Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning

Kisung Lee
Georgia Institute of Technology
kslee@gatech.edu

Ling Liu
Georgia Institute of Technology
lingliu@cc.gatech.edu

ABSTRACT

Massive volumes of big RDF data are growing beyond the performance capacity of conventional RDF data management systems operating on a single node. Applications using large RDF data demand efficient data partitioning solutions for supporting RDF data access on a cluster of compute nodes. In this paper we present a novel semantic hash partitioning approach and implement a Semantic HAsh Partitioning-Enabled distributed RDF data management system, called SHAPE. This paper makes three original contributions. First, the semantic hash partitioning approach we propose extends the simple hash partitioning method through direction-based triple groups and direction-based triple replications. The latter enhances the former by controlled data replication through intelligent utilization of data access locality, such that queries over big RDF graphs can be processed with zero or very small amount of inter-machine communication cost. Second, we generate locality-optimized query execution plans that are more efficient than popular multi-node RDF data management systems by effectively minimizing the inter-machine communication cost for query processing. Third but not the least, we provide a suite of locality-aware optimization techniques to further reduce the partition size and cut down on the inter-machine communication cost during distributed query processing. Experimental results show that our system scales well and can process big RDF datasets more efficiently than existing approaches.

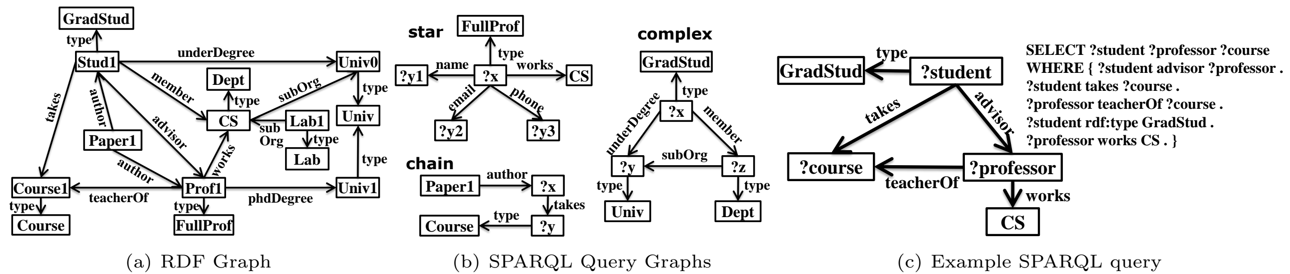
1. INTRODUCTION

The creation of RDF (Resource Description Framework) [5] data is escalating at an unprecedented rate, led by the semantic web community and Linked Open Data initiatives [3]. On one hand, the continuous explosion of RDF data opens door for new innovations in big data and Semantic Web initiatives, and on the other hand, it easily overwhelms the memory and computation resources on commodity servers, and causes performance bottlenecks in many existing RDF stores with query interfaces such as SPARQL [6]. Furthermore, many scientific and commercial online services must answer queries over big RDF data in near real time and achieving fast query response time requires careful partitioning and distribution of big RDF data across a cluster of servers.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 39th International Conference on Very Large Data Bases, August 26th - 30th 2013, Riva del Garda, Trento, Italy.
Proceedings of the VLDB Endowment, Vol. 6, No. 14
Copyright 2013 VLDB Endowment 2150-8097/13/14... $ 10.00.

A number of distributed RDF systems are using Hadoop MapReduce as their query execution layer to coordinate query processing across a cluster of server nodes. Several independent studies have shown that a sharp difference in query performance is observed between queries that are processed completely in parallel without any coordination among server nodes and queries that require even a small amount of coordination. When the size of intermediate results is large, the inter-node communication cost for transferring intermediate results of queries across multiple server nodes can be prohibitively high. Therefore, we argue that a scalable RDF data partitioning approach should be able to partition big RDF data into performance-optimized partitions such that the number of queries that hit partition boundaries is minimized and the cost of multiple rounds of data shipping across a cluster of sever nodes is eliminated or reduced significantly.

In this paper we present a semantic hash partitioning approach that combines locality-optimized RDF graph partitioning with cost-aware query partitioning for scaling queries over big RDF graphs. At the data partitioning phase, we develop a semantic hash partitioning method that utilizes access locality to partition big RDF graphs across multiple compute nodes by maximizing the intra-partition processing capability and minimizing the inter-partition communication cost. Our semantic hash partitioning approach introduces direction-based triple groups and direction-based triple replications to enhance the baseline hash partitioning algorithm by controlled data replication through intelligent utilization of data access locality. We also provide a suite of semantic optimization techniques to further reduce the partition size and increase the opportunities for intra-partition processing. As a result, queries over big RDF graphs can be processed with zero or very small amount of inter-partition communication cost. At the cost-aware query partitioning phase, we generate locality-optimized query execution plans that can effectively minimize the inter-partition communication cost for distributed query processing and are more efficient than those produced by popular multi-node RDF data management systems. To validate our semantic hash partitioning architecture, we develop SHAPE, a Semantic HAsh Partitioning-Enabled distributed RDF data management system. We experimentally evaluate our system to understand the effects of various system parameters and com-



(a) RDF Graph (b) SPARQL Query Graphs (c) Example SPARQL query

Figure 1: RDF and SPARQL

pare against other popular RDF data partitioning schemes, such as simple hash partitioning and min-cut graph partitioning. Experimental results show that our system scales well and can process big RDF datasets more efficiently than existing approaches. Although this paper focuses on RDF data and SPARQL queries, we conjecture that many of our technical developments are applicable to scaling queries and subgraph matching over general applications of big graphs.

The rest of the paper proceeds as follows. We give a brief overview of RDF, SPARQL and the related work in Section 2. Section 3 describes the SHAPE system architecture that implements the semantic hash partitioning. We present the locality-optimized semantic hash partitioning scheme in Section 4 and the partition-aware distributed query processing mechanisms in Section 5. We report our experimental results in Section 6 and conclude the paper in Section 7.

2. PRELIMINARY

2.1 RDF and SPARQL

RDF is a standard data model, proposed by World Wide Web Consortium (W3C). An RDF dataset consists of RDF triples and each triple has a subject, a predicate and an object, representing a relationship, denoted by the predicate, between the subject and the object. An RDF dataset forms a directed, labeled RDF graph, where subjects and objects are vertices and predicates are labels on the directed edges, each emanating from its subject vertex to its object vertex. The schema-free model makes RDF attractive as a flexible mechanism for describing entities and relationships among entities. Fig. 1(a) shows an example RDF graph, extracted from the Lehigh University Benchmark (LUBM) [8].

SPARQL [6] is a SQL-like standard query language for RDF, recommended by W3C. SPARQL queries consist of triple patterns, in which subject, predicate and object may be a variable. A SPARQL query is said to match subgraphs of the RDF data when the terms in the subgraphs may be substituted for the variables. Processing a SPARQL query Q involves graph pattern matching and the result of Q is a set of subgraphs of the big RDF graph, which match the triple patterns of Q.

SPARQL queries can be categorized into star, chain and complex queries as shown in Fig. 1(b). *Star* queries often consist of subject-subject joins and each join variable is the subject of all the triple patterns involved. *Chain* queries often consist of subject-object joins (i.e., the subject of a triple pattern is joined to the object of another triple pattern) and their triple patterns are connected one by one like a chain. We refer to the remaining queries, which are combinations of star and chain queries, as *complex* queries.

2.2 Related Work

Data partitioning is an important problem with applications in many areas. Hash partitioning is one of the dominating approaches in RDF graph partitioning. It divides an RDF graph into smaller and similar sized partitions by hashing over the subject, predicate or object of RDF triples. We classify existing distributed RDF systems into two categories based on how the RDF dataset is partitioned and how partitions are stored and accessed.

The first category generally partitions an RDF dataset across multiple servers using horizontal (random) partitioning, stores partitions using distributed file systems such as Hadoop Distributed File System (HDFS), and processes queries by parallel access to the clustered servers using distributed programming model such as Hadoop MapReduce [20, 12]. SHARD [20] directly stores RDF triples in HDFS as flat text files and runs one Hadoop job for each clause (triple pattern) of a SPARQL query. [12] stores RDF triples in HDFS by hashing on predicates and runs one Hadoop job for each join of a SPARQL query. Existing approaches in this category suffers from prohibitively high inter-node communication cost for processing queries.

The second category partitions an RDF dataset across multiple nodes using hash partitioning on subject, object, predicate or any combination of them. However, the partitions are stored locally in a database, such as a key-value store like HBase or an RDF store like RDF-3X [18] and accessed via a local query interface. In contrast to the first type of systems, these systems only resort to distributed computing frameworks, such as Hadoop MapReduce, to perform cross-server coordination and data transfer required for distributed query execution, such as joins of intermediate query results from two or more partition servers [7, 9, 19, 15]. Concretely, Virtuoso Cluster [7], YARS2 [9], Clustered TDB [19] and CumulusRDF [15] are distributed RDF systems which use simple hashing as their triple partitioning strategy, but differ from one another in terms of their index structures. Virtuoso Cluster partitions each index of all RDBMS tables containing RDF data using hashing. YARS2 uses hashing on the first element of all six alternately ordered indices to distribute triples to all servers. Clustered TDB uses hashing on subject, object and predicate to distribute each triple three times to the cluster of servers. CumulusRDF distributes three alternately ordered indices using a key-value store. Surprisingly, none of the existing data partitioning techniques by design aim at minimizing the amount of inter-partition coordination and data transfer involved in distributed query processing. Thus most existing work suffers from the high cost of cross-server coordination and data transfer for complex queries. Such heavy inter-partition communication incurs excessive network I/O

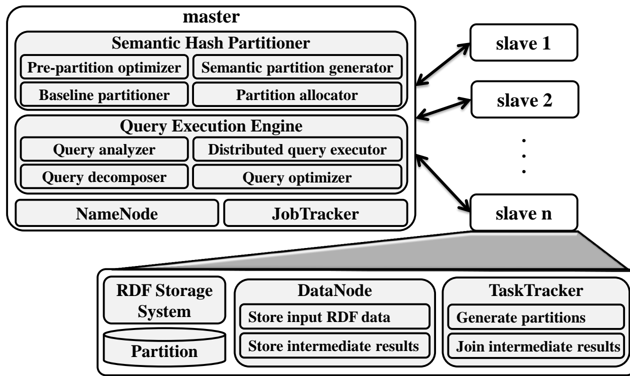


Figure 2: System Architecture

operations, leading to long query latencies.

Graph partitioning has been studied extensively in several communities for decades [10, 13]. A typical graph partitioner divides a graph into smaller partitions that have minimum connections between them, as adopted by METIS [13, 4] or Chaco [10]. Various efforts on graph partitioning have been dedicated to partitioning a graph into similar sized partitions such that the workload of servers hosting these partitions will be better balanced. [11] promotes the use of min-cut based graph partitioner for distributing big RDF data across a cluster of machines. It shows experimentally that min-cut based graph partitioning outperforms the simple hash partitioning approach. However, the main weaknesses of existing graph partitioners are the high overhead of loading the RDF data into the data format of graph partitioners and the poor scalability to large datasets. For example, we show in Section 6 that it is time consuming to load large RDF datasets to a graph partitioner and also the partitioner crashes when RDF datasets exceed a half billion triples. Orthogonal to graph partitioning efforts such as min-cut algorithms, several vertex-centric programming models are proposed for efficient graph processing on a cluster of commodity servers [17, 16] or for minimizing disk IOs required by in-memory graph computation [14]. Concretely, [17, 14] are known for their iterative graph computation techniques that can speed up certain types of graph computations. The techniques developed in [16] partition heterogeneous graphs by constructing customizable types of vertex blocks.

In comparison, this is the first work, to the best of our knowledge, which introduces a semantic hash partitioning method combined with a locality-aware query partitioning method. The semantic hash partitioning method extends simple hash partitioning by combining direction-based triple grouping with direction-based triple replication. The locality-aware query partitioning method generates semantic hash partition-aware query plans, which minimize inter-partition communication cost for distributed query processing.

3. OVERVIEW

We implement the first prototype system of our semantic hash partitioning method on top of Hadoop MapReduce with the master server as the coordinator and the set of slave servers as the workers. Fig. 2 shows a sketch of our system architecture.

Data partitioning. RDF triples are fetched into the data partitioning module hosted on the master server, which partitions the data stored across the set of slave servers. To work with big data that exceeds the performance capacity (e.g., memory, CPU) of a single server, we provide a distributed implementation of our semantic hash partitioning algorithm to perform data partitioning using a cluster of servers. The semantic hash partitioner performs three main tasks: (i) *Pre-partition optimizer* prepares the input RDF dataset for hash partitioning, aiming at increasing the access locality of each baseline partition generated in the next step. (ii) *Baseline hash partition generator* uses a simple hash partitioner to create a set of baseline hash partitions. In the first prototype implementation, we set the number of partitions to be exactly the number of available slave servers. (iii) *Semantic hash partition generator* utilizes the triple replication policies (see Section 4) to determine how to expand each baseline partition to generate its semantic hash partition with high access locality. We utilize the selective triple replication optimization technique to balance between the access locality and the partition size. On each slave server, either an RDF-specific storage system or a relational DBMS can be installed to store the partition generated by the data partitioning algorithms, process SPARQL queries over the local partition hosted by the slave server and generate partial (or intermediate) results. RDF-3X [18] is installed on each slave server of the cluster in Shape.

Distributed query processing. The master server also serves as the interface for SPARQL queries and performs distributed query execution planning for each query received. We categorize SPARQL query processing on a cluster of servers into two types: *intra*-partition processing and *inter*-partition processing.

By intra-partition processing, we mean that a query Q can be fully executed in parallel on each server by locally searching the subgraphs matching the triple patterns of Q, without any inter-partition coordination. The only inter-server communication cost required to process Q is for the master server to send Q to each slave server and for each slave server to send its local matching results to the master server, which simply merges the partial results received from all slave servers to generate the final results of Q.

By inter-partition processing, we mean that a query Q as a whole cannot be executed on any partition server, and it needs to be decomposed into a set of subqueries such that each subquery can be evaluated by intra-partition processing. Thus, the processing of Q requires multiple rounds of coordination and data transfer across a set of partition servers. In contrast to intra-partition processing, the communication cost for inter-partition processing can be extremely high, especially when the number of subqueries is not small and the size of intermediate results to be transferred across a network of partition servers is large.

4. SEMANTIC HASH PARTITIONING

The semantic hash partitioning algorithm performs data partitioning in three main steps: (i) Building a set of triple groups which are baseline building blocks for semantic hash partitioning. (ii) Grouping the baseline building blocks to generate baseline hash partitions. To further increase the access locality of baseline building blocks, we also develop an RDF-specific optimization technique that applies URI hierarchy-based grouping to merge those triple groups whose

anchor vertices share the same URI prefix prior to generating the baseline hash partitions. (iii) Generating k-hop semantic hash partitions, which expands each baseline hash partition via controlled triple replication. To further balance the amount of triple replication and the efficiency of query processing, we also develop the `rdf:type`-based triple filter during the k-hop triple replication. To ease the readability, we first describe the three core tasks and then discuss the two optimizations at the end of this section.

4.1 Building Triple Groups

An intuitive way to partition a large RDF dataset is to group a set of triples anchored at the same subject or object vertex and place the grouped triples in the same partition. We call such groups *triple groups*, each with an anchor vertex. Triple groups are used as baseline building blocks for our semantic hash partitioning. An obvious advantage of using the triple groups as baseline building blocks is that star queries can be efficiently executed in parallel using solely intra-partition processing at each sever because it is guaranteed that all required triples, from each anchor vertex, to evaluate a star query are located in the same partition.

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). $(u, v) \in E$ denotes a directed edge from u to 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 and thus E is a multiset instead of a set. Also the size of E ($|E|$) represents the total number of triples in the RDF graph G.

For each vertex v in a given RDF graph, we define three types of triple groups based on the role of v with respect to the triples anchored at v: (i) *subject-based* triple group (s-TG) of v consists of those triples in which their subject is v (i.e., outgoing edges from v) (ii) *object-based* triple group (o-TG) of v consists of those triples in which their object is v (i.e., incoming edges to v) (iii) *subject-object-based* triple group (so-TG) of v consists of those triples in which their subject or object is v (i.e., all connected edges of v). We formally define triple groups as follows.

Definition 2. (Triple Group) Given an RDF graph $G = (V, E, \Sigma_E, l_E)$, s-TG of vertex $v \in V$ is a set of triples in which their subject is v, denoted by $s\text{-}TG(v) = \{(u, w)|(u, w) \in E, u = v\}$. We call v the *anchor* vertex of s-$TG(v)$. Similarly, o-TG and so-TG of v are defined as $o\text{-}TG(v) = \{(u, w)|(u, w) \in E, w = v\}$ and $so\text{-}TG(v) = \{(u, w)|(u, w) \in E, v \in \{u, w\}\}$ respectively.

We generate a triple group for each vertex in an RDF graph and use the set of generated triple groups as baseline building blocks to generate k-hop semantic hash partitions. The subject-based triple groups are anchored at subject of the triples and are efficient for *subject-based* star queries in which the center vertex is the subject of all triple patterns (i.e., subject-subject joins). The total number of s-TG equals to the total number of distinct subjects. Similarly, o-TG and so-TG are efficient for *object-based* star queries (i.e., object-object joins), in which the center vertex is the object of all triple patterns, and *subject-object-based* star queries, in which the center vertex is the subject of some triple patterns and object of the other triple patterns (i.e., there exists at least one subject-object join) respectively.

4.2 Constructing Baseline Hash Partitions

The *baseline* hash partitioning takes the triple groups generated in the first step and applies a hash function on the anchor vertex of each triple group and place those triple groups having the same hash value in the same partition. We can view the baseline partitioning as a technique to bundle different triple groups into one partition. With three types of triple groups, we can construct three types of baseline partitions: subject-based partitions, object-based partitions and subject-object-based partitions.

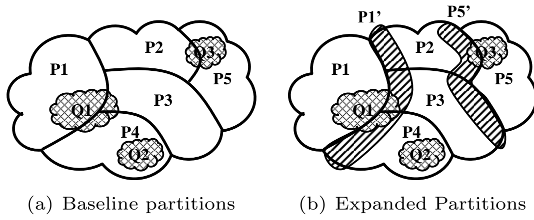
Definition 3. (Baseline hash partitions) Let $G = (V, E, \Sigma_E, l_E)$ denote an RDF graph and $TG(v)$ denote the triple group anchored at vertex $v \in V$. The baseline hash partitioning P of graph G results in a set of n partitions, denoted by $\{P_1, P_2, \ldots, P_n\}$ such that $P_i = (V_i, E_i, \Sigma_{E_i}, l_{E_i})$, $\bigcup_i V_i = V$, $\bigcup_i E_i = E$. If $V_i = \{v|hash(v) = i, v \in V\} \bigcup \{w|(v, w) \in TG(v), hash(v) = i, v \in V\}$ and $E_i = \{(v, w)|v, w \in V_i, (v, w) \in E\}$, we call the baseline partitioning P the s-TG hash partitioning. If $V_i = \{v|hash(v) = i, v \in V\} \bigcup \{u|(u, v) \in TG(v), hash(v) = i, v \in V\}$ and $E_i = \{(u, v)|u, v \in V_i, (u, v) \in E\}$, we call the baseline partitioning P the o-TG hash partitioning. In the above two cases, $E_i \bigcap E_j = \emptyset$ for $1 \leq i, j \leq n$, $i \neq j$. If $V_i = \{v|hash(v) = i, v \in V\} \bigcup \{w|(v, w) \in TG(v) \vee (w, v) \in TG(v), hash(v) = i, v \in V\}$ and $E_i = \{(v, w)|v, w \in V_i, (v, w) \in E\}$, we call the baseline partitioning P the so-TG hash partitioning.

We can verify the correctness of the baseline partitioning by checking the full coverage of baseline partitions and the disjoint properties for subject-based and object-based baseline partitions. In addition, we can further improve the partition balance across a cluster of servers by fine-tuning of triple groups with high degree anchor vertex. Due to space constraint, we omit further discussion on the correctness verification and quality assurance step in this paper.

4.3 Generating Semantic Hash Partitions

Using a hash function to map triple groups to baseline partitions has two advantages. It is simple and it generates well balanced partitions. However, a serious weakness of simple hash-based partitioning is the poor performance for complex non-star queries.

Considering a complex SPARQL query asking the list of graduate students who have taken a course taught by their CS advisor in Fig. 1(c), its query graph consists of two star query patterns chained together: one consists of three triple patterns emanating from variable vertex `?student`, and the other consists of two triple patterns emanating from variable vertex `?professor`. Assuming that the original RDF data in Fig. 1(a) is partitioned using the simple hash partitioning based on s-TGs, we know that the triples with predicates `advisor` and `takes` emanating from their subject vertex `Stud1` are located in the same partition. However, it is highly likely that the triple `teacherOf` and the triple `works` emanating from a different but related subject vertex `Prof1`, the advisor of the student `Stud1`, are located in a different partition, because the hash value for `Stud1` is different from the hash value of `Prof1`. Thus, this complex query needs to be evaluated by performing inter-partition



(a) Baseline partitions (b) Expanded Partitions

Figure 3: Partition Expansion

processing, which involves splitting the query into a set of subqueries as well as cross-server communication and data shipping. Assume that we choose to decompose the query into the following two subqueries: 1) SELECT ?student ?professor ?course WHERE {?student advisor ?professor . ?student takes ?course . ?student rdf:type GradStud . } 2) SELECT ?professor ?course WHERE {?professor teacherOf ?course . ?professor works CS .}. Although each subquery can be performed in parallel on all partition servers, we need to ship the intermediate results generated from each subquery across a network of partition servers in order to join the intermediate results of the subqueries, which can lead to a high cost of inter-server communication.

Taking a closer look at the query graph in Fig.1(c), it is intuitive to observe that if the triples emanating from the vertex `Stud1` and the triples emanating from its one hop neighbor vertex `Prof1` are residing in the same partition, we can effectively eliminate the inter-partition processing cost and evaluate this complex query by only intra-partition processing. This motivates us to develop a locality-aware semantic hash partitioning algorithm through hop-based controlled triple replication.

4.3.1 Hop-based Triple Replication

The main goal of using hop-based triple replication is to create a set of semantic hash partitions such that the number of queries that can be evaluated by intra-partition processing is increased and maximized. In contrast, with the baseline partitions only star queries can be guaranteed for intra-partition processing.

Fig. 3 presents an intuitive illustration of the concept and benefit of the semantic hash partitioning. By the baseline hash partitioning, we have five baseline partitions P1, P2, P3, P4 and P5 and three queries shown in Fig. 3(a). For brevity, we assume that the baseline partitions are generated using *s-TG*s or *o-TG*s in which each triple is included in only one triple group. Clearly, Q_2 is an intra-partition query and Q_1 and Q_3 are inter-partition queries. Evaluating Q_1 requires to access triples located in and nearby the boundaries of the three partitions: P1, P3 and P4. One way to process Q_1 is to use the baseline partitions. Thus, Q_1 should be split into three subqueries, and upon completion of the subqueries, their intermediate results are joined using Hadoop jobs. The communication cost for inter-partition processing depends on the number of subqueries, the size of the intermediate results and the size of the cluster (number of partition servers involved).

Alternatively, we can expand the triple groups in each baseline partition by using hop-based triple replication and execute queries over the semantic hash partitions instead. In Fig. 3(b), the shaded regions, P1' and P5', represent a set of replicated triples added to partition P1 and P5 respectively. Thus, P1 is replaced by its semantic hash partition, denoted by P1 $\bigcup$ P1'. Similarly, P5 is replaced by P5 $\bigcup$ P5'. With the semantic hash partitions, all three queries can be executed by intra-partition processing without any coordination with other partitions and any join of intermediate results, because all triples required to evaluate the queries are located in the expanded partition.

Before we formally introduce the k-hop semantic hash partitioning, we first define some basic concepts of RDF graphs.

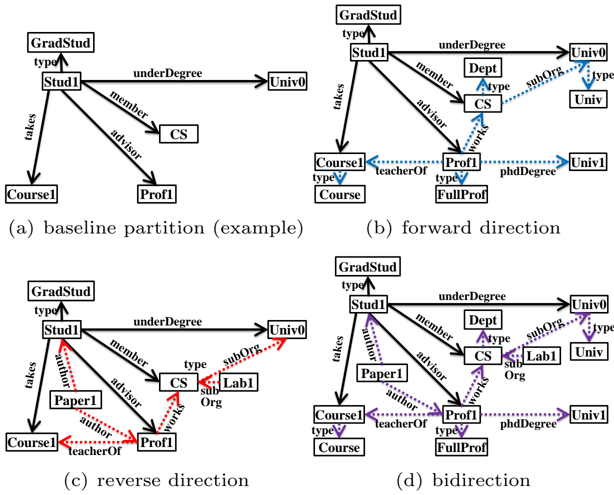
Definition 4. (Path) Given an RDF graph $G = (V, E, \Sigma_E, l_E)$, a **path** from vertex $u \in V$ to another vertex $w \in V$ is a sequence of vertices, denoted by $v_0, v_1, \dots, v_k$, such that $v_0 = u$, $v_k = w$, $\forall m \in [0, k-1] : (v_m, v_{m+1}) \in E$. We also call this path the *forward direction path*. A *reverse direction path* from vertex u to vertex w is a sequence of vertices, denoted by $v_0, v_1, \dots, v_k$, such that $v_0 = u$, $v_k = w$, $\forall m \in [0, k-1] : (v_{m+1}, v_m) \in E$. A *bidirection path* from vertex u to vertex w is a sequence of vertices, denoted by $v_0, v_1, \dots, v_k$, such that $v_0 = u$, $v_k = w$, $\forall m \in [0, k-1] : (v_m, v_{m+1}) \in E$ or $(v_{m+1}, v_m) \in E$. The **length** of the path $v_0, v_1, \dots, v_k$ is k.

Definition 5. (Hop count) Given an RDF graph $G = (V, E, \Sigma_E, l_E)$, we define the **hop count** from vertex $u \in V$ to vertex $v \in V$, denoted by $hop(u, v)$, as the minimum length of all possible forward direction paths from u to v. We also define the hop count from vertex u to edge $(v, w) \in E$, denoted by $hop(u, vw)$, as "$1 + hop(u, v)$". The reverse hop count from vertex u to vertex v, $reverse_hop(u, v)$, is the minimum length of all possible reverse direction paths from u to v. The bidirection hop count from vertex u to vertex v, $bidirection_hop(u, v)$, is the minimum length of all possible bidirection paths between u to v. The hop count $hop(u, v)$ is zero if $u = v$ and ∞ if there is no forward direction path from u to v. Similar exceptions exist for $reverse_hop(u, v)$ and $bidirection_hop(u, v)$.

Now we introduce k-hop expansion to control the level of triple replication and balance between the query performance and the cost of storage. Concretely, each *expanded* partition will contain all triples that are within k hops from *any* anchor vertex of its triple groups. k is a system-defined parameter and $k = 2$ is the default setting in our first prototype. One way to optimize the setting of k is to utilize the statistics collected from representative historical queries such as frequent query patterns.

We support three approaches to generate k-hop semantic hash partitions based on the direction of triple expansion: i) forward direction-based, ii) reverse direction-based, and iii) bidirection-based. The main advantage of using direction-based triple replication is to enable us to selectively replicate the triples within k hops. This selective replication strategy offers a configurable and customizable means for users and applications of our semantic hash partitioner to control the amount of triple replications desired. This is especially useful when considering a better tradeoff between the gain of minimizing inter-partition processing and the cost of local storage and local query processing. Furthermore, by enabling direction-based triple expansion, we provide k-hop semantic hash partitioning with a flexible combination of tripe groups of different types and k-hop triple expansion to baseline partitions along different directions.

Let $G = (V, E, \Sigma_E, l_E)$ be the RDF graph of the original dataset and $\{P_1, P_2, \dots, P_m\}$ denote the baseline partitions



(a) baseline partition (example) (b) forward direction

(c) reverse direction (d) bidirection

Figure 4: Semantic hash partitions from `Stud1`

on G. We formally define the k-hop *forward* semantic hash partitions as follows.

DEFINITION 6. (k-hop *forward* semantic hash partition) The k-hop forward semantic hash partitions on G are expanded partitions from $\{P_1, P_2, \ldots, P_m\}$, by adding (replicating) triples that are within k hops from any *anchor* vertex in each baseline partition along the *forward* direction, denoted by $\{P_1^k, P_2^k, \ldots, P_m^k\}$, where each baseline partition $P_i = (V_i, E_i, \Sigma_{E_i}, l_{E_i})$ is expanded into $P_i^k = (V_i^k, E_i^k, \Sigma_{E_i^k}, l_{E_i^k})$ such that $E_i^k = \{e | e \in E, \exists v_{anchor} \in V_i : hash(v_{anchor}) = i \text{ and } hop(v_{anchor}, e) \leq k\}$, and $V_i^k = \{v | (v, v') \in E_i^k$ or $(v'', v) \in E_i^k\}$.

We omit the formal definitions of the k-hop *reverse* and *bidirection* semantic hash partitions, in which the only difference is using $reverse_hop(v_{anchor}, e)$ and $bidirection_hop(v_{anchor}, e)$, instead of using $hop(v_{anchor}, e)$, respectively.

Fig. 4 illustrates three direction-based 2-hop expansions from a triple group with anchor vertex `Stud1` shown in Fig. 4(a). Fig. 4(b) shows the 2-hop forward semantic hash partition, where dotted edges represent replicated triples by 2-hop expansion from the baseline partition. Fig. 4(c) shows the 2-hop reverse semantic hash partition (i.e., from object to subject). Fig. 4(d) shows the semantic hash partition generated by 2-hop bidirection expansion from `Stud1`.

4.3.2 Benefits of k-hop semantic hash partitions

The main idea of the semantic hash partitioning approach is to use a flexible triple replication scheme to maximize intra-partition processing and minimize inter-partition processing for RDF queries. Compared to existing data partitioning algorithms that produce disjoint partitions, the biggest advantage of using the k-hop semantic hash partitioning is that, by selectively replicating some triples across multiple partitions, more queries can be executed using intra-partition processing.

We employ the concept of eccentricity, radius and center vertex to formally characterize the benefits of the k-hop semantic hash partitioning scheme. Let $G = (V, E, \Sigma_E, l_E)$ denote an RDF graph.

DEFINITION 7. (Eccentricity) The **eccentricity** ϵ of a vertex $v \in V$ is the greatest bidirection hop count from v to any edge in G and formally defined as follows:

$$\epsilon(v) = \max_{e \in E} bidirection_hop(v, e)$$

The eccentricity of a vertex in an RDF graph shows how far a vertex is from the vertex most distant from it in the graph. In the above definition, if we use the forward or reverse hop count instead, we can obtain the *forward* or *reverse* eccentricity respectively.

DEFINITION 8. (Radius and Center vertex) We define the **radius** of G, $r(G)$, as the minimum (bidirection) eccentricity of any vertex $v \in V$. The **center vertices** of G are the vertices whose (bidirection) eccentricity is equal to the radius of G.

$$r(G) = \min_{v \in V} \epsilon(v), \;\; center(G) = \{v | v \in V, \epsilon(v) = r(G)\}$$

When the *forward* or *reverse* eccentricity is used to define the radius of an RDF graph G, we refer to this radius as the forward or reverse direction radius respectively.

Now we use the query radius to formalize the gain of the semantic hash partitioning. Given a query Q issued over a set of k-hop semantic hash partitions, if the *radius* of Q's query graph is equal to or less than k, then Q can be executed on the partitions by using intra-partition processing.

THEOREM 1. Let $\{P_1^k, P_2^k, \ldots, P_m^k\}$ denote the semantic hash partitions of G, generated by k-hop expansion from the baseline partitions $\{P_1, P_2, \ldots, P_m\}$ on G, G_Q denote the query graph of a query Q and $r(G_Q)$ denote the radius of the query graph G_Q. Q can be evaluated using intra-partition processing over $\{P_1^k, P_2^k, \ldots, P_m^k\}$ if $r(G_Q) \leq k$.

We give a brief sketch of proof. By the k-hop forward (or reverse or bidirection) semantic hash partitioning, for any anchor vertex u in baseline partition P_i, all triples which are within k hops from u along the forward direction (or reverse or bidirection) are included in P_i^k. Therefore, it is guaranteed that all required triples to evaluate Q from u reside in the expanded partition if $r(G_Q) \leq k$.

4.3.3 Selective k-hop Expansion

Instead of replicating triples by expanding k hops in an exhaustive manner, we promote to further control the k-hop expansion by using some context-aware filters. For example, we can filter out some `rdf:type`-like triples that are rarely used in most of queries in the k-hop *reverse* expansion step to reduce the total number of triples to be replicated, based on the two observations. First, `rdf:type` predicate is widely used in most of RDF datasets to represent membership (or class) information of resources. Second, there are few object-object joins where more than one `rdf:type`-like triples are connected by an object variable, such as {Greg type ?x. Brian type ?x .}. By identifying such type of uncommon case, we can set a triple filter that will not replicate those `rdf:type`-like triples if their *object* vertices are the border vertices of the partition. However, we keep the `rdf:type`-like triples when performing forward direction expansion (i.e., from subject to object), because those triples are essential to provide fast pruning of irrelevant results due to the fact that the `rdf:type`-like triples in the forward direction typically are given as query conditions for most SPARQL queries. Our experimental results in Section 6 display significant reduction of replicated triples compared to the k-hop semantic hash partitioning without the object-based `rdf:type`-like triple filter.

4.3.4 URI Hierarchy-based Optimization

In an RDF graph, URI (Uniform Resource Identifier) references are used to identify vertices (except literals and blank nodes) and edges. URI references usually have a path hierarchy, and URI references having a common ancestor are often connected together, presenting high access locality. We conjecture that if such URI references (vertices) are placed in the same partition, we may reduce the number of replicated triples because a good portion of triples that need to be replicated by k-hop expansion from a vertex v are already located in the same partition of v. For example, the most common form of URI references in RDF datasets are URLs (Uniform Resource Locators) with http as their schema, such as "http://www.Department1.University2.edu/FullProfessor2/Publication14". The typical structure of URLs is "$http : //domainname/path1/path2/\dots/pathN\#fragmentID$". We first extract the hierarchy of the domain name based on its levels and then add the path components and the fragment ID by keeping their order in the full URL path. For instance, the hierarchy of the previous example URL, starting from the top level, will be "edu", "University2", "Department1", "FullProfessor2", "Publication14". Based on this hierarchy, we measure the percentage of RDF triples whose subject vertex and object vertex share the same ancestor for different levels of the hierarchy. If, at any level of the hierarchy, the percentage of such triples is larger than a system-supplied threshold (empirically defined) and the number of distinct URLs sharing this common hierarchical structure is greater than or equal to the number of partition servers, we can use the selected portion of the hierarchy from the top to the chosen level, instead of full URI references, to participate in the baseline hash partitioning process. This is because the URI hierarchy-based optimization can increase the access locality of baseline hash partitions by placing triples whose subjects are sharing the same prefix structure of URLs into the same partitions, while distributing the large collection of RDF triples across all partition servers in a balanced manner. We call such preprocessing *the URI hierarchy optimization*.

In summary, when using a hash function to build the baseline partitions, we calculate the hash value on the selected part of URI references and place those triples having the same hash value on the selected part of URI references in the same partition. Our experiments reported in Section 6 show that with the URI hierarchy optimization, we can obtain a significant reduction of replicated triples at the k-hop expansion phase.

4.3.5 Algorithm and Implementation

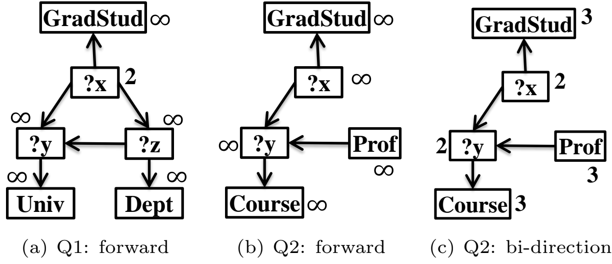
Algorithm 1 shows the pseudocode for our semantic hash partitioning scheme. It includes the configuration of parameters at the initialization step and the k-hop semantic hash partitioning, which carries out in multiple Hadoop jobs. The first Hadoop job will perform two tasks: generating triple groups and generating baseline partitions by hashing anchor vertices of triple groups. The subsequent Hadoop job will generate k-hop semantic hash partitions ($k \geq 2$).

We assume that the input RDF graph has loaded into HDFS. The map function of the first Hadoop job reads each triple and emits a key-value pair in which the key is subject (for s-TG) or object (for o-TG) of the triple and the value is the remaining part of the triple. If we use so-TG for generating baseline partitions, the map function emits two key-value pairs, one using its subject as the key and the other using its object as the key (line 3-7). Next we generate triple groups based on the subject (or object or both subject and object) during the shuffling phase such that triples with the same anchor vertex are grouped together and assigned to the partition indexed by the hash value of their anchor vertex. The reduce function records the assigned partition of the grouped triples using the hash value of their anchor vertex (line 8). If $k = 1$, we simply output the set of semantic hash partitions by merging all triples assigned to the same partition. Otherwise, the reduce function also records, for each anchor vertex, a set of vertices which should be expanded in the next hop expansion (line 9-15). We call such vertices *border vertices* of the anchor vertex. Concretely, for each triple in the triple group associated with the anchor vertex, the reduce function records the other vertex (e.g., the object vertex if the anchor vertex is the subject) as a border vertex of the anchor vertex because triples anchored at the border vertex may be selected for expansion in the next hop.

In the next Hadoop job, we implement k-hop semantic hash partitioning by controlled triple replication along the

Algorithm 1 Semantic Hash Partitioning

```
    Input: an RDF graph G, k, type (s-TG, o-TG or so-TG), direction
    (forward, reverse or bidirection)
    Output: a set of semantic hash partitions
 1: Initially, semantic partitions are empty
 2: Initially, there is no (anchor, border) pair
    Round=1 // generating baseline partitions
    Map
    Input: triple t(s,p,o)
 3: switch type do
 4:     case s−TG: emit(s,t)
 5:     case o−TG: emit(o,t)
 6:     case so−TG: emit(s,t), emit(o,t)
 7: end switch
    Reduce
    Input: key: anchor vertex anchor, value: triples
 8: add (hash(anchor), triples)
 9: if k = 1 then
10:     output baseline partitions P_1, ..., P_n
11: else
12:     read triples
13:     emit (anchor, borderSet)
14:     Round = Round + 1
15: end if
16: while Round ≤ k do //start k-hop triple replication
        Map
        Input: (anchor, border) pair or triple t(s,p,o)
17:     if (anchor, border) pair is read then
18:         emit(border, anchor)
19:     else
20:         switch direction do
21:             case forward: emit(s,t)
22:             case reverse: emit(o,t)
23:             case bidirection: emit(s,t), emit(o,t)
24:         end switch
25:     end if
        Reduce
        Input: key: border vertex border, value: anchors and triples
26:     for each anchor in anchors do
27:         add (hash(anchor), triples)
28:         if k < Round then
29:             read triples
30:             emit (anchor, borderSet)
31:         end if
32:     end for
33:     if k = Round then
34:         output semantic partitions P_1^k, ..., P_n^k
35:     end if
36:     Round = Round + 1
37: end while
```



(a) Q1: forward (b) Q2: forward (c) Q2: bi-direction

Figure 5: Calculating query radius

given expansion direction. The map function examines each baseline partition and reads a (anchor vertex, border vertex) pair, and emits a key-value pair in which the key is the border vertex and the value is the anchor vertex (line 17-25). During the shuffling phase, a set of anchor vertices which have the same border vertex are grouped together. The reduce function adds the triples connecting the border vertex to the partition if they are new to the partition and records the partition index of the triple using the hash value of the anchor vertex (line 27). If $k = 2$, we output the set of semantic partitions obtained so far. Otherwise, we record a set of new border vertices for each anchor vertex and repeat this job until k-hop semantic hash partitions are generated (line 28-31).

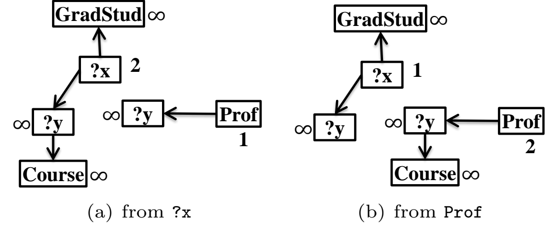
5. DISTRIBUTED QUERY PROCESSING

The distributed query processing component consists of three main tasks: query analysis, query decomposition and generating distributed query execution plans. The query analyzer determines whether or not a query Q can be executed using intra-partition processing. All queries that can be evaluated by intra-partition processing will be sent to the distributed query plan execution module. For those queries that require inter-partition processing, the query decomposer is invoked to split Q into a set of subqueries, each can be evaluated by intra-partition processing. The distributed query execution planner will coordinate the joining of intermediate results from executions of subqueries to produce the final result of the query.

5.1 Query Analysis

Given a query Q and its query graph, we first examine whether the query can be executed using intra-partition processing. According to Theorem 1, we calculate the *radius* and the center vertices of the query graph based on Definition 8, denoted by $r(Q)$ and $center(Q)$ respectively. If the dataset is partitioned using the k-hop expansion, then we evaluate whether $r(Q) \leq k$ holds. If yes, the query Q as a whole can be executed using the intra-partition processing. Otherwise, the query Q is passed to the query decomposer.

Fig. 5 presents three example queries with their query graphs respectively. We place the eccentricity value of each vertex next to the vertex. Since the *forward* radius of the query graph in Fig. 5(a) is 2, we can execute the query using intra-partition processing if the query is issued against the k-hop forward semantic hash partitions and k is equal to or larger than 2. In Fig. 5(b), the *forward* radius of the query graph is *infinity* because there is no vertex which has at least one forward direction path to all other vertices. Therefore, we cannot execute the query over the k-hop forward semantic hash partitions using intra-partition processing regardless of the hop count value of k. This query is passed to the query decomposer for further query analysis. Fig. 5(c) shows the eccentricity of vertices in the query graph under the *bidirection* semantic hash partitions. The *bidirection* radius is 2 and there are two center vertices: `?x` and `?y`. Therefore we can execute the query using intra-partition processing if k is equal to or larger than 2 under the bidirection semantic hash partitions.



(a) from `?x` (b) from `Prof`

Figure 6: Query decomposition

5.2 Query Decomposition

The first issue in evaluating a query Q using *inter-partition* processing is to determine the number of subqueries Q needs to be decomposed into. Given that there are more than one way to split Q into a set of subqueries, an intuitive approach is to first check whether Q can be decomposed into *two* subqueries such that each subquery can be evaluated using intra-partition processing. If there is no such decomposition, then we increase the number of subqueries by one and check again to see whether the decomposition enables each subquery to be evaluated by intra-partition processing. We repeat this process until a desirable decomposition is found.

Concretely, we start the query decomposition by putting all vertices in the query graph of Q into a set of candidate vertices to be examined in order to find such a decomposition having two subqueries. For each candidate vertex v, we find the largest subgraph from v, in the query graph of Q, which can be executed using intra-partition processing under the current k-hop semantic hash partitions. For the remaining part of the query graph, which is not covered by the subgraph, we check whether there is any vertex whose expanded subgraph under the current k-hop expansion can fully cover the remaining part. If there is such a decomposition, we treat each subgraph as a subquery of Q. Otherwise, we increase the number of subqueries by one and then repeat the above process until we find a possible decomposition. If we find several possible decompositions having the equal number of subqueries, then we choose the one in which the standard deviation of the size (i.e., the number of triple patterns) of subqueries is the smallest, under the assumption that a small subquery may generate large intermediate results. We leave as future work the query optimization problem where we can utilize additional metadata such as query selectivity information.

For example, in Fig. 5(b) where the query cannot be executed using intra-partition processing under the *forward* semantic hash partitions, assume that partitions are generated using the 2-hop forward direction expansion. To decompose the query, if we start with vertex `?x`, we will get a decomposition which consists of two subqueries as shown in Fig. 6(a). If we start with vertex `Prof`, we will also get two subqueries as shown in Fig. 6(b). Based on the smallest subquery standard deviation criterion outlined above, we choose the latter

Algorithm 2 Join Processing

Input: two intermediate results, join variable list, output variable list
Output: joined results
Map
Input: one tuple from one of the two intermediate results
1: Extracts a list of values, from the tuple, which are corresponding to the join variables
2: *emit*(join values, the remaining values of the tuple)
Reduce
Input: key: join values, value: two sets of tuples
3: Generates the Cartesian product of the two sets
4: Projects only columns that are included in the output variables
5: **return** joined (and projected) results

because two subqueries are of the same size.

5.3 Distributed Query Execution

Intra-partition processing steps: Let the number of partition servers be N. If the query Q can be executed using *intra-partition* processing, we send Q to each of the N partition servers in parallel. Upon the completion of local query execution, each partition server will send the partial results generated locally to the master server, which merges the results from all partition servers to generate the final results. The entire processing does not involve any coordination and communication among partition servers. The only communication happens between the master server and all its slave servers to ship the query to all slave servers and ship partial results from slaves to the master server.

Inter-partition processing steps: If the query Q cannot be executed using intra-partition processing, the query decomposer will be invoked to split Q into a set of subqueries. Each subquery is executed in all partitions using intra-partition processing and then the intermediate results of all sub-queries are loaded into HDFS and joined using Hadoop MapReduce. To join the two intermediate results, the map function of a Hadoop job reads each tuple from the two results and extracts a list of values, from the tuple, which are corresponding to the join variables. Then the map function emits a key-value pair in which the key is the list of extracted values (i.e., join key) and the value is the remaining part of the tuple. Through the shuffling phase of MapReduce, two sets of tuples sharing the same join values are grouped together: one is from the first intermediate results and the other is from the second intermediate results. The reduce function of the job generates the Cartesian product of the two sets and projects only columns that are included in the output variables or will be used in subsequent joins. Finally, the reduce function records the projected tuples. Algorithm 2 shows the pseudocode for our join processing during inter-partition processing. Since we use one Hadoop job to join the intermediate results of two subqueries, more subqueries usually imply more query processing and higher query latency due to the large overhead of Hadoop jobs.

6. EXPERIMENTAL EVALUATION

This section reports the experimental evaluation of our semantic hash partitioning scheme using our prototype system Shape. We divide the experimental results into four sets: (i) We present the experimental results on loading time, redundancy and triple distribution. (ii) We conduct the experiments on query processing latency, showing that by combining the semantic hash partitioning with the intra-partition processing-aware query partitioning, our approach reduces the query processing latency considerably compared to existing simple hash partitioning and graph partitioning schemes. (iii) We also evaluate the scalability of our approach with respect to varying dataset sizes and varying cluster sizes. (iv) We also evaluate the effectiveness of our optimization techniques used for reducing the partition size and the amount of triple replication.

6.1 Experimental Setup and Datasets

We use a cluster of 21 physical servers (one master server) on Emulab [22]: 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. We use RDF-3X version 0.3.5, installed on each slave server. We use Hadoop version 1.0.4 running on Java 1.6.0 to run various partitioning algorithms and join the intermediate results generated by subqueries.

We experiment with our 2-hop forward (*2f*), 3-hop forward (*3f*), 4-hop forward (*4f*), 2-hop bidirection (*2b*) and 3-hop bidirection (*3b*) semantic hash partitions, with the `rdf:type`-like triple optimization and the URI hierarchy optimization, expanded from the baseline partitions on subject-based triple groups. To compare our semantic hash partitions, we have implemented the random partitioning (*rand*), the simple hash partitioning on subjects (*hash-s*), the simple hash partitioning on both subjects and objects (*hash-so*), and the graph partitioning [11] with undirected 2-hop guarantee (*graph*). For fair comparison, we apply the `rdf:type`-like triple optimization to *graph*.

To run the vertex partitioning of *graph*, we also use the graph partitioner METIS [4] version 5.0.2 with its default configuration. We do not directly compare with other partitioning techniques which do not use the RDF-specific storage system to store RDF triples, such as SHARD [20], because it is reported in [11] that they are much slower than the graph partitioning for all benchmark queries. The random partitioning (*rand*) is similar to using HDFS for partitioning, but more optimized in the storage level by using the RDF-specific storage system.

For our evaluation, we use eight datasets of different sizes from four domains as shown in Table. 1. **LUBM** [8] and **SP^{2}Bench** [21] are benchmark generators and **DBLP** [1], containing bibliographic information in computer science, and **Freebase** [2], a large knowledge base, are the two real RDF datasets. As a data cleaning step, we remove any duplicate triples using one Hadoop job.

Table 1: Datasets

Dataset	#Triples	#subjects	#rdf:type triples
LUBM267M	267M	43M	46M
LUBM534M	534M	87M	92M
LUBM1068M	1068M	174M	184M
SP2B200M	200M	36M	36M
SP2B500M	500M	94M	94M
SP2B1000M	1000M	190M	190M
DBLP	57M	3M	6M
Freebase	101M	23M	8M

6.2 Data Loading Time

Table 2 shows the data loading time of the datasets for different partitioning algorithms. Due to the space limit, we

report the results of the largest dataset among three benchmark datasets. The data loading time basically consists of the data partitioning time and the partition loading time into RDF-3X. For *graph*, one additional step is required to run METIS for vertex partitioning. Note that the graph partitioning approach using METIS fail to work on larger datasets, such as LUBM534M, LUBM1068M, SP2B500M and SP2B1000M, due to the insufficient memory. The random partitioning (*rand*) and the simple hash partitioning on subjects (*hash-s*) have the fastest loading time because they just need to read each triple and assign the triple to a partition randomly (*rand*) or based on the hash value of the triple's subject (*hash-s*). Our forward direction-based approaches have fast loading time. The graph partitioning (*graph*) has the longest loading time if METIS can process the input dataset. For example, it takes about 25 hours to convert the Freebase dataset to a METIS input format and about 44 minutes to run METIS on the input. Note that our converter (from RDF to METIS input format), implemented using Hadoop MapReduce, is not the problem of this slow conversion time because, for LUBM267M, it takes 38 minutes (33 minutes for conversion and 5 minutes for running METIS), much faster than the reported time (1 hour) in [11].

Table 2: Partitioning and Loading Time (in min)

Algorithm	METIS	Partitioning	Loading	Total
LUBM1068M				
single server	-	-	779	779
rand	-	17	47	64
hash-s	-	19	34	53
hash-so	-	84	131	215
graph	fail	N/A	N/A	N/A
2-forward	-	94	32	126
3-forward	-	117	32	149
4-forward	-	133	32	165
2-bidirection	-	121	61	182
3-bidirection	-	396	554	950
SP2B1000M				
single server	-	-	665	665
rand	-	16	39	55
hash-s	-	16	28	44
hash-so	-	74	81	155
graph	fail	N/A	N/A	N/A
2-forward	-	89	34	123
3-forward	-	111	34	145
4-forward	-	127	34	161
2-bidirection	-	109	53	162
3-bidirection	-	195	135	330
Freebase				
single server	-	-	73	73
rand	-	2	4	6
hash-s	-	2	3	5
hash-so	-	5	9	14
graph	1573	38	52	1663
2-forward	-	9	4	13
3-forward	-	11	4	15
4-forward	-	14	4	18
2-bidirection	-	22	17	39
3-bidirection	-	59	75	134
DBLP				
single server	-	-	34	34
rand	-	2	2	4
hash-s	-	2	1	3
hash-so	-	4	3	7
graph	452	22	35	509
2-forward	-	7	2	9
3-forward	-	8	2	10
4-forward	-	10	2	12
2-bidirection	-	13	8	21
3-bidirection	-	36	35	71

6.3 Redundancy and Triple Distribution

Table 3 shows, for each partitioning algorithm, the ratio of the number of triples in all generated partitions to the total number of triples in the original datasets. The random partitioning (*rand*) and the simple hash partitioning on subjects (*hash-s*) have the ratio of 1 because there is no replicated triple. This result shows that our forward direction-based approaches can reduce the number of replicated triples considerably while maintaining the hop guarantee. For example, even though we expand the baseline partitions to satisfy 4-hop guarantee (forward direction), the replication ratio is less than 1.6 for all the datasets. On the other hand, this result also shows that we should be careful when we expand the baseline partitions using both directions. Since the original data can be almost fully replicated on all the partitions when we use 3-hop bidirection expansion, the number of hops should be decided carefully by considering the tradeoff between the overhead of local processing and inter-partition communication. We leave how to find an optimal k value, given a dataset and a set of queries, as future work.

Table 3: Redundancy (Ratio to original dataset)

Dataset	2f	3f	4f	2b	3b	hash-so	graph
LUBM267M	1.00	1.00	1.00	1.67	8.87	1.78	3.39
LUBM534M	1.00	1.00	1.00	1.67	8.73	1.78	N/A
LUBM1068M	1.00	1.00	1.00	1.67	8.66	1.78	N/A
SP2B200M	1.18	1.19	1.19	1.76	3.81	1.78	1.32
SP2B500M	1.16	1.17	1.17	1.70	3.58	1.77	N/A
SP2B1000M	1.15	1.15	1.16	1.69	3.50	1.77	N/A
DBLP	1.48	1.53	1.55	5.35	18.28	1.86	5.96
Freebase	1.18	1.26	1.28	5.33	17.18	1.87	7.75

Table. 4 shows the coefficient of variation (the ratio of the standard deviation to the mean) of generated partitions in terms of the number of triples to measure the dispersion of the partitions. Having uniformly distributed triples across all partitions is one of the key performance factors because the large partitions in the skewed distribution can be performance bottlenecks during query processing. Our semantic hash partitioning approaches have almost perfect uniform distributions. On the other hands, the results indicate that partitions generated using *graph* are very different in size. For example, among the partitions generated using *graph* for DBLP, the largest partition is 3.8 times bigger than the smallest partition.

Table 4: Distribution (Coefficient of Variation)

Dataset	2f	3f	4f	2b	3b	hash-so	graph
LUBM267M	0.01	0.01	0.01	0.00	0.01	0.20	0.26
LUBM534M	0.01	0.01	0.01	0.00	0.01	0.20	N/A
LUBM1068M	0.01	0.01	0.01	0.00	0.01	0.20	N/A
SP2B200M	0.00	0.00	0.00	0.00	0.00	0.01	0.05
SP2B500M	0.00	0.00	0.00	0.00	0.00	0.01	N/A
SP2B1000M	0.00	0.00	0.00	0.00	0.00	0.01	N/A
DBLP	0.00	0.00	0.00	0.00	0.00	0.09	0.50
Freebase	0.00	0.00	0.00	0.00	0.00	0.16	0.24

6.4 Query Processing

For our query evaluation of the three LUBM datasets, we report the results of all 14 benchmark queries provided by **LUBM**. Among the 14 queries, 8 queries (Q1, Q3, Q4, Q5, Q6, Q10, Q13 and Q14) are star queries. The *forward* radii of Q2, Q7, Q8, Q9, Q11 and Q12 are 2, ∞, 2, 2, 2 and 2 respectively. Their *bidirection* radii are all 2. Due to the space limit, for the other datasets, we report the results of one star query and one or two complex queries including chain-like patterns. We pick three queries among a set of benchmark queries provided by **SP^{2}Bench** and create star and complex queries for the real datasets. Table 5 shows the queries used for our query evaluation. The *forward* radii of SP2B Complex1 and Complex2 are ∞ and 2 respectively. The *bidirection* radii of SP2B Complex1 and Complex2 are 3 and 2 respectively.

Fig. 7 shows the query processing time of all 14 benchmark queries for different partitioning approaches on LUBM534M dataset. Since the results of our 2-hop forward (*2f*), 3-hop forward (*3f*) and 4-hop forward (*4f*) partitions are almost the same, we merge them into one. Our forward direction-

Table 6: Query Processing Time (sec)

Dataset	2f	3f	4f	2b	3b	hash-s	hash-so	random	graph	single server
SP2B200M Star	64.367	67.995	71.013	69.076	344.701	**58.556**	129.423	834.99	73.928	500.825
SP2B200M Complex1	**222.962**	224.025	225.316	226.977	659.767	1257.993	2763.815	2323.422	223.598	fail
SP2B200M Complex2	**42.697**	53.125	56.387	52.393	136.136	208.383	357.72	341.831	49.785	431.441
SP2B500M Star	183.413	187.017	197.716	197.605	967.291	**166.922**	378.614	1625.757	N/A	fail
SP2B500M Complex1	**487.078**	493.477	522.398	529.83	1272.774	3365.152	7215.617	5613.11	N/A	fail
SP2B500M Complex2	**112.99**	115.269	116.955	125.736	410.66	449.633	921.241	703.548	N/A	1690.236
SP2B1000M Star	456.121	479.469	482.588	459.583	2142.445	**413.237**	685.492	2925.019	N/A	fail
SP2B1000M Complex1	**897.884**	911.012	917.834	1006.927	2391.878	6418.564	14682.103	11739.827	N/A	fail
SP2B1000M Complex2	**258.611**	265.821	270.21	282.414	905.218	808.229	1986.949	1353.028	N/A	fail
DBLP Star	12.875	12.91	13.589	17.938	41.184	**3.617**	5.328	56.006	30.509	22.711
DBLP Complex	**3.48**	3.571	3.726	9.9	31.659	61.281	74.186	117.481	20.866	21.384
Freebase Star	10.666	11.151	11.954	22.06	129.111	**8.234**	9.024	143.602	105.413	42.608
Freebase Complex1	**6.989**	7.78	8.069	13.681	71.443	54.361	61.783	57.537	25.408	43.592
Freebase Complex2	**63.804**	66.87	67.281	80.484	501.563	216.555	238.568	568.919	195.98	23212.521

Table 5: Queries

LUBM	**All 14** benchmark queries
SP2B Star	Benchmark Query2 (without Order by and Optional) Select ?inproc ?author ?booktitle ?title ?proc ?ee ?page ?url ?yr Where { ?inproc rdf:type Inproceedings . ?inproc creator ?author . ?inproc booktitle ?booktitle . ?inproc title ?title . ?inproc partOf ?proc . ?inproc seeAlso ?ee . ?inproc pages ?page . ?inproc homepage ?url . ?inproc issued ?yr }
SP2B Complex1	Benchmark Query4 (without Filter) Select DISTINCT ?name1 ?name2 Where { ?article1 rdf:type Article . ?article2 rdf:type Article . ?article1 creator ?author1 . ?author1 name ?name1 . ?article2 creator ?author2 . ?author2 name ?name2 . ?article1 journal ?journal . ?article2 journal ?journal }
SP2B Complex2	Benchmark Query6 (without Optional) Select ?yr ?name ?document Where { ?class subClassOf Document . ?document rdf:type ?class . ?document issued ?yr . ?document creator ?author . ?author name ?name }
DBLP Star	Select ?author ?name Where { ?author rdf:type Agent . ?author name ?name }
DBLP Complex	Select ?paper ?conf ?editor Where { ?paper partOf ?conf . ?conf editor ?editor . ?paper creator ?editor }
Freebase Star	Select ?person ?name Where { ?person gender male . ?person rdf:type book.author . ?person rdf:type people.person . ?person name ?name }
Freebase Complex1	Select ?loc1 ?loc2 ?postal Where { ?loc1 headquarters ?loc2 . ?loc2 postalcode ?postal . }
Freebase Complex2	Select ?name1 ?name2 ?birthplace ?inst Where { ?person1 birth ?birthplace . ?person2 birth ?birthplace . ?person1 education ?edu1 . ?edu1 institution ?inst . ?person2 education ?edu2 . ?edu2 institution ?inst . ?person1 name ?name1 . ?person2 name ?name2 . ?edu1 rdf:type education . ?edu2 rdf:type education }

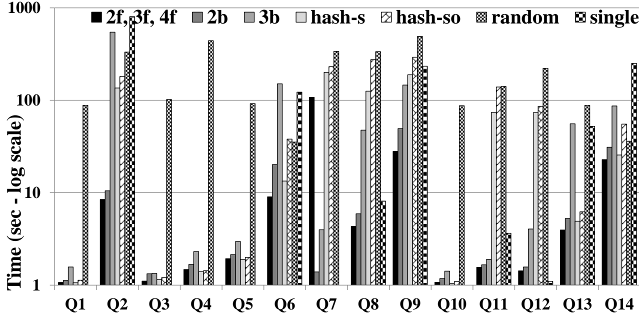
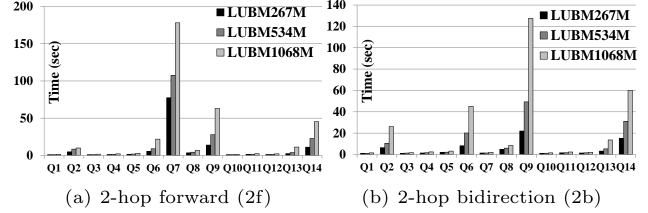


Figure 7: Query Processing Time (LUBM534M)

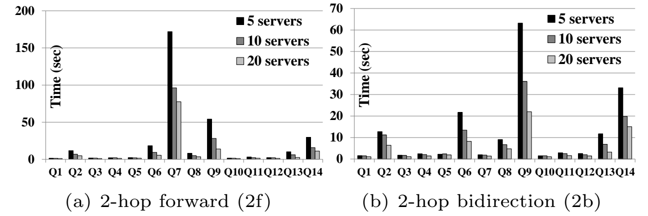
based partitioning approaches (*2f*, *3f* and *4f*) have faster query processing time than the other partitioning techniques for all the benchmark queries except Q7 in which inter-partition processing is required for *2f*, *3f* and *4f*. Our 2-hop bidirection (*2b*) approach also has good performance because it ensures intra-partition processing for all benchmark queries.

For Q7, since our forward direction-based partitioning approaches need to run one Hadoop job to join the intermediate results of two subqueries and the size of the intermediate results is about 2.4 GB (much larger compared to the final result size of 907 bytes), its query processing time for Q7 is very slow compared to other approaches (*2b* and *3b*) using intra-partition processing. However, our approaches (*2f*, *3f* and *4f*) are faster than the simple hash partitioning (*hash-s* and *hash-so*) which requires two Hadoop jobs to process Q7. Recall that the graph partitioning does not work for LUBM534M because METIS failed due to the insufficient memory.



(a) 2-hop forward (2f) (b) 2-hop bidirection (2b)

Figure 8: Scalability with varying dataset sizes



(a) 2-hop forward (2f) (b) 2-hop bidirection (2b)

Figure 9: Scalability with varying cluster sizes

Table 6 shows the query processing times of the other datasets. The fastest query processing time for each query is marked in bold. Our forward direction-based partitioning approaches (*2f*, *3f* and *4f*) are faster than the other partitioning techniques for all complex queries. For example, for SP2B1000M Complex1, our approach *2f* is about 7, 16 and 13 times faster than *hash-s*, *hash-so* and *random* respectively. Note that executing SP2B1000M Complex1 fails on a single server due to the insufficient memory and *graph* does not work for SP2B1000M. Our 2-hop bidirection (*2b*) approach also has comparable query processing performance with *2f*, *3f* and *4f*. Even though our 3-hop bidirection (*3b*) approach is much slower than *2f*, *3f*, *4f* and *2b* due to its large partition size, it is faster than *random* for most queries. For star queries, *hash-s* is slightly faster than our approaches because it is optimized only for star queries and there is no replicated triple.

6.5 Scalability

We evaluate the scalability of our partitioning approach by varying dataset sizes and cluster sizes. Fig. 8 shows that the increase of the query processing time of star queries Q6, Q13 and Q14 is almost proportional to the dataset size. For Q7, under the 2-hop bidirection (*2b*) expansion, the query processing time increases only slightly because its results do not change by the dataset size. On the other hand, under the 2-hop forward (*2f*) expansion, there is a considerable increase in the query processing time because the intermediate

results increase according to the dataset size even though the final results are the same regardless of the dataset size.

Fig. 9 shows the results of scalability experiment with varying numbers of slave servers from 5 to 20 on LUBM267M dataset. For star queries whose selectivity is high (Q1, Q3, Q4, Q5 and Q10), the processing time slightly decreases with an increasing number of servers due to the reduced partition size. For star queries with low selectivity (Q6, Q13 and Q14), the decrease of the query processing time is almost proportional to the number of slave servers.

6.6 Effects of optimizations

Table 7 shows the effects of different optimization techniques under the 2-hop bidirection (*2b*) expansion in terms of the replication ratio. Without any optimization, large partitions are generated because lots of triples are replicated and so it will considerably increase the query processing time. Using the `rdf:type`-like triple optimization, we can reduce the partition size by excluding `rdf:type`-like triples during the expansion. The result of applying the URI hierarchy optimization shows that we place close vertices in the same partition and so prevent the replication of many triples. By combining both optimization techniques, we substantially reduce the partition size and so increase the performance of query processing.

Table 7: Effects of optimizations (Replication Ratio)

Dataset	No Opt.	rdf:type	URI hierarchy	Both
LUBM1068M	11.46	8.46	4.94	1.67
SP2B1000M	6.95	3.70	5.12	1.69
DBLP	7.24	5.35	N/A	5.35
Freebase	6.88	5.33	N/A	5.33

7. CONCLUSION

In this paper we have shown that when data needs to be partitioned across multiple server nodes, the choice of data partitioning algorithms can make a big difference in terms of the cost of data shipping across a network of servers. We have presented a novel semantic hash partitioning approach, which starts with the simple hash partitioning and expands each partition by replicating only necessary triples to increase access locality and promote intra-partition processing of SPARQL queries. We also developed a partition-aware distributed query processing facility to generate locality-optimized query execution plans. In addition, we provide a suite of locality-aware optimization techniques to further reduce the partition size and cut down on the inter-partition communication cost during distributed query processing. Our experimental results show that the semantic hash partitioning approach improves the query latency and is more efficient than existing popular simple hash partitioning and graph partitioning schemes.

The first prototype system for our semantic hash partitioning does not support aggregate queries and update operations. We plan to implement new features introduced in SPARQL 1.1. Both rdf:type filter and URI hierarchy-based merging of triple groups are provided as a configuration parameter. One of our future work is to utilize statistics collected over representative set of queries to derive a near-optimal setting of k for k-hop semantic hash partitioning. Finally, we conjecture that the effectiveness of RDF data partitioning can be further enhanced by exploring different strategies for access locality-guided triple grouping and triple replication.

Acknowledgments

This work is partially supported by grants from NSF Network Science and Engineering (NetSE) and NSF Secure and Trustworthy Computing (SaTC), an IBM faculty award and a grant from Intel ISTC on Cloud Computing. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation and our industry sponsors.

8. REFERENCES

[1] About FacetedDBLP. *http://dblp.l3s.de/dblp++.php*.
[2] BTC 2012. *http://km.aifb.kit.edu/projects/btc-2012/*.
[3] Linking Open Data. *http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData*.
[4] METIS. *http://www.cs.umn.edu/˜metis*.
[5] RDF. *http://www.w3.org/RDF/*.
[6] SPARQL. *http://www.w3.org/TR/rdf-sparql-query/*.
[7] O. Erling and I. Mikhailov. Towards web scale RDF. *Proc. SSWS*, 2008.
[8] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semant.*, 2005.
[9] A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: a federated repository for querying graph structured data from the web. In *ISWC*, 2007.
[10] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Supercomputing*, 1995.
[11] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. PVLDB, 2011.
[12] M. Husain, J. McGlothlin, M. M. Masud, et al. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *IEEE TKDE*, 2011.
[13] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 1998.
[14] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: large-scale graph computation on just a PC. In *OSDI*, 2012.
[15] G. Ladwig and A. Harth. CumulusRDF: Linked data management on nested key-value stores. In *SSWS*, 2011.
[16] K. Lee and L. Liu. Efficient Data Partitioning Model for Heterogeneous Graphs in the Cloud. In *ACM/IEEE SC*, 2013.
[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*, 2010.
[18] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDBJ*, 2010.
[19] A. Owens, A. Seaborne, N. Gibbins, et al. Clustered TDB: A Clustered Triple Store for Jena. 2008.
[20] K. Rohloff and R. E. Schantz. Clause-iteration with MapReduce to scalably query datagraphs in the SHARD graph-store. In *DIDC*, 2011.
[21] M. Schmidt, T. Hornung, G. Lausen, et al. SP^2Bench: A SPARQL Performance Benchmark. In *ICDE*, 2009.
[22] B. White, J. Lepreau, L. Stoller, R. Ricci, et al. An integrated experimental environment for distributed systems and networks. In *OSDI*, 2002.