SemStore: A Semantic-Preserving Distributed RDF Triple Store

Buwen Wu [†#], Yongluan Zhou [#], Pingpeng Yuan [†], Hai Jin [†], Ling Liu [§]

[†]SCTS/CGCL, Huazhong University of Science and Technology, Wuhan, China
hjin@hust.edu.cn

[#]University of Southern Denmark, Denmark
zhou@imada.sdu.dk

[§]Georgia Institute of Technology, Atlanta, USA
lingliu@cc.gatech.edu

ABSTRACT

The flexibility of the RDF data model has attracted an increasing number of organizations to store their data in an RDF format. With the rapid growth of RDF datasets, we envision that it is inevitable to deploy a cluster of computing nodes to process large-scale RDF data in order to deliver desirable query performance. In this paper, we address the challenging problems of data partitioning and query optimization in a scale-out RDF engine. We identify that existing approaches only focus on using fine-grained structural information for data partitioning, and hence fail to localize many types of complex queries. We then propose a radically different approach, where a coarse-grained structure, namely Rooted Sub-Graph (RSG), is used as the partition unit. By doing so, we can capture structural information at a much greater scale and hence are able to localize many complex queries. We also propose a k-means partitioning algorithm for allocating the RSGs onto the computing nodes as well as a query optimization strategy to minimize the inter-node communication during query processing. An extensive experimental study using benchmark datasets and real dataset shows that our engine, SemStore, outperforms existing systems by orders of magnitudes in terms of query response time.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*distributed databases, query processing*

Keywords

RDF, SPARQL, Partitioning, Query Processing

1. INTRODUCTION

RDF (Resource Description Framework) is a simple yet powerful data model for representing information in the form of triple statements: (subject, predicate, object). Such simple statements can be used to represent complicated relationship among different resources. An RDF dataset can be modeled as a labeled and directed graph with each triple modeled as two vertices and a directed edge, labeled by the predicate, from its subject vertex to its object vertex. SPARQL is the standard query language of RDF data. Due to the flexibility and dynamicity of the RDF data model, an increasing number of communities making their data available in the RDF format. The statistics from Linked Open Data Project[1] show that more than 31 billion triples had been published in RDF till Sep. 2011.

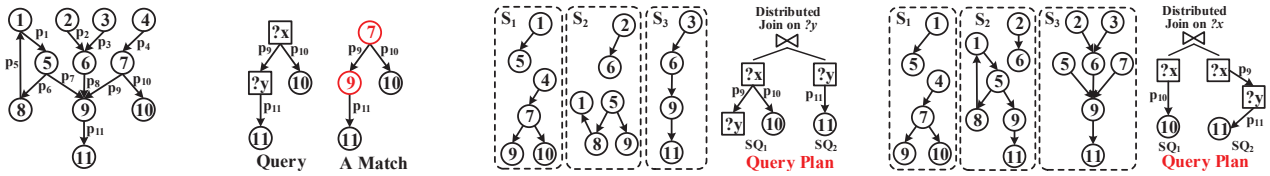
In the past decade, many researches focused on techniques for RDF data management and SPARQL query optimization in a centralized environment, such as [4, 5, 16, 22, 25]. With the rapid growth of the data sizes, a single machine would not be capable to deliver satisfactory query performance [5]. Consequently, it is highly desirable to develop techniques for distributed RDF data processing that are scalable to large RDF datasets. Recently, there exist many efforts devoted to building scale-out RDF engines [9,10,12–15,18,24]. In these engines, data are partitioned and allocated to multiple computing nodes and the processing of SPARQL queries often involves distributed join processing. As indicated in previous results [12,15,17], distributed joins are much more expensive than local joins due to their excessive communication cost. Therefore, the fundamental challenge in building a scale-out RDF engine is how to partition the RDF data across a computing cluster such that queries can be evaluated with minimum communication cost incurred by distributed joins.

The most common data partitioning algorithm is edge-based hash partitioning algorithm [9, 10, 14], which uses the edges in RDF graph as partition unit, and assigns the edges to each node by computing a hash key over either subject or object of the edges. More recent approaches [12, 15] use n-hop blocks as the partition unit. For each vertex v in RDF graph, an n-hop block anchored at v is formed by including all the vertices whose distance from v is less than or equal to n. These approaches then allocate the n-hop blocks to different partitions by using a hash function or a graph partitioning algorithm. Using edges and n-hop blocks as the partition unit works well for queries with small scopes, which involve joins on a single vertex, such as star queries, or joins on vertices with limited distances. For queries that involve joins on vertices that are of greater distances, especially those with a long sequence of subject-object joins, they often have to resort to expensive distributed joins across multiple computing nodes. This can be attributed to the fact that they

CIKM'14, November 3–7, 2014, Shanghai, China.

http://dx.doi.org/10.1145/2661829.2661876.

[1] http://lod-cloud.net/state/



(a) RDF Graph (b) Query and Match (c) Edge-based Hash Partitioning (d) 1-hop based Hash Partitioning

Figure 1: RDF Graph, SPARQL Query, Query Matches and Query Decomposition Plans with regard to Two Data Partitioning Algorithms: Edge-Based Hash Partitioning and 1-hop Based Hash Partitioning.

fail to make use of larger-scale structural information in the RDF graph but rather focusing on fine-grained information.

To address this problem, we propose a drastically different approach. Rather than using a fine-grained partition unit, such as edges or n-hop blocks, we adopt a coarse-grained unit, namely *Rooted Sub-Graph* (*RSG*), which is capable to capture structural information with a greater scope. We prove that any star, chain, tree and cycle queries would only involve joins on vertices within RSGs. Therefore, placing a complete RSG to only a single node would localize all these four common types of queries. In this paper, we particularly address the two major challenges in realizing this approach. First of all, as an RDF triple could appear in multiple different RSGs, there would be overlaps between RSGs and hence may potentially involve duplicate triples. To reduce the data redundancy and further localize more complex queries, we propose a k-means clustering algorithm to place the RSGs into partitions and design a suitable distance measure and centroid to capture the correlations among different RSGs. Second, for the queries where distributed joins are unavoidable, we design a partition-aware query optimizer that can capitalize on the data partitioning strategy to generate optimized plans at both compile time and run time.

In summary, our major contributions in this paper include the following:

- We present a radically new RDF data partition method, which takes Rooted Sub-Graph as the partition unit, and prove that by doing so can efficiently localize the four aforementioned common types of SPARQL queries.
- We present a k-means partitioning algorithm with the tailored distance measure and centroid to place the RSGs across a computing cluster. The algorithm can significantly reduce the data redundancy among the computing nodes and maximize the possibility of queries other than the four common ones to be processed locally. To deal with large datasets, we also present how to implement this algorithm using the popular MapReduce framework.
- We provide a partition-aware query decomposition algorithm and a dynamic two-stage (compile-time and runtime) query optimization technique to reduce the cost of distributed join processing.
- We introduce the architecture of SemStore, a semantic-preserving distributed triple store that implements all these techniques as integral parts, where each computing node in the cluster running an instance of a centralized RDF engine.
- We conduct a comprehensive experimental study by comparing the performance of SemStore over two datasets, namely LUBM [8] and UniProt [2], with a number of existing systems or approaches, including TripleBit [22], RDF-3X [16], Trinity.RDF [24], SHARD [18], Hybrid system [12, 15] and Hive.

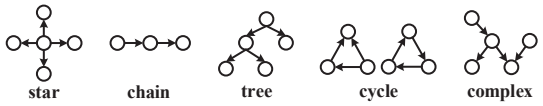


Figure 2: Query Types

2. BACKGROUND

2.1 RDF and SPARQL

RDF dataset is a set of triples in the form of (subject, predicate, object). In fact, an RDF dataset can be represented as a directed labeled graph with *subject* and *object* as the vertices connected by an edge labeled by *predicate*. An RDF graph can be defined as follows:

DEFINITION 2.1 (RDF GRAPH). *An RDF graph is a directed labeled graph, denoted as $G = (V, E, L_E)$, where V is a set of vertices, corresponding to subjects and objects, and E is a set of directed edges from the subjects to the objects. L_E is a set of edge labels, referring to the predicates associated with the edges. A vertex with indegree as zero is called a* **source vertex**.

Figure 1(a) shows an example of the RDF graph. Here we use IDs instead of URIs and Literals for simplicity.

SPARQL is a standard query language for RDF data. A SPARQL query consists of triple patterns, each of which is a triple that contains variables in the subject, predicate or object. A SPARQL query Q can be represented as $Q = \{tp_1, ..., tp_n\}$, where tp_i $(1 \leqslant i \leqslant n)$ is a triple pattern and triple patterns are connected by common vertex (a constant or a variable). Similarly, a SPARQL query can be represented as a directed graph. Thus, to evaluate a SPARQL query is to find all matches of a query graph pattern and each match is a subgraph of the RDF graph. Figure 1(b) shows a query graph and a match of this query graph.

We note that triple patterns are connected by shared subjects or objects, which means joins between triple patterns are on their shared subjects or objects. Thus, the typical types of join in SPARQL are subject-subject join (S-S join), subject-object join (S-O join) and object-object join (O-O join). In this paper, we do not consider the predicate join, since predicate join is not common [7]. Based on the join types, we can classify the SPARQL queries into five categories, star query (only contains S-S join), chain query (only contains S-O join), tree query (contains S-S join and S-O join), cycle query (contains S-S join, S-O join and O-O join) and complex query denote the rest of more complex queries. Figure 2 shows an example.

2.2 Related Work

Distributed RDF Engines. With the rapid increase of the volume of RDF data, there are many recent efforts in designing scalable distributed RDF engines for processing big

RDF datasets. To benefit from the scalability and fault tolerance of the MapReduce framework [6], several systems [13, 18] directly store RDF data as files in HDFS (Hadoop Distributed File System) and process SPARQL queries using Hadoop's MapReduce programming model. The authors in [12,15] propose systems integrating Hadoop and RDF-3X, an efficient local RDF engine to achieve significant performance improvement in comparing to the pure Hadoop-based systems [13,18]. The main idea is to push as much joins as possible to the individual local RDF-3X engines by a well designed RDF data partitioning algorithm. The remaining distributed joins would be handled using MapReduce jobs. Even though such distributed joins would be much more expensive than the local ones, the authors did not provide any query optimization algorithm to improve their performance. Trinity.RDF [24] is one of the most efficient distributed graph engine for web-scale RDF data, which uses main memory to store the RDF data and hence can achieve very low data access latency. Instead of using joins, the authors proposed an efficient operator, namely graph exploration, to perform SPARQL queries based on the MPI protocol. While [24] is mainly focused on designing a scale-out system, our data partitioning algorithm can be employed in this system to further improve its performance by reducing the communication cost.

RDF Data Partitioning Approaches. The most common data partitioning approach is the edge-based hash partitioning [9,10,14]. It distributes edges across different partitions by computing a hash key of the subject (or object) vertex of each edge. The principle of edge-based hash partitioning is to group the edges that contain same subject as a "star" and each of such star is placed in a single computing node. In this way, the edge-based hash scheme can localize subject-subject joins [12]. Figure 1(c) shows an example of the edge-based partitioning. The query in Figure 1(b) has to be decomposed into two subqueries, which only have subject-subject joins, as shown in Figure 1(c). Each of these subqueries can be executed in parallel without need for distributed join. However, to obtain the final results, we should execute distributed join between the intermediate outputs produced by SQ_1 and SQ_2 on join key $?y$, which leads to network overhead.

In [12, 15], the authors proposed RDF data partitioning approaches using a different partition unit: n-hop blocks. For each vertex v in the RDF graph, an n-hop block anchored at v is formed by including all the vertices whose distances from v is less than or equal to n. These approaches essentially allocate the n-hop blocks to different partitions by using a graph partitioner [12] (e.g. METIS) or a hash function [15]. These approaches can localize queries where there is a vertex whose distances to all the other vertices are at most n. Wang et al. [19] propose a more efficient graph partitioner that is able to partition billion-node graphs. Figure 1(d) shows an example using the 1-hop hash partitioning method. The query in Figure 1(b) does not satisfy the condition that there exists a vertex whose distances to the other vertices are at most 1. Actually, this query should be decomposed into two subqueries illustrated in Figure 1(d), both of which satisfy the condition. As proposed in [12, 15], they will be executed in parallel on the computing nodes using the local RDF database engine and a distributed join will be performed on the intermediate results to generate the final ones.

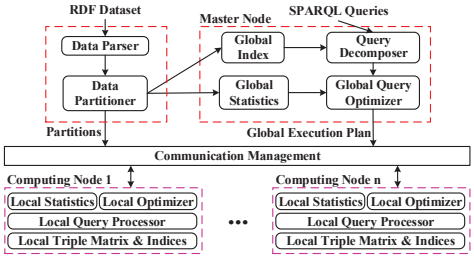


Figure 3: The Architecture of SemStore

Using the methods based on n-hop blocks, in order to localize more complex queries that has a greater diameter, one has to increase the value of n. However, even a moderate n value may incur a very large block size, especially for those blocks anchored at vertices with high degrees. This would lead to a large number of duplicate triples and a skewed data distribution among the partitions. The duplicate triples would incur higher query processing cost as each local engine has to process more data and the distributed joins may be executed over more intermediate results. Furthermore, the skewed data distribution may render some computing nodes become the system's bottleneck. Consequently, the best n value reported in [12, 15] is around 2. More experimental results in Section 6.1 further verify these problems.

Another research direction is dynamic run-time data partitioning, which adapts the data partitioning scheme according to the run-time changes of system workload [20,21]. Our data partitioning algorithm can also be used as the initial partitioning method in [20, 21]. Moreover, the idea of using a coarse-grained partition unit, *RSG*, can be applied to the dynamic partitioning algorithms to further improve their performance.

3. SYSTEM OVERVIEW

SemStore is designed to provide a massively scale-out system that can run on a cluster of servers for managing big RDF datasets. Figure 3 illustrates the architectural design of the prototype of SemStore. SemStore has three main components: data partitioner, a master node and a number of computing nodes. The data partitioner partitions the entire RDF dataset into multiple subsets. Each computing node receives one subset of triples and builds the local data indices and statistics for local join processing and optimization. The master node is responsible for constructing a distributed query plan and coordinating distributed data transmission and processing among the computing nodes.

In our prototype system, each computing node is running a single-node RDF engine. While our system architecture allows the embedding of different single-node RDF engines, in our experiments, we use TripleBit [22], an open-source centralized RDF engine.

Data Partitioner. When RDF datasets are loaded into the system, we first build some mapping dictionaries to replace all strings (URIs, literals and blank nodes) by IDs. The data partitioner will partition and allocate the triples to the computing nodes. The partitioning method in SemStore adopts a new partition unit, called Rooted Sub-Graph (RSG). By using RSGs as the partition unit, we can effectively localize all the queries in the form of a star, a chain, a tree and a cycle, which are very common in SPARQL queries. To localize more queries of other types and reduce data redundancy, we leverage a k-means partitioning algorithm for allocating

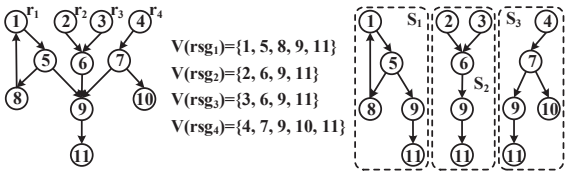


Figure 4: RSG-based K-Means Partitioning

the highly correlated RSGs into the same computing node. After the partitioning, the data partitioner will also build a global bitmap index over the vertices of the RDF graph and collect the global statistics.

Query Processing and Optimization. In SemStore, all SPARQL queries will be first submitted to the master node. Upon receiving a query, the master node parses the query and determines which category this query belongs to, **local-query** or **distributed-query**. If a SPARQL query can be executed in parallel and locally without collaboration between computing nodes, this query is a local-query. Otherwise, it is a distributed-query, which needs to join data from multiple partitions and hence has to be evaluated collaboratively by more than one computing nodes. For a distributed-query, The query decomposer will generate a query decomposition plan consists of several subqueries, each being a local-query. Then distributed query optimizer will generate a global query plan at compile-time which determines the joining order of results returned by the subqueries. Furthermore, each local-query will be sent to the computing nodes holding its results and the local optimizer on each computing node will optimize each local query at run-time based on the local statistics.

4. DATA PARTITIONING

In this section, we present our data partitioning method. We start by presenting the rationale behind the choice of RSG as the partition unit. Then we present a scalable k-means partitioning algorithm to assign partition units, that can be implemented on the MapReduce framework.

4.1 Rooted Sub-Graph

As discussed earlier, the reason that the aforementioned approaches do not work well for queries involving vertices of greater distances, such as chain, tree, cycle and some complex queries, is because they only take use of fine-grained structural information and fail to capture the RDF graph structure with a greater scope. Furthermore, a previous empirical study [7] analyzed queries generated by both humans and machine agents over two datasets, and concluded that the most common types of joins are S-S joins (60%) and S-O joins (35%). S-S and S-O joins would mainly form queries as star queries, chain queries and tree queries. Hence a good partition unit should be optimized for these types of queries.

To address this problem, we propose a new partition unit, namely Rooted Sub-Graph (RSG), defined as follows:

Definition 4.1 (Rooted Sub-Graph). *Given an RDF graph $G = (V, E)$ and a vertex $r \in V$ as the root, a sub-graph $rsg(r) = (V_{rsg(r)}, E_{rsg(r)})$ is called a rooted sub-graph of G rooted at r if and only if the following conditions hold: (1) for each $v \in V$, if there is a directed path from r to v then $v \in V_{rsg(r)}$, (2) for each pair of vertices (u, v), if $u, v \in V_{rsg(r)}$ and $(u, v) \in E$, then $(u, v) \in E_{rsg(r)}$.*

Algorithm 1: RSGs Generation

```
   Input: G = (V, E), Source vertex set Source
1  foreach v ∈ V do // init all vertices as input
2  |  Activate(v) ;
3  |  if v ∈ Source then Out_0.Add((v, (v, v))) ;
4  |  else Out_0.Add((v, (v, null))) ;
5  i ← 0 ;
6  repeat
7  |  Propagation(E, Out_i) ;
8  |  i ← i + 1 ;
9  until no update signal;
Function: Propagation(E, Out_i)
   Map1: Input: edges E = {(u, w)}, Out_i = {(v, (v_m, r))}
1  if input is an edge (u, w) then Emit(u,w) ;
2  else Emit(v,(v_m, r))
   Reduce1: Input: key, values
1  foreach val ∈ values do // propa.  v_m and r to neigh.
2  |  m ← ∞ ; root ← ∅ ;
3  |  if val is a vertex and val is active then
4  |  |  m ← v_m ; root ← r ;
5  |  |  Emit(key,val) ;
6  |  if val is a neighbor w and val is active then
7  |  |  Emit(w,(min{w, m}, root)) ; // update v_m and r
   Map2: Input: {(v, (v_m, r))}
1  Emit(v,(v_m, r)) ;
   Reduce2: Input: key, values
1  Update the minimal vertex and root set (v_m, r) of key;
2  if nothing was updated then
3  |  Deactivate(key) ; Out_{i+1} ← Emit(key,(v_m, r)) ;
4  else
5  |  Activate(key); Signal; Out_{i+1} ← Emit(key,(v_m, r)) ;
```

Below we propose an algorithm to generate a set of RSGs from a general RDF graph such that all star, chain, tree and cycle queries are localized.

The critical step of the algorithm is to select a set of root vertices. First, all the source vertices, i.e. vertices with in-degrees as zero, are included in the set of root vertices. Then all the vertices that are reachable from each root vertex will form a RSG. For example, in Figure 4, the sub-graph rooted at source vertex 4 in partition S_3 is one of the RSGs of the RDF graph.

Second, there could be some vertices that not reachable from any source vertex. For example, in Figure 4, none of the source vertices (i.e. vertex $2, 3, 4$) can reach the directed cycle $1 \to 5 \to 8 \to 1$. To handle such situation, we choose the vertex with minimal ID in such a cycle as a root, i.e. vertex 1 in this example. Additional RSGs can be generated accordingly for the given set of roots produced by this step. Finally we can get a collection of RSGs that cover all the vertices in the RDF graph.

The benefits of our RSGs generation algorithm are two-fold. First, the algorithm can be easily parallelized, for example using the MapReduce framework. Second, by using the RSGs generated by our algorithm as the partition units, all the four types of common queries can be processed as local-queries. In the rest of this section, we first present the details of the algorithm and then prove its ability to localize complication queries.

Algorithm. The pseudo code of the RSG generation algorithm is illustrated in Algorithm 1. A set of vertices of an RSG rsg_r is denoted as $V(rsg_r)$, where r is its root. An example is given in Figure 4. It is implemented by a series of MapReduce jobs, which iteratively propagate information of each vertex v to its out-neighbors. The information include

the minimum-ID of the vertices that can reach v, i.e. v_m in the psuedo code, and the currently chosen roots of the RSGs that can cover v, i.e. r in the psuedo code. Specifically, this algorithm involves the following steps:

Step 1 (line 1-4): generate the information attached to each vertex v in the form of $(v,(v_m,r))$. Initially v_m is v itself and r contains v if v is a source vertex or otherwise r is an empty set. Then all vertices are flagged as active.

Step 2 (Map1 and Reduce1): if a vertex is active, generate the candidate v_m and r for all its out-neighbors.

Step 3 (Map2 and Reduce2): for each vertex v, merge all messages of v. If there is a new minimum-id or root, then set v active and update v. Otherwise, deactivate v.

Step 4 (line 6-9): iterate Step 2 and Step 3, until there is no update occurred.

After invoking Algorithm 1, the RSGs rooted at all source vertices are found. Then for the vertices in the cycles that are not reachable from any source vertex, we choose the minimum-id as the input root, then invoke Algorithm 1 again to generate the remaining RSGs.

THEOREM 4.1 (COMPLETENESS). *Given an RDF graph $G=(V,E)$, if it is decomposed into a set of RSGs by Algorithm 1, then each vertex v and edge (u,w) in G can be found in at least one RSG.*

PROOF. We prove this by contradiction. (i) Assume there exists one vertex $v \in V$, which cannot be found in any RSG. If v is a source, then v is in an RSG rooted at v. Thereby we derive a contradiction. If v is not a source and there exists a source r can reach v, then v is in the RSG rooted at r. If v is not a source and none of the sources can reach v, then there must exist a minimal-ID vertex can reach v and v is in the RSG rooted at that vertex. Again we derive a contradiction. (ii) Assume there exists one edge $(u,w) \in E$, which cannot be found in any RSG. By (i) we have that u must at least be in one RSG. we conclude that (u,w) is also in that RSG by Definition 4.1. Thus, we derive a contradiction. □

THEOREM 4.2. *Using the RSGs generated by Algorithm 1 as the partition unit, then all star, chain, tree and cycle queries are local-queries.*

PROOF. For brevity, we give a brief sketch of the proof. For each type of queries, there must be a vertex v in this query that can reach all other vertices. If v is a constant, then there must be a root vertex r that can reach v. If v is a variable, for each binding of v, there must be a root vertex r that can reach that binding. Thus, each match of this query is a subgraph of one RSG. □

4.2 RSGs Partitioning Problem Formulation

In this seciton, we will present how to optimized the allocation of RSGs to the computing nodes. Firstly, a k-way RSGs partitioning plan can be defined as follows.

DEFINITION 4.2 (K-WAY RSGS PARTITIONING PLAN). *Given an RDF graph G and its set of RSGs S, a k-way RSGs partitioning plan P contains k nonempty and disjoint subsets of RSGs, $P=\{S_1,...,S_k\}$, where $\bigcup_{i=1}^{k} S_i = S$.*

To optimize the RSGs partitioning plan, we should consider the following metrics.

Balance. A well balanced data distribution plays an important role for efficient parallel query processing. A substantial skewed data distribution may lead to some undesirably overloaded partitions, which become the performance bottlenecks for the whole cluster. This is because that the overloaded partitions may exceed the memory capacity of a single machine. To avoid the substantial overloaded partition, here, we use the number of RSGs contained in a partition S_i to measure its payload, which is denoted by $|S_i|$.

Data Redundancy. If two RSGs have overlaps and they are assigned to different partitions, then duplicate triples will be stored at both partitions. Such data redundancy would cause excessive redundant computations cost. Hence, we should control the overlaps of RSGs by assigning the correlated RSGs to the same partition. In particular, we have the following observation.

OBSERVATION 4.1. *If two RSGs share a vertex v, then they also share the RSG rooted at v.*

For example, in Figure 4, rsg_2 and rsg_3 share the vertex 6, then they share the RSG rooted at vertex 6, i.e., 6→9→11.

In addition, joins always occur on shared vertices between the triple patterns specified in an SPARQL query (recall Section 2.1). Hence, to localize more joins, we have the following observation.

OBSERVATION 4.2. *If all the triples that contain shared vertices between any pair of query triple patterns in a query are allocated in the same partition, then this query is a local-query.*

Based on the above observations, we know that if a vertex is shared by multiple different RSGs, then by placing all these RSGs in one partition can localize all the possible joins on this vertex. It will also reduce the duplicate triples connected with this vertex. Therefore, to measure the correlation between two RSGs, we use the difference between their sets of vertices. It is also called the distance between the two RSGs.

To better understand the distance function, we first give a simple way of extending the distance between RSGs to a real-value function. Let $V(rsg)$ be the set of all the vertices in rsg. $V(rsg)$ can be denoted as a $|V|$-dimensional binary vector $\boldsymbol{v_{rsg}}=\langle v_{rsg}^{(1)}, v_{rsg}^{(2)}, ..., v_{rsg}^{(|V|)}\rangle$, where $|V|$ is the number of all the vertices in RDF graph G and $v_{rsg}^{(j)} \in \{0,1\}$. If $v_j \in V$ is in $V(rsg)$, then the $v_{rsg}^{(j)}$ is 1 or 0 otherwise. Then, given two RSGs rsg and rsg', their distance can be defined as follows:

$$d(rsg,rsg') = d(\boldsymbol{v_{rsg}},\boldsymbol{v_{rsg'}}) = \sum\nolimits_{i=1}^{|V|}(v_{rsg}^{(i)} - v_{rsg'}^{(i)})^2 \quad (4.1)$$

Then the intra-correlation (called IC) of a RSGs partitioning plan P is defined as:

$$IC(P) = \frac{1}{k}\sum_{i=1}^{k}\frac{1}{|S_i|^2}\sum\nolimits_{\forall(rsg,rsg')\in S_i\times S_i} d(rsg,rsg') \quad (4.2)$$

A plan with a lower IC tend to be more efficient in executing SPARQL queries, since it collocates more correlated RSGs, hence localizes more joins and reduce more duplicate triples.

4.2.1 Problem Statement

Given an RDF graph G, our goal is to find the optimal partitioning plan $P=\{S_1,...,S_k\}$ for the RSGs partitioning problem. According to the metrics, the objective function for this problem is defined as follows:

$$\textbf{minimize } IC(P) \ \ \textbf{s.t.} \ \ |S_i| \leqslant \lceil\frac{|R|}{k}\rceil, 1 \leqslant i \leqslant k \quad (4.3)$$

Algorithm 2: RSG-based K-means Partitioning

Input: RSGs set $\{V(rsg_r)|r \in R\}$
Output: a partition result $P = \{S_1, ..., S_k\}$
Map: **Input**: RSG $V(rsg_r)$, centroid set $\{\boldsymbol{C_1}, ..., \boldsymbol{C_k}\}$
1 $N \leftarrow \emptyset$; $C \leftarrow \{\boldsymbol{C_1}, ..., \boldsymbol{C_k}\}$;
2 **while** *true* **do**
3 $N \leftarrow$ `FindNearestCentroid` $(C, V(rsg_r))$;
4 $\forall C_i \in N$, choose the minimal loaded partition S_i ;
5 **if** $S_i < \lceil \frac{|R|}{k} \rceil$ **then** `emit`$(i, V(rsg_r))$;
6 **else** $C \leftarrow C - N$;

Reduce: **Input**: partition S_i, $\{V(rsg_r)|rsg_r \in S_i\}$
1 $C_i \leftarrow$ `UpdateCentroid` $(\{V(rsg_r)|rsg_r \in S_i\})$;

where $|R|$ indicates the number of RSGs in this graph.

We can reduce the balanced graph partition problem, which is an NP-hard problem, into our problem (the proof is omitted due to the limited space). Hence our problem is also NP-hard. Our goal is to find a reasonable heuristic-based approach.

4.3 RSG-based K-means Partitioning

We use the k-means partitioning algorithm to find an approximate solution.

Distance and Centroid. The k-means partitioning algorithm requires a proper definition of the centroid of a cluster to represent all vertices belong to this cluster. Here, we use a vector $\boldsymbol{C_i}$=$\langle c_i^{(1)}, c_i^{(2)}, ..., c_i^{(|V|)} \rangle$ as the centroid of partition S_i. Different from the vector $\boldsymbol{v_{rsg}}$, each dimensional value of $\boldsymbol{C_i}$ is a real number in $[0, 1]$. Then we can calculate the value of $\boldsymbol{C_i}$ on each dimension as follows:

$$c_i^{(j)} = \frac{1}{|S_i|} \sum_{rsg \in S_i} v_{rsg}^{(j)} \tag{4.4}$$

According to the above formula, the centroid not only represents all vertices belong to this cluster, but it also indicates the number of RSGs each vertex belongs to in this cluster.

In the k-means partitioning algorithm, we use the objective function in Eq. 4.2 to measure the distance between RSGs and the centroid of a partition. Formally, for each partition S_i, the objective function of the partitioning can be defined as follows:

$$D = \sum_{i=1}^{k} D_i = \sum_{i=1}^{k} \frac{1}{|S_i|} \sum\nolimits_{rsg \in S_i} d(\boldsymbol{v_{rsg}}, \boldsymbol{C_i}) \tag{4.5}$$

Full Algorithm. By combining the elements defined above, we can now present the RSG-based k-means partitioning algorithm. The pseudo code is presented in Algorithm 2, which runs in the following steps:

Step 1: Randomly assign the RSGs into k partitions, where k is the number of computing nodes in the cluster.

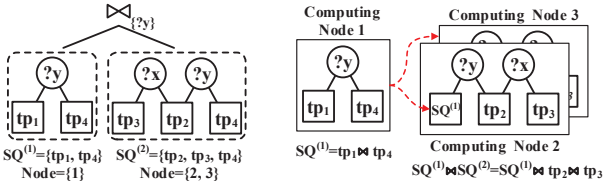
Step 2: For each RSG rsg, we find the nearest and minimal loaded partition S_i from the set of non-fullly loaded partitions (i.e., $|S_i| < \lceil \frac{|R|}{k} \rceil$).

Step 3: For each S_i, update the centroid C_i.

Step 4: Repeat Step 2 and Step 3 until it converges.

Step 5: Finally, we use the partitioned sets of roots to get the final partitioning results, i.e. extending the $V(rsg_j)$ to a real RSG with triples.

Figure 4 gives a partitioning result using this method. One can see that the query in Figure 1(b) can be executed locally and the duplicate triples produced are less than those produced by 1-hop based hash partitioning (Figure 1(d)).



(a) Compile-Time (b) Run-Time

Figure 5: Distributed Query Processing

5. QUERY PROCESSING

5.1 Query Decomposition

We present a partition-aware query decomposition scheme to maximize the localization of query operations and to reduce the number of local subqueries such that the intermediate results shipped across computing nodes will be reduced significantly. A SPARQL query Q can be represented as a graph that contains variables in the subject, predicate and object. Thus, we first identify all the source vertices of query graph Q and use them as the roots to partition Q into a set of RSGs (subqueries) SQ, each of which is a local-query (Theorem 4.2). For example, consider the following query:

```
tp1: (<AssociateProfessor0> ub:teacherOf ?y)
tp2: (?x ub:takesCourse ?y)
tp3: (?x rdf:type ub:GraduateStudent)
tp4: (?y rdf:type ub:GraduateCourse)
```

This query can be decomposed into two subqueries. The first is {*tp1, tp4*} with root "*<AssociateProfessor0>*" and the second is {*tp2, tp3, tp4*} with root "*?x*".

In the next step, for each individual subquery in SQ, we search for the computing nodes that contain its results. To speed up this process, we construct an index to map each vertex in the RDF graph to the computing nodes holding the partitions that contains the vertex. By using this index, the exact computing nodes that contain the results of a subquery SQ_i can be retrieved as follows:

Case 1: if SQ_i's root is a constant, then each computing node that contains this root has complete results of SQ_i. We call each of such computing nodes as a matching node to SQ_i.

Case 2: if SQ_i's root is a variable, then the intersection of the locations of all the constant in SQ_i are the nodes holding the results. Here, if the intersection contains exactly one node, then we call it the matching node of SQ_i.

Then, for each pair of subqueries, if they have a shared matching node, then we can merge these two subqueries into a new subquery, which is guaranteed to be a local-query. Finally, this will produce a minimal number of subqueries based on our partitioning algorithm, and each of such subqueries is a local-query.

5.2 Query Optimization and Evaluation

If a query is decomposed into two or more subqueries, we need to define the global execution workflow to process the distributed and local joins such that the query will be processed with minimal communication overhead. The critical issue is to determine ordering of query operations [16].

Here we adopt a typical two-stage scheme for physical query plan generation. First, the join order between sub-queries is calculated by the master based on the global statistics at compile-time. Then the join ordering within each subquery is computed locally by individual computing nodes at run-time using the statistics maintained at each computing node.

Compile-Time Optimization. After query decomposition, we represent each subquery as a bipartite graph whose vertices can be divided into two disjoint sets T and Var. Each vertex in T represents a triple pattern in the subquery and each vertex in Var represents a variable appears in the subquery. If a variable appears in a triple pattern, then an edge connects their corresponding vertices. Figure 5(a) shows an example.

In the master, we only generate the order of the joins between the subqueries and the cardinality of the results of these subqueries can be estimated by the statistics stored in the master (Section 5.3). In particular, we use dynamic programming to build a left-deep tree, in which each leaf node is a subquery. Assume that a query is decomposed into n subqueries, a subplan L_k that contains k subqueries is an ordered sequence of subqueries $SQ^{(1)}, ..., SQ^{(k)}$, and the remaining set of subqueries is denoted by SQ_{rem}. Then the state transition equation can be defined as follows:

$$L_{k+1} = \min_{SQ_i \in SQ_{rem}} \{L_k + Card(L_k \bowtie SQ_i)\} \quad (5.1)$$

After building the left-deep tree, we will search the tree bottom-up to remove the redundant triple patterns which might lead to redundant computations. For example, in Figure 5(a), the tp_4 in $SQ^{(2)}$ will be removed.

Run-Time Optimization and Evaluation. The join ordering within each subquery will be generated locally at each computing node. Specifically, when $SQ^{(i)}$ is performed, the results of $SQ^{(i)}$ will be transferred to the computing nodes holding the results of $SQ^{(i+1)}$. In each of these computing nodes, the results of $SQ^{(i)}$ will be joined with other triple patterns. The local optimizer in each computing node will optimize the local joins involved in $SQ^{(i+1)}$ independently. The join ordering problem has been well studied and is out of the scope of this paper. In our prototype, we use the join ordering algorithm proposed in TripleBit. Due to the optimization at compile-time, the size of transferred intermediate results is minimized. Figure 5(b) shows an example that $SQ^{(1)}$'s results are forwarded to node 2 and node 3 which contain the results of $SQ^{(2)}$.

5.3 Cardinality Estimation

Global Statistics. In the prototype, we build the global statistics to store the number of triples matching each possible single triple pattern. In SPARQL queries, there are seven forms of triple patterns, which are $(?s, ?p, ?o)$, $(s, ?p, ?o)$, $(s, p, ?o)$, $(?s, ?p, o)$, $(?s, p, o)$, $(?s, p, ?o)$ and $(s, ?p, o)$ respectively. For the triple patterns that only have one variable, such as $(s, p, ?o)$, we store the number of triples matching each triple pattern. For the triple patterns containing two or more variables, in addition to the number of triples matching this triple pattern, we also store the number of distinct bindings for each variable. Take $(s, ?p, ?o)$ as an example. For each constant s in the RDF data, we will store the number of triples with s as their subjects as well as the number of distinct predicates and the number distinct objects within these triples.

Estimation of Cardinality. The cardinality of a selection or a join operation is the number of results that satisfy the selection or join condition. Using the aforementioned global statistics, we can find the cardinality of a given triple pattern tp, denoted as $Card(tp)$. If two triple patterns tp_1 and tp_2 join on shared variable(s), denoted as $J = Var(tp_1) \cap Var(tp_2)$, then the cardinality of the results of the join between these two triple patterns is estimated in the following way:

- If $J = Var(tp_1) = Var(tp_2)$, then:

$$Card(tp_1 \bowtie tp_2) = \min\{Card(tp_1), Card(tp_2)\} \quad (5.2)$$

- If $J = Var(tp_1)$ and $J \subsetneq Var(tp_2)$, then

$$Card(tp_1 \bowtie tp_2) = \min\{Card(tp_1) \times \frac{\mathscr{B}(J, tp_1)}{\mathscr{B}(J, tp_2)}, Card(tp_2)\} \quad (5.3)$$

- If $J \subsetneq Var(tp_1)$ and $J \subsetneq Var(tp_2)$, then

$$Card(tp_1 \bowtie tp_2) = \frac{Card(tp_1) \times Card(tp_2)}{\max\{\mathscr{B}(J, tp_1), \mathscr{B}(J, tp_2)\}} \quad (5.4)$$

where $\mathscr{B}(J, tp)$ indicates the number of distinct bindings of J in the results of tp.

The number of distinct bindings of a variable (or a set of variables) in all permutation of single triple patterns can be found in the statistics. However, the number of distinct bindings of a variable (or a set of variables) J in the results of a join between tp_1 and tp_2, can be estimated as follows:

$$\mathscr{B}(J, tp_1 \bowtie tp_2) = \min\{\mathscr{B}(Var(tp_1), tp_1) \times \mathscr{B}(J - Var(tp_1), tp_2), \mathscr{B}(Var(tp_2), tp_2) \times \mathscr{B}(J - Var(tp_2), tp_1), Card(tp_1 \bowtie tp_2)\} \quad (5.5)$$

where $J \subseteq Var(tp_1) \cup Var(tp_2)$ and $\mathscr{B}(\emptyset, tp) = 0$.

6. EXPERIMENTAL EVALUATION

We compare **SemStore** with six state-of-the-art RDF engines, including two centralized engines, **RDF-3X** [16] and **TripleBit** [22,23] and four distributed engines, **SHARD** [18], **Hybrid** system [12,15], which is an engine integrating Hadoop and RDF-3X, **Hive** and **Trinity.RDF** [24]. SHARD adopts an edge-based partitioning. Hive uses vertical partitioning [3] and RCFile [11] techniques. For the Hybrid system, we implement two typical data partitioning algorithms as proposed in [12], which uses METIS [1] as the graph partitioning tool and applies undirected one-hop block and undirected two-hop block as the partition unit, denoted as **un-one** and **un-two** respectively. We also implement semantic hash partitioning algorithm in [15], 2-hop forward, denoted as **2f**, which is reported in [15] to perform the best in most cases. Finally, SemStore is implemented using C++, compiled with GCC, using -O2 option to optimize. We refer to the RSG-based k-means partitioning algorithm as **RSG-KM**. We also implement a baseline RSG-based partitioning algorithm, called **RSG-Hash**, which assigns each RSG to the partitions by hashing the root of the RSG.

Unless otherwise stated, our experiments use a default cluster consisting of 1 master and 16 computing nodes, each of which has one processor at 2.4GHz and 4GB RAM, and we run the centralized engines on a powerful single machine with 4 way 4-core 2.13GHz CPU and 64GB RAM.

We choose two datasets LUBM [8] and UniProt [2] for the experiments. LUBM is a benchmark generator. In

Table 1: Queries

	Star	Chain	Tree	Cycle	Complex
LUBM	Q2, Q4, Q5	Q8	Q6,Q9	Q1, Q3, Q7	Q10
UniProt	Q2	Q5	Q1, Q3, Q4, Q6		

Table 2: Data Loading Time

	SemStore	SHARD	Hybrid (un-one)	Hybrid (un-two)	Hybrid (2f)	Hive
LUBM-2000	85 min	542 min	330 min	445 min	79 min	67 min
LUBM-10240	364 min	N/A	N/A	N/A	283 min	256 min
UniProt	952 min	N/A	N/A	N/A	852 min	469 min

the experiments, we generate LUBM datasets with 1000, 2000, 5000 and 10240 universities with the size ranging from 138 million of triples to about 1.4 billion of triples, which are denoted as LUBM-1000, LUBM-2000, LUBM-5000 and LUBM-10240 respectively. The UniProt dataset is a real-life protein dataset that contains nearly 2 billion of triples. Our experiments feature 16 queries (listed in Appendix), covering most types of queries mentioned earlier in this paper, as shown in Table 1.

6.1 Analysis of Data Partitioning Algorithms

Due to the large memory consumption of METIS, we cannot get it to work on large datasets and we can only perform the Hybrid (un-one) and Hybrid (un-two) over the LUBM-2000 dataset.

Data Loading Time. In Table 2, we compare the data loading time of the different distributed engines, which include the time to perform the data partitioning algorithms. Hive has the fastest data loading time. This is because the vertical partitioning scheme is simple and there is no redundancy. SemStore and Hybrid (2f) need more loading time than Hive, because these two algorithms are much more complex than vertical partitioning. SHARD is the slowest due to the fact that it requires to covert the RDF data to a special HDFS file format and hence incurs an excessive data loading cost. Moreover, Hybrid (un-one and un-two) spend more time than Hybrid (2f) due to the extra cost of graph partitioning and the large amount of redundant triples that needs to be stored and processed.

Redundancy. Since 2f, un-one, un-two, RSG-KM and RSG-Hash generate overlapping partitions, we report the ratio of total number of triples produced by each partitioning algorithm to the original datasets, as shown in Table 3. Due to the existence of high degree vertices, the numbers of redundant triples in un-one and un-two are very large. Moreover, with the hop count increasing from one to two, the number of duplicate triples grows rapidly. The 2f algorithm reduces the duplicate triples significantly, which is consistent with what is reported in [15]. Since the overlaps between RSGs are very large, using hash functions to allocate RSGs will incur redundant triples. The RSG-KM achieves a dramatic reduction on the duplicate triples in comparing to RSG-Hash, because the k-means partitioning algorithm attempts to assign highly correlated RSGs into one partition.

Data Distribution. Table 4 shows the standard deviation of the number of triples allocated to each computing node. The results show that RSG-KM achieves the most balanced data distribution across computing nodes. RSG-Hash, 2f and Hive are not as balanced as RSG-KM, but they are better than un-one and un-two. Due to the existence of high degree vertices, the data skewness in un-one and un-two is high.

Table 3: Redundancy

	RSG-KM	RSG-Hash	un-one	un-two	2f
LUBM-2000	1.03x	1.66x	1.22x	3x	1.12x
UniProt	1.29x	3.02x	N/A	N/A	1.45x

Table 4: Data Distribution (Standard Deviation)

	RSG-KM	RSG-Hash	un-one	un-two	2f	Hive
LUBM-2000	2.8E5	1.4E6	7.4E6	3.6E7	1.2E6	2.5E6

6.2 Performance Comparisons

Comparison with Centralized Engines (LUBM-2000). We first compare SemStore with centralized engines, TripleBit and RDF-3X over LUBM-2000. The results are shown in Figure 6(a). To better analyze the differences in performance, we classify the queries into two categories. Queries of the first type are Q4, Q5, Q6, Q9 and Q10, which are highly selective and called low-load queries. Q1, Q2, Q3, Q7 and Q8 are the second type with low-selectivity and a large input, called high-load queries.

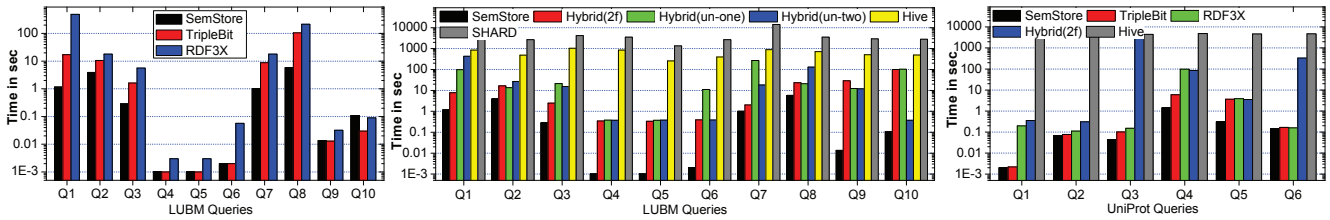
For low-load queries (Q4, Q5, Q6 and Q9), the performances of SemStore and TripleBit are better than RDF-3X. This is because these queries are local-queries for SemStore. And the intermediate results are significantly reduced by the selective triple pattern scanning. An interesting data point is Q10, which has to be decomposed into two subqueries in SemStore. The more subqueries incur more network communication cost. However, benefiting from query decomposition and optimization strategies, intermediate results shipped across network are reduced dramatically. Thus Q10 in SemStore takes an acceptable query response time which is less than 0.2 second.

On the contrary, high-load queries with low selectivity and big input and output sizes are executed more efficiently and effectively by SemStore. This can be attributed to the high degree of parallelism of SemStore. Even though the cluster running SemStore and the single server running the centralized engines have the same total amount of main memory, SemStore can better take use of the more number of CPUs and the higher I/O bandwidth in the cluster.

Comparison with Distributed Systems (LUBM-2000). Then, we compare SemStore with the existing distributed RDF data engines: Hybrid (un-one), Hybrid (un-two), Hybrid (2f), Hive and SHARD. Figure 6(b) shows the experimental results. The first observation is that SemStore outperforms other systems dramatically, even more than an order of magnitude on some benchmark queries. This is because, except Q4, all queries are local-queries for SemStore, such that each of them can be processed in parallel without cross-node interactions.

For the Hybrid system, queries are decomposed into multiple subqueries and each subquery is processed in parallel by the individual RDF-3X engines on the computing node. Then the results of each subquery are joined by the Hadoop system to generate the final results. Benefit from performance of RDF-3X and the partitioning algorithm, the Hybrid system with different partitioning methods can outperform Hive and SHARD. Hive has a better performance than SHARD due to the query optimization and the RCFile [11] techniques. Without query optimization, SHARD produces a large amount of intermediate results, which lead to a significant impact on query performance.

Furthermore, the three different partitioning algorithms in the Hybrid system also have significantly different performance. First of all, different data partitioning algorithms



(a) Centralized Engines (LUBM-2000) (b) Distributed Engines (LUBM-2000) (c) UniProt

Figure 6: Performance Comparisons (Time in *sec*)

Table 5: Performance on LUBM-10240 (Time in *ms*)

	SemStore	Trinity.RDF	Hybrid(2f)	Hive	TripleBit	RDF-3X
Q1	**5,672**	12,648	396,145	4.1E6	29,387	1.9E6
Q2	**5,826**	6,018	17,841	3.7E6	43,854	63,978
Q3	**1,979**	8,735	7,871	4.6E6	17,486	15,205
Q4	**1**	5	389	5.1E6	**1**	40
Q5	**1**	4	391	1.2E6	**1**	3
Q6	**1.9**	9	451	2.4E6	2	191
Q7	**7,485**	31,214	14,356	4.5E6	153,613	231,572
Geo. Mean	**139**	450	4,753	3.3E6	491	3,711

imply different query decomposition plans for each query, which would lead to significantly different cost for the distributed joins running on the MapReduce framework. Particularly, Q1, Q3, Q6, Q7, Q9 and Q10 need to be decomposed in Hybrid (un-one). Q9 needs to be decomposed in Hybrid (un-two) and Q9 and Q10 need to be decomposed in Hybrid (2f). The second factor is the skewness of data distribution and the number redundant triples. For high-load queries, such as Q1 and Q8, although they do not need to be decomposed in Hybrid (un-two), their performance in Hybrid (un-two) is the worst among all the partitioning methods in the Hybrid system. This is because un-two method causes skewed data distribution and a large amount of redundant (see Table 3 and Table 4), and the processing time depends on the computing node with the largest amount of triples, which is the bottleneck in the cluster.

All in all, the data partitioning algorithm has a rather decisive effect on the performance of distributed RDF engines.

Performance on Larger Datasets. We experiment on two large dataset, UniProt and LUBM-10240, to analyze the performance of SemStore over large-scale datasets. We exclude SHARD in this section due to its generally low performance. For the UniProt dataset, Q1, Q2, Q3 and Q6 are low-load queries, and Q4, Q5 are high-load queries. The results are presented in Figure 6(c). One can make a conclusion similar to our previous experiments. Hybrid (2f) performs especially worse for Q3 and Q6 simply because they have to be decomposed in Hybrid (2f) and the distributed joins of the large intermediate results are pretty costly.

Table 5 summaries the experimental results over LUBM-10240 dataset. Trinity.RDF [24] is the current state-of-the-art of distributed RDF engine. However, Trinity.RDF is not openly available. To compare with Trinity.RDF, we use the the same dataset and benchmark queries over a similar cluster setup (5 computing nodes with 24GB RAM connected by 1GBit network bandwidth in our cluster as opposed to 5 computing nodes with 96GB RAM connected by 40GBit network bandwidth in [24]). And the query run times of Trinity.RDF in Table 5 are those reported in [24]. One can see that, SemStore outperforms other systems for all the queries. This is because that each of these queries can be executed locally in SemStore. Moreover, due to the well-balanced partitioning scheme, there is no bottleneck in the cluster.

6.3 Scalability

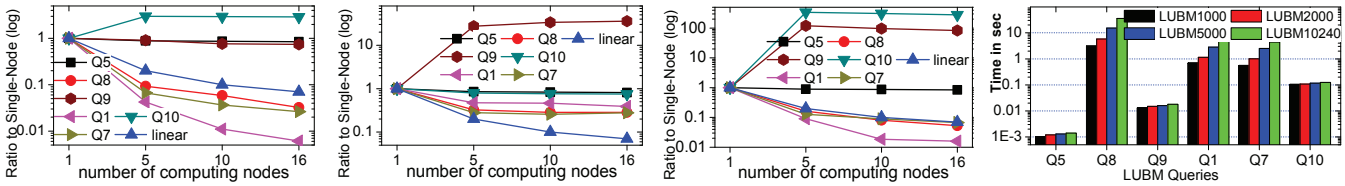
Varying Size of Cluster. This section reports the comparison of three engines with varying size of computing cluster. In particular, we run queries on the LUBM-2000 dataset on clusters with 1, 5, 10 and 16 computing nodes respectively. For brevity, we only use six queries, which cover all types of queries. The results are displayed in Figure 7(a), 7(b) and 7(c). We normalize the query execution time of each query to single-node execution time, and include a linear speedup line (1, 0.2, 0.1 and 0.06 for 1, 5, 10 and 16 computing nodes) for comparison. Hybrid (un-one) and Hybrid (un-two) have similar experimental results. Hive achieves a poor performance. Thus, we only show the Hybrid (un-two) and Hybrid (2f) in this comparison.

For SemStore in Figure 7(a), high-load queries (Q1, Q8 and Q7) achieve super linear speedup as the number of computing nodes increases. This is because, on one hand, each query can be performed locally without communication cost. On the other hand, due to the well-balanced partitioning scheme, there is no bottleneck computing node. In Q5 and Q9, due to the high selectivity of each query, even in a single-machine these queries can be executed efficiently (within 2 ms). Thus, the execution times from 1 to 16 computing nodes are roughly the same. Q10 is a low-load query which needs to be decomposed in SemStore. Since the intermediate results shipped across computing nodes are reduced significantly by the query decomposer and optimizer, the performance of multi-nodes is little lower than that of the single-machine.

For Hybrid (un-two), Q1, Q7, Q5, Q8 and Q10 have few benefit when going from 1 to 16 computing nodes. This is because the data distribution of Hybrid(un-two) is extraordinarily skewed, as discussed in Section 6.1. The overloaded partitions become the bottleneck of query processing. For Q9, this query needs to be decomposed when processing, which leads to large amount of intermediate results transferring. Therefore, the performance of multi-nodes is much lower than the performance of the single-machine which do not need to partition the dataset.

For Hybrid (2f), since 2f algorithm reduces the duplicate triples significantly and achieves a more balanced data distribution in comparing to un-two, the high-load queries (i.e. Q1, Q7 and Q8) achieve linear speedup. Q9 and Q10 are low-load queries for 2f, however, they need to be decomposed. Thus, the performance of multi-nodes is also much lower than that of the single-machine.

Varying Data Sizes. Figure 7(d) shows the query execution time of SemStore over four LUBM datasets. As the number of triples increases, the execution time of each query also increases. We observe that the increase of execution time is sublinear or nearly linear, which means SemStore achieves a good scalability. In particular, Q5, Q9 and Q10



(a) SemStore (b) Hybrid (un-two) (c) Hybrid (2f) (d) Varying Size of Data

Figure 7: Scalability

are low-load queries and hence with growth of data size, their processing time do not increase much. Q1, Q8 and Q7 are high-load queries. Hence, benefiting from the well-balanced data distribution, low data redundancy and minimal query decomposition, the processing time increases almost linearly.

7. CONCLUSION

In this paper, we present SemStore, a semantic-preserving distributed RDF triple store for scalable SPARQL query processing. We design a partitioning algorithm for RDF data using RSG, a coarse-grained structure, as the partition unit. Furthermore, we proposed a k-means partitioning algorithm to place highly correlated RSGs in the same partition. A query decomposition algorithm and a query optimization strategy are proposed to minimize the communication cost for query execution. The experimental study shows that SemStore outperforms the state-of-the-art systems by orders of magnitude in terms of query response time.

8. ACKNOWLEDGMENTS

The research is supported by the NSFC under grant No. 61133008 and National High Technology Research and Development Program of China (863 Program) under grant No.2012AA011003. Ling Liu acknowledges the partial support by the National Science Foundation under grants IIS-0905493, CNS-1115375, IIP-1230740, and a grant from Intel ISTC on Cloud Computing.

9. REFERENCES

[1] *METIS*. `http://www.cs.umn.edu/~metis/`.
[2] *UniProt*. `http://www.uniprot.org/`.
[3] D. Abadi, A. Marcus, et al. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.
[4] D. Abadi, A. Marcus, et al. SW-Store: a vertically partitioned DBMS for semantic web data management. *The VLDB Journal*, 18(2):385–406, 2009.
[5] M. Atre, V. Chaoji, et al. Matrix bit loaded: a scalable lightweight join query processor for RDF data. In *WWW*, 2010.
[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[7] M. A. Gallego, J. D. Fernández, et al. An empirical study of real-world SPARQL queries. In *USEWOD*, 2011.
[8] Y. Guo, Z. Pan, et al. LUBM: a benchmark for OWL knowledge base systems. *Web Semantics*, 3(2):158–182, 2005.
[9] S. Harris, N. Lamb, et al. 4store: The design and implementation of a clustered RDF store. In *SSWS*, 2009.
[10] A. Harth, J. Umbrich, et al. YARS2: A federated repository for querying graph structured data from the web. In *ISWC*, 2007.
[11] Y. He, R. Lee, et al. RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. In *ICDE*, 2011.
[12] J. Huang, D. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. In *VLDB*, 2011.
[13] M. Husain, J. McGlothlin, et al. Heuristics based query processing for large RDF graphs using cloud computing. *IEEE TKDE*, 23(9):1312–1327, 2011.
[14] Z. Kaoudi, K. Kyzirakos, and M. Koubarakis. SPARQL query optimization on top of DHTs. In *ISWC*, 2010.
[15] K. Lee and L. Liu. Scaling queries over big rdf graphs with semantic hash partitioning. In *VLDB*, 2014.
[16] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010.
[17] J. Rao, C. Zhang, et al. Automating physical database design in a parallel database. In *SIGMOD*, 2002.
[18] K. Rohloff and R. E. Schantz. Clause-iteration with mapreduce to scalably query datagraphs in the SHARD graph-store. In *DIDC*, 2011.
[19] L. Wang, Y. Xiao, B. Shao, and H. Wang. How to partition a billion-node graph. In *ICDE*, 2014.
[20] S. Yang, X. Yan, et al. Towards effective partition management for large graphs. In *SIGMOD*, 2012.
[21] T. Yang, J. Chen, et al. Efficient SPARQL query evaluation via automatic data partitioning. In *DASFAA*, 2013.
[22] P. Yuan, P. Liu, et al. Triplebit: a fast and compact system for large scale rdf data. In *VLDB*, 2013.
[23] P. Yuan, C. Xie, et al. Dynamic and fast processing of queries on large-scale RDF data. *KAIS*, 2014.
[24] K. Zeng, J. Yang, et al. A distributed graph engine for web scale rdf data. In *VLDB*, 2013.
[25] L. Zou, J. Mo, et al. gstore: answering SPARQL queries via subgraph matching. In *VLDB*, 2011.

Appendix: Benchmark Queries

LUBM:
Q1-Q7: Same as Q1-Q7 in [5] respectively.
Q8: SELECT ?x WHERE { ?x rdf:type ub:UndergraduateStudent . }
Q9: SELECT ?x ?w WHERE { ?x ub:advisor ?y. ?y ub:worksFor ?z. ?x rdf:type ub:GraduateStudent. ?z ub:subOrganizationOf ?w. ?w ub:name ?u. ?z rdf:type ub:Department.?w rdf:type ub:University. <http://www.Department12.University0.edu/FullProfessor0/Publication0> ub:publicationAuthor ?x. }
Q10: SELECT ?x ?y WHERE { ?x rdf:type ub:GraduateStudent. <http://www.Department0.University0.edu/AssociateProfessor0> ub:teacherOf ?y. ?y rdf:type ub:GraduateCourse. ?x ub:takesCourse ?y. }
UniProt:
Q1: SELECT ?p2 ?i ?p1 WHERE { ?p1 rdf:type uni:Protein. ?p1 uni:enzyme <http://purl.uniprot.org/enzyme/2.7.7.-->. ?i uni:participant ?p1. ?i rdf:type uni:Interaction. ?i uni:participant ?p2. ?p2 rdf:type uni:Protein. ?p2 uni:enzyme <http://purl.uniprot.org/enzyme/3.1.3.16>. }
Q2: SELECT ?a ?vo WHERE { ?a uni:encodedBy ?vo. ?a schema:seeAlso <http://purl.uniprot.org/refseq/NP_346136.1>. ?a schema:seeAlso <http://purl.uniprot.org/tigr/SP_1698>. ?a schema:seeAlso <http://purl.uniprot.org/pfam/PF00842>. ?a schema:seeAlso <http://purl.uniprot.org/prints/PR00992>. }
Q3: SELECT ?a ?vo WHERE { ?a uni:annotation ?vo. ?a schema:seeAlso <http://purl.uniprot.org/interpro/IPR000842>. ?a schema:seeAlso <http://purl.uniprot.org/geneid/945772>. ?a uni:citation <http://purl.uniprot.org/citations/9298646>. }
Q4: SELECT ?p ?a WHERE { ?p uni:annotation ?a. ?p rdf:type uni:Protein. ?a uni:range ?r. ?a rdf:type <http://purl.uniprot.org/core/Transmembrane_Annotation>. }
Q5: SELECT ?p ?a WHERE { ?p uni:annotation ?a. ?p rdf:type uni:Protein. ?p uni:organism taxon:9606. ?a rdfs:comment ?t. ?a rdf:type <http://purl.uniprot.org/core/Disease_Annotation>. }
Q6: SELECT ?a ?vo WHERE { ?a uni:modified ?vo. ?a uni:reviewed "false". ?a uni:mnemonic "CDL1_HUMAN". ?a rdf:type uni:Protein. ?a uni:obsolete "true". }