Community-Based RDF Graph Partitioning

Fredah Banda and Boris Motik

University of Oxford, Oxford, UK
firstname.lastname@cs.ox.ac.uk

Abstract. A common approach to processing large RDF datasets is to partition the data in a cluster of shared-nothing servers and then use a distributed query evaluation algorithm. It is commonly assumed in the literature that the performance of query processing in such systems is limited mainly by network communication. In this paper, we show that this assumption does not always hold: we present a new RDF partitioning method based on Louvain community detection, which drastically reduces communication, but without a corresponding decrease in query running times. We show that strongly connected partitions can incur workload imbalance among the servers during query processing. We thus further refined our technique to strike a balance between reducing communication and spreading processing more evenly, and we show that this technique can reduce both communication and query times.

Keywords: RDF · Louvain modularity · Community partitioning

1 Introduction

Resource Description Framework (RDF) is a widely-used data model for publishing data on the Web. Applications typically rely on RDF systems to store and answer SPARQL queries over RDF data; Wylot et al. [28] have recently surveyed the state of the art in RDF systems. Centralised systems such as Jena [7], Sesame [6], RDFSuite [3], and RDF-3X [21] store and process the entire dataset on a single server. However, the size of RDF datasets has been steadily increasing. Thus, datasets containing several billions of triples are routinely encountered in practice, and they often exceed the capacity of a single server.

A common solution to that problem is to partition a dataset across a cluster of shared-nothing, interconnected servers and use a distributed query processing strategy. Numerous distributed RDF systems have been developed along these principles, such as YARS2 [14], 4store [13], H-RDF-3X [16], Trinity.RDF [29], SHARD [25], SHAPE [18], Partout [8], AdPart [2], TriAD [11], SemStore [27], DREAM [12], WARP [15], and a prototype based on RDFox [24]. Abdelaziz et al. [1] surveyed 22 and evaluated 11 distributed RDF stores on a variety of data and query loads, and they showed AdPart [2] and TriAD [11] to be among the best performing ones. Computing joins presents the main technical difficulty in distributed query processing since all triples participating in a join need to be transmitted over the network to a common server. Since network communication is much slower than accessing local disk or RAM, it is commonly assumed that reducing or even eliminating communication is key to the scalability of distributed RDF stores. Thus, many existing RDF partitioning techniques aim to place groups of related triples on the same server and thus decrease communication cost. Moreover, to further reduce communication, many systems duplicate data—that is, they place the same triple on more than one server. While data duplication can even completely eliminate all communication, it can also lead to a significant increase in storage requirements [16].

Motivated by these common assumptions, we developed a new RDF partitioning technique based on Louvain community detection [5]—a well-known algorithm for detecting groups of highly connected vertices in a graph. By assigning communities to servers, we hoped to minimise communication and thus also query answering times. We target the distributed query answering technique by Potter et al. [24]: this technique aims to avoid communication when triples to be joined are colocated, which is achieved by using a special index for locating join counterparts and by employing a specific left-deep query evaluation plan.

We evaluated our technique on the LUBM [10] and WatDiv [4] benchmarks against two well-known data partitioning techniques: subject hash partitioning (which assigns each triple to the server obtained by hashing the subject value), and the weighted vertex partitioning with pruning by Potter et al. [24] (see Section 3). To our surprise, our results suggested that the relationship between network communication and query answering performance is not as straightforward as commonly assumed. Our technique was very effective at reducing network communication, which was consistently significantly below the other two techniques. However, this reduction in communication did not translate into a reduction in query answering times; in fact, our technique was sometimes even worst-performing. A careful analysis revealed this to be due to the workload imbalance between the servers. In particular, when servers contain strongly interconnected communities, query answers are only produced by data on one or just a handful of servers; thus, only some servers are involved in query processing, while others are largely idle. In contrast, with subject hash partitioning, the data contributing to query answers is likely to be evenly distributed across the cluster, and so query processing tends to be evenly distributed as well. Since the overall amount of work is the same in both cases and server processing is parallelised, subject hash partitioning can be more efficient than community partitioning, even though the latter involves significantly less communication.

Motivated by this observation, we further refined our community partitioning algorithm with the aim to strike a balance between producing strongly connected communities with distributing data evenly in the cluster. Our further experiments show that, on certain queries, such an approach can reduce network communication as well as improve query answering times.

2 Preliminaries

The Resource Description Framework (RDF) data model is commonly used for publishing structured data on the Web. RDF data is constructed using *resources*, which can be *IRIs*, *blank nodes*, or *literals* (e.g., strings or integers). An *RDF triple* is an expression of the form $\langle s, p, o \rangle$, where s, p, and o are resources called *subject*, *predicate*, and *object*, respectively. An *RDF graph* G is a finite set of triples. SPARQL is the standard language for querying RDF graphs. In this paper, we consider only the fragment of SPARQL covering *conjunctive queries* (CQs)—that is, queries of the form

$$\mathsf{SELECT}\ ?X_1 \ldots ?X_n\ \mathsf{WHERE}\ \{\ s_1\ p_1\ o_1\ .\ \ \ldots\ \ s_n\ p_n\ o_n\ \},$$

where each s_i, p_i, and o_i is a resource or a variable. Each $s_i\ p_i\ o_i$ is called a *triple pattern*. Conjunctive queries correspond to the select–project–join fragment of the relational algebra, and they form the basis of SPARQL.

In this paper, we study the problem of answering CQs over an RDF graph G stored in a cluster consisting of ℓ shared-nothing servers connected by a network. For convenience, we identify each server with a unique integer k between 1 and ℓ. We assume that the RDF graph is partitioned into ℓ *partition elements* $P_1, \ldots, P_\ell$—that is, each P_k is the subset of G stored at server k, and $\bigcup_{1 \leq k \leq \ell} P_k = G$. In our work we assume that data is stored *without replication*, so $P_k \cap P_{k'} = \emptyset$ for all $1 \leq k < k' \leq \ell$.

Distributed query evaluation algorithms need to know how to locate triples that can match different triple patterns, and so they often depend on the details of data partitioning. In this paper, we evaluate conjunctive SPARQL queries using the distributed query answering technique by Potter et al. [24], which aims to avoid communication when triples participating in a join reside on the same server. We next briefly summarise the main aspects of this algorithm on an example of evaluating a conjunctive query

$$\mathsf{SELECT}\ ?X_1\ ?X_2\ ?X_3\ \mathsf{WHERE}\ \{\ ?X_1\ r\ ?X_2\ .\ ?X_2\ s\ ?X_3\ \}$$

over an RDF graph split into partition elements $P_1 = \{\langle a, r, b \rangle, \langle b, s, c \rangle\}$ and $P_2 = \{\langle b, s, d \rangle\}$. The query is evaluated using index nested loop joins over a left-deep query evaluation plan. The query is sent to all servers, and each server starts asynchronously matching the first triple pattern of the query. In our example, server 2 cannot match the first triple pattern in P_2, so it stops evaluation. In contrast, server 1 matches the first triple pattern to triple $\langle a, r, b \rangle$ and thus produces a *partial binding* μ that maps variables $?X_1$ and $?X_2$ to resources a and b, respectively. Server 1 then proceeds to the second triple pattern, which can be matched both in P_1 and P_2. To take this into account, server 1 consults a special index called *occurrence mappings*, which informs the algorithm of the location of possible join counterparts. In our example, this index tells server 1 that triples containing b (i.e., the current value of $?X_2$) occurs in the subject position in P_1 and P_2. Therefore, server 1 sends a message to server 2 that contains the partial μ and instructs server 2 to continue matching the second triple pattern. Upon the receipt of that message, server 2 tries to match the second triple pattern in P_2; that is, it extends μ by additionally mapping $?X_3$ to d and thus produces one answer to the query. Moreover, server 1 also matches the second triple pattern in P_1; that is, it extends μ by additionally mapping

$?X_3$ to c and thus produces another answer to the query. No further matches are possible, so the servers use a specially designed protocol to inform each other that query evaluation is complete.

While the above summary omits several details, it highlights certain important properties of the approach. First, servers use a special index to locate join counterparts, which allows the algorithm to be applied to any partition of an RDF graph without replication (i.e., the algorithm makes no assumptions about how triples are allocated to servers as long as the index is constructed appropriately). Second, partial bindings correspond to the answers of the prefixes of the triple patterns; in our example, μ is an answer to the prefix containing just the first triple pattern. Third, distributed joins are processed not by moving data, but by moving computation. In our example, join processing 'hops' from server 1 to server 2. Fourth, the processing at all servers is asynchronous: a server can process messages with partial bindings as soon as they are received, and it does so independently from the other servers. As a result of these optimisations, this approach outperformed TriAD and AdPart on certain data and query loads [24].

3 Related Work

Wylot et al. [28] have recently surveyed the state of the art in both centralised and distributed RDF stores, and Abdelaziz et al. [1] have presented a detailed survey of the distributed RDF stores. Prominent distributed RDF stores include YARS2 [14], 4store [13], H-RDF-3X [16], Trinity.RDF [29], SHARD [25], SHAPE [18], AdPart [2], TriAD [11], SemStore [27], DREAM [12], WARP [15], Partout [8], and RDFox [24], to name just a few. In this section, we briefly summarise the data partitioning methods commonly used in these systems.

To partition RDF data, most systems aim to place triples with the same subject onto the same server. This is motivated by an empirical study of over three million SPARQL queries, which showed that approximately 60% of joins in queries are subject–subject joins, 35% are subject–object joins, and less than 5% are object–object joins [9]. Thus, by assigning triples with the same subject to the same partition element, no communication is needed for the most common type of join occurring in practice, which is widely seen as critical for good performance of query answering in distributed RDF systems [16].

Many RDF systems partition data using *subject hashing* [14, 29, 26, 23]. In particular, to partition an RDF graph G into ℓ partition elements, each triple $\langle s, p, o \rangle \in G$ is stored at server $(h(s) \mod \ell) + 1$, where h is a suitable hash function that maps the triple's subject to an integer. As mentioned in the previous paragraph, subject hashing requires no communication for subject–subject joins. Moreover, the technique is very simple and can be implemented in streaming fashion, which makes it very popular in practice. Finally, the technique typically produces very balanced partitions. The main drawback is that communication is usually needed for subject–object or object–object joins. To improve this, certain systems (e.g., TriAD [11]) also hash triples by object; this increases the likelihood that triples participating in a join are co-located, but it effectively duplicates the amount of data in the system, which can be prohibitive.

The RDF partitioning methods we discuss in the rest of this section use the METIS graph partitioning algorithm [17], so we next briefly discuss the latter. This algorithm is not directly applicable to RDF: it accepts as input a directed, unlabelled graph $G = (V, E, w)$ consisting of a finite vertex set V, an edge set $E \subseteq V \times V$, and a weighting function w that assigns to each vertex $v \in V$ a non-negative weight $w(v)$. The algorithm aims to split the set of vertices V into mutually disjoint sets $V_1, \dots, V_n$ such that

- $\sum_{v \in V_i} w(v)$ is roughly the same for each V_i, and
- the number of edges connecting vertices in distinct V_i and V_j is minimised.

The decision version of this problem is NP-hard. However, the METIS algorithm has been highly tuned to produce good solutions on graphs that are typically encountered in practice.

METIS is used in *n-hop vertex partitioning* [16]. In particular, this technique first removes all triples whose predicate is *rdf*:*type*: the objects of such triples typically have many incoming edges, which can confuse METIS. The resulting graph is converted into a directed, unlabelled graph in the obvious way, with the weight of each vertex set to one. The vertices of this graph are next partitioned using METIS into ℓ partitions $V_1, \dots, V_\ell$. Finally, each triple $\langle s, p, o \rangle \in G$ is assigned to partition element P_k such that $s \in V_k$. Intuitively, partitioning vertices using METIS minimises the number of triples spanning partitioning elements, which should in turn minimise communication during query answering. However, since not all communication is eliminated in this way, the approach further replicates data. In particular, for a chosen integer n, the approach adds to each partition element P_k all triples of G that are necessary to ensure that any query containing n 'hops' (i.e., joins whose computation requires traversing at most n connected vertices) can be evaluated purely within each server. While this is very effective at reducing communication, it can be impractical as it incurs a significant overhead in terms of data storage [16]. A variant of this approach has been used in the D-SPARQ system by Mutharaju et al. [20].

Weighted vertex partitioning with pruning [24] is also based on METIS. It also first prunes the given RDF graph by removing all triples of the form $\langle x, \mathit{rdf}{:}\mathit{type}, y \rangle$, as well all triples of the form $\langle x, p, \mathit{lit} \rangle$ where *lit* is a literal: such triples are also prone to introducing hub vertices, but they rarely participate in object–object joins, so it is beneficial to remove them before partitioning. The resulting graph is partitioned using METIS, but the weight of each vertex v is set to the number of triples in G that have v as subject. Since the METIS algorithm aims to make the sum of weights of vertices in each partition roughly the same, this effectively ensures that final partition elements are of roughly the same size. Finally, triples of the input RDF graph are assigned to partition elements as in the previous paragraph, but without any replication.

4 Community-Based RDF Graph Partitioning

We now present two variants of our approach for partitioning RDF data. To make this paper self-contained, in Section 4.1 we first summarise the well-known community detection technique based on Louvain modularity, which we use as the basis for our work. Then, in Section 4.2 we discuss how we adapt this technique to the problem of partitioning RDF data.

4.1 Louvain Algorithm for Community Detection

Graph analysis tasks often involve the concept of a *community*, which is intuitively a set of vertices that are more interconnected among themselves than to the remaining vertices of a graph. This notion can be captured using the concept of graph *modularity*, which has been formalised by Newman [22] as follows.

Consider a graph with n vertices where the weight of the connection between vertex i and vertex j is given by $A_{i,j}$. We assume that the graph is undirected, so $A_{i,j} = A_{j,i}$ for all $1 \leq i, j \leq n$. Note that the graph is allowed to contain self-loops—that is, $A_{i,i} \neq 0$ is allowed. Moreover, we assume that no weight is negative. The *degree* of a vertex i is the sum of the weights of the connections of i—that is, $k_i = \sum_{1 \leq j \leq n} A_{i,j}$. Finally, $m = 0.5 \times \sum_{1 \leq i \leq n} k_i$ is the sum of all the edge weights, where factor 0.5 takes into account that the graph is undirected so the sum includes the weight of each edge twice.

Now let C be a set of *community labels*, and let us assume that each vertex i has been assigned to some community $c_i \in C$. The *modularity* of such a community assignment is the fraction of the graph edges that connect vertices in the same community minus the expected fraction had the edges been distributed at random. This value can be computed by

$$M = \frac{1}{2m} \sum_{1 \leq i,j \leq n} \left(A_{i,j} - \frac{k_i k_j}{2m} \right) \delta(c_i, c_j), \tag{1}$$

where $\delta(c_i, c_j)$ is the Kronecker delta symbol—that is, $\delta(c_i, c_j) = 1$ if $c_i = c_j$, and $\delta(c_i, c_j) = 0$ otherwise. For unweighted graphs (i.e., if each $A_{i,j}$ is either 0 or 1), modularity is a number between -0.5 and 1. Given two community assignments, the one with the higher modularity better captures the community structure inside the graph. Thus, the problem of community detection can be reduced to the problem of finding a community assignment with maximum modularity.

The *Louvain algorithm* [5] solves this problem approximately using a hierarchical greedy method. The algorithm first assigns each vertex to its own community, after which it iteratively performs the following two phases.

In the first phase, the algorithm tries to increase modularity by moving a vertex to a community of its neighbour. The increase in modularity of moving vertex i into community c can be efficiently calculated by

$$\mathit{\Delta} M(i,c) = \left[\frac{\sum_{in} + k_{i,in}}{2m} - \left(\frac{\sum_{tot} + k_i}{2m} \right)^2 \right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m} \right)^2 - \left(\frac{k_i}{2m} \right)^2 \right]$$

where $\sum_{in}$ is the sum the weights of all the edges within community c, $\sum_{tot}$ is the sum of all the weights of the edges incident to some vertex in community c, and $k_{i,in}$ is the sum of the weights of the edges from i to all vertices in community c. Based on this idea, the Louvain algorithm considers in the first phase each vertex i and each neighbour j of i, and it calculates the increase in modularity $\Delta M(i, c_j)$ (where, as above, c_j is the current community of vertex j). Once all neighbours of i have been considered, if $\Delta M(i, c_j)$ for some neighbour j is maximal and positive, vertex i is moved into community c_j. This process is repeated as long as moving a vertex is possible.

In the second phase, the Louvain algorithm groups all vertices inside the same community and creates a new graph whose vertices are communities detected in the first phase. The weight of an edge connecting two communities c and c' is defined as the sum of the weights of all edges between vertices i and j that were assigned to c and c', respectively, in the first phase. The algorithm then applies the first phase to this new graph, and the entire process is repeated as long as applying the first phase leads to a further increase in modularity.

The notion of modularity has been extended to directed graphs [19]. One might intuitively expect directed modularity to be more suitable to RDF since RDF graphs are directed. However, note that triple patterns in queries can traverse an edge of an RDF graph in either direction. Because of that, we intuitively expect undirected modularity to be more applicable to our setting.

4.2 Our Partitioning Algorithms for RDF

In this section, we describe two variants of a new partitioning algorithm for RDF datasets that are both based on the Louvain community detection algorithm. The pseudo-code of both approaches is shown in Algorithm 1. Function COMMUNITY-RDF-PARTITIONING accepts as input an RDF graph G and the desired number of partitions ℓ, and it produces partition elements $P_1, \dots, P_\ell$ that can be used for distributed query answering. From a high-level point of view, partition elements are produced by first partitioning the resources of the input graph into communities using the Louvain algorithm, allocating each community to a server, and finally allocating each triple to the server containing the community of the subject. We next discuss these steps in more detail.

First, in line 2, the algorithm creates a pruned RDF graph G' by removing all triples of the form $\langle x, \mathit{rdf}{:}\mathit{type}, y\rangle$ or $\langle x, p, \mathit{lit}\rangle$ where *lit* is a literal. Such triples can make y and *lit* hubs, which can confuse the community detection algorithm. The same step is used in weighted vertex partitioning with pruning [24], and in *n-hop vertex partitioning* [16].

Second, in line 4, the algorithm applies the Louvain community detection algorithm to G'. Our initial experiments revealed that the Louvain algorithm, as described in Section 4.1, can create very large communities, and moreover that the number of vertices in each community can vary significantly. Assigning triples to servers based on such communities is likely to produce partition elements of very different sizes, which is undesirable. Therefore, we use a slightly modified variant of the Louvain algorithm: vertex i is moved to a community c_j of a

Algorithm 1 Community-Based RDF Partitioning

```
 1: procedure COMMUNITY-RDF-PARTITIONING($G, \ell$)
 2:     $G' := \{\langle s,p,o \rangle \in G \mid p \neq rdf{:}type$ and $o$ is not a literal$\}$
 3:     Determine $maxSize$ depending on the allocation variant
 4:     $\mathcal{S} :=$ LOUVAINCOMMUNITYDETECTION($G', maxSize$)
 5:     for each $1 \leq k \leq \ell$ do $A_k := \emptyset$
 6:     ALLOCATECOMMUNITIES
 7:     for each $1 \leq k \leq \ell$ do
 8:         $P_k := \{\langle s,p,o \rangle \in G \mid s \in A_k\}$

 9: procedure ALLOCATECOMMUNITIES-TIGHT
10:     for each $T \in \mathcal{S}$ do
11:         $\mathcal{O}_T := \{o \mid \exists s,p : \langle s,p,o \rangle \in G'$ and $s \in T\}$
12:     for each $1 \leq k \leq \ell$ do $R_k := \emptyset$
13:     while $\mathcal{S} \neq \emptyset$ do
14:         for each $T \in \mathcal{S}$ and each $1 \leq k \leq \ell$ do
15:             if $|R_k \cup T \cup \mathcal{O}_T| \leq maxSize$ then
16:                 $rank^k_T := |R_k \cap (T \cup \mathcal{O}_T)|$
17:             else
18:                 $rank^k_T := 0$
19:         Choose $T \in \mathcal{S}$ and $k$ such that $rank^k_T$ is largest
20:         $A_k := A_k \cup T$
21:         $R_k := R_k \cup T \cup \mathcal{O}_T$
22:         Remove $T$ from $\mathcal{S}$

23: procedure ALLOCATECOMMUNITIES-LOOSE
24:     for $S \in \mathcal{S}$ do
25:         Choose $k$ such that $|A_k| \leq |A_{k'}|$ for each $1 \leq k' \leq \ell$
26:         $A_k := A_k \cup S$
```

neighbour j of i only if community c_j contains at most $maxSize$ vertices after the move. As a result, our modified algorithm creates communities of at most $maxSize$ elements. The value of $maxSize$ should be selected in line 3 depending on which community allocation variant is used in line 6, and we discuss the details in Sections 4.2.1 and 4.2.2. The community detection returns a set $\mathcal{S}$ of communities—that is, each set $T \in \mathcal{S}$ contains all resources of exactly one community. Note that, for all $T, T' \in \mathcal{S}$, we have $T \cap T' = \emptyset$ whenever $T \neq T'$—that is, each community contains distinct resources.

Third, the algorithm assigns the resources in each community to a server. To this end, for each server $1 \leq k \leq \ell$, the algorithm introduces a set A_k that will contain the resources assigned to server k. All sets A_k are initially empty in line 5, and they are populated inside the allocation step in line 6. We developed two variants of community allocation, tight and loose, the details of which we discuss shortly. Either way, after the community allocation step, each resource r occurring in G in subject position can be found in exactly one set A_k.

Fourth, the algorithm allocates in line 8 each triple of G to the partition element P_k such that A_k contains the subject of the triple.

We next discuss our two variants of community allocation. Both take as input the set $\mathcal{S}$ of communities produced by the Louvain algorithm.

4.2.1 Tight Community Allocation

Tight community allocation aims to distribute communities to servers such that, after triples are allocated to servers in line 8, the resources assigned to each of the servers are highly connected. During the triple allocation process note that, even though each $T \in \mathcal{S}$ contains distinct resources, allocating T to some server k will bring into P_k resources occurring as objects of triples whose subjects are in T. To ensure high connectivity of resources assigned to any particular server, in the tight variant we set *maxSize* to N/ℓ in line 3 where N is the number of resources of G'—that is, each community is limited to at most N/ℓ resources.

The tight community allocation variant aims to ensure that the resources introduced as part of the triple allocation step occur on as few servers as possible. To achieve this, for each community $T \in \mathcal{S}$, the algorithm first constructs in line 11 the set $\mathcal{O}_T$ of resources occurring as objects in triples whose subjects are in T. Moreover, in line 12, for each server k, the algorithm introduces a set R_k that is analogous to A_k, but that will accumulate resources occurring in both subject and object position on server k. Then, the algorithm greedily assigns communities to servers using the loop in lines 13–22. In each pass through the loop, the algorithm considers each unassigned community $T \in \mathcal{S}$ and each possible target server k. For each T and k, the algorithm calculates rank_T^k in line 16 as the overlap between the resources R_k that are currently assigned to server k and the resources occurring in triples whose subjects are in T. In line 19, the algorithm chooses the community T and the server k with the best rank (with ties broken arbitrarily), and then it allocates T to k by adding T to A_k (line 20) and by recording all relevant resources in R_k (line 20).

There is one final detail: if adding T to server k would assign more than *maxSize* resources to server k, then rank_T^k is set to zero (line 18). This is key to ensuring that final partition elements are of roughly the same sizes (i.e., that no partition element is much larger than any other element).

4.2.2 Loose Community Allocation

As we have already mentioned in Section 1 and as we discuss in detail in Section 5, when an RDF graph is partitioned into highly connected communities, computational load during query answering can be focused to only some servers, which can have a negative effect on query answering times. As a possible solution to this problem, we developed the loose community allocation approach, which aims to strike a balance between reducing communication and spreading the computational workload more evenly to servers.

The loose approach is quite simple: each community $T \in \mathcal{S}$ is assigned to server k that currently has fewest resources assigned to it. Ties between servers are broken arbitrarily. To make this approach effective, we must ensure that the Louvain method detects sufficiently many communities. Thus, when loose approach is used, *maxSize* is set in line 3 to 30. We found empirically that this value seems to produce the best results across a range of datasets.

5 Experimental Evaluation

We investigated empirically how different data partitioning approaches affect query answering performance. To this end, we compared our tight (C-T) and loose (C-L) community-based partitioning variants, the weighted vertex partitioning with pruning (PRN), and subject hash partitioning (HSH); the last two techniques were described in Section 3. Our main objective was to see how different methods affect query evaluation time and the amount of communication. In this section, we first describe our experimental setting, then present the results of our experiments, and finally discuss our findings.

5.1 Experimental Setup

Test Datasets and Queries We evaluate our approaches on two well-known Semantic Web datasets. First, we used WatDiv [4], a well-known synthetic benchmark that aims to simulate real-world data and query workloads. WatDiv provides a data generator, which we used to produce an RDF graph containing 1,099,208,068 triples. Moreover, WatDiv provides 20 query templates, which have been divided into four classes: star (S), linear (L), snowflake (F), and complex (C) queries. Since star queries contain only subject–subject joins, they do not incur any communication on either of the data partitioning approaches we consider and are thus not interesting for this paper. Consequently, we considered only the 13 query templates consisting of linear, snowflake, and complex queries. These templates contain a parameter that is usually replaced with a resource from the graph; however, queries obtained in such a way tend to be quite localised and thus do not incur a lot of distributed processing. To obtain more complex and thus more interesting queries, we replaced the parameter in each of the query templates with a fresh variable.

Second, we used the widely used Lehigh University Benchmark (LUBM) [10]. We generated a dataset with 10,000, universities which resulted in 1,382,072,494 triples. The LUBM benchmark comes with 14 queries; however, most of them do not return any results when reasoning is not enabled. Thus, we used the seven queries (T1–T7) by Zeng et al. [29] that compensate for the lack of reasoning, and three new, complex cyclic queries (N1–N3) introduced by Potter et al. [24].

Test Systems We produced the Prune (PRN) and Hash (HSH) partitions of the datasets, as well as ran all queries using the distributed RDFox prototype by Potter et al. [24]. The system was written in C++. We implemented C-L and C-T in Python, making sure to generate partitions compatible with RDFox. We used a well-known library by Thomas Aynaud for Louvain community detection,[1] which we modified as discussed in Section 4 to limit the maximum community size.

Hardware We performed our experiments on ten memory-optimised r5.8xlarge servers of the Amazon Elastic Cloud. Each of the ten instances was equipped with 32 CPUs and 256 GB of RAM, and it was running Ubuntu Server 16.04 LTS (HVM). The servers were connected using 10 Gigabit Ethernet.

5.2 Evaluation Protocol and Results

In each experiment, we partitioned a dataset into ten partition elements using each of the four data partitioning techniques. We then loaded each partition into the ten servers and ran all queries while measuring wall-clock query answering times. To compensate for the variability in query answering performance, we rerun each query ten times, and we report the average time.

In addition, for each query, we also measured the number of partial binding messages that each server sends to any other server. These numbers are stable across repeated runs, and we report the total number of messages sent from all servers. These numbers do not include messages needed to communicate query answers. The number of latter messages is determined primarily by the dataset, and we expect it to be largely the same for all data partitioning strategies assuming that query answers are distributed evenly in the cluster. In this way, the number of partial binding messages reflects the amount of communication incurred by distributed join processing, which is the main objective of our study.

The average times (in seconds) for answering all queries on all datasets are shown in Table 1, and the corresponding total numbers of partial binding messages are shown in Table 2.

5.3 Discussion

As shown in Table 1, query evaluation times vary significantly across the datasets and partitioning techniques. On WatDiv, C-L and HSH are fastest on five queries each, C-T is fastest on two queries and PRN is fastest on one query. Moreover, the times are generally close across queries and partitioning styles: the biggest difference is on query F2, where C-T is 6.95 times slower than HSH. On LUBM, if we disregard trivial queries T4 to T6, PRN is fastest on four queries, and C-T is fastest on the remaining three queries. Moreover, the query times seem to differ more than on WatDiv: on query N3, PRN is 16.9 times faster than HSH; and on queries N1 and N2 the fastest partition approach is more than eight times faster than the slowest.

The picture is completely different for the numbers of partial binding messages shown in Table 2. On the WatDiv dataset, C-T requires the least number

[1] https://github.com/taynaud/python-louvain

Table 1. Query Evaluation Times in Seconds

WatDiv					**LUBM**				
	C-L	**C-T**	**PRN**	**HSH**		**C-L**	**C-T**	**PRN**	**HSH**
C1	**0.157**	0.539	0.343	0.488	T1	4.481	4.186	**3.855**	5.716
C2	**3.179**	9.359	8.520	5.817	T2	4.131	**1.798**	3.441	3.903
C3	0.789	2.082	1.389	**0.717**	T3	3.497	3.587	**3.445**	3.637
F1	1.673	5.668	3.916	**1.612**	T4	**0.003**	**0.003**	**0.003**	**0.003**
F2	0.633	3.317	1.937	**0.477**	T5	0.002	0.002	**0.001**	**0.001**
F3	**0.621**	3.298	1.856	0.622	T6	**0.002**	0.003	**0.002**	**0.002**
F4	0.530	2.332	1.414	**0.527**	T7	2.374	2.160	**1.905**	32.210
F5	**6.629**	12.315	13.029	8.858	N1	4.341	4.049	**3.255**	26.713
L1	**0.733**	2.024	1.506	0.806	N2	15.625	**12.206**	13.249	113.174
L2	1.189	0.847	**0.792**	1.124	N3	2.774	**2.148**	2.194	9.869
L3	1.174	**1.003**	1.228	1.262					
L4	0.347	1.576	0.950	**0.289**					
L5	10.495	**9.095**	9,353	9.231					

Table 2. The Number Partial Binding Messages Sent

WatDiv					**LUBM**				
	C-L	**C-T**	**PRN**	**HSH**		**C-L**	**C-T**	**PRN**	**HSH**
C1	366.68 k	**350.20 k**	391.94 k	468.03 k	**T1**	2.09 k	**1.78 k**	72.38 k	22.67 M
C2	4.72 M	**991.43 k**	4.05 M	8.97 M	**T2**	**10.00**	**10.00**	**10.00**	**10.00**
C3	**10.00**	**10.00**	**10.00**	**10.00**	**T3**	**10.00**	**10.00**	**10.00**	**10.00**
F1	4.20 M	**3.09 M**	4.28 M	7.46 M	**T4**	**0.00**	**0.00**	**0.00**	**0.00**
F2	20.53 k	**23.60 k**	32.88 k	33.54 k	**T5**	**0.00**	**0.00**	**0.00**	**0.00**
F3	244.53 k	**128.90 k**	259.89 k	474.88 k	**T6**	**0.00**	**0.00**	**0.00**	**0.00**
F4	88.58 k	**68.02 k**	98.16 k	148.10 k	**T7**	**10.00**	**10.00**	53.15 k	28.06 M
F5	5.83 M	**233.53 k**	2.86 M	8.42 M	**N1**	**15.00**	18.00	184.55 k	55.85 M
L1	1.94 M	**1.68 M**	1.86 M	2.01 M	**N2**	60.00	**58.00**	598.60 k	150.84 M
L2	1.02 k	820.00	712.00	1.02 k	**N3**	**12.00**	17.00	169.26 k	37.37 M
L3	**10.00**	**10.00**	**10.00**	**10.00**					
L4	**10.00**	**10.00**	**10.00**	**10.00**					
L5	54.78 M	42.89 M	**36.20 M**	54.93 M					
TOT	**72.19 M**	**49.45 M**	**50.03 M**	**82.91 M**		**2.20 k**	**1.91 k**	**1.08 M**	**294.79 M**

of messages for all but two queries, and C-L is slightly more efficient than C-T on one more query. The difference is particularly pronounced on query F5, where C-T outperforms HSH in terms of the message number by a factor of 36. Across all queries, HSH sends 4.2 times more messages than C-T. On the LUBM dataset, the difference is even more pronounced. On queries N1–N3, C-T and C-L are about six orders of magnitude more efficient than HSH, and by around four orders or magnitude than PRN in terms of the message number.

These results show that community partitioning is exceptionally effective at reducing communication during join processing, but unfortunately this does not seem to translate to an analogous reduction in query answering times. To understand why this is the case, we first analysed the number of partial binding messages sent by each server: while Table 2 shows only the numbers aggregated for all servers, the numbers tend to be fairly uniform across servers for HSH, but not for C-L and C-T. This suggested an unexpected side-effect of community partitioning: query processing tends to be confined to only some communities, which can lead to significant difference in server workload.

To verify this hypothesis, we measured the workload of all servers. The distributed query answering approach by Potter et al. [24] evaluates queries left-to-right matching atoms using index nested loops, so the number of times that an atom is matched at each server provides a good proxy for the server workload. We modified the RDFox prototype to capture the number of atom matches, and we rerun all queries for all datasets. The number of matches on each server does not depend on the exact run, so we run each query just once. Moreover, the total amount of work (i.e., the total number of atom matches) depends only on the input RDF graph, and not on the data partitioning technique.

Tables 3 and 4 summarise the results of this experiment for WatDiv and LUBM, respectively. Showing the matches for each of the ten servers would be inconvenient, so, for each query and partitioning technique, we show only the maximum, minimum, and median number of matches across all servers.

These results confirmed our initial hypothesis. On WatDiv, C-L and C-T incur considerable difference in server workload on most queries. This difference also exists with PRN, but it is somewhat less pronounced. Finally, there is almost no difference in the workload for HSH, which is unsurprising given that triples are spread uniformly across the cluster. In contrast, the workload was roughly the same for all data partitioning techniques on LUBM.

We interpret these results as follows. The overall query answering time seems primarily determined by (i) the distribution of the workload in the cluster, and (ii) the communication cost. Thus, if the communication cost of the two strategies is roughly the same, then the strategy with a more even workload distribution is more efficient: the total amount of work is the same, so the query answering time depends on the time of the slowest server. This seems to be happening on WatDiv: C-T requires only about 4.2 times fewer partial binding messages than HSH, but this improvement is insufficient to offset the stark difference in server workload. As a result, HSH often outperforms C-L and C-T on WatDiv in terms of query times. This is particularly pronounced on queries F1–F5 and L4, where the gap in both the performance and workload distribution seems significant. In contrast, if workload distribution is roughly the same, then the strategy with less communication is more efficient. This seems to be the case for LUBM: all four data partitioning strategies distribute the workload in roughly the same way; however, C-L and C-T incur significantly less communication than HSH on queries N1–N3, which seems to correlate with a significant boost in query performance. Although C-L and C-T incur significantly less communication compared to PRN, the absolute numbers of partial answer messages for PRN are quite small, and so the performance in these cases seems to mainly depend on other factors.

These results are quite surprising. It was commonly assumed in the literature that communication between servers is the main factor determining the performance of distributed join processing, but we show that performance can also depend on the workload distribution in the cluster. In particular, the C-T

Table 3. Atom Matches for WatDiv

	C-L			C-T		
	MAX	**MIN**	**MED**	**MAX**	**MIN**	**MED**
C1	85.24 k	47.00	59.27 k	420.55 k	2.00	58.00
C2	2.03 M	18.79 k	1.92 M	14.32 M	13.17 k	50.95 k
C3	789.06 k	573.18 k	577.96 k	871.84 k	190.71 k	597.10 k
F1	1.68 M	884.00	825.62 k	7.60 M	29.00	683.50
F2	822.87 k	4.23 k	466.56 k	4.02 M	209.00	8.16 k
F3	678.19 k	3.63 k	407.05 k	3.08 M	1.66 k	7.99 k
F4	506.83 k	15.60 k	220.86 k	2.08 M	7.54 k	15.56 k
F5	3.12 M	500.00	3.05 M	16.89 M	8.00	255.00
L1	476.34 k	433.13 k	439.02 k	930.26 k	171.98 k	339.63 k
L2	1.85 M	1.37 M	1.36 M	1.40 M	183.15 k	956.57 k
L3	476.32 k	413.02 k	416.77 k	930.23 k	171.97 k	321.37 k
L4	383.57 k	37.00	158.25 k	1.58 M	0.00	0.50
L5	10.12 M	3.12 M	5.37 M	10.63 M	17.68 k	3.85 M
	PRN			**HSH**		
	MAX	**MIN**	**MED**	**MAX**	**MIN**	**MED**
C1	271.20 k	5.86 k	13.30 k	57.38 k	54.74 k	55.79 k
C2	9.55 M	91.07 k	473.47 k	1.85 M	1.68 M	1.72 M
C3	829.22 k	175.83 k	632.51 k	600.23 k	597.52 k	599.47 k
F1	5.08 M	18.67 k	144.93 k	828.11 k	824.86 k	827.55 k
F2	2.66 M	23.04 k	102.81 k	456.38 k	453.59 k	455.10 k
F3	2.01 M	27.41 k	219.02 k	393.71 k	392.30 k	393.20 k
F4	1.39 M	15.75 k	44.02 k	229.54 k	227.40 k	228.90 k
F5	12.87 M	12.47 k	138.87 k	2.83 M	2.78 M	2.81 M
L1	505.46 k	391.51 k	446.57 k	449.26 k	440.29 k	443.58 k
L2	1.14 M	205.81 k	912.82 k	1.43 M	1.43 M	1.43 M
L3	504.70 k	302.83 k	439.79 k	426.24 k	420.57 k	422.85 k
L4	1.06 M	798.00	21.01 k	165.05 k	164.24 k	164.55 k
L5	11.63 M	21.28 k	3.10 M	6.62 M	4.12 M	5.62 M

Table 4. Atom Matches for LUBM in millions (M)

	C-L			C-T			PRN			HSH		
	MAX	**MIN**	**MED**	**MAX**	**MIN**	**MED**	**MAX**	**MIN**	**MED**	**MAX**	**MIN**	**MED**
T1	7.57	7.55	7.56	7.77	7.42	7.47	7.80	7.21	7.78	7.59	7.53	7.56
T2	2.16	2.16	2.16	2.22	2.11	2.13	2.24	2.05	2.23	2.16	2.15	2.16
T3	3.24	3.24	3.24	3.33	3.18	3.20	3.34	3.09	3.34	3.25	3.23	3.24
T7	3.08	3.07	3.07	3.11	3.05	3.06	3.18	2.92	3.17	3.08	3.06	3.07
N1	5.15	5.14	5.14	5.20	5.10	5.12	5.31	4.89	5.31	5.16	5.12	5.14
N2	26.95	26.80	26.88	27.52	26.42	26.62	27.72	25.63	27.66	26.99	26.76	26.87
N3	3.54	3.53	3.53	3.63	3.46	3.50	3.65	3.36	3.63	3.55	3.52	3.53

variant of our technique was extremely effective at reducing communication, but this was often coupled with worse query answering times. We use this as the main motivation for the C-L variant of our technique, which distributes communities across servers evenly and thus aims to strike a balance between reducing communication and distributing workload. Indeed, as one can see in Table 3, the workload is generally more evenly distributed for C-L than for C-T.

6 Conclusion

In this paper, we present two new techniques for partitioning RDF data based on graph community detection. Both techniques aim to reduce communication during distributed join processing, and we compared them empirically with several well-known data partitioning techniques. While communication is commonly assumed to be the main source of overhead, we show that the distribution of workload across servers in a cluster can also have a significant impact on the performance of query answering in distributed RDF systems.

In future, we will refine our data partitioning technique to further improve the balance between network communication and server workload and thus ensure consistent performance on a wide range of datasets and query loads. A particular challenge is to develop techniques that can partition an RDF graph in a *streaming* fashion—that is, by reading the input graph sequentially and keeping only small subsets of the graph in memory at once.

References

1. Abdelaziz, I., Harbi, R., Khayyat, Z., Kalnis, P.: A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data. PVLDB **10**(13), 2049–2060 (2017)
2. Al-Harbi, R., Abdelaziz, I., Kalnis, P., Mamoulis, N., Ebrahim, Y., Sahli, M.: Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. VLDB Journal **25**(3), 355–380 (2016)
3. Alexaki, S., Christophides, V., Karvounarakis, G., Plexousakis, D., Tolle, K.: The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In: SemWeb (2001)
4. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified Stress Testing of RDF Data Management Systems. In: ISWC. pp. 197–212 (2014)
5. Blondel, V.D., Guillaume, J.L., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. J. Stat. Mech. **2008**(10), P10008 (2008)
6. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: ISWC. pp. 54–68 (2002)
7. Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: Implementing the Semantic Web Recommendations. In: WWW. pp. 74–83 (2004)
8. Galárraga, L., Hose, K., Schenkel, R.: Partout: a distributed engine for efficient RDF processing. In: WWW. pp. 267–268 (2014)
9. Gallego, M.A., Fernández, J.D., Martìnez-Prieto, M.A., de la Fuente, P.: An Empirical Study of Real-World SPARQL Queries. CoRR **abs/1103.5043** (2011)
10. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. J. Web Semantics **3**(2), 158–182 (2005)
11. Gurajada, S., Seufert, S., Miliaraki, I., Theobald, M.: TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing. In: SIGMOD. pp. 289–300 (2014)
12. Hammoud, M., Rabbou, D.A., Nouri, R., Beheshti, S., Sakr, S.: DREAM: Distributed RDF Engine with Adaptive Query Planner and Minimal Communication. PVLDB **8**(6), 654–665 (2015)
13. Harris, S., Lamb, N., Shadbolt, N.: 4store: The Design and Implementation of a Clustered RDF Store. In: SSWS. CEUR Workshop Proceedings, vol. 517, pp. 94–109 (2009)
14. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In: ISWC. pp. 211–224 (2007)
15. Hose, K., Schenkel, R.: WARP: Workload-aware replication and partitioning for RDF. In: Workshop at ICDE. pp. 1–6 (2013)

16. Huang, J., Abadi, D.J., Ren, K.: Scalable SPARQL Querying of Large RDF Graphs. PVLDB **4**(11), 1123–1134 (2011)
17. Karypis, G., Kumar, V.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998)
18. Lee, K., Liu, L.: Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. PVLDB **6**(14), 1894–1905 (2013)
19. Leicht, E.A., Newman, M.E.J.: Community Structure in Directed Networks. Phys. Rev. Lett. **100**, 118703 (2008)
20. Mutharaju, R., Sakr, S., Sala, A., Hitzler, P.: D-sparq: distributed, scalable and efficient rdf query engine. In: ISWC (Posters & Demos). CEUR Workshop Proceedings, vol. 1035, pp. 261–264 (2013)
21. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. VLDB Journal **19**(1), 91–113 (2010)
22. Newman, M.E.J.: Analysis of weighted networks. Phys. Rev. E **70**, 056131 (2004)
23. Papailiou, N., Konstantinou, I., Tsoumakos, D., Karras, P., Koziris, N.: H_2RDF+: High-performance Distributed Joins over Large-scale RDF Graphs. In: Big Data. pp. 255–263 (2013)
24. Potter, A., Motik, B., Nenov, Y., Horrocks, I.: Dynamic Data Exchange in Distributed RDF Stores. IEEE TKDE **30**(12), 2312–2325 (2018)
25. Rohloff, K., Schantz, R.E.: Clause-Iteration with MapReduce to Scalably Query Data Graphs in the SHARD Graph-Store. In: DIDC. pp. 35–44 (2011)
26. Sun, J., Jin, Q.: Scalable rdf store based on hbase and mapreduce. In: 2010 3rd international conference on advanced computer theory and engineering (ICACTE). vol. 1, pp. V1–633. IEEE (2010)
27. Wu, B., Zhou, Y., Yuan, P., Jin, H., Liu, L.: SemStore: A Semantic-Preserving Distributed RDF Triple Store. In: CIKM. pp. 509–518 (2014)
28. Wylot, M., Hauswirth, M., Cudré-Mauroux, P., Sakr, S.: RDF data storage and query processing schemes: A survey. ACM Computing Surveys (CSUR) **51**(4), 1–36 (2018)
29. Zeng, K., Yang, J., Wang, H., hao, B., Wang, Z.: A Distributed Graph Engine for Web Scale RDF Data. PVLDB **6**(4), 265–276 (2013)