Distributed RDF Query Answering with Dynamic Data Exchange

Anthony Potter(✉), Boris Motik, Yavor Nenov, and Ian Horrocks

University of Oxford, Oxford, UK
{anthony.potter,boris.motik,yavor.nenov,ian.horrocks}@cs.ox.ac.uk

Abstract. Evaluating joins over RDF data stored in a shared-nothing server cluster is key to processing truly large RDF datasets. To the best of our knowledge, the existing approaches use a variant of the *data exchange operator* that is inserted into the query plan *statically* (i.e., at query compile time) to shuffle data between servers. We argue that such approaches often miss opportunities for local computation, and we present a novel solution to distributed query answering that consists of two main components. First, we present a query answering algorithm based on *dynamic data exchange*, which exploits data locality to maximise the amount of computation on a single server. Second, we present a partitioning algorithm for RDF data based on graph partitioning whose aim is to increase data locality. We have implemented our approach in the RDFox system, and our performance evaluation suggests that our techniques outperform the state of the art by up to an order of magnitude in terms of query evaluation times, network communication, and memory use.

1 Introduction

RDF datasets used in practice are often too large to fit on a single server. For example, in performance-critical applications, it is common to use an in-memory RDF store, but the comparatively high cost of RAM limits the capacity of such systems. Moreover, linked data applications often require integrating several large datasets that cannot be processed jointly even using disk-based systems. To attain scalability sufficient for such applications, numerous approaches for storing and querying RDF data in a shared-nothing server cluster have been developed [7–10,12,15,17–19,21,22].

Such approaches typically consist of a query answering algorithm and a data partitioning strategy, both of which must address a specific set of challenges. First, triples participating in a join may be stored on different servers, so network communication during join evaluation should be minimised. Second, to ensure that servers can progress independently of each other, one must minimise synchronisation between the servers. Third, the intermediate results produced during join evaluation often grow with the overall data size and so they may easily exceed the capacity of individual servers.

The Volcano [5] database system was one of the first to address these challenges by introducing the *data exchange operator* that encapsulates the communication

P. Groth et al. (Eds.): ISWC 2016, Part I, LNCS 9981, pp. 480–497, 2016.
DOI: 10.1007/978-3-319-46523-4_29

between query execution processes.[1] Data exchange operators are added into the query plan to move the data within the system in order to ensure that each operator in the query plan receives all the relevant data. Data exchange can be avoided if the data partitioning strategy guarantees that the triples participating in a join are colocated on the same sever. For example, exchange is not needed for subject–subject joins if all triples with the same subject are assigned to the same server. Data partitioning strategies often replicate data across servers to increase the level of guarantees they offer. As we discuss in detail in Sect. 3, all existing distributed RDF systems we are aware of can be seen as using a variant of the data exchange operator, and they aim to balance the trade-off between data replication and data exchange. Moreover, in all of the existing approaches, the decision about when and how to exchange data is made *statically*—that is, at compile time and independently from the data encountered during query evaluation. In Sect. 3 we argue that this can incur a communication cost even when the data is stored in such a way that no communication is needed in principle.

In this paper we present a new approach to query answering in distributed RDF systems. We focus here on *conjunctive* SPARQL queries (i.e., *basic graph patterns* extended with projection), but we believe that our approach can be extended to handle all SPARQL constructs. As is common in the literature, our solution also consists of a query answering algorithm and a data partitioning strategy.

In Sect. 4 we present a novel distributed query answering algorithm that employs *dynamic data exchange*: the decision when and how to exchange data is made during query processing, rather than statically at query compile time. In this way, each join between triples stored on the same server is computed on that server. Unlike in the existing solutions, local computation in our algorithm is independent of any guarantees about data partitioning, and is determined solely by the actual placement of the data. Our algorithm thus gives the data partitioning strategy more freedom regarding data placement. Moreover, our algorithm uses asynchronous communication between servers, ensuring that a server's progress in query evaluation is largely independent of that of other servers. Finally, our algorithm uses a novel technique that limits the amount of memory each server needs to store intermediate results.

In Sect. 5 we present a novel RDF data partitioning method that aims to maximise data locality by using *graph partitioning* [11]—the task of dividing the vertices of a graph into sets while satisfying certain balancing constraints and simultaneously minimising the number of edges between the sets. Graph partitioning has already been used for partitioning RDF data [7,10], but these approaches duplicate data across servers to increase the chance of local processing. In contrast, our approach does not duplicate any data at all, and it uses a special pruning step to reduce the size of the graph being partitioned. Finally, a balanced partition of vertices does not necessarily lead to a balanced partition of triples so, to achieve the latter, we use *weighted* graph partitioning.

[1] These processes may be threads within a single server or processes running on different servers, and so intra- and inter-server communication is handled using the same abstraction.

We have implemented our approach in the in-memory RDF management system RDFox [13], and have compared its performance with that of TriAD [7]—a system that was shown to outperform other state of the art distributed RDF systems on a mix of data and query loads. In Sect. 6 we present the results of our evaluation using the LUBM [6] and WatDiv [1] benchmarks. We show that our approach is competitive with TriAD in terms of query evaluation times, network communication, and memory usage; in fact, RDFox often outperforms TriAD by an order of magnitude.

2 Preliminaries

To make this paper self-contained, in this section we recapitulate certain definitions and notation. For f a function, $\mathsf{dom}(f)$ is its domain; for D a set, $f|_D$ is the restriction of f to $D \cap \mathsf{dom}(f)$; and if f' is a function such that $f(x) = f'(x)$ for each $x \in \mathsf{dom}(f) \cap \mathsf{dom}(f')$, then $f \cup f'$ is a function as well.

The vertices of RDF graphs are taken from a countable set of *resources* $\mathcal{R}$ that consists of *IRI references*, *blank nodes*, and *literals*. A *triple* has the form $\langle t_s, t_p, t_o \rangle$, where t_s, t_p, and t_o are resources. An *RDF graph* G is a finite set of triples. The *vocabulary* $\mathsf{voc}(G)$ of G is the set of all resources that occur in G; moreover, for a position $\beta \in \{s, p, o\}$, set $\mathsf{voc}_\beta(G)$ contains each resource r for which a triple $\langle t_s, t_p, t_o \rangle \in G$ exists such that $t_\beta = r$. SPARQL is an expressive language for querying RDF graphs; for example, the following SPARQL query retrieves all people that have a sister:

```
SELECT ?x WHERE { ?x rdf:type :Person . ?x :hasSister ?y }
```

SPARQL syntax is verbose, so we use a more compact notation. An *(RDF) term* is a resource or a *variable*. An *atom* (aka *triple pattern*) A is an expression of the form $\langle t_s, t_p, t_o \rangle$, where t_s, t_p, and t_o are terms; thus, each triple is an atom. For A an atom, let $\mathsf{vars}(A)$ be the set of variables occurring in A; and for $\beta \in \{s, p, o\}$, let $\mathsf{term}_\beta(A) = t_\beta$. A *conjunctive query* (CQ) has the form $Q(\vec{x}) = A_1 \wedge \cdots \wedge A_n$, where each A_i is an atom. Our definition of CQs captures *basic graph patterns* with projection in SPARQL; e.g., $Q(x) = \langle x, \mathit{rdf{:}type}, \mathit{:Person} \rangle \wedge \langle x, \mathit{:hasSister}, y \rangle$ captures the above query. A *subject-join* query is a query where the same term occurs in the subject position of all query atoms; such queries are used very frequently in practice.

Evaluation of CQs on an RDF graph produces partial mappings of variables to resources called *(variable) assignments*. For α a term or an atom and σ an assignment, $\alpha\sigma$ is the result of replacing each variable x in α with $\sigma(x)$. An assignment σ is an *answer* to a CQ $Q(\vec{x}) = A_1 \wedge \cdots \wedge A_n$ on an RDF graph G if an assignment ν exists such that $\sigma = \nu|_{\vec{x}}$, $\mathsf{dom}(\nu) = \mathsf{vars}(A_1) \cup \cdots \cup \mathsf{vars}(A_n)$, and $\{A_1\nu, \ldots, A_n\nu\} \subseteq G$ holds. SPARQL uses *bag* semantics, so $\mathsf{ans}(Q, G)$ is the multiset that contains each answer σ to Q on G with multiplicity equal to the number of such assignments ν.

Finally, we formalise the computational problems we consider in this paper. Let C be a finite set called a *cluster*; each element $k \in C$ is called a *server*.

A *partition* of an RDF graph G is a function $\mathbf{G}$ that assigns to each server $k \in C$ an RDF graph $\mathbf{G}_k$, called a *partition element*, such that $G = \bigcup_{k \in C} \mathbf{G}_k$. Partition $\mathbf{G}$ is *strict* if $\mathbf{G}_k \cap \mathbf{G}_{k'} = \emptyset$ for all $k, k' \in C$ with $k \neq k'$. A *(data) partitioning strategy* takes an RDF graph G and produces a partition $\mathbf{G}$. Given a CQ Q, a *distributed query answering algorithm* computes $\mathsf{ans}(Q, G)$ on a cluster C where each server $k \in C$ stores $\mathbf{G}_k$. An answer σ to Q on G is *local* if $k \in C$ exists such that σ is an answer to Q on $\mathbf{G}_k$.

3 Motivation and Related Work

We now illustrate the difficulties of distributed query answering and present an overview of the existing approaches.

Data Exchange Operator by Example. To make our discussion concrete, let G be the RDF graph from Fig. 1a partitioned to elements $\mathbf{G}_1$ and $\mathbf{G}_2$ by *subject hashing*; resource c is shown in grey because it occurs in both partition elements. Subject hashing is one of the simplest data partitioning strategies that assigns triple $\langle t_s, t_p, t_o \rangle$ to partition element $(h(t_s) \; mod \; 2) + 1$ for a suitable hash function h. It was initially studied in the YARS2 [9] system, but modern distributed RDF systems use more elaborate strategies.

To understand the main issue that distributed query processing must address, let $Q_1(x, y, z) = \langle x, S, y \rangle \wedge \langle y, R, z \rangle$. Answer $\sigma_1 = \{x \mapsto b, y \mapsto c, z \mapsto e\}$ spans partition elements so servers must exchange intermediate answers to compute σ_1. The Volcano system [5] proposed the solution in form of a data exchange operator that encapsulates all communication between servers in the query pipeline. In particular, variable y occurs in the second atom of Q_1 in the subject position so, for each triple $\langle t_x, S, t_y \rangle$ matching the first atom of Q_1, subject hashing ensures that any join counterparts are found on server $(h(t_y) \; mod \; 2) + 1$. Thus, we can answer Q_1 using the query plan shown in Fig. 1b, where $\otimes$ is a *data exchange operator* that (i) sends each variable assignment σ from its input to server $(h(\sigma(y)) \; mod \; 2) + 1$, and (ii) receives variable assignments sent from other servers and forwards them to the parent join operator. Thus, the rest of the query plan is completely isolated from any data exchange issues.

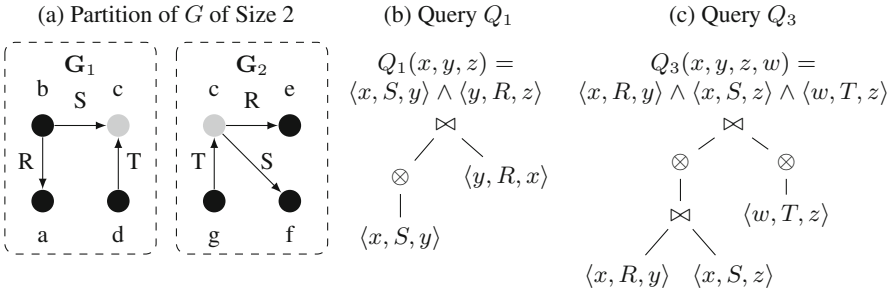

Fig. 1. Example RDF data and query plans

Guarantees about data partitioning can be used to avoid data exchange in some cases. For example, subject hashing ensures that all triples with the same subject are colocated, so subject-join queries can be evaluated without any data exchange. Thus, we can evaluate $Q_2(x, y, z) = \langle x, R, y\rangle \wedge \langle x, S, z\rangle$ independently over $\mathbf{G}_1$ and $\mathbf{G}_2$.

The decision when to introduce the data exchange operators is made *statically* (i.e., at query compile time), which can introduce unnecessary data exchange. For example, let $Q_3(x, y, z, w) = \langle x, R, y\rangle \wedge \langle x, S, z\rangle \wedge \langle w, T, z\rangle$. As in Q_2, we can evaluate the first two atoms locally. In contrast, join variable z occurs in Q_3 only in the object position so, given a value for z, subject hashing does not tell us where to find the relevant triples. Consequently, we need a query plan from Fig. 1c with two data exchange operators that hash their inputs based on the value of z, which allows us to compute answers $\sigma_2 = \{x \mapsto b, y \mapsto a, z \mapsto c, w \mapsto d\}$ and $\sigma_3 = \{x \mapsto b, y \mapsto a, z \mapsto c, w \mapsto g\}$. Note that data exchange is necessary for σ_3; however, σ_2 can be obtained by evaluating Q in $\mathbf{G}_1$, but resource c is hashed to server 2 so σ_2 is unnecessarily computed on server 2.

Data Exchange in Related Approaches. Static data exchange has been extensively used in practice. For example, the map phase in MapReduce [3] assigns to each data record a key that is used to redistribute data records in the shuffle phase; hence, distributed MapReduce-based RDF systems [10,15,17,18] can be seen as using a variant of static data exchange. Moreover, systems such as Sempala [19] implemented on top of big data databases such as Impala and Spark, as well as custom-built systems such as TriAD [7], SemStore [21], and SHAPE [12], use similar ideas. Trinity.RDF [22] uses one master and a number of worker servers: the workers first to compute candidate bindings for each variable using graph exploration, and they then send these bindings to the master to compute the final join, which is a variant of static data exchange.

Some approaches provide stronger locality guarantees by data duplication. For example, Huang et al. [10] distribute the ownership of the resources of G to partition elements using graph partitioning, and they assign each triple to the element that owns the triple's subject. Moreover, they duplicate data using *n-hop duplication*: each server containing a resource r is extended with all triples so that it contains all paths of length n from r. Thus, each query with paths of length less than n can be answered locally, and all other queries are answered using MapReduce. Duplication, however, is costly: for example, query Q_3 needs 2-hop duplication, and Huang et al. [10] show that this can increase the data in the system by a factor of 4.8; this factor is unlikely to scale linearly with the total data size since RDF graphs typically have small diameters. Furthermore, SemStore [21] partitions every rooted subgraph in the original graph. SHAPE [12] partitions subject, object or subject-object groups, extending each group with n-hop duplication and applying optimisations to reduce duplication. Trinity.RDF [22] hashes all triples on subject and object. TriAD [7] first divides resources into groups using graph partitioning, then it computes a *summary* of the input graph by merging all resources in each group, and it assigns groups

from the summary to servers by hashing on subject and object. Hashing by subject and object at most doubles the data, and so it is more likely to scale, and it also reduces data exchange: for Q_3, we can use the query plan from Fig. 1c, but without the right-hand data exchange operator.

Our Contribution. In contrast to all of these approaches that use static data exchange, in Sect. 4 we present a novel algorithm for distributed query answering that decides when and how to exchange data independently from any locality guarantees provided by data partitioning. On our example query Q_3, our algorithm computes answer σ_2 on server 1 by discovering that all the data needed for σ_2 is colocated on server 1, and it exchanges only the data necessary for σ_3. Similarly to TriAD, servers in our system exchange messages asynchronously, without coordinating progress through the query plan. This promotes concurrency, but it complicates detecting termination since an idle server can always receive a message from other servers and become busy. We solve this problem using a novel, fully decentralised termination condition. Finally, by processing messages in a specific order we limit the amount of memory needed to store messages.

Although our query answering algorithm does not rely on locality guarantees, ensuring that most answers to a query are local is critical to its efficiency. Thus, in Sect. 5 we present a novel data partitioning strategy based on graph partitioning. Our approach uses no replication, and it produces partition elements that are more balanced in sizes than those produced by related strategies based on graph partitioning [7,10,16].

4 Query Answering Algorithm

We now present our distributed query answering algorithm that uses *dynamic data exchange*. Throughout this section, we fix a cluster C of shared-nothing servers, a strict partition $\mathbf{G}$ of an RDF graph G distributed over C, and a CQ Q. Our algorithm outputs $\mathsf{ans}(G, Q)$ as pairs $\langle \sigma, m \rangle$ of assignments and multiplicities; each σ can be output several times, but the sum of all m for σ is equal to the multiplicity of σ in $\mathsf{ans}(G, Q)$.

4.1 Intuition

We evaluate Q over G using nested index loop joins: starting with an empty assignment, we recursively extend the assignment by matching the atoms of Q; we call each assignment that matches a prefix of the atoms of Q a *partial answer*. By letting all servers evaluate Q in parallel over their respective partition element, we obtain all local answers to Q without any network communication or synchronisation between the servers. To also obtain answers that are not local, whenever some server k attempts to extend a partial answer σ so that it matches some atom A of Q, the server must take into account that other servers in the cluster may contain facts matching A as well. To identify such situations, server k uses the key notion of *occurrences* that, for any resource r in $\mathbf{G}_k$, allow server

k to identify all servers that contain r. Server k can thus use the occurrences of the resources in A and σ to identify the servers that can potentially extend σ to a match for A, so server k forwards σ only to those servers; each server receiving σ then continues matching the remaining atoms of Q. The occurrences are thus used to avoid sending σ to servers that definitely cannot extend σ to an answer of Q on G, which considerably reduces communication in the cluster.

Consider again our example query Q_3 from Fig. 1c. Evaluating the first two atoms of Q_3 in $\mathbf{G}_1$ left-to-right produces a partial answer $\sigma = \{x \mapsto b, y \mapsto a, z \mapsto c\}$, so we must next evaluate $\langle w, T, z\rangle\sigma = \langle w, T, c\rangle$. By keeping the occurrences of the resources from $\mathbf{G}_1$, server 1 determines that resource c occurs in both $\mathbf{G}_1$ and $\mathbf{G}_2$ so it branches its execution: it continues evaluating the query locally and thus computes σ_2, but it also sends the partial answer σ and atom index 3 to server 2. Upon receiving this message, server 2 continues evaluating the query starting from atom 3 and produces answer σ_3.

Data exchange in our setting is thus dynamic (i.e., it is determined by the occurrences), which allows servers to always compute all local answers locally. Moreover, messages are exchanged asynchronously, without predetermined synchronisation points in the query plan: partial answers can be sent and processed as soon as they are produced, which promotes parallelisation. However, as we shall see, the asynchronous nature of our algorithm makes detecting termination nontrivial.

Our notion of occurrences exhibits two important properties. First, we require each server k to store only the occurrences for the resources that are present in $\mathbf{G}_k$. As we discuss in Sect. 4.3, this complicates determining where to forward partial answers; however, this assumption is critical for scalability because it makes the size of the occurrences at server k proportional to the size of $\mathbf{G}_k$, rather than to the size of G. Second, we track the occurrences of resources for subject, predicate, and object position independently, which we use to further limit communication. For example, if an atom A to be matched next contains a resource r in the subject position, then a partial answer is sent to server k' only if r occurs in $\mathbf{G}_{k'}$ in the subject position. As a consequence of this optimisation, if the data is partitioned such that all triples containing the same resource in the subject are colocated, subject-join queries are answered without any communication.

4.2 Setting

Before presenting our approach in detail, we discuss the assumptions we make on each server in the cluster.

We assume that each server $k \in C$ stores the partition element $\mathbf{G}_k$. For A an atom and X a set of variables, $\textsc{Evaluate}(A, \mathbf{G}_k, X)$ evaluates A in $\mathbf{G}_k$ and returns the multiset containing one occurrence of $\rho|_X$ for each assignment ρ with $\mathsf{dom}(\rho) = \mathsf{vars}(A)$ and $A\rho \in \mathbf{G}_k$. For reasons we discuss in Sect. 4.3, this multiset must be represented as a set of pairs $\langle \rho|_X, c\rangle$ where c is the multiplicity of $\rho|_X$.

In addition to $\mathbf{G}_k$, for each $\beta \in \{s, p, o\}$, server k must also store the *occurrences mapping* $\mu_{k,\beta} : \mathsf{voc}_\beta(\mathbf{G}_k) \to 2^C$ that, for each resource $r \in \mathsf{voc}_\beta(\mathbf{G}_k)$, returns the *occurrences* of r as $\mu_{k,\beta}(r) = \{k' \in C \mid r \in \mathsf{voc}_\beta(\mathbf{G}_{k'})\}$.

Finally, we assume that each server can use $\textsc{Send}(L, msg)$ to send a message msg to all servers listed in set L. Message delivery must be guaranteed: each sent message must be eventually received and processed; however, we make no assumptions about the order of message delivery, not even for messages sent from the same server. For the moment, we assume that the call always succeeds—that is, each sent message is delivered to all the servers in L in a finite amount of time. In Sect. 4.5, we show how $\textsc{Send}(L, msg)$ can be realised so that it handles the case where each server can accept only a bounded number of messages.

4.3 Computing Query Answers

The client can submit Q for processing to any sever k_c in the cluster, and so server k_c becomes the *coordinator* for Q; the client will receive all answers from server k_c. Coordinator processes Q using Algorithm 1. In line 2, the coordinator determines an efficient ordering of the query atoms; this can be done using any of the well-known query planning techniques. In line 4 the coordinator sends the reordered query to all servers; this is done synchronously so that no server starts sending partial answers to servers that have not yet accepted Q. Finally, to start the processing of Q in the cluster, in line 5 the coordinator sends to each server in the cluster the empty partial answer.

Each server $k \in C$ (including the coordinator) accepts Q for processing using procedure $\textsc{Start}(k_c, \vec{x}, A_1, \ldots, A_n)$ from Algorithm 2. The procedure initialises certain local variables, starts a number of message processing threads, and then terminates; all further processing at server k is driven by the messages that the server receives. The server processes messages in lines 15–21. The ANS messages represent partial answers produced at other servers and we discuss them shortly; moreover, the FIN messages are used to detect termination and we discuss them in Sect. 4.4. Each message is associated with a *stage* integer i that satisfies $1 \leq i \leq n + 1$.

Message $\mathsf{ANS}[i, \sigma, m, \lambda_s, \lambda_p, \lambda_o]$ informs a server that σ is a partial answer with multiplicity m. As we discuss later, the algorithm eagerly removes certain variables from partial answers to save bandwidth; thus, although σ does not necessarily cover all the variables of $A_1, \ldots, A_{i-1}$, for each σ there exists an assignment ν that coincides with σ on $\mathsf{dom}(\sigma)$ and that satisfies $\{A_1\nu, \ldots, A_{i-1}\nu\} \subseteq G$. Finally, for $\beta \in \{s, p, o\}$ a position, $\lambda_\beta : \mathcal{R} \to 2^C$ is a partial function that determines the location of certain resources in σ; we discuss the role of λ_β shortly. Such a message is forwarded in line 16 to the $\textsc{MatchAtom}$ procedure that implements index nested loop join. Line 23 determines the recursion base: if $i = n + 1$, then σ is an answer to Q on G and it is output to the client in line 24. Otherwise, in line 27 atom $A_i\sigma$ is evaluated in $\mathbf{G}_k$ and, for each match ρ, assignment σ is extended with ρ to σ' in line 28 so that the remaining atoms can be evaluated

recursively. Due to data distribution, however, recursion may also need to continue on other servers. The set L of relevant servers is identified in lines 29–35 using the following observations.

- If all atoms have been matched, then line 30 ensures that the answer σ' is forwarded to the coordinator so that it can be delivered to the client.
- Otherwise, atom $A_{i+1}\sigma'$ containing a resource r in position β cannot be matched at a server $\ell \in C$ that does not contain r in position β; hence, lines 32–35 determine the servers that contain all resources occurring in $A_{i+1}\sigma'$ at the respective positions.

After the set L of relevant servers has been computed, the computation branches to the servers in $L \setminus \{k\}$ by sending them an ANS message in line 36; and if $k \in L$, processing also continues on server k via a recursive call in line 38.

The Role of λ_s, λ_p, and λ_o. As we have already explained, each server tracks the occurrences only for the resources that it contains, which introduces a complication. For example, consider evaluating query Q_4 over the following partition:

$$Q_4(x, y, z) = \langle x, R, y\rangle \wedge \langle y, S, z\rangle \wedge \langle x, T, z\rangle$$
$$\mathbf{G}_1 = \{\langle a, R, b\rangle, \langle a, T, c\rangle\} \qquad \mathbf{G}_2 = \{\langle b, S, c\rangle\} \qquad \mathbf{G}_3 = \{\langle e, T, f\rangle\}$$

Now let $\sigma' = \{x \mapsto a, y \mapsto b\}$ be the partial answer obtained by matching the first two atoms in $\mathbf{G}_1$ and $\mathbf{G}_2$, respectively, and consider processing in line 28. Then, we have $A_{i+1}\sigma' = \langle a, T, z\rangle$, but resource a does not occur in $\mathbf{G}_2$, and so server 2 has no way of knowing where to forward σ. To remedy this, our algorithm tracks the location of resources matched thus far using partial mappings λ_s, λ_p, and λ_p. When $A_i\sigma$ is matched at server k, the server's mappings $\mu_{k,\beta}$ contain information about each resource r occurring in $A_i\sigma'$; now if r also occurs in $A_j\sigma'$ with $j > i + 1$, then the information about the location of r might be relevant when evaluating such A_j. Therefore, the algorithm records the location of r in λ'_β, which is sent along with partial answers.

Handling Projected Variables. To optimise variable projection, line 26 determines the set X of variables that are needed after A_i. Variables not occurring in X are removed from σ' in line 28 in order to reduce message size. Furthermore, $A_i\sigma$ is evaluated in line 27 using EVALUATE by grouping the resulting assignments X, which can considerably improve performance. For example, let $Q_5(x) = \langle x, R, y\rangle \wedge \langle x, S, z\rangle$, and let $\mathbf{G}_k$ contain triples $\langle a, R, b_i\rangle$ and $\langle a, S, c_j\rangle$ for $1 \leq i \leq u$ and $1 \leq j \leq v$. A naïve evaluation of the index nested loop join requires $u \cdot v$ steps, producing the same number of answer messages. In contrast, our algorithm uses $u + v$ steps: evaluating the first atom returns the pair $\langle \rho = \{x \mapsto a\}, u\rangle$ using u steps, and evaluating the second atom returns the pair $\langle \rho' = \emptyset, v\rangle$ using v steps. In addition, our algorithm sends just one answer message in this case, which is particularly important in a distributed setting.

4.4 Detecting Termination

Termination is detected by tracking the per-server completion of each atom (stage) in the query. In particular, server k can finish processing stage i if (i)

Algorithm 1. Initiating the Query at Coordinator k_c

```
1: procedure AnswerQuery(Q)
2:     Reorder the query atoms as Q(x⃗) = A_1 ∧ ··· ∧ A_n to obtain an efficient plan
3:     for k ∈ C do
4:         Call Start(k_c, x⃗, A_1, ..., A_n) on server k synchronously
5:     Send(C, ANS[1, ∅, 1, ∅, ∅, ∅])
```

it knows that all servers in C have finished processing stages up to $i-1$ by receiving the respective FIN messages, and (ii) it has processed all received messages for this stage. At this point, server k sends a FIN message to all other servers informing them that they will not receive further messages from k for stage i. To this end, each server k keeps several counters: P_i and R_i count the ANS messages for stage i that the server processed and received, respectively; and N_i counts the FIN messages that servers have sent to inform k that they have finished processing stage i. Thus, if $N_i = |C|$ holds at server k, then all other servers have finished sending all messages for all stages prior to i and so server k will not get further partial answers to process for stages up to i. If in addition $P_i = R_i$, then server k has finished stage i and it then sends his FIN message for i. Only one thread must detect this condition line 40, which is ensured by Swap(F_i, true): this operation atomically reads F_i, stores *true* into F_i, and returns the previous value of F_i. Hence, this operation returns *false* just once, in which case server k then informs in line 47 all servers (or just the coordinator if $i = n$) of this by sending a message FIN$[i, S_{i,\ell}]$, where $S_{i,\ell}$ is the number of ANS messages that server k sent to ℓ for stage i. Server ℓ processes this message in line 19 by incrementing R_i and N_i, which can cause further termination messages to be sent. Since each server sends $|C|$ messages to all other servers per stage, detecting termination requires $\Theta(n|C|^2)$ messages in total.

4.5 Dealing with Finite Resources

Nested index loop joins require just one iterator per query atom, so a query with n atoms can be answered using $O(n)$ memory; this is particularly important when servers store their data in RAM. The algorithm as presented thus far does not have this property: partial answers produced in line 36 must be stored on the sending and/or the receiving server before they are processed. In the worst case, queries can produce exponentially many answers and so the number of messages sent in line 36 can be large; consequently, the cumulative size of all messages sent to a server can exceed the server's capacity. We now describe how our query answering algorithm overcomes this drawback.

To formulate our idea abstractly, we assume that each server in the cluster contains $n+1$ finite *queues*. Moreover, function PutIntoQueue(ℓ, i, msg) instructs the message passing infrastructure to insert message msg into queue i on server ℓ. The function returns *true* if the infrastructure can guarantee that msg will be placed into the appropriate queue eventually, otherwise it returns

Algorithm 2. Processing at Server k

procedure START($k_c, \vec{x}, A_1, \ldots, A_n$)
  for $1 \le i \le n+1$ **do**
    for $\ell \in C$ **do** $S_{i,\ell} := 0$ ▷ # (partial) answers sent to server ℓ for stage i
    $P_i := 0$ ▷ # processed (partial) answers for stage i
    $R_i := (i = 1 \ ? \ 1 : 0)$ ▷ # received (partial) answers for stage i
    $N_i := (i = 1 \ ? \ |C| : 0)$ ▷ # servers finished sending messages for stage i
    $F_i := \mathit{false}$ ▷ has this server finished stage i?
  Start message processing threads that, until **exit** is called, repeatedly extract an unprocessed message msg and call PROCESSMESSAGE(msg)

procedure PROCESSMESSAGE(msg)
  if $msg = \mathsf{ANS}[i, \sigma, m, \lambda_s, \lambda_p, \lambda_o]$ **then** ▷ Partial/query answer
    MATCHATOM($i, \sigma, m, \lambda_s, \lambda_p, \lambda_o$)
    $P_i := P_i + 1$
    CHECKTERMINATION(i
  else if $msg = \mathsf{FIN}[i, m]$) **then** ▷ Atom/query termination
    $R_i := R_i + m, \qquad N_i := N_i + 1$
    CHECKTERMINATION(i)

procedure MATCHATOM($i, \sigma, m, \lambda_s, \lambda_p, \lambda_o$)
  if $i = n + 1$ **then**
    Output answer $\langle \sigma, m \rangle$ to the client
  else
    $X := \vec{x} \cup \mathsf{vars}(A_{i+1}) \cup \cdots \cup \mathsf{vars}(A_n)$
    for each $\langle \rho, h \rangle \in$ EVALUATE($A_i\sigma, \mathbf{G}_k, X$) **do**
      $\sigma' := (\sigma \cup \rho)|_X, \qquad m' := m \cdot h$
      if $i = n$ **then**
        $L := \{k_c\}, \qquad \lambda'_s := \lambda'_p := \lambda'_o := \emptyset$
      else
        $L := C$
        for $\beta \in \{s, p, o\}$ **do**
          $\lambda'_\beta := (\lambda_\beta \cup \mu_{k,\beta})|_Y$ for $Y = \mathcal{R} \cap \{\mathsf{term}_\beta(A_j\sigma') \mid i + 1 < j \le n\}$
          if $\mathsf{term}_\beta(A_{i+1}\sigma') \in \mathsf{dom}(\lambda'_\beta)$ **then** $L := L \cap \lambda'_\beta(\mathsf{term}_\beta(A_{i+1}\sigma'))$
      SEND($L \setminus \{k\}, \mathsf{ANS}[i + 1, \sigma', m', \lambda'_s, \lambda'_p, \lambda'_o]$)
      for $\ell \in L \setminus \{k\}$ **do** $S_{i+1,\ell} := S_{i+1,\ell} + 1$
      if $k \in L$ **then** MATCHATOM($i + 1, \sigma', m', \lambda'_s, \lambda'_p, \lambda'_o$)

procedure CHECKTERMINATION(i)
  if $P_i = R_i$ **and** $N_i = |C|$ **and** SWAP(F_i, true) = *false* **then**
    if $i = n + 1$ **then**
      Tell client that Q has been answered and **exit**
    else if $i = n$ **then**
      SEND($\{k_c\}, \mathsf{FIN}[i + 1, S_{i+1,k_c}]$)
      if $k \ne k_c$ **then exit**
    else
      for $\ell \in C$ **do** SEND($\{\ell\}, \mathsf{FIN}[i + 1, S_{i+1,\ell}]$)

Algorithm 3. Message Sending for Resource-Constrained Servers

```
48: procedure SEND(L, msg)
49:     i := the stage index that message msg is associated with
50:     loop
51:         for all ℓ ∈ L do
52:             if PUTINTOQUEUE(ℓ, i, msg) then L := L \ {ℓ}
53:         if L = ∅ then return
54:         If an unprocessed message for stage j with j > i exists,
                extract one such message msg and call PROCESSMESSAGE(msg)
```

false. Note that the return value of *true* does not imply that the message has actually been delivered; thus, message passing can still be asynchronous. Queues can be implemented in many different ways using common networking infrastructure. For example, TCP uses *sliding window protocol* for congestion control, so one TCP connection could provide a pair of queues. Another solution is to multiplex $n+1$ queues onto a single TCP connection. Yet another solution is to use explicit signalling: when a server sees that it is running out of queue space, it tells the sender not to send any more data until further notice.

To handle finite resources, our algorithm implements $\textsc{Send}(L, msg)$ in terms of PUTINTOQUEUE as shown in Algorithm 3: as long as some queue for stage i is blocked, server k keeps processing messages for stages larger than i. This ensures recursion depth of at most $n+1$, so each server's thread uses $O(n^2)$ memory. To see why Algorithm 2 necessarily terminates, even with queues of bounded size, we make two observations. First, processing a message for stage i only calls $\textsc{PutIntoQueue}(\ell, j, msg)$ for $j > i$, which fails only if queue j on server ℓ is full. Second, at any given point in time the cluster contains at least one highest-indexed nonempty queue across the cluster. As a result of these observations, messages from the highest-indexed nonempty queue can always be processed. Thus, although individual servers in the cluster can become blocked at different points in time, at least one server in the cluster makes progress at any given point in time, which eventually ensures termination.

The following theorem captures the properties of our algorithm, and its proof is given online at http://www.cs.ox.ac.uk/people/anthony.potter/rdfox-tr.pdf.

Theorem 1. *When Algorithm 1 is applied to a strict partition* $\mathbf{G}$ *of an RDF graph* G *distributed over a cluster* C *of servers where each server has* $n+1$ *finite message queues, the following claims hold:*

1. *the coordinator for* Q *correctly outputs* $\mathsf{ans}(Q, G)$,
2. *each server sends* $\Theta(n|C|^2)$ FIN *messages and then terminates, and*
3. *each server thread uses* $O(n^2)$ *memory.*

5 Data Partitioning Algorithm

Ensuring that computation is not passed from server to server often is key to ensuring efficiency of our approach. Therefore, in this section we present a new

data partitioning strategy based on *weighted graph partitioning* that (i) aims to maximise the number of local answers on common queries, (ii) does not duplicate triples, and (iii) produces partitions balanced in the number of triples. Throughout this section, let G be an RDF graph that we wish to partition into $|C|$ elements. Our algorithm proceeds in three steps.

First, we transform G into an undirected weighted graph (V, E, w) as follows: we define the set of vertices as $V = \mathsf{voc}_s(G)$—that is, V is the set of resources occurring in G in the subject position; we add to the set of edges E an undirected edge $\{s, o\}$ for each triple $\langle s, p, o\rangle \in G$ if $p \neq \mathit{rdf{:}type}$ and o is not a literal (e.g., a string or an integer); and we define the weight $w(r)$ of each resource $r \in V$ as the number of triples in G that contain r in the subject position. Classes and literals often occur in RDF graphs in many triples, and the presence of such hubs can easily confuse partitioners such as METIS, so we *prune* such resources from (V, E, w). As we discuss shortly, this does not affect the performance of distributed query answering for the queries commonly used in practice.

Second, we partition (V, E, w) by *weighted graph partitioning* [11]—that is, we compute a function $\pi : V \to C$ such that (i) the number of edges spanning partitions is minimised, while (ii) the sum of the weights of the vertices assigned to each partition is approximately the same for all partitions.

Third, we compute each partition element by assigning triples based on subject—that is, we assign each triple $\langle s, p, o\rangle \in G$ to partition element $\mathbf{G}_{\pi(s)}$. Note that this ensures no duplication between partition elements.

This data partitioning strategy is tailored to efficiently support common query loads. By analysing more than 3 million real-world SPARQL queries, it was shown [4] that approximately 60 % of joins are subject–subject joins, 35 % are subject–object joins, and less than 5 % are object–object joins. Now pruning classes and literals before graph partitioning makes it more likely that such resources will end up in different partitions; however, this can affect the performance only of object–object joins, which are the least common in practice. In other words, pruning does not affect the performance of 95 % of the queries occurring in practice, but it increases the chance of obtaining a good partition, as well as reduces the size of (V, E, w). Furthermore, by placing all triples with the same subject on a single server in the third step, we can answer the most common type of join without any communication; this includes subject-join queries, which are particularly important in practice. Finally, the weight $w(r)$ of each vertex r in (V, E, w) determines exactly the number of triples are added to $\mathbf{G}_{\pi(r)}$ as a consequence of assigning r to partition $\pi(r)$; since weighted graph partitioning balances the sum of the weights of vertices in each partition, this ensures that the resulting partition elements are balanced in terms of their size. As we experimentally show in Sect. 6, our partitions are indeed much more balanced than the ones produced by the existing approaches based on graph partitioning [7,10,16]. This is important because it ensures that the servers in the cluster use roughly the same amount of memory for storing their respective partition elements.

6 Evaluation

We implemented our query answering and data partitioning algorithms in our RDFox system.[2] The authors of TriAD [7] have already shown that their system outperforms Trinity.RDF [22], SHARD [17], H-RDF-3X [10], 4store [8], RDF-3X [14], BitMat [2], and MonetDB [20]; therefore, we have evaluated our approach by comparing it with TriAD only. We have conducted our experiments using the m4.2xlarge servers of the Amazon Elastic Compute Cloud.[3] Each server had 32 GB of RAM and eight virtual cores of 2.4GHz Intel Xeon E5-2676v3 CPUs.

We generated the WatDiv-10K dataset of the WatDiv [1] v0.6 benchmark, and used the 20 basic testing queries, which are divided into four groups: complex (C), snowflake (F), linear (L), and star (S) queries. We also generated the LUBM-10K dataset of the widely used LUBM [6] benchmark. Many of the LUBM queries return no answers if the dataset is not extended via reasoning, so we used the seven queries from [22] that compensate for the lack of reasoning (Q1–Q7), and we manually generated three additional complex queries (Q8–Q10). All queries that we used in the evaluation are given online at http://www.cs.ox.ac.uk/people/anthony.potter/rdfox-tr.pdf.

6.1 Evaluating Query Answering

To evaluate the effectiveness of our distributed query answering algorithm, we compared RDFox and TriAD using a cluster of ten servers. For RDFox, we partitioned the data into ten partition elements as described in Sect. 5. For TriAD, one master server partitioned the data across nine workers using TriAD's summary mode. Both systems produced the answers on one server, but without printing them. For each query, we recorded the wall-clock query time, the total amount of data sent over the network, and the maximum amount of memory used by a server for query processing.

WatDiv-10K results are summarised in Table 1. TriAD threw an exception on queries F4 and S5, which is why the respective entries are empty. Both RDFox and TriAD offer comparable performance for linear and star queries, which in both cases require little network communication. On complex queries, RDFox was faster in two out of three cases despite the fact that TriAD uses a summary graph optimisation [7] to aggressively prune the search space on complex queries. RDFox could process queries F2, F3, and F5 by up to two orders of magnitude quicker and with up to two orders of magnitude less data sent over the network. Moreover, all queries apart from C3 do not return large datasets, so the memory used for query processing was comparable.

LUBM-10K results are summarised in Table 2. RDFox was quicker than TriAD on all queries apart from Q5 and Q8, on which the two systems were roughly comparable. The difference was most pronounced on Q1, Q7, Q9, and Q10, on

[2] http://www.cs.ox.ac.uk/isg/tools/RDFox/.
[3] http://aws.amazon.com/ec2/.

Table 1. Query answering results on WatDiv-10K

		RDFox			TriAD		
Query	Answer count	Time (ms)	Network use (KB)	Max Mem. (MB)	Time (ms)	Network use (KB)	Max Mem. (MB)
C1	1,504	**148**	9,043	31	248	**3,170**	**27**
C2	288	493	**32,866**	**2**	**343**	45,520	97
C3	42,441,808	**373**	1,190	13	419	**423**	**8**
F1	324	62	4,013	1	**15**	**265**	1
F2	188	**10**	**92**	**1**	263	11,461	25
F3	865	**15**	**199**	**1**	208	337	29
F4	2,879	**25**	**471**	**1**	–	–	–
F5	65	**5**	**61**	**1**	348	29,900	76
L1	2	**3**	**29**	1	11	227	1
L2	16,132	41	**259**	1	**15**	1,106	1
L3	24	**2**	**20**	1	6	76	1
L4	5,782	14	**92**	1	**5**	299	1
L5	12,957	21	**306**	1	**17**	940	1
S1	12	**5**	**79**	1	41	142	1
S2	6,685	**12**	**183**	1	33	517	1
S3	0	25	**35**	1	**8**	91	1
S4	153	**19**	3,096	1	22	**108**	1
S5	0	**10**	**37**	**1**	–	–	–
S6	453	8	**37**	1	8	151	1
S7	0	**2**	**29**	1	3	58	1

which TriAD used significant amounts of memory. This is because TriAD evaluated queries using bushy query plans consisting of hash joins; for example, on Q10 TriAD used over 6 GB—more than half the amount needed to store the data. In contrast, RDFox uses index nested loop joins that require very little memory: at most 147 MB were used in all cases, mainly to store the messages passed between the servers. Furthermore, on most queries RDFox sent less data over the network, leading us to believe that dynamic data exchange can considerably reduce communication during query processing.

6.2 Effectiveness of Data Partitioning

To evaluate our data partitioning algorithm, we have partitioned our test data into ten elements in four different ways: with both weighted partitioning and pruning as described in Sect. 5, without pruning, without weighted partitioning, and by subject hashing. For each partitioning obtained in this way, Table 3 shows the minimum and maximum number of triples, the average number of resources

Table 2. Query answering results on LUBM-10K

		RDFox			TriAD		
Query	Answer count	Time (ms)	Network use (KB)	Max Mem. (MB)	Time (ms)	Network use (KB)	Max Mem. (MB)
Q1	2,528	**1,927**	**2,261**	**33**	13,410	197,762	1,144
Q2	10,799,863	**701**	150,565	**147**	927	**104,657**	154
Q3	0	**443**	1,809	**1**	771	**466**	708
Q4	10	**4**	**45**	1	7	115	1
Q5	10	2	**18**	1	2	63	1
Q6	125	**4**	**39**	1	85	153	1
Q7	439,994	**975**	**10,860**	**8**	7,294	29,592	844
Q8	2,528	1,771	**5,497**	**20**	**1,755**	8,154	232
Q9	4,111,592	**6,281**	**141,603**	**103**	23,711	184,661	3,501
Q10	2,225,206	**1,096**	**38,030**	**29**	33,661	111,571	6,645

Table 3. Comparing the partitioning strategies of RDFox and TriAD

	WatDiv				LUBM			
Partitioning	Min	Max	Avg. Res.	P	Min	Max	Avg. Res.	P
Weighted, pruning	103.1 M	113.0 M	20.9 M	72.1 %	126.4 M	138.2 M	32.9 M	0.3 %
Weighted, no pruning	102.1 M	113.0 M	21.6 M	72.3 %	123.6 M	139.8 M	35.7 M	13.3 %
Unweighted, no pruning	22.5 M	410.7 M	18.1 M	63.0 %	123.7 M	142.3 M	36.0 M	14.5 %
Subject hashing	109.0 M	109.3 M	24.2 M	79.2 %	133.3 M	133.7 M	52.5 M	46.8 %

Table 4. Comparing the idle memory use of RDFox and TriAD

	WatDiv			LUBM		
System	Mean (GB)	Max (GB)	Sdev (GB)	Mean (GB)	Max (GB)	Sdev (GB)
RDFox	4.39	5.42	0.54	5.49	5.61	0.15
TriAD	9.57	10.99	0.73	12.04	19.26	3.98

per partition, and the average percentage of the resources that occur in more than one partition. In all cases, subject hashing produces very balanced partitions, but the percentage of resources that occur on more than one server is large. Weighted partitioning reduces this percentage on LUBM dramatically to 0.3 %. Our partitioning is not as effective on WatDiv, but it still offers some improvement. Partitions are well balanced in all cases, apart from WatDiv with unweighted partitioning: WatDiv contains several hubs, so a balanced number of resources in partitions does not ensure a balanced number of triples.

We also compared the idle memory use (excluding dictionaries) of RDFox and TriAD's workers, in order to indirectly compare the partitioning approaches used by the two systems. Table 4 shows the minimal and maximal memory use per server after the data is loaded, as well as the standard deviation across all

servers. As one can see, RDFox uses about half of the memory of TriAD. We believe is due to the fact that our partitioning strategy does not duplicate data, whereas TriAD hashes its groups by subject and object. Furthermore, memory use per server is more balanced for RDFox, which we believe is due to weighted graph partitioning.

7 Conclusion

We have presented a new technique for query answering in distributed RDF systems based on dynamic data exchange, which ensures that all local answers to a query are computed locally and thus reduces the amount of data transferred between servers. Using index nested loops and message prioritisation, the algorithm is very memory-efficient while still guaranteeing termination. Furthermore, we have presented an algorithm for partitioning RDF data based on weighted graph partitioning. The results of our performance evaluation show that our algorithms outperform the state of the art, sometimes by orders of magnitude. In our future work, we shall focus on adapting the known query planning techniques to the distributed setting. Moreover, we shall evaluate our approach against modern big data systems such as Spark and Impala.

Acknowledgments. This work was funded by the EPSRC projects MaSI^3, DBOnto, and ED3, an EPSRC doctoral training grant, and a grant by Roke Manor Research Ltd.

References

1. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified stress testing of RDF data management systems. In: Mika, P., et al. (eds.) ISWC 2014, Part I. LNCS, vol. 8796, pp. 197–212. Springer, Heidelberg (2014)
2. Atre, M., Chaoji, V., Zaki, M.J., Hendler, J.A.: Matrix "Bit" loaded: a scalable lightweight join query processor for RDF data. In: Proceedings of WWW, pp. 41–50 (2010)
3. Dean, J., Ghemawat, S.: MapReduce: a flexible data processing tool. Commun. ACM **53**(1), 72–77 (2010)
4. Gallego, M.A., Fernández, J.D., Martínez-Prieto, M.A., de la Fuente, P.: An empirical study of real-world SPARQL queries. CoRR abs/1103.5043 (2011)
5. Graefe, G., Davison, D.L.: Encapsulation of parallelism and architecture-independence in extensible database query execution. IEEE Trans. Softw. Eng. **19**(8), 749–764 (1990)
6. Guo, Y., Pan, Z., Heflin, J.: LUBM: a benchmark for OWL knowledge base systems. J. Web Semant. **3**(2), 158–182 (2005)
7. Gurajada, S., Seufert, S., Miliaraki, I., Theobald, M.: TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing. In: Proceedings SIGMOD, pp. 289–300 (2014)
8. Harris, S., Lamb, N., Shadbolt, N.: 4store: the design and implementation of a clustered RDF store. In: Proceedings of SSWS (2009)

9. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: a federated repository for querying graph structured data from the web. In: Aberer, K., et al. (eds.) ASWC 2007 and ISWC 2007. LNCS, vol. 4825, pp. 211–224. Springer, Heidelberg (2007)
10. Huang, J., Abadi, D.J., Ren, K.: Scalable SPARQL querying of large RDF graphs. PVLDB **4**(11), 1123–1134 (2011)
11. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998)
12. Lee, K., Liu, L.: Scaling queries over big RDF graphs with semantic hash partitioning. PVLDB **6**(14), 1894–1905 (2013)
13. Nenov, Y., Piro, R., Motik, B., Horrocks, I., Wu, Z., Banerjee, J.: RDFox: a highly-scalable RDF Stored. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9367, pp. 3–20. Springer, Heidelberg (2015). doi:10.1007/978-3-319-25010-6_1
14. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. VLDB J. **19**(1), 91–113 (2010)
15. Papailiou, N., Konstantinou, I., Tsoumakos, D., Karras, P., Koziris, N.: H_2RDF+: high-performance distributed joins over large-scale RDF graphs. In: Proceedings of Big Data, pp. 255–263 (2013)
16. Potter, A., Motik, B., Horrocks, I.: Querying distributed RDF graphs: the effects of partitioning. In: Proceedings of SSWS (2014)
17. Rohloff, K., Schantz, R.E.: Clause-iteration with MapReduce to scalably query data graphs in the SHARD graph-store. In: Proceedings of DIDC, pp. 35–44 (2011)
18. Schätzle, A., Przyjaciel-Zablocki, M., Hornung, T., Lausen, G.: PigSPARQL: a SPARQL query processing baseline for big data. In: Proceedings of ISWC (Poster), pp. 241–244 (2013)
19. Schätzle, A., Przyjaciel-Zablocki, M., Neu, A., Lausen, G.: Sempala: interactive SPARQL query processing on hadoop. In: Mika, P., et al. (eds.) ISWC 2014, Part I. LNCS, vol. 8796, pp. 164–179. Springer, Heidelberg (2014)
20. Sidirourgos, L., Gonçalves, R., Kersten, M., Nes, N., Manegold, S.: Column-store support for RDF data management: not all swans are white. PVLDB **1**(2), 1553–1563 (2008)
21. Wu, B., Zhou, Y., Yuan, P., Jin, H., Liu, L.: SemStore: a semantic-preserving distributed RDF triple store. In: Proceedings of CIKM, pp. 509–518 (2014)
22. Zeng, K., Yang, J., Wang, H., Shao, B., Wang, Z.: A distributed graph engine for web scale RDF data. PVLDB **6**(4), 265–276 (2013)