph can be covered by any k -hop extended vertex block. Concretely, if there is any vertex in the query graph whose k -hop extended vertex block includes all triple patterns (edges) of the query graph, then we can say that the query can be evaluated using intra-VM processing under the current k -hop data partitioning scheme.

In the first prototype of SPA system, we implement our RDF data partitioning schemes using Hadoop MapReduce and HDFS. First, users can use the SPA configuration to set some system-defined parameters, including the number of worker nodes used for partitioning, say n , the type of extended vertex blocks (out-vertex, in-vertex or bi-vertex) and the k value of extended vertex blocks. Once the selection is entered, the SPA data partitioner will launch a set of Hadoop jobs to partition the triples of the given dataset into n partitions based on the direction and the k value of the extended vertex blocks.

The first Hadoop job reads the input RDF dataset stored in HDFS chunks. For each chunk, it examines triples one by one and groups triples by two partitioning parameters: (i) the type of extended vertex blocks (out-vertex, in-vertex or bi-vertex), which defines the direction of the k -hop extension on the input RDF graph; (ii) the k value of k -hop extended vertex blocks. If the parameters are set to generate 1-hop extended *bi-vertex* blocks, each triple is assigned to two extended vertex blocks (or one block if the subject and object of the triple is the same entity). This can be viewed as indexing this triple by both its subject and object. However, if the parameters are set to generate 1-hop extended *out-vertex* or *in-vertex* blocks, each triple will be assigned to one extended vertex block based on its subject or object respectively. We below describe how the SPA data partitioner takes the input RDF data and uses a cluster of compute nodes to partition it into n partitions based on the settings of 2-hop extended *out-vertex* blocks. In our first prototype, the SPA data partitioner is implemented in two phases: Phase 1 generates vertex blocks or 1-hop extended vertex blocks, and Phase 2 generates k -hop extended vertex blocks for $k > 1$. If $k = 1$, only Phase 1 is needed.

In the following discussion, we assume that the RDF dataset to be partitioned has been loaded into HDFS. The first Hadoop job is initialized with a set of map tasks in which each map task reads a fixed-size data chunk from HDFS (the chunk size is initialized as a part of the HDFS configuration). For each map task, it examines triples in a chunk one by one and returns a key-value pair where the key is the subject vertex and the value is the remaining part (predicate and object) of the triple. The local combine function of this Hadoop job groups all key-value pairs that have the same key into a local subject-based

triple group for each subject. The reduce function reads the outputs of all maps to further merge all local subject-based groups to form a vertex block with the subject as the anchor vertex. Each vertex group will have one anchor vertex and be stored in HDFS as a whole. Upon the completion of this first Hadoop job, we obtain the full set of 1-hop extended *out-vertex* blocks for the RDF dataset.

In the next Hadoop job, we examine each triple against the set of 1-hop extended out-vertex blocks, obtained in the previous phase, to determine which extended vertex blocks should include this triple in order to meet the k -hop guarantee. We call this phase the controlled triple replication. Concretely, we perform a full scan of the RDF dataset and examine in which 1-hop extended out-vertex blocks this triple should be added in two steps. First, for each triple, one key-value pair is generated with the subject of the triple as the key and the remaining part (predicate and object) of the triple as the value. Then we group all triples with the same subject into a local subject-based triple group. In the second step, for each 1-hop extended out-vertex block and each subject with its local triple group, we test if the subject vertex matches the extended out-vertex block. By matching, we mean that the subject vertex is the border vertex of the extended out-vertex block. If a match is found, then we create a key-value pair with the unique ID of the extended out-vertex block as the key and the subject vertex of the given local subject-based triple group as the value. This ensures that all out-triples of this subject vertex are added to the matching extended out-vertex block. If there is no matching extended out-vertex block for the subject vertex, the triple group of the subject vertex is discarded. This process is repeated until all the subject-based local triple groups are examined against the set of extended out-vertex blocks. Finally, we get a set of 2-hop extended out-vertex blocks by the reduce function, which merges all key-value pairs with the same key together such that the subject-based triple groups with the same extended out-vertex block ID (key) are integrated into the extended out-vertex block to obtain the 2-hop extended out-vertex block and store it as a HDFS file. For $k > 2$, we will repeat the above process until the k -hop extended vertex blocks are constructed. In our experiment section we report the performance of the SPA partitioner in terms of both the time complexity and data characteristics of generated partitions to illustrate why partitions generated by the SPA partitioner is much more efficient for RDF graph queries than random partitioning by HDFS chunks.

B. Distributing Extended Vertex Blocks

After we construct the extended vertex block for each vertex, we need to distribute all the extended vertex blocks across the cluster of worker nodes. Since there are usually a huge number of vertices in RDF data, we need to carefully assign each extended vertex block to a worker node. We consider three objectives for distribution of extended vertex blocks: (i) balanced partitions in terms of the storage cost, (ii) reduced replication and (iii) the time required to carry out the distribution of extended vertex blocks to worker nodes.

First, balanced partitions are essential for efficient query processing because one big partition in the imbalanced partitions may be a bottleneck for query processing and so will increase the overall query processing time. Second, by

the definition of the extended vertex block, a triple may be replicated in several extended vertex blocks. If vertex blocks having many common triples can be assigned to the same partition, the storage cost can be reduced considerably because duplicate triples are eliminated on each worker node. When many triples are replicated on multiple partitions, the enlarged partition size will increase the query processing cost on each worker node. Therefore, reducing the number of replicated triples is also an important factor critical to efficient query processing. Third, when selecting the distribution mechanism, we need to consider the processing time for the distribution of extended vertex blocks. This criterion is also important for RDF datasets that require more frequent updates.

In the first prototype of SPA partitioner, we implement two alternative techniques for distribution of extended vertex blocks to the set of worker nodes: hashing-based and graph partitioning-based. The hashing-based technique can achieve the balanced partitions and take less time to complete the distribution. It assigns the extended vertex block of vertex v to a partition using the hash value of v . We can also achieve the second goal – reducing replication by developing a smart hash function based on the semantics of the given dataset. For example, if we know that vertices having the same URI prefix are closely connected in the RDF graph, then we can use the URI prefix to get the hash value, which will assign those vertices to the same partition.

The graph partitioning-based distribution technique utilizes the partitioning result of the minimum cut vertex partitioning algorithm [15]. The minimum cut vertex partitioning algorithm divides the vertices of a graph into n smaller partitions by minimizing the number of edges between different partitions. We generate a vertex-to-worker-node map based on the result of the minimum cut vertex partitioning algorithm. Then we assign the extended vertex block of each vertex to the worker node using the vertex-to-node map. Since it is likely that close vertices are included in the same partition and thus placed on the same worker node, this technique focuses on reducing replication by assigning close vertices, which are more likely to have common triples in their extended vertex blocks, to the same partition.

C. Distributed Query Processing

Given that intra-VM processing and inter-VM processing are two types of distributed query processing in our framework, for a SPARQL query Q , assuming that the partitions are generated based on the k -hop extended vertex block, we construct a k -hop extended vertex block for each vertex in the query graph of Q . If any k -hop extended vertex block includes all edges in the query graph of Q , we can process Q using intra-VM processing because it is guaranteed that all triples which are required to process Q are located in the same partition. If there is no such k -hop extended vertex block, we need to process Q using inter-VM processing by splitting Q into smaller subqueries. In this paper, we split the query with an aim of minimizing the number of subqueries. We use Hadoop MapReduce and HDFS to join the intermediate results generated by the subqueries. Concretely, each subquery is executed on each VM and then the intermediate results of the subquery are stored in HDFS. To join the intermediate results of all subqueries, we use the join variable values as

key values of a Hadoop job.

IV. EXPERIMENTS

In this section, we report the experimental evaluation of our RDF data partitioning framework. Our experiments are performed using not only different benchmark datasets but also various real datasets. We first introduce basic characteristics of datasets used for our evaluation. We categorize the experimental results into two sets: (1) We show the effects of different types of vertex blocks and the distribution techniques in terms of the balanced partitions, replicated triples and data partitioning and distribution time. (2) We conduct the experiments on query processing latency.

A. Setup

We use a cluster of 21 machines (one is the coordinator) on Emulab [21]: each has 12 GB RAM, one 2.4 GHz 64-bit quad core Xeon E5530 processor and two 250GB 7200 rpm SATA disks. The network bandwidth is about 40 MB/s. When we measure the query processing time, we perform five cold runs under the same setting and show the *fastest* time to remove any possible bias posed by OS and/or network activity.

To store each generated partition on each machine, we use RDF-3X [18] version 0.3.5 which is a RDF storage system. We also use Hadoop version 0.20.203 running on Java 1.6.0 to run various partitioning algorithms and join the intermediate results generated by subqueries. In addition to our RDF data partitioning framework, we have implemented a random partitioning for comparison, which randomly assigns each triple to a partition. Random partitioning is often used as the default data partitioner in Hadoop MapReduce and HDFS.

To show the benefits of different building blocks of our partitioning framework, we experiment with five different extended vertex blocks: 1-hop extended out-vertex block (*1-out*), 1-hop extended in-vertex block (*1-in*), 1-hop extended bi-vertex block (*1-bi*), 2-hop extended out-vertex block (*2-out*) and 2-hop extended in-vertex block (*2-in*). Also hashing-based and graph partitioning-based distribution techniques are compared in terms of balanced partitions, triple replication and the time to complete the distribution. We use the graph partitioner METIS [4] version 5.0.2 with its default configuration to implement the graph partitioning-based distribution.

B. Datasets

We use two different benchmark generators and three different real datasets for our experiments. To generate benchmark datasets, we utilize two famous RDF benchmarks: LUBM (Lehigh University Benchmark) [11] having a university domain and SP²Bench [20] having a DBLP domain. We generate two datasets having different sizes from each benchmark: (i) LUBM500 (67M triples) and LUBM2000 (267M triples) covering 500 and 2000 universities respectively (ii) SP2B-100M and SP2B-200M having about 100 and 200 million triples respectively. For real datasets, we use (iii) DBLP (57M triples) [1] containing bibliographic descriptions in computer science, (iv) Freebase (101M triples) [2] which is a large knowledge base, and (v) DBpedia (288M triples) [3] which is structured information from Wikipedia. We have deleted any duplicate triples using one Hadoop job.

Fig. 4 shows some basic metrics of the seven RDF datasets. The datasets generated from each benchmark have almost the

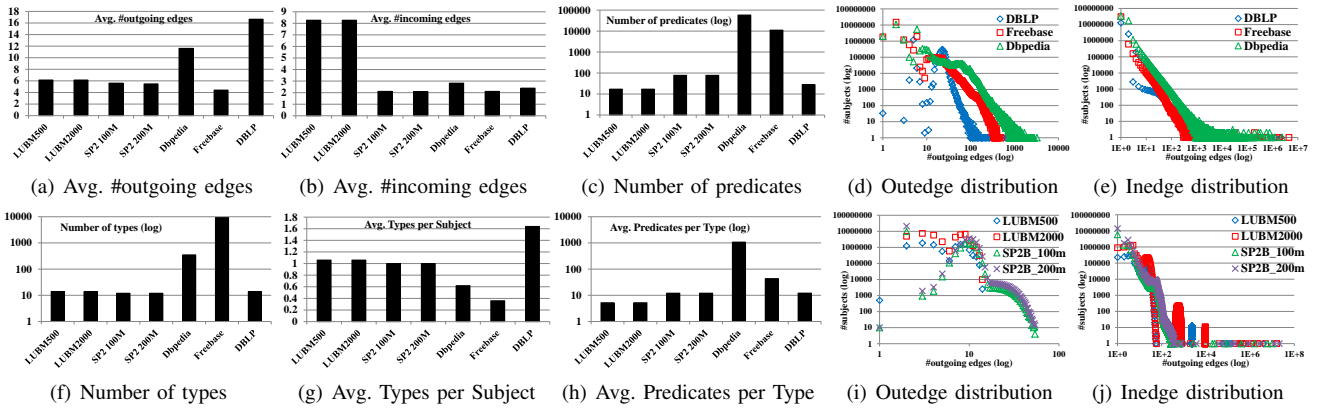


Fig. 4: Basic Characteristics of Seven RDF datasets

same characteristics regardless of their data size, similar to those reported in [9]. On the other hand, the real datasets have totally different characteristics. For example, while DBLP has 28 predicates, Dbpedia has about 58,000 predicates. To measure the metrics based on types (Fig. 4(f), 4(g), 4(h)), we have counted those triples having *rdf:type* as their predicate. The distribution of the number of outgoing/incoming edges in the three real datasets (Fig. 4(d) and 4(e)) have very similar power law-like distributions. It is interesting to note that, even though a huge number of objects have less than 100 incoming edges, there are still many objects having a significant number of incoming edges (e.g. more than 100,000 edges) in real datasets. Benchmark datasets have totally different distributions as shown in Fig. 4(i) and 4(j).

C. Generating Partitions

Fig. 5 shows the partition distribution in terms of the number of triples for different building blocks using the hashing-based and graph partitioning-based distribution techniques. Hereafter, we omit the results on LUBM500 and SP2B-100M which are almost the same with those on LUBM2000 and SP2B-200M respectively. The results show that we can get almost perfectly balanced partitions if we extend only outgoing triples when we construct the extended vertex blocks and use the hashing-based distribution technique. These results are related to the outgoing edge distribution in Fig. 4(d) and 4(i). Since most subjects have less than 100 outgoing triples and there is no subject having a huge number of outgoing triples, we can generate balanced partitions by distributing triples based on subjects. For example, the maximum number of outgoing triples on LUBM2000 is just 14 and 99.99% of subjects on Freebase have less than 200 outgoing triples. The generated partitions using the graph partitioning-based distribution technique are also well balanced when we extend only outgoing edges because METIS has generated almost uniform vertex partitions, even though they are a little less balanced than those using the hashing-based technique.

However, the results clearly show that it is hard to generate balanced partitions if we include incoming triples for our extended vertex blocks, regardless of the distribution technique. Since there are many objects having a huge number of incoming triples as shown in Fig. 4(e) and 4(j), partitions having such high in-degree vertices may be much bigger than the other partitions. For example, each of two object vertices on LUBM2000 has about 16 million incoming triples which is

about 6% of all triples. Table I shows the normalized standard deviation (i.e., the standard deviation divided by the mean) in terms of the number of triples on generated partitions to compare the balance between using the hashing-based and graph partitioning-based distribution techniques.

TABLE I: Balance of partitions (Normalized standard deviation)

| Dataset | 1-out | 1-in | 1-bi | 2-out | 2-in |
|---------------------|-------|-------|-------|-------|-------|
| LUBM2000 (hashing) | 0.000 | 0.515 | 0.203 | 0.002 | 0.424 |
| LUBM2000 (graph) | 0.003 | 0.520 | 0.325 | 0.003 | 0.578 |
| SP2B-200M (hashing) | 0.000 | 0.578 | 0.012 | 0.000 | 0.738 |
| SP2B-200M (graph) | 0.060 | 0.549 | 0.309 | 0.060 | 0.811 |
| DBLP (hashing) | 0.003 | 0.259 | 0.090 | 0.003 | 0.398 |
| DBLP (graph) | 0.036 | 0.268 | 0.196 | 0.041 | 0.626 |
| Freebase (hashing) | 0.002 | 0.335 | 0.165 | 0.003 | 0.379 |
| Freebase (graph) | 0.138 | 0.362 | 0.218 | 0.138 | 0.321 |
| Dbpedia (hashing) | 0.002 | 0.164 | 0.075 | 0.002 | 0.083 |
| Dbpedia (graph) | 0.284 | 0.201 | 0.201 | 0.290 | 0.133 |

Table II shows the extended vertex block construction time for different building blocks using the hashing-based and graph partitioning-based distribution techniques. Since the hashing-based distribution technique is incorporated into our implementation using Hadoop for efficient processing, we do not separate the distribution time and the extended vertex block construction time.

TABLE II: Extended vertex block construction time (sec)

| Dataset | 1-out | 1-in | 1-bi | 2-out | 2-in |
|---------------------|-------|------|------|-------|------|
| LUBM2000 (hashing) | 276 | 350 | 1015 | 1727 | 2453 |
| LUBM2000 (graph) | 962 | 1683 | 1933 | 1363 | 2895 |
| SP2B-200M (hashing) | 196 | 264 | 797 | 1102 | 1420 |
| SP2B-200M (graph) | 700 | 849 | 1050 | 958 | 1445 |
| DBLP (hashing) | 88 | 89 | 254 | 414 | 489 |
| DBLP (graph) | 289 | 312 | 365 | 409 | 487 |
| Freebase (hashing) | 108 | 115 | 326 | 568 | 599 |
| Freebase (graph) | 399 | 403 | 505 | 567 | 585 |
| Dbpedia (hashing) | 266 | 272 | 961 | 2453 | 2189 |
| Dbpedia (graph) | 961 | 1055 | 1367 | 1593 | 1979 |

For the graph partitioning-based distribution technique, on the other hand, there is an additional processing time to run METIS as shown below. Concretely, to run the graph partitioning using METIS, we first need to convert the RDF datasets into METIS input files. It is interesting to note that the format conversion time to run METIS considerably varies for different datasets. Since each line in a METIS input file represents a vertex and its connected vertices, we need to group all connected vertices for each vertex in the RDF graph. We have implemented this conversion step using Hadoop for efficient processing. Our implementation consists of four Hadoop MapReduce jobs: (1) assign a unique integer ID to each vertex, (2) convert each subject to its ID, (3) convert each object to its ID, and 4) group all connected vertices of

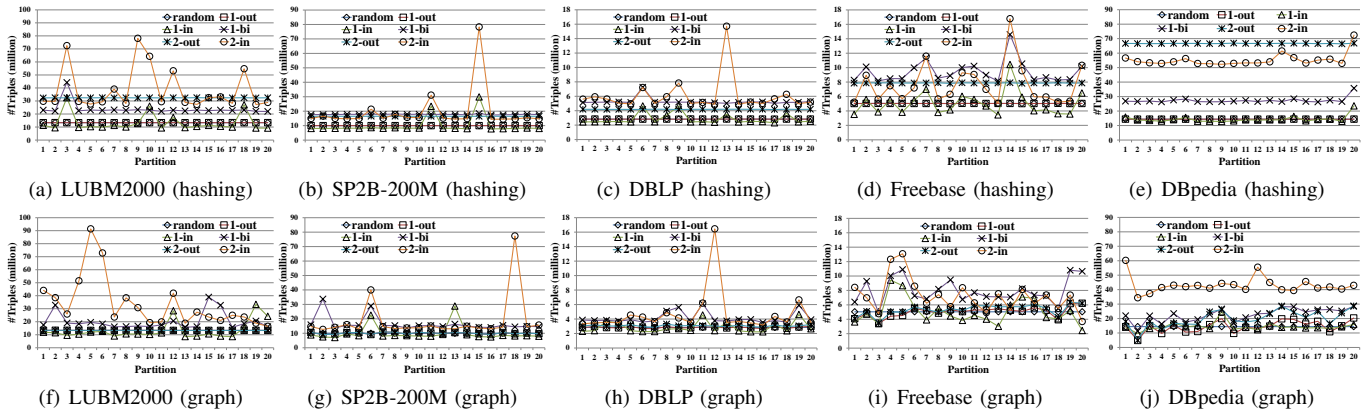


Fig. 5: Partition Distribution

each vertex. Table III shows the conversion time and running time of METIS. The conversion time of DBLP is about 7.5 hours even though it has the smallest number of triples. It took about 35 hours to convert the Freebase to a METIS input file. We believe that this is because there are much more vertices which are connected to a huge number of vertices on Freebase and DBLP as shown in Fig. 4(e). Since most existing graph partitioning tools require similar input formats, this result indicates that the graph partitioning-based distribution technique may be infeasible for some RDF datasets.

TABLE III: Graph partitioning time (sec)

| Dataset | conversion time | METIS running time |
|-----------|-----------------|--------------------|
| LUBM2000 | 2012 | 280 |
| SP2B-200M | 792 | 96 |
| DBLP | 27218 | 130 |
| Freebase | 91826 | 2610 |
| DBpedia | 3055 | 504 |

Fig. 6 shows, for different building blocks, the ratio of the total number of triples of all generated partitions to that of the original RDF data. A large ratio means that many triples are replicated across the cluster of machines through the data partitioning. Partitions using the 1-hop extended out-vertex or in-vertex blocks have 1 as the ratio because each triple is assigned to only one partition based on its subject or object respectively. As we mentioned above, the results show that the graph partitioning-based distribution technique can reduce the number of replicated triples considerably, compared to the hashing-based technique. This is because the graph partitioner generates components in which close vertices, which have high probability to share some triples in their extended vertex blocks, are assigned to the same component. Especially, there are only a few replicated triples for the 2-hop extended out-vertex blocks. However, for the other extended vertex blocks, the reduction is not so significant if we consider the overhead of the graph partitioning-based distribution technique.

D. Query Processing

To evaluate query processing performance on the generated partitions, we choose different queries from one benchmark dataset (LUBM2000) and one real dataset (DBpedia) as shown in Table IV. Since LUBM provides a set of benchmark queries, we select three among them. Note that Q11 and Q13 are slightly changed because our focus is not RDF reasoning. We create three queries for DBpedia because there is no representative benchmark queries.

Fig. 7 shows the query processing time of the queries for

TABLE IV: Queries

| |
|---|
| LUBM Q11: SELECT ?x ?y WHERE { ?x <subOrganizationOf> ?y . ?y <subOrganizationOf> <http://www.University0.edu> } |
| LUBM Q13: SELECT ?x WHERE { ?x <rdf:type> <GraduateStudent> . ?x <undergraduateDegreeFrom> <http://www.University0.edu> } |
| LUBM Q14: SELECT ?x WHERE { ?x <rdf:type> <UndergraduateStudent> } |
| DBpedia Q1: SELECT ?subject ?similar WHERE { <Barack_Obama> <subject> ?subject. ?similar <subject> ?subject. } |
| DBpedia Q2: SELECT ?person WHERE { ?person <rdf:type> <Person> } |
| DBpedia Q3: SELECT ?suc2 ?suc1 ?person WHERE { ?suc2 <successor> ?suc1 . ?suc1 <successor> ?person } |

different building blocks and different distribution techniques. All building blocks ensure intra-VM processing for LUBM Q14 and DBpedia Q2 because they are simple retrieval queries having only one triple pattern. For LUBM Q13, *random*, *1-in* and *2-in* needs inter-VM processing using Hadoop because Q13 is a star query and so all triples sharing the same subject should be located in the same partition for intra-VM processing. On the other hand, for DBpedia Q1 which is a *reverse* star query where all triple patterns share the same object, *random*, *1-out* and *2-out* needs inter-VM processing. For LUBM Q11 and DBpedia Q3 where the object of one triple pattern is linked to the subject of the other triple pattern, inter-VM processing is required for *random*, *1-out* and *1-in*. The results clearly show that using inter-VM processing is slower than using intra-VM processing because of the communication cost across the cluster of machines to join intermediate results generated from subqueries. Furthermore, each Hadoop job usually has about 10 seconds of initialization overhead regardless of its input data size. In terms of the intra-VM processing, the difference between the hashing-based and the graph partitioning-based distribution techniques is not significant for most cases even though they have different balance and replication levels as shown in Table I and Fig. 6. We think that this is because each partition is efficiently stored and indexed in the RDF-specific storage system, thus such level of difference on the balance and replication does not have a great effect on query processing. The only notable difference is when we use *2-out* for partitioning because the graph partitioning-based technique reduces the number of replicated triples considerably. Especially, on DBpedia where *2-out* using the hashing-based distribution technique generates many replicated triples, *2-out* using the graph partitioning-based distribution technique is three times faster than that using the hashing-based technique for DBpedia Q3.

In summary, we conclude that the most important factor of RDF data partitioning is to ensure the intra-VM processing

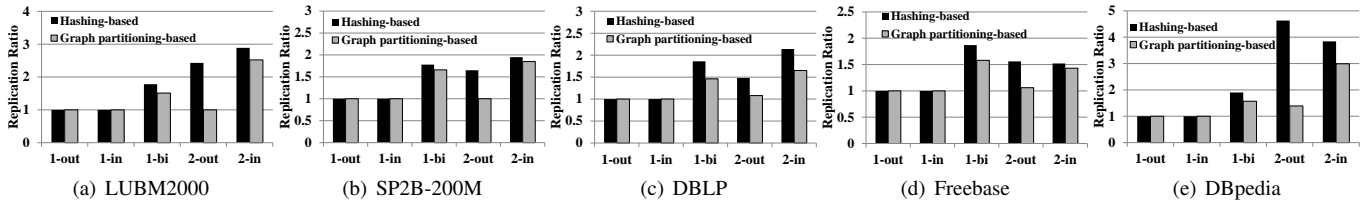


Fig. 6: Triple Replication

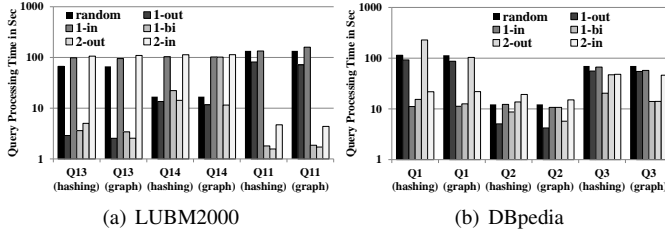


Fig. 7: Query Processing Time (sec)

for all (or most) given SPARQL queries because inter-VM processing using Hadoop is usually much slower than intra-VM processing. If all (or most) queries can be covered by only outgoing extension, the graph partitioning-based distribution technique should be preferred because it can reduce the number of replicated triples considerably. However, for all the other cases, the hashing-based distribution technique is a better distribution method if we consider the huge overhead of the graph partitioning-based technique. Also, it should be noted that running existing graph partitioner can be infeasible for some RDF datasets due to their complex structure.

V. RELATED WORK

Graph partitioning has been extensively studied in several communities for several decades [13], [15]. Recently a number of techniques have been proposed to process RDF queries on a cluster of machines. Most of them utilize existing distributed file systems such as HDFS to store and distribute RDF data. SHARD [19] directly stores RDF triples in HDFS as flat text files and runs one Hadoop job for each clause (triple pattern) of a SPARQL query. [10] uses two HBase tables to store RDF triples and, given a query, iteratively evaluates each triple pattern of the query on the two tables to generate final results. Because they use general distributed file systems which are not optimized for RDF data, their query processing is much less efficient than the state-of-the-art RDF storage and indexing technology [18]. [14] generates partitions using existing graph partitioning algorithms and stores them on RDF-3X. As we reported in Sec. IV, running an existing graph partitioner for large RDF data has a huge amount of overhead and is not feasible for some real RDF datasets.

In addition, two recent papers [17], [16] have proposed two different approaches to general processing of big graph data. [17] promotes to store a vertex and all its outgoing edges together and implemented a distributed system to show the parallel processing opportunity of their approach. [16] proposes to store a vertex and all its incoming edges together and shows that this can speed up some graph processing on a single server, such as PageRank. In comparison, [17] presents a system built using out-vertex block for both subject and object vertices, whereas [16] presents a single server solution using the in-vertex block for both subject and object vertices.

Both are special cases of our SPA partitioner. We argue that our SPA framework presents a systematic approach to graph partitioning and is more general and easily configurable with customizable performance guarantee.

VI. CONCLUSION

We have presented SPA, an efficient and customizable RDF data partitioning model for distributed processing of big RDF graph data in the Cloud. The main contribution is to develop a selection of scalable techniques to distribute the partition blocks across a cluster of compute nodes in a manner that minimizes inter-node communication cost by localizing most of the queries on distributed partitions. We show that the SPA approach is not only efficient for partitioning and distributing big RDF datasets of diverse sizes and structures but also effective for processing RDF queries of different types and complexity.

Acknowledgment: This work is partially sponsored by grants from NSF CISE NetSE program, SaTC program, and Intel ISTC on Cloud Computing.

REFERENCES

- [1] "About FacetedDBLP," <http://dblp.l3s.de/dblp++.php>.
- [2] "BTC 2012 Dataset," <http://km.aifb.kit.edu/projects/btc-2012/>.
- [3] "DBpedia 3.8 Downloads," <http://wiki.dbpedia.org/Downloads38>.
- [4] "METIS," <http://www.cs.umn.edu/metis>.
- [5] "Opening up Government," <http://data.gov.uk/sparql>.
- [6] "Resource Description Framework (RDF)," <http://www.w3.org/RDF/>.
- [7] "The Data-gov Wiki," <http://data.gov.tw.rpi.edu/wiki>.
- [8] "UK Legislation," <http://www.legislation.gov.uk/developer/formats/rdf>.
- [9] S. Duan, Kementsietsidis, Srinivas, and Udrea, "Apples and oranges: a comparison of rdf benchmarks and real rdf datasets," in *SIGMOD*, 2011.
- [10] C. Franke and et al., "Distributed Semantic Web Data Management in HBase and MySQL Cluster," in *IEEE CLOUD*, 2011.
- [11] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems," *Web Semant.*, vol. 3, no. 2-3, Oct. 2005.
- [12] T. Heath, C. Bizer, and J. Hendler, *Linked Data*, 1st ed., 2011.
- [13] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," in *Supercomputing*, 1995.
- [14] J. Huang, D. J. Abadi, and K. Ren, "Scalable SPARQL Querying of Large RDF Graphs," *PVLDB*, 4(21), pp. 1123–1134, August 2011.
- [15] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM J. Sci. Comput.*, 1998.
- [16] A. Kyrola, G. Blueloch, and C. Guestrin, "Graphchi: large-scale graph computation on just a pc," in *OSDI*. Berkeley, CA, USA: USENIX Association, 2012, pp. 31–46.
- [17] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*. New York, NY, USA: ACM, 2010, pp. 135–146.
- [18] T. Neumann and G. Weikum, "The RDF-3X engine for scalable management of RDF data," *The VLDB Journal*, vol. 19, no. 1, Feb. 2010.
- [19] K. Rohloff and R. E. Schantz, "Clause-iteration with MapReduce to scalably query datagraphs in the SHARD graph-store," in *DIDC*, 2011.
- [20] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel, "SP²Bench: A SPARQL Performance Benchmark," in *ICDE*, 2009.
- [21] B. White and et al., "An integrated experimental environment for distributed systems and networks," in *OSDI*, 2002.