Dynamic Data Exchange in Distributed RDF Stores

Anthony Potter, Boris Motik, Yavor Nenov, and Ian Horrocks

Abstract—When RDF datasets become too large to be managed by centralised systems, they are often distributed in a cluster of shared-nothing servers, and queries are answered using a distributed join algorithm. Although such solutions have been extensively studied in relational and RDF databases, we argue that existing approaches exhibit two drawbacks. First, they usually decide *statically* (i.e., at query compile time) how to shuffle the data, which can lead to missed opportunities for local computation. Second, they often materialise large intermediate relations whose size is determined by the entire dataset (and not the data stored in each server), so these relations can easily exceed the memory of individual servers. As a possible remedy, we present a novel distributed join algorithm for RDF. Our approach decides when to shuffle data *dynamically*, which ensures that query answers that can be wholly produced within a server involve only local computation. It also uses a novel flow control mechanism to ensure that every query can be answered even if each server has a bounded amount of memory that is much smaller than the intermediate relations. We complement our algorithm with a new query planning approach that balances the cost of communication against the cost of local processing at each server. Moreover, as in several existing approaches, we distribute RDF data using graph partitioning so as to maximise local computation, but we refine the partitioning algorithm to produce more balanced partitions. We show empirically that our techniques can outperform the state of the art by orders of magnitude in terms of query evaluation times, network communication, and memory use. In particular, bounding the memory use in individual servers can mean the difference between success and failure for answering queries with large answer sets.

Index Terms—RDF, distributed query answering, sharding

1 Introduction

Resource Description Framework (RDF) is a popular graph-like data model. RDF applications often need to integrate large datasets that cannot be jointly processed on a single sever. This problem is particularly acute in applications that require high levels of performance that can only be obtained by storing data in RAM. A common solution is to store the data in a cluster of shared-nothing servers. However, triples needed to answer a query can then reside on distinct servers so the servers may need to exchange partial query answers, and key challenges are to reduce communication overheads and promote parallelisation.

Based on this idea, a number of distributed RDF systems have been developed [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18]. The adopted technical approaches vary significantly, from reusing big data frameworks such as Hadoop and Spark to building specialised solutions. Abdelaziz et al. [19] recently surveyed 22 and evaluated 11 systems on a variety of data and query loads. AdPart [16] and TriAD [18] consistently offered the best performance. Both are specialised, RAM-based systems that support distributed query evaluation: AdPart uses left-deep plans consisting of hash joins and semijoins, and TriAD uses bushy query plans consisting of distributed merge and hash joins. In Section 3.2, we survey existing systems and argue that most exhibit two important drawbacks.

First, decisions as to when and how partial answers should be exchanged between the servers are made *statically* (i.e., during query planning). For example, partial answers are exchanged after each Map phase in Hadoop-based systems, and the strategies used to partition the data govern data exchange in AdPart and TriAD. We argue in Section 3.3 that such an approach can miss opportunities for local computation: servers might exchange partial answers even when all triples necessary to compute a query answer are collocated on one server. Network communication is one of the main bottlenecks in distributed RDF stores, so measures to reduce and/or eliminate communication directly affect the performance and scalability of distributed systems.

Second, servers often need to store intermediate relations (e.g., as hash tables in hash joins). The size of such relations is worst-case exponential in the size of the entire graph, rather than the data stored in the servers. Thus, increasing capacity by adding new servers can disproportionally increase the size of the intermediate relations, which can severely limit the type of queries that can be processed. In our experiments (see Section 8), TriAD and AdPart could efficiently answer queries with small result sets, but both systems failed due to memory exhaustion on nonselective chain queries returning hundreds of millions of answers.

In Sections 4 and 5 we present a new technique for distributed query answering that aims to address these two challenges. We address the first challenge by making data exchange *dynamic*: decisions about data exchange are made during query processing, rather than at compile time. The *track join* algorithm [20] has a similar aim, but it can still incur unnecessary communication in the so-called tracking phase to identify pairs of servers that need to exchange partial answers. In contrast, our approach tracks the *occurrences* of RDF resources in a lightweight data structure, and it computes an answer locally whenever all relevant triples are collocated. We address the second challenge using index nested loop joins. Similar to TriAD and AdPart, we index the data in RAM using exhaustive hash-based indexes; hence,

• *All authors are with the Department of Computer Science, University of Oxford; E-mail: first.last@cs.ox.ac.uk*

index nested loop joins can be seen as hash joins where hash tables are constructed in advance. Importantly, index sizes are determined only by the size of the data stored in each server (rather than the total data size), and memory requirements for index nested loop joins are determined only by the size of the query (and not the data) since intermediate relations are never stored explicitly. To also limit the amount of memory used for communication between the servers, we use a novel flow control strategy where a key challenge is to ensure that the system remains deadlock-free. In this way, our approach can process complex queries with large answer sets even if the memory available at each server is severely limited. Finally, query processing in our approach is fully asynchronous (i.e., no synchronisation at various points in a query plan is needed); this benefits parallelism, although it does complicate detecting termination.

We complement our approach with two additional contributions. First, in Section 6 we present a query planning approach that balances the costs of local computation and network communication. A key ingredient of our cost model is the number of messages exchanged between the servers. To estimate this number using an off-the-shelf query cardinality estimator, we reduce the problem of estimating the number of messages to the problem of estimating the cardinality of a specific query. Second, as in existing systems [9], [18], [21], we distribute the data using *graph partitioning* [22] to maximise data locality; however, we observe that existing approaches often produce unbalanced partitions, which can hinder scalability. To remedy this, in Section 7 we present a refinement based on *weighted* graph partitioning, which ensures that partitions are balanced in the numbers of triples, rather than the numbers of resources.

In Section 8 we present the results of a performance evaluation where we compared our approach with AdPart, TriAD, S2RDF [5], and PEDA [15] on the WatDiv [23] and LUBM [24] benchmarks. Our approach often outperformed the others in terms of query evaluation times, network communication, and memory usage, sometimes by orders of magnitude. Moreover, our system was the only one to successfully answer several complex chain queries that produce hundreds of millions of answers, which shows the benefits of limiting memory use in distributed join algorithms.

2 Preliminaries

We next present an overview of the definitions and notations that allow us to present our approach precisely. An (*RDF*) *resource* is an *IRI reference*, a *blank node*, or a *literal* (e.g., string or integer). A *triple* has the form $\langle t_s, t_p, t_o \rangle$ where t_s, t_p, and t_o are resources, and an *RDF graph* G is a finite set of triples. SPARQL is an expressive RDF query language. For example, the following query retrieves all people with a sister.

```
SELECT ?X WHERE { ?X rdf:type :Person . ?X :sister ?Y }
```

Since the SPARQL syntax is very verbose, we next introduce a more compact notation for queries. We denote variables by possibly indexed uppercase letters without a question mark. A *term* is a resource or a *variable*, and an *atom* (also called a *triple pattern*) has the form $\langle t_s, t_p, t_o \rangle$ where t_s, t_p, and t_o are terms (so each triple is an atom). A *conjunctive query* (CQ) has the form $Q(X_1, \ldots, X_m) = A_1 \wedge \cdots \wedge A_n$, where each X_i is an *answer variable* and each A_j is an atom. CQs capture *basic graph patterns* with projection in SPARQL. For example, the above SPARQL query is captured by $Q(X) = \langle X, \mathit{rdf{:}type}, \mathit{:Person} \rangle \wedge \langle X, \mathit{:sister}, Y \rangle$.

To define conjunctive query answers, we first define an *assignment* as a finite mapping of variables to resources. For σ an assignment and γ a term or an atom, $\gamma\sigma$ is the result of replacing in γ each occurrence of a variable X on which σ is defined with $\sigma(X)$. Assignment σ is an *answer* to a query $Q(X_1, \ldots, X_m) = A_1 \wedge \cdots \wedge A_n$ on an RDF graph G if σ is defined on $X_1, \ldots, X_m$ and it can be extended to an assignment ν defined on the variables of $A_1, \ldots, A_n$ such that $A_j\nu \in G$ for $1 \leq j \leq n$. SPARQL uses the *bag* semantics, so we define $\mathsf{ans}(Q, G)$ as the multiset containing each answer σ to Q on G with multiplicity equal to the number of such ν. We define the *cardinality* $[Q]_G$ of Q on G as the sum of the multiplicities of all answers to Q on G.

We study CQ answering when an RDF graph G is stored in a cluster C of shared-nothing servers connected by a network. Data distribution is determined by a *partition* P of G, which assigns to each server $k \in C$ an RDF graph P_k called a *partition element*. A *(data) partitioning strategy* is an algorithm that computes a partition P given G and C. Each triple must be stored on some server—that is, $G = \bigcup_{k \in C} P_k$ must hold. While partitioning strategies can store a triple on several servers in general, P is *strict* if $P_k \cap P_{k'} = \emptyset$ holds for all $k, k' \in C$ with $k \neq k'$. Given a conjunctive query Q, we consider ways of computing $\mathsf{ans}(Q, G)$ using P.

We next introduce some useful notation. The set of *positions* is defined as $\Pi = \{s, p, o\}$. For $A = \langle t_s, t_p, t_o \rangle$ an atom, $\mathsf{vars}(A)$ is the set of variables of A, and $\mathsf{term}_\pi(A) = t_\pi$ for $\pi \in \Pi$. The *vocabulary* of an RDF graph G is defined by $\mathsf{voc}_\pi(G) = \{\mathsf{term}_\pi(A) \mid A \in G\}$ for $\pi \in \Pi$, and $\mathsf{voc}(G) = \bigcup_{\pi \in \Pi} \mathsf{voc}_\pi(G)$. We often abbreviate a vector of elements $\alpha_1, \ldots, \alpha_n$ as $\boldsymbol{\alpha}$ and treat it as a set (e.g., we write $\alpha_i \in \boldsymbol{\alpha}$). For f a function, $\mathsf{dom}(f)$ is the domain of f; for D a set, $f|_D$ is the function f restricted to the set $D \cap \mathsf{dom}(f)$; and we often write f as a set of mappings $\{\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n\}$. If g is a function where $f(\alpha) = g(\alpha)$ for each $\alpha \in \mathsf{dom}(f) \cap \mathsf{dom}(g)$, then $f \cup g$ is a function whose domain is $\mathsf{dom}(f) \cup \mathsf{dom}(g)$.

By convention, in this paper we denote RDF graphs by G; partitions and their elements by P; clusters by C; servers by k and ℓ; atoms by A; positions by s, p, and o, and π; the set of positions by Π; variables by X, Y, Z, and W; terms by t; queries by Q; and resources by the remaining lowercase letters. All letters can be indexed if needed.

3 Related Work and Our Contribution

We now discuss the main difficulties in distributed query answering, survey the existing approaches and their drawbacks, and summarise our technical contributions.

3.1 Problems of Distributed Join Evaluation

Let G be the RDF graph in Figure 1a partitioned into two elements by *subject hashing*: triple $\langle t_s, t_p, t_o \rangle$ is assigned to $P_{(h(t_s) \bmod 2)+1}$, where h is a suitable hash function. Resource c is shown in grey as it occurs in both P_1 and P_2.

We say that answer $\sigma \in \mathsf{ans}(Q, G)$ is *local to server* k if $\sigma \in \mathsf{ans}(Q, P_k)$, and that σ is *local* if it is local to some

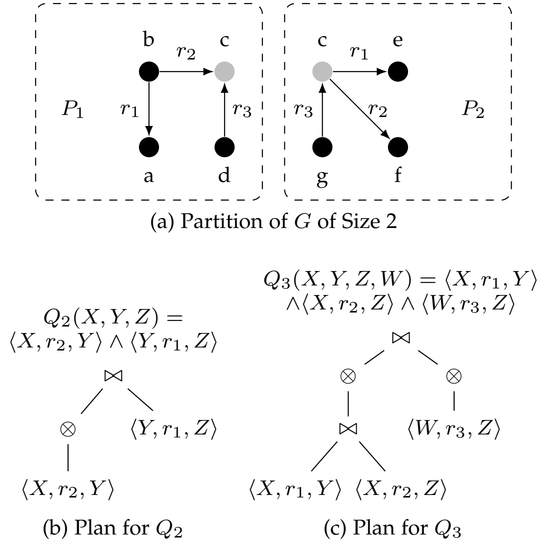

Fig. 1: Example RDF Data and Query Plans

$k \in C$. If all answers of Q are local, we can evaluate Q by taking the union of the answers obtained by evaluating Q at independently at all servers. For example, let $Q_1(X, Y, Z) = \langle X, r_1, Y \rangle \wedge \langle X, r_2, Z \rangle$. Triples are assigned to partition elements by subject hashing and the query involves only a subject–subject join, so all triples participating in a join are collocated and all query answers are local.

Now let $Q_2(X, Y, Z) = \langle X, r_2, Y \rangle \wedge \langle Y, r_1, Z \rangle$. Answer $\sigma_1 = \{X \mapsto b, Y \mapsto c, Z \mapsto e\}$ to Q_2 on G is not local: the relevant triples reside in different partition elements, so servers must communicate to compute σ_1.

Many distributed join techniques are known [25]. Query plans can be constructed using standard operators such as hash or merge join, and a new *data exchange* (or *shuffle*) *operator* [26] encapsulates all communication. Based on data partitioning, these operators are inserted into plans as necessary for the other operators to receive all relevant information. Semijoins [27], [28] can reduce communication: to join relations R_1 and R_2 residing at two servers, server 1 first sends the projection of R_1 on the join attributes to server 2, and the latter sends only the relevant part of R_2 to the former; one can sometimes only send a Bloom filter of the projected relation R_1. *Track join* [20] uses *tracking phases* to identify pairs of servers that must exchange data.

Figure 1b shows a possible evaluation plan for the example query Q_2. Variable Y occurs in the second atom of Q_2 in the subject position so, due to subject hashing, the join counterparts of each triple $\langle t_X, r_2, t_Y \rangle$ matching the first atom of Q_2 are found in $P_{(h(t_Y) \bmod 2)+1}$. The data exchange operator $\otimes$ thus sends each assignment σ from its input to server $(h(\sigma(Y)) \bmod 2) + 1$, and receives assignments sent from other servers and forwards them to the parent join operator. The plan can be evaluated in parallel on all servers. If the root join is blocking (e.g., if it is a hash join), the servers must synchronise after exchanging data.

3.2 Data Exchange in Distributed RDF Systems

Many distributed RDF systems have been developed, with significant variation in implementation techniques. We classify them broadly into the following five groups.

The first, prominent group comprises systems that store data in a distributed file system such as HDFS and use a big data framework to process queries: CliqueSquare [1], EAGRE [2], HadoopRDF [3], H_2RDF+ [10], and SHARD [4] use MapReduce on Hadoop; S2RDF [5] uses Spark SQL; S2X [6] uses the GraphX framework on Spark; PigSPARQL [7] uses Apache Pig; and Sempala [8] uses Apache Impala.

Systems in the second group compute the join on a single server after retrieving the assignments for the atoms from the cluster. In addition, 4store [12] optimises subject–subject joins; Trinity.RDF [13] prunes the retrieved assignments using graph exploration that roughly corresponds to semijoins; and YARS2 [14] uses index nested loop joins.

Systems in the third group distribute the data so that answers to common queries are local. Other queries are split into subqueries with only local answers, the subqueries are evaluated in the cluster, and their answers are combined in a 'final join' phase. This phase is realised in several ways: H-RDF-3X [9] and SHAPE [11] use MapReduce joins; SemStore [29] passes the answers between servers; and DREAM [30] and WARP [31] use a single server. These systems replicate data to increase the likelihood of queries having only local answers. H-RDF-3X first splits the graph using the METIS [22] algorithm and then applies *n-hop duplication*; thus, each query containing a term that is at most n hops away from any other term has only local answers, but already for $n = 2$ data size can increase by a factor of 4.8 [9]. SHAPE hashes resources grouped based on their URI structure and uses a form of n-hop duplication. SemStore partitions coarse-grained rooted subgraphs. DREAM replicates all data to all servers, so the main motivation is to increase parallelism. WARP replicates data based on a query workload.

The fourth group contains PECA and PEDA [15]. In both, servers partially evaluate a query (without any decomposition) using a custom form of query evaluation that can map variables to wildcards. Another custom join operator combines such answers in a 'final join' phase, which is centralised in PECA and distributed in PEDA. These custom operators differ from relational algebra, thus departing from the principles in Section 3.1. Our investigation of a closely related idea [21] suggests that query evaluation with wildcards often produces very large numbers of partial answers.

Systems in the fifth group closely follow the principles from Section 3.1 by computing and exchanging partial answers in a distributed way. AdPart [16] uses left-deep hash joins and semijoins. It hashes the data by subject, but also redistributes the data based on an observed query load. TriAD [18] uses bushy plans consisting of merge joins, hash joins, and data exchange operators. It hashes the data by subject and object, thus replicating the data twice, and it can prune irrelevant bindings using a summary graph. Both systems store data in RAM. Partout [17] partitions data based on a query workload, uses distributed query plans, and stores the data on disk using the RDF-3X [32] store.

Abdelaziz et al. [19] recently surveyed 22 of these systems and conducted a comprehensive performance evaluation on 11 of them. They observed that big data frameworks usually incur considerabe overheads (e.g., by running MapReduce jobs), and that a 'final join' phase is a common bottleneck on complex queries. AdPart and TriAD convincingly outperformed all other systems on most queries.

3.3 Drawbacks of the Existing Techniques

Although the systems from Section 3.2 can process large datasets, they still face two main scalability bottlenecks.

First, they consult the data partitioning strategy to decide *statically* (i.e., at query compile time) when to exchange data, which can be suboptimal. Consider evaluating

$$Q_3(X, Y, Z, W) = \langle X, r_1, Y \rangle \wedge \langle X, r_2, Z \rangle \wedge \langle W, r_3, Z \rangle$$

on the RDF graph G in Figure 1a. As in Q_1, the join between the first two atoms can be computed locally. Variable Z, however, occurs in Q_3 only in the object position, so subject partitioning does not tell us where to find the triples matching the third atom. This is addressed in the plan from Figure 1c: both data exchange operators distribute their inputs based on the value of Z, ensuring that nonlocal answer $\sigma_2 = \{X \mapsto b, Y \mapsto a, Z \mapsto c, W \mapsto g\}$ is computed. However, answer $\sigma_3 = \{X \mapsto b, Y \mapsto a, Z \mapsto c, W \mapsto d\}$ is local to P_1, but, since resource c is hashed to server 2, the partial answers are unnecessarily sent to server 2 to compute the join. Track join [20] aims to address such situations and it will indeed detect that answer σ_2 is local, but the tracking phases may still incur unnecessary communication.

Second, intermediate relations produced during query evaluation can be very large (i.e., worst-case exponential in the size of G). Thus, adding data to a system by adding servers can disproportionally increase the sizes of the intermediate relations that the servers must store during query evaluation (e.g., as hash tables in hash join operators). The sizes of such relations will eventually exceed the servers' capacity, which, as we show experimentally in Section 8, can prevent answering queries returning many answers. Thus, limiting the amount of memory used for query processing is critical to ensuring scalability of distributed RDF systems.

3.4 Our Contribution

To address these drawbacks, our distributed join algorithm evaluates a query Q (without any decomposition) on all servers using index nested loop joins. That is, on server k, the empty assignment is extended by recursively matching the atoms of Q in P_k; we call each match of a subset of the atoms of Q a *partial answer*. Such approaches have been extensively used in centralised systems, and their main advantage is that their memory use is linear in the number of query atoms and is independent from the size of G (or even P_k); hence, this addresses the memory issues mentioned in Section 3.3. For efficiency, the data must be indexed so that, given an atom A, we can find each substitution ρ such that $A\rho \in P_k$. When data is stored on disk, the high latency of index lookups can make index nested loop joins inefficient. However, they become feasible when data is stored in RAM: Motik et al. [33] describe a scheme for exhaustive indexing of RDF data in RAM based on hashing, so index nested loop joins can be understood as RAM-based hash joins where hash tables are built in advance. While AdPart and TriAD construct hash tables over potentially very large intermediate relations, index sizes in our approach are determined by the sizes of partition elements. Intermediate relations are not materialised, which considerably reduces memory use. This style of query evaluation incorporates sideways information passing [34], [35] as a side-effect.

Our main contribution is in adapting index nested loop joins to a distributed setting. By letting all servers evaluate Q in parallel over their partition elements, we obtain all local answers to Q without any communication or synchronisation. To also compute the remaining answers, when a server k attempts to extend a partial answer σ to an atom A of Q, in order to take into account that other servers may contain matching triples, the server consults locally stored *occurrence mappings* to identify all servers that contain all resources in $A\sigma$. Server k forwards σ to all such servers, which continue matching the remaining atoms of Q. The occurrences are thus used to ensure the algorithm's completeness, as well as to reduce communication by avoiding sending σ to servers that definitely cannot extend σ to an answer to Q. Data exchange is thus determined by the occurrences at runtime, allowing servers to compute all local answers locally.

Consider again the query Q_3 from Figure 1c. Evaluating the first two atoms of Q_3 in P_1 produces a partial answer $\sigma = \{X \mapsto b, Y \mapsto a, Z \mapsto c\}$, so we must next evaluate $\langle W, r_3, Z \rangle\sigma = \langle W, r_3, c \rangle$. By consulting the occurrence mappings, server 1 determines that resources c and r_3 occur in both P_1 and P_2 so it branches its execution: it continues evaluating the query locally and thus computes σ_3, 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 computes answer σ_2.

We track the occurrences for each position separately, which allows subject–subject joins to be answered without any communication when triples are collocated by subject. Messages are exchanged asynchronously, which benefits parallelisation, but also requires a novel distributed termination condition. Finally, storing partial answer messages is akin to storing intermediate answers in related approaches, and a novel flow control strategy ensures that each query can be answered with a limited amount of message storage. Our technique belongs to the fifth group from Section 3.2, but with a slightly different take on data exchange.

We also present two complementary techniques. First, in Section 6 we reduce the problem of estimating the number of messages sent between the servers to the problem of estimating the cardinality of a conjunctive query, thus allowing us to estimate the former using any query cardinality estimator. We also present a model of query evaluation cost and discuss query planning issues. Second, in Section 7 we refine the known RDF partitioning techniques based on graph partitioning [9], [18], [21] to produce more balanced partitions and thus promote scalability.

4 Query Answering Algorithm

We now present our approach for computing $\mathsf{ans}(G, Q)$, where an RDF graph G is distributed by a strict partition P in a cluster C. In this section we present the core concepts, and in Section 5 we present the full, optimised version.

4.1 Setting

Before presenting our algorithm, we discuss the operations that servers in the cluster must support.

Each server $k \in C$ stores the partition element P_k. For A an atom, function $\textsc{Evaluate}(A, P_k)$ should return each assignment ρ such that $\mathsf{dom}(\rho) = \mathsf{vars}(A)$ and $A\rho \in P_k$.

For each position $\pi \in \Pi$, server k must also store the *occurrence mapping* $\mu_\pi : \mathsf{voc}(G) \rightarrow 2^C$ that provides the *occurrences* $\mu_\pi(r) = \{k' \in C \mid r \in \mathsf{voc}_\pi(P_{k'})\}$ for each resource $r \in \mathsf{voc}(G)$. To present our main ideas clearly, in this section we assume that each server stores μ_π fully. However, the size of μ_π is determined by the number of resources in G, which can significantly reduce scalability; hence, in Section 5.1 we present an optimisation that allows each server k to store occurrences only for the resources in P_k.

Servers must be able to exchange messages. To process a query with n atoms, each server must provide $n+1$ independent message queues indexed from 1 to $n+1$. Queues can be of arbitrary size, as long as they can store at least one message. Then, $\text{PUTINQUEUE}(\ell, i, msg)$ attempts to insert message msg into queue i on server ℓ. The call returns $true$ if the infrastructure can guarantee that msg will be delivered eventually; otherwise, the call returns *false* (e.g., if the destination queue is full). Finally, $\text{GETFROMQUEUE}(i)$ extracts and returns a message from a queue with index i or higher, or it returns *null* if no such queue or message exists.

4.2 Computing Query Answers

Using Algorithm 1, a client submits a query Q to any sever k_c, which becomes the *coordinator* for Q. The coordinator first identifies a left-deep join plan (line 2) using an approach we discuss in Section 6, and it then distributes the plan to all servers (line 3) synchronously. Once all servers become ready, the coordinator kick-starts distributed query processing by sending to each server the empty partial answer (line 4), and then returns the control to the client. The client receives $\mathsf{ans}(Q, G)$ from k_c asynchronously (i.e., as answers are produced) as pairs $\langle \sigma, m \rangle$, where σ is an assignment and m is a positive integer called the *multiplicity*. Some σ can be output more than once, but the corresponding multiplicities will add up to the multiplicity of σ in the multiset $\mathsf{ans}(Q, G)$. The basic algorithm in this section always produces $m = 1$, but this changes with the optimisations in Section 5.

Each server $k \in C$ (including the coordinator) accepts Q for processing using the procedure START in Algorithm 2. The procedure initialises several variables (lines 6–11), prepares data structures used in the optimisations we discuss in Section 5 (line 12), starts a number of message processing threads (line 13), and then terminates. All further processing at server k is driven by message processing treads, which repeatedly call $\text{GETFROMQUEUE}(1)$ and pass the result to PROCESSMESSAGE. There are three types of message.

- $\mathsf{PAR}[i, \sigma, m, \boldsymbol{\lambda}]$ says that σ is a partial answer up to the i-th atom of Q, with integer multiplicity m and partial occurrence mappings $\boldsymbol{\lambda}$ (cf. Section 5).
- $\mathsf{FIN}[i, S]$ is used to detect termination (cf. Section 4.3).
- $\mathsf{ANS}[\sigma, m]$ says that σ is an answer to the query Q with multiplicity m.

Each message has an integer *stage*: the stage of PAR and FIN messages is i, and the stage of ANS messages is $n+1$.

A partial answer message $\mathsf{PAR}[i, \sigma, m, \boldsymbol{\lambda}]$ is passed to the MATCHATOM procedure, but parameters m and $\boldsymbol{\lambda}$ are only used by the optimisations described in Section 5. The procedure computes recursively all extensions of the partial answer σ to the n atoms $A_i, \ldots, A_n$ of Q using index nested loop joins. It evaluates $A_i\sigma$ in P_k (line 44) and, for each match ρ, it extends σ with ρ to an assignment σ' covering all atoms of Q up to A_i. The recursion base is given by $i = n$ (line 46): σ' is then an answer to Q on G, so the projection $\sigma'|_{\boldsymbol{X}}$ of σ' to the answer variables $\boldsymbol{X}$ is output to the client if server k is the coordinator (line 48); otherwise, $\sigma'|_{\boldsymbol{X}}$ is sent to the coordinator (line 50). If $i \neq n$, atom $A_{i+1}\sigma'$ must be matched recursively, and this might be done on servers other than k due to data distribution. Function OCCURS identifies the set L of relevant servers (line 52): server ℓ is included in L only if, for each position $\pi \in \Pi$ such that $t = \mathsf{term}_\pi(A_{i+1}\sigma')$ is a resource, we have $\ell \in \mu_\pi(t)$—that is, resource t occurs in partition element P_ℓ in position π. Computation is then branched to each server in $L \setminus \{k\}$ via a $\mathsf{PAR}[i+1, \sigma', 1, \boldsymbol{\emptyset}]$ message (line 53); moreover, matching also proceeds on server k via a recursive call if $k \in L$ (line 54). Note that, if triples are partitioned by subject, then for each resource r, set $\mu_s(r)$ contains at most one server, so subject–subject joins incur no communication.

Algorithm 1 Initiating the Query at Coordinator k_c

```
1: procedure ANSWERQUERY(Q, X)
2:     Reorder the atoms of Q as A = A_1 ∧ ··· ∧ A_n
           to obtain an efficient evaluation plan
3:     for k ∈ C do Call START(k_c, X, A) on server k synchronously
4:     SEND(C, PAR[1, ∅, 1, ∅])
```

Algorithm 2 Processing at Server k

```
5: procedure START(k_c, X, A)
6:     for 1 ≤ i ≤ n + 1 do
7:         for ℓ ∈ C do S_{i,ℓ} := 0
8:         D_i := 0
9:         R_i := (i = 1 ? 1 : 0)
10:        N_i := (i = 1 ? |C| : 0)
11:        F_i := false
12:    PREPAREOPTIMISATIONS(X, A)
13:    Start message processing threads

14: procedure PROCESSMESSAGE(msg)
15:    if msg = PAR[i, σ, m, λ] then
16:        MATCHATOM(i, σ, m, λ)
17:        D_i := D_i + 1
18:        CHECKTERMINATION(i)
19:    else if msg = ANS[σ, m] then
20:        Output answer ⟨σ, m⟩ to the client
21:        D_{n+1} := D_{n+1} + 1
22:        CHECKTERMINATION(n + 1)
23:    else if msg = FIN[i, S] then
24:        N_i := N_i + 1
25:        R_i := R_i + S
26:        CHECKTERMINATION(i)

27: procedure CHECKTERMINATION(i)
28:    if D_i = R_i ∧ N_i = |C| ∧ SWAP(F_i, true) = false then
29:        if i = n + 1 then
30:            Tell client that Q has been answered and exit
31:        else if i = n then
32:            SEND({k_c}, FIN[n + 1, S_{n+1,k_c}])
33:            if k ≠ k_c then exit
34:        else
35:            for ℓ ∈ C do SEND({ℓ}, FIN[i + 1, S_{i+1,ℓ}])

36: procedure SEND(L, msg)
37:    i := the stage index of msg
38:    for ℓ ∈ L do
39:        while PUTINTOQUEUE(ℓ, i, msg) = false do
40:            msg' := GETFROMQUEUE(i + 1)
41:            if msg' ≠ null then PROCESSMESSAGE(msg)
42:        if msg is a PAR or an ANS message then S_{i,ℓ} := S_{i,ℓ} + 1
```

Algorithm 3 Simplified Atom Matching at Server k

```
43: procedure MatchAtom(i, σ)
44:     for ρ ∈ Evaluate(A_i σ, P_k) do
45:         σ' := σ ∪ ρ
46:         if i = n then
47:             if k = k_c then
48:                 Output answer ⟨σ'|_X, 1⟩ to the client
49:             else
50:                 Send({k_c}, ANS[σ'|_X, 1])
51:         else
52:             L := Occurs(A_{i+1} σ')
53:             Send(L \ {k}, PAR[i + 1, σ', 1, ∅])
54:             if k ∈ L then MatchAtom(i + 1, σ')
55: function Occurs(A)
56:     L := C
57:     for π ∈ Π and t = term_π(A) do
58:         if t ∈ dom(μ_π) then L := L ∩ μ_π(t)
59:     return L
```

4.3 Detecting Termination

Eliminating global coordination benefits parallelisation, but complicates termination: even if a server is (temporarily) idle, it could be reactivated by receiving a partial answer message. Our query answering algorithm addresses this issue by a novel asynchronous termination condition.

Server k has finished processing a stage i when (i) it has processed all received partial answers for stage i, and (ii) all servers have finished processing stages up to $i-1$ and will thus not send to server k any more partial answers for stage i. To detect these two conditions, when server ℓ finishes processing a stage $i-1$, it sends to server k a $\mathsf{FIN}[i, S]$ message, where S is the total number of partial answers for stage i that server ℓ sent to server k. Server k processes this message by incrementing counter N_i and adding S to counter R_i (lines 24–25); thus, N_i counts the servers that informed server k of finishing stage $i-1$, and R_i counts the messages that server k will receive for stage i. Thus, when $N_i = |C|$ holds at server k, all other servers have processing stages up to $i-1$. Moreover, whenever server k is done processing a partial answer message for stage i, it increments a counter D_i (line 17 or 21). If $D_i = R_i$ holds as well, then server k has finished stage i, and it sends a FIN message to the coordinator if $i = n$ (line 32) or to each server ℓ otherwise (line 35); these FIN messages include the number of sent partial answers, so these are also counted (line 42). Operation $\textsc{Swap}(F_i, \mathit{true})$ (line 28) atomically stores *true* into F_i and returns the previous value of F_i; thus, *false* is returned just once, so only one thread of server k can send a termination message for a stage. Each server sends a message to all other servers per stage, so detecting termination requires $\Theta(n|C|^2)$ messages.

4.4 Memory and Termination Guarantees

A nested index loop join in a centralised system requires one iterator per query atom. A query with n atoms thus needs $O(n)$ memory, which is independent of the data size and is particularly important in RAM-based RDF stores. However, our distributed algorithm does not exhibit this property: partial answers produced in line 53 must be stored in queues of the receiving server, which is tantamount to storing potentially large intermediate relations. If G is of size m, a query with n atoms can produce m^n answers in the worst case, so the cumulative size of all messages sent to a server can easily exceed the server's capacity. Flow control techniques such as the sliding window protocol can limit the space required, but can lead to deadlocks where two servers wait indefinitely for the other server's queues to free up.

We address this problem by a sophisticated flow control mechanism. As mentioned in Section 4.1, for a query with n atoms, each server must provide $n+1$ distinct message queues, one per stage. Function $\textsc{Send}$ delivers each message msg to the queue determined by the stage of msg, and it will try doing so until the target queue becomes free (lines 39–41); hence, this is the only place in our algorithm where synchronisation between servers may be necessary. To avoid deadlocks, after each unsuccessful delivery attempt, the function will process a message for stages $i+1$ to n, if any exist. The recursion depth of each thread is thus $O(n)$, and each level requires $O(n)$ memory, so we need at most $O(n^2)$ memory per thread: more memory can benefit parallelism, but is not strictly needed to process the query.

We next argue that this solution avoids deadlocks regardless of the message queue sizes. First, processing a message for stage i can produce messages only for stages $j > i$. Second, at any given point in time, the cluster contains at least one highest-indexed nonempty queue across the cluster, and messages from this 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.

5 Optimisations

We now extend the basic dynamic data exchange approach presented in Section 4 with several optimisations that avoid unnecessary computation, and reduce storage requirements and network communication. Algorithm 4 shows the optimised version of the $\textsc{MatchAtom}$ procedure, as well as the $\textsc{PrepareOptimisations}$ procedure that precomputes data structures used during optimised atom matching. Thus, our optimised approach consists of Algorithms 1, 2, and 4.

5.1 Partial Occurrences

In Section 4.1 we assumed that server k stores occurrence mappings covering all resources of G. In such a case, the size of G, rather than the size of P_k, determines the memory requirements of server k, which can reduce scalability.

To address this, we let server k store only the mapping $\mu_{k,\pi} = \mu_\pi|_{\mathsf{voc}(P_k)}$ for each position $\pi \in \Pi$. As $\mu_{k,\pi}$ covers only the resources contained in $\mathsf{voc}(P_k)$, its size is determined by the size of P_k. Note that we require $\mu_{k,\pi}(r)$ to be defined for a resource $r \in \mathsf{voc}(P_k)$ even if $r \notin \mathsf{voc}_\pi(P_k)$.

Storing $\mu_{k,\pi}$ instead of μ_π at server k introduces a problem, which we illustrate using the following query Q_4 and partition elements P_1 and P_2.

$$Q_4(x) = \langle X, r_1, Y\rangle \wedge \langle Y, r_2, Z\rangle \wedge \langle X, r_2, W\rangle$$
$$P_1 = \{\langle a, r_1, b\rangle, \langle a, r_2, d\rangle\} \qquad P_2 = \{\langle b, r_2, c\rangle\}$$

Matching the first two atoms in P_1 and P_2 produces a partial answer $\sigma'' = \{X \mapsto a, Y \mapsto b, Z \mapsto c\}$, which, applied to the third atom, produces $\langle a, r_2, W\rangle$. Hence, the occurrences

Algorithm 4 Optimised Atom Matching at Server k

```
60: procedure PREPAREOPTIMISATIONS(X, A)
61:     for X ∈ vars(A) do
62:         I_X := min{j | X ∈ vars(A_j)}
63:         U_X := {π ∈ Π | ∃i > I_X such that X = term_π(A_i)}
64:     for 1 ≤ i ≤ n do
65:         B_i := max[{0} ∪ {I_X | X ∈ vars(A_i) and I_X < i}]
66:         V_i := X ∪ vars(A_{i+1}) ∪ ··· ∪ vars(A_n)
67: function MATCHATOM(i, σ, m, λ)
68:     E := EVALUATE(A_iσ, P_k, V_i)
69:     if E = ∅ ∧ B_i ≠ i − 1 ∧ OCCURS(A_iσ, λ) ⊆ {k} then return B_i
70:     for ⟨ρ, c⟩ ∈ E such that CANPRUNE(ρ) = false do
71:         σ' := (σ ∪ ρ)|_{V_i}
72:         m' := m · c
73:         if i = n then
74:             if k = k_c then
75:                 Output answer ⟨σ', m'⟩ to the client
76:             else
77:                 SEND({k_c}, ANS[σ', m'])
78:         else
79:             L := OCCURS(A_{i+1}σ', λ)
80:             for π ∈ Π do
81:                 λ'_π := (λ_π ∪ μ_{k,π})|_T where
                        T := {term_π(A_jσ') | i + 1 < j ≤ n}
82:             SEND(L \ {k}, PAR[i + 1, σ', m', λ'])
83:             if k ∈ L then
84:                 j := MATCHATOM(i + 1, σ', m', λ)
85:                 if j < i then return j
86:     return i − 1
87: function CANPRUNE(ρ)
88:     for x ∈ dom(ρ) and π ∈ U_X do
89:         if μ_{k,π}(ρ(x)) = ∅ then return true
90:     return false
91: function OCCURS(A, λ)
92:     L := C
93:     for π ∈ Π and t = term_π(A) do
94:         if t ∈ dom(λ_π) then L := L ∩ λ_π(t)
95:         else if t ∈ dom(μ_{k,π}) then L := L ∩ μ_{k,π}(t)
96:     return L
```

of a and r_2 determine how to proceed, but resource a does not occur in P_2 so $\mu_{2,s}(a)$ is undefined. The only solution is to send σ to all servers containing r_2, which is inefficient.

To address this, each message $\mathsf{PAR}[i, \sigma, m, \boldsymbol{\lambda}]$ includes a vector $\boldsymbol{\lambda} = \lambda_s, \lambda_p, \lambda_o$ of *partial occurrence mappings*, where each λ_π records the occurrences of the resources appearing in $A_{i+1}\sigma, \dots, A_n\sigma$ at position π. Upon receiving this message, when server k extends σ to σ' by matching $A_i\sigma$ in P_k, the occurrences of the resources in $A_{i+1}\sigma'$ determine where to send σ', and the occurrences of resources in $A_{i+2}\sigma', \dots, A_n\sigma'$ determine where to send any subsequent messages. These observations are used as follows.

First, $\textsc{Occurs}(A_{i+1}\sigma', \boldsymbol{\lambda})$ in line 79 determines the set L of servers that must receive the partial answer σ' by combining the occurrences $\boldsymbol{\lambda}$ sent to server k with the occurrences $\mu_{k,\pi}$ stored locally. Note that if $A_{i+1}\sigma'$ contains a resource r at position π on which neither $\lambda_\pi(r)$ nor $\mu_{k,\pi}(r)$ is defined, set L can contain a server ℓ such that $\ell \notin \mu_\pi(r)$, but this can only cause superfluous PAR messages.

Second, for each position $\pi \in \Pi$, the received partial occurrences λ_π are combined with the locally stored occurrences $\mu_{k,\pi}$ and the result is projected to the set of terms T, which is constructed to contain all terms occurring in atoms $A_{i+2}\sigma', \dots, A_n\sigma'$ at position π (line 81). The resulting partial occurrences $\lambda'_s, \lambda'_p, \lambda'_o$ are sent to the servers matching atom $i + 1$ (line 82). The recursive call in line 84 uses $\boldsymbol{\lambda}$ instead of $\boldsymbol{\lambda}'$ since this is necessary for the correct operation of backjumping in line 69, as described in Section 5.3.

Consider again our example query Q_4. Server 1 matches the first atom of Q_4 in P_1 producing the partial answer $\sigma' = \{X \mapsto a, Y \mapsto b\}$. The server uses the occurrences of $\langle Y, r_2, Z\rangle\sigma' = \langle b, r_2, Z\rangle$ to identify $L = \{2\}$ as the set of servers that should receive σ'. Thus, server 1 sends σ' to server 2 together with partial occurrences $\lambda'_s = \{\, a \mapsto \{1\} \,\}$, $\lambda'_p = \{\, r_2 \mapsto \{1, 2\} \,\}$, and $\lambda'_o = \emptyset$. Note that λ'_s is not defined on b: the rest of the query $\langle X, r_2, W\rangle\sigma' = \langle a, r_2, W\rangle$ does not contain b so its occurrences are not relevant for future messages. Server 2 receives σ' and $\boldsymbol{\lambda}'$, extends σ' to σ'' as outlined earlier, and uses $\lambda'_s(a) = \{1\}$ to determine that the third atom of Q_4 can be matched only on server 1.

5.2 Projecting Variables Eagerly

Projecting variables eagerly can reduce local processing and network communication, as the following example shows.

$$Q_5(X) = \langle X, r_2, Y\rangle \wedge \langle X, r_3, Z\rangle$$
$$P_1 = \{\langle a, r_2, b_i\rangle \mid 1 \le i \le u\} \quad P_2 = \{\langle a, r_3, c_j\rangle \mid 1 \le j \le v\}$$

SPARQL uses bag semantics, so query Q_5 has just one answer $\sigma = \{X \mapsto a\}$ with multiplicity $u \cdot v$, and our unoptimised algorithm uses $u \cdot v$ steps and u messages to compute it. We can, however, evaluate $\langle X, r_2, Y\rangle$ in P_1 and project Y, and thus obtain just one partial answer $\sigma_1 = \{X \mapsto a\}$ with multiplicity u. We send σ_1 and u to P_2, and then we evaluate $\langle a, r_3, Z\rangle$ and project Z, and thus obtain just one match $\rho = \emptyset$ with multiplicity v. Finally, we combine σ_1 and u with ρ and v into the answer $\sigma = \sigma_1 \cup \rho$ with multiplicity $u \cdot v$. We thus need only $u + v$ steps and just one message.

Grouping assignments after variable projection can be costly, so we only project variables when matching single atoms. In particular, we match an atom A in a partition element P_k using function $\textsc{Evaluate}(A, P_k, \boldsymbol{V})$, where set $\boldsymbol{V}$ contains the relevant variables of A. The function returns a set of pairs $\langle \rho, c\rangle$ where ρ is an assignment with $\mathsf{dom}(\rho) = \boldsymbol{V} \cap \mathsf{vars}(A)$ and c is the (positive) number of distinct extensions ρ' of ρ such that $A\rho' \in P_k$. In our example, $\textsc{Evaluate}(\langle X, r_1, Y\rangle, P_1, \{X\})$ returns just one pair $\langle \{X \mapsto a\}, u\rangle$. This operation can be easily implemented in RDF stores with hierarchical indexes [32], [33], [36].

For each atom A_i, set $\boldsymbol{V}_i$ is computed (line 66) to contain the variables relevant to the rest of the query, and it is used to project unnecessary variables from the matches of $A_i\sigma$ (line 68) and the partial answer σ' (line 71). Also, each partial answer message $\mathsf{PAR}[i, \sigma, m, \boldsymbol{\lambda}]$ includes the multiplicity m of σ, which is combined (line 72) with the multiplicity c of a match ρ of $A_i\sigma$ to obtain the multiplicity m' of σ'.

5.3 Backjumping

Index nested loop joins can sometimes perform unnecessary work, as the following example shows.

$$Q_6(X, Y_1, Y_2, Y_3) = \langle X, r_1, Y_1\rangle \wedge \langle X, r_2, Y_2\rangle \wedge \langle X, r_3, Y_3\rangle$$
$$P_1 = \{\langle a, r_1, b\rangle, \langle a, r_2, c_1\rangle, \dots, \langle a, r_2, c_k\rangle, \langle d, r_1, b\rangle, \langle d, r_2, c_1\rangle, \dots, \langle d, r_2, c_k\rangle\} \quad P_2 = \{\langle d, r_3, e\rangle\}$$

Let $\sigma_i = \{X \mapsto a, Y_1 \mapsto b, Y_2 \mapsto c_i\}$. Matching the first two atoms of Q_6 produces partial answer σ_1. Since $\langle X, r_3, Y_3\rangle\sigma_1$

cannot be matched, the unoptimised algorithm tries σ_2. However, σ_2 differs from σ_1 only on Y_2, which does not occur in $\langle X, r_3, Y_3 \rangle$, so $\langle X, r_3, Y_3 \rangle\sigma_2$ still cannot be matched. Thus, the unoptimised algorithm explores all σ_i in vain.

This can be avoided by observing that, when matching $\langle X, r_3, Y_3 \rangle\sigma_1$ fails, we must change the value of X to have a chance of a match; thus, we can *backjump* to the first atom and continue evaluation there. To generalise this idea, our algorithm computes, for each variable X, the index I_X of the atom in the left-to-right plan that produces a binding for X (line 62). Moreover, for each atom A_i, it also computes the index B_i of the closest preceding atom that binds a variable in A_i (line 65). Then, if $A_i\sigma$ cannot be matched (line 69), function MATCHATOM returns the index of the atom where query evaluation should continue (line 69), which is used to unwind the recursive calls to the desired stage (line 85).

Our example also illustrates a subtlety that arises in a distributed setting. Let $\nu_i = \{X \mapsto d, Y_1 \mapsto b, Y_2 \mapsto c_i\}$. Server 1 computes ν_1 and sends it to server 2, which computes $\nu_1 \cup \{Y_3 \mapsto e\}$. Now ν_1 cannot be extended on server 1, but backjumping to the first atom misses ν_j for $j > 1$, each leading to an answer $\nu_j \cup \{Y_3 \mapsto e\}$. Intuitively, backjumping is possible only if an atom cannot be matched in the entire graph, and not just on the current server. Backjumping is thus initiated only if OCCURS$(A_i\sigma, \boldsymbol{\lambda})$ determines that no server other than the current one can match $A_i\sigma$ (line 69); this is why the recursive call to MATCHATOM in line 84 uses $\boldsymbol{\lambda}$ instead of $\boldsymbol{\lambda}'$. This check can be expensive, so it is performed only if backjumping can have an effect (i.e., if $B_i \neq i - 1$). With this change, our algorithm backjumps after considering σ_1 (resource a occurs only on server 1), but not after ν_1 (resource d also occurs on server 2).

5.4 Early Pruning

Occurrences can be used to further reduce the work during matching, as the following example shows.

$$Q_7(x) = \langle X, r_1, Y \rangle \wedge \langle Y, r_2, Z \rangle \wedge \langle Z, r_3, X \rangle$$

Assume now that server k matches $\langle X, r_1, Y \rangle$ via assignment ρ. Since X occurs in $\langle Z, r_3, X \rangle$ in object position, $\rho(X)$ must occur in G in object position for ρ to be extended to an answer. Also, $\langle X, r_1, Y \rangle\rho \in P_k$ ensures $\rho(X) \in \mathsf{voc}(P_k)$, so $\mu_{k,o}(\rho(X))$ is defined; hence, server k can check whether $\rho(X)$ occurs on some partition element in object position.

Our algorithm thus computes, for each variable X, the set of positions U_X at which X occurs in the query (line 63). Then, a match ρ of $A_i\sigma$ is skipped (line 70) if a variable X matched by ρ and a position $\pi \in U_X$ exist such that $\rho(X)$ does not occur in G in position π (lines 88–90).

5.5 Correctness

Theorem 1 captures the properties of our approach, and it is proved in the appendix available from the IEEE Web site.

Theorem 1. *If Algorithms 1, 2, and 4 are applied to a strict partition P of an RDF graph G distributed over a cluster C of servers where each server has $n + 1$ finite message queues,*

1) *servers terminate after sending $\Theta(n|C|^2)$ FIN messages,*
2) *the coordinator for Q correctly outputs $\mathsf{ans}(Q, G)$, and*
3) *at most $O(n^2)$ memory is needed per server thread.*

6 QUERY PLANNING

RDF stores typically use a *query planner* to identify a query plan that reduces the time/space required for query evaluation. A typical planner comprises (i) a *query cardinality estimator*, which uses statistics about the data to estimate the number of answers to the query or its subparts, (ii) a *cost model*, which combines these estimates into a numeric measure of the time/space needed, and (iii) a *query planning algorithm*, which identifies (or approximates) a plan with the least cost. We can apply these principles in our setting, but distributed processing raises several important issues.

First, the cost of processing at a server is determined by the number of messages that the server produces as well as the number of partial answers that it considers. Thus, in Section 6.1 we present a novel technique that can estimate these numbers by reusing any query cardinality estimator.

Second, when combining the costs of processing and communication of all servers, we must take into account that servers (and sometimes even communication) operate in parallel. Thus, in Section 6.2 we discuss how to define the plan cost so that it approximates the system's performance.

For the rest of this section, we fix a query $Q(\boldsymbol{X})$ to be evaluated over a strict partition P of an RDF graph G. Since we use left-deep index nested loop joins, a plan for Q is a reordering $\boldsymbol{A} = A_1, \ldots, A_n$ of the atoms of Q. Also, we consider the MATCHATOM variant from Algorithm 3: capturing the optimisations from Section 5 seems very challenging, but doing so would not significantly affect plan quality.

6.1 Counting Partial Answers and Sent Messages

Consider calling MATCHATOM for stage i of plan $\boldsymbol{A}$ on server k. We present a query $\mathsf{P}_{\boldsymbol{A},i,k}$ that, on an RDF graph G' obtained by extending G with information about resource occurrences, returns precisely the partial answers σ' considered in line 45; hence, the number of such σ' determines the number of passes through the loop in lines 44–54. We also present a query $\mathsf{S}_{\boldsymbol{A},i,k,\ell}$ whose number of answers is equal to the number of messages sent from server k to server ℓ in line 50 or 53. As we discuss in Section 6.2, these numbers are the basic building blocks of our cost model. Moreover, since $\mathsf{P}_{\boldsymbol{A},i,k}$ and $\mathsf{S}_{\boldsymbol{A},i,k,\ell}$ are just CQs, we can estimate the numbers of their answers using an arbitrary cardinality estimator (which is likely to require statistics about G').

We first define G'. Let occ_s, occ_p, occ_o, and srv_k for $k \in C$ be fresh resources not occurring in G. Then, G' is obtained by extending G with triples $\langle r, \mathsf{occ}_\pi, \mathsf{srv}_k \rangle$ for each position $\pi \in \Pi$, resource $r \in \mathsf{voc}_\pi(G)$, and server $k \in \mu_\pi(r)$. These triples encode resource occurrences, but we need triple occurrences as well. For simplicity, we first assume that all triples containing a resource r in their subject are assigned to the same partition element P_k; thus, $\mu_s(r) = \{k\}$—that is, r occurs in the subject position only on server k. Such triple assignment is practically beneficial as it is allows answering subject–subject joins without any communication. We discuss later ways around this restriction.

Now let $\mathsf{P}_{\boldsymbol{A},i,k}$ be the following query, where $\boldsymbol{X}_i$ are all variables occurring in $A_1 \wedge \cdots \wedge A_i$.

$$\mathsf{P}_{\boldsymbol{A},i,k}(\boldsymbol{X}_i) = A_1 \wedge \cdots \wedge A_i \wedge \langle \mathsf{term}_s(A_i), \mathsf{occ}_s, \mathsf{srv}_k \rangle \quad (1)$$

Evaluating $A_1 \wedge \cdots \wedge A_i$ on G clearly produces each assignment σ' considered in stage i in line 45 of Algorithm 3 on any server. Moreover, by our assumption on data partitioning, the last atom of $\mathsf{P}_{\boldsymbol{A},i,k}$ is true whenever atom A_i is matched in P_k. Thus, σ' is an answer to $\mathsf{P}_{\boldsymbol{A},i,k}$ on G' if and only if σ' is considered in stage i on server k.

Also, for $i < n$, let $\mathsf{S}_{\boldsymbol{A},i,k,\ell}$ be as follows, where set B contains each *bound* position π of A_{i+i}—that is, $\pi \in B$ iff $\mathsf{term}_\pi(A_{i+1})$ is a resource or a variable occurring in $\boldsymbol{X}_i$.

$$\mathsf{S}_{\boldsymbol{A},i,k,\ell}(\boldsymbol{X}_i) = \mathsf{P}_{\boldsymbol{A},i,k} \wedge \bigwedge_{\pi \in B} \langle \mathsf{term}_\pi(A_{i+1}), \mathsf{occ}_\pi, \mathsf{srv}_\ell \rangle \quad (2)$$

For each σ', server k sends a PAR message for stage $i+1$ to server ℓ if all resources of $A_{i+1}\sigma'$ occur on server ℓ. Since B contains each position at which $A_{i+1}\sigma'$ contains a resource, σ' is an answer to $\mathsf{S}_{\boldsymbol{A},i,k,\ell}$ on G' iff server k sends a PAR message for stage $i+1$ containing σ' to server ℓ. For $i = n$, the ANS messages that server k sends to the coordinator (if k is not the coordinator) are determined by just $\mathsf{P}_{\boldsymbol{A},n,k}$.

We now discuss how to handle cases when triples are not assigned to partition elements by subject. Query $\mathsf{S}_{\boldsymbol{A},i,k,\ell}$ then remains unchanged since it uses resource occurrences only, but $\mathsf{P}_{\boldsymbol{A},i,k}$ needs to be adapted. Intuitively, G' must then also capture locations of triples, which can be done in (at least) one of two ways. First, we can use reification: for each triple $\langle t_s, t_p, t_o \rangle \in P_\ell$, we introduce a fresh resource t and transform the triple as $\langle t, \mathit{rdf}{:}\mathit{subject}, t_s \rangle$, $\langle t, \mathit{rdf}{:}\mathit{predicate}, t_p \rangle$, and $\langle t, \mathit{rdf}{:}\mathit{object}, t_o \rangle$, and we further record the location of the triple by $\langle t, \mathsf{occ}, \mathsf{srv}_\ell \rangle$. We also transform the atoms of (1) accordingly, and we require atom A_i to be matched on server k. Second, we can use *quads*—an extension of RDF where basic data units are quadruples of resources. Thus, we transform each $\langle t_s, t_p, t_o \rangle \in P_\ell$ into $\langle t_s, t_p, t_o, \mathsf{srv}_\ell \rangle$, and we adapt the atoms of (1) accordingly.

6.2 The Cost Model

We next use the cardinalities of $\mathsf{P}_{\boldsymbol{A},i,k}$ and $\mathsf{S}_{\boldsymbol{A},i,k,\ell}$ to determine the cost of a query evaluation plan $\boldsymbol{A}$.

6.2.1 The Cost at Each Server

We first consider the processing at each server in the cluster. Capturing a server's behaviour on $\boldsymbol{A}$ precisely is challenging, so we make several simplifying assumptions. First, we assume that all servers in C have the same numbers of threads, each taking an equal share of the server's workload. This will hold if local processing can be parallelised, but a discussion of the issues involved is out of scope here. Second, we assume that a server never waits for messages—that is, a message is available whenever a server's thread is idle. This can be expected to hold whenever the number of sent messages is not very small. Third, we assume that the queues of all servers are large enough so that line 39 of Algorithm 2 always succeeds. As long as the queues are reasonably large, this assumption should not significantly affect query planning: flow control will be needed only on plans producing many partial answers, but the cost of such plans is likely to be high anyway. Fourth, we assume that retrieving each assignment ρ in line 44 of Algorithm 3 requires a fixed amount of time. This can be expected to hold in most implementations that match atoms using indexes.

With these assumptions in mind, the cost of local processing at server k is proportional to the number of passes through the loop in lines 44–54, which is given by

$$\mathsf{local}_k(\boldsymbol{A}) = \sum_{i=1}^{n} [\mathsf{P}_{\boldsymbol{A},i,k}]_{G'}. \quad (3)$$

To estimate the amount of data sent by server k, let M_i be the average size of a PAR/ANS message for stage $1 \le i \le n+1$. This should be easy given that the message size depends on the number of variables being sent, which is determined by i; in fact, we can derive M_i based on $|\boldsymbol{V}_i|$, where $\boldsymbol{V}_i$ is defined in line 66 of Algorithm 4. Then, if server k is the coordinator, it will send a total of

$$\mathsf{send}_k(\boldsymbol{A}) = \sum_{i=1}^{n-1} \mathsf{M}_{i+1} \cdot \sum_{\ell \in C \setminus \{k\}} [\mathsf{S}_{\boldsymbol{A},i,k,\ell}]_{G'} \quad (4)$$

bytes to other servers. If k is not the coordinator, we further extend (4) by $\mathsf{M}_{n+1} \cdot [\mathsf{P}_{\boldsymbol{A},i,k}]_{G'}$ to also account for the ANS messages that server k sends to the coordinator.

6.2.2 Combining the Cost of all Servers

The cost of a plan $\boldsymbol{A}$ must reflect the fact that servers are largely independent and work in parallel. Ganguly et al. [37] present a general query planning framework for such a setting. They represent the cost of a plan $\boldsymbol{A}$ as a vector $\boldsymbol{O}_{\boldsymbol{A}}$, and they order such costs *partially*; as a consequence, the cost of two plans can be incomparable. Moreover, they show that such a framework is incompatible with standard dynamic programming query planning algorithms, and they extend such algorithms to handle partially ordered cost models. To apply this well-known technique, we must define $\boldsymbol{O}_{\boldsymbol{A}}$, which requires addressing the following two issues.

First, $\mathsf{local}_k(\boldsymbol{A})$ and $\mathsf{send}_k(\boldsymbol{A})$ are incomparable: the former is the number of passes through a loop, and the latter is the number of bytes. We therefore scale $\mathsf{send}_k(\boldsymbol{A})$ by a factor f so that both numbers reflect processing time. To select f, we can measure the average time for processing one loop iteration and compare it with the cost of network communication. For example, if each pass through the loop takes about 10 μs, all servers use just one thread, the cluster uses gigabit Ethernet, and we disregard network congestion, then we can take $f = 0.08$ as the ratio of the times required to send one byte and to process one loop iteration.

Second, we must combine $\mathsf{local}_k(\boldsymbol{A})$ and $\mathsf{send}_k(\boldsymbol{A})$ as appropriate for the implementation at hand. We see two possibilities for this. First, if a call to PUTINTOQUEUE is likely to block the sending thread (e.g., if sent messages are written directly onto a TCP connection), then the two costs should be added; hence, we define $\boldsymbol{O}_{\boldsymbol{A}}$ as containing $\mathsf{local}_k(\boldsymbol{A}) + f \cdot \mathsf{send}_k(\boldsymbol{A})$ for each server $k \in C$. Second, if PUTINTOQUEUE just copies messages into a large queue that is processed in the background, then message sending is just another parallel task and we define $\boldsymbol{O}_{\boldsymbol{A}}$ as containing both $\mathsf{local}_k(\boldsymbol{A})$ and $f \cdot \mathsf{send}_k(\boldsymbol{A})$ for each $k \in C$.

7 Optimised Partitioning of RDF Data

Ensuring that most query answers are local can be critical to the efficiency of our algorithm, so we complement our

approach by a new data partitioning strategy. Several existing approaches [2], [9], [18], [21] achieve this by using graph partitioning: they divide the resources of an RDF graph into sets of roughly the same size while minimising the number of triples with resources from more than one set; then, they assign triples to partitions based on their subject/object. We also use graph partitioning, but we overcome several drawbacks. First, all approaches mentioned in Section 3.2 replicate data to more than one server, which can be costly; for example, 2-hop replication can increase the data size by a factor of 4.8 [9]. In contrast, our approach produces strict partitions, which benefits scalability. Second, a balanced partition of resources does not guarantee that the sizes of partition elements will be balanced as well: if triples are assigned to partition elements based on their subject, then high-degree subjects will bring more triples into a partition element than the low-degree ones. We use *weighted graph partitioning* to produce partitions balanced in the number of triples. While we do not expect this to have a major impact on the performance of query answering, in Section 8.4 we show experimentally that it produces more balanced partitions, which benefits scalability by ensuring that the servers use roughly the same amount of memory.

Let G be an RDF graph that we wish to partition into $|C|$ partition elements. We proceed in three steps.

First, we transform G into an undirected weighted graph (V, E, w). We define $V = \mathsf{voc}_s(G)$—that is, vertices are the resources occurring in G in the subject position. We add to E an undirected edge $\{t_s, t_o\}$ for each triple $\langle t_s, t_p, t_o \rangle \in G$ where $t_p \neq \mathit{rdf}{:}\mathit{type}$, $t_o \in V$, and t_o is not a literal. We set the weight $w(r)$ of each resource $r \in V$ to the number of triples in G that contain r in the subject position. We prune classes (i.e., objects of the $\mathit{rdf}{:}\mathit{type}$ property) and literals because these resources often have a large number of incoming connections, which can confuse graph partitioning algorithms. As we discuss shortly, this does not affect the number of local answers on common queries.

Second, we use *weighted graph partitioning* on (V, E, w): we compute a function $\tau : V \to C$ 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 [22].

Third, we compute each partition element by assigning triples based on subject—that is, we assign each triple $\langle t_s, t_p, t_o \rangle \in G$ to partition element $P_{\tau(t_s)}$. Note that triples are thus not duplicated between partition elements.

This strategy is tailored to common query loads: a study of more than 3 million real-world SPARQL queries revealed that approximately 60% of joins are subject–subject joins, 35% are subject–object joins, and less than 5% are object–object joins [38]. Pruning classes and literals before graph partitioning makes it more likely that these will end up in different partitions, but this can affect the performance only of object–object joins, which are the least common in practice. In other words, pruning does not affect 95% of the joins in practice, but it increases the chance of obtaining a good partition, and it also makes (V, E, w) smaller. Moreover, by placing all triples with the same subject on a single server, we can answer the most common subject–subject joins without any communication. Finally, the weight $w(r)$ of each vertex r in (V, E, w) determines exactly the number of triples that are added to $P_{\tau(r)}$ due to assigning r to partition $\tau(r)$; since weighted graph partitioning balances the sum of the weights of vertices in each partition, the partition elements are balanced in the numbers of triples.

8 Evaluation

We implemented our algorithms in the RDFox system.[1] The system's core was written in C++ and it uses the METIS [22] graph partitioner. The query planner was written in Java as it reuses the recent SumRDF [39] query cardinality estimator. To investigate the effects of data partitioning, we evaluated the RDFox$_{GP}$ and RDFox$_{HP}$ versions of RDFox, which respectively use graph partitioning from Section 7 and hash partitioning by subject. Our goals were to (i) compare the performance of RDFox query answering with state of the art systems, (ii) investigate how our approach scales with increasing data load, and (iii) compare the uniformity of partitions produced by different partitioning strategies.

8.1 Test Datasets

We based our evaluation on two well-known benchmarks. WatDiv v0.6 [23] aims to simulate realistic data and query loads. We used the original 20 query templates classified into four categories: linear (L), star (S), snowflake (F), and complex (C); each template contains at most one parameter that is replaced with a resource from the RDF graph. In addition to these queries, we also used the 18 incremental linear (IL) queries developed by Schätzle et al. [5].

LUBM [24] is a widely used benchmark in the Semantic Web community. It comes with 14 predefined queries, but most of them do not return any results if reasoning is not used. Thus, we instead used seven queries (T1–T7) by Zeng et al. [13] that compensate for the lack of reasoning, and we manually generated three new complex, cyclic queries.

Table 1 shows the sizes of our test graphs. All queries we used are available in the literature, apart from the three new LUBM queries, which are shown in Table 3.

8.2 Query Answering Experiments

Comparison Systems. In a recent, extensive comparison of 11 distributed RDF stores [19], AdPart [16] and TriAD [18] consistently outperformed all other systems. Independently, Gurajada et at. [18] argued that TriAD outperforms two centralised (i.e., RDF-3X [32] and BitMat [40]) and four distributed stores (i.e., SHARD [4], H-RDF-3X [9], 4store [12], and Trinity.RDF [13]), the MonetDB [41] column store, and 'raw' Apache Hadoop and Spark. We thus used AdPart and TriAD as our main points of comparison. We configured TriAD to use the summary graph. The adaptive partitioning of AdPart affects query evaluation only on workloads containing thousands of queries [16], so we did not use it: this issue is orthogonal to core query evaluation and it could be easily adapted to TriAD and RDFox as well.

To compare our approach against systems that use different implementation styles, we also considered S2RDF [5] (which was shown to outperform H_2RDF+ [10], Sempala [8], PigSPARQL [7], and SHARD [4]) and PEDA [15] (which

1. http://www.cs.ox.ac.uk/isg/tools/RDFox/

TABLE 1: Numbers of Resources and Triples in Our Test Datasets, and Idle Memory Use Per Server (GB)

Dataset	Resources	Triples	RDFox$_{GP}$			RDFox$_{HP}$			AdPart			TriAD		
			Mean	Max	Sdev	Mean	Max	Sdev	Mean	Max	Sdev	Mean	Max	Sdev
WatDiv-10K	97.74 M	1.09 G	4.39	5.42	0.54	4.71	4.72	0.01	13.90	13.91	0.00	9.57	10.99	0.73
LUBM-10K	328.62 M	1.33 G	5.49	5.61	0.15	5.86	5.89	0.06	26.08	26.10	0.01	12.04	19.26	3.98

TABLE 2: The Performance of Query Answering

	Query	Answers	Query Evaluation Time (ms)						Total Network Communication (kB)				Total RAM Use (MB)			
			RDFox$_{GP}$	RDFox$_{HP}$	AdPart	TriAD	S2RDF	PEDA	RDFox$_{GP}$	RDFox$_{HP}$	AdPart	TriAD	RDFox$_{GP}$	RDFox$_{HP}$	AdPart	TriAD
WatDiv-10K Basic Queries	L1	2	2	2	5	11	471	15,356	31	30	62	227	1	2	1	1
	L2	16,132	45	38	15	15	498	1,622	262	257	203	1,106	1	1	1	1
	L3	24	2	2	2	6	549	16,889	17	17	11	76	1	1	1	1
	L4	5,782	16	4	3	5	209	261	93	98	58	299	1	1	1	1
	L5	12,957	23	28	18	17	270	49,539	304	300	218	940	1	1	1	1
	S1	12	3	5	23	41	2,208	43,803	77	81	366	142	1	1	1	1
	S2	6,685	11	12	5	33	607	74,479	184	180	96	517	1	1	1	1
	S3	0	26	9	4	8	311	8,087	37	37	11	91	1	1	1	1
	S4	153	21	23	17	22	329	16,520	3,061	3,010	420	108	1	2	1	1
	S5	0	12	8	5	—	260	1,861	37	38	11	—	1	2	1	—
	S6	453	8	4	2	8	235	50,865	37	38	17	151	1	1	1	1
	S7	0	2	2	1	3	420	56,784	28	29	4	58	1	1	1	1
	F1	324	42	15	11	15	590	64,748	855	1,693	127	265	1	2	1	1
	F2	188	8	5	15	263	1,226	207,725	109	102	130	11,461	1	2	1	25
	F3	865	12	6	14	208	1,969	4,831,257	197	269	280	337	1	1	1	29
	F4	2,879	25	11	31	—	1,265	260,410	437	399	466	—	1	1	1	—
	F5	65	3	4	16	348	2,254	26,208	59	62	238	29,900	1	1	1	76
	C1	1,504	154	201	56	248	2,508	212,129	6,103	17,209	2,721	3,170	16	5	1	27
	C2	288	175	198	69	343	2,740	1,787,692	8,564	12,772	2,290	45,520	2	14	33	97
	C3	42.44 M	270	295	9,078	419	16,407	123,349	1,186	1,175	1,197,297	423	2	72	1,685	8
WatDiv-10K Incremental Linear Queries	IL-1-5	7,469	3	4	13	8,322	12,543	n/a	285	284	399	2,276,527	800	881	3	18,492
	IL-1-6	0	3	4	4	19,957	12,252	n/a	54	55	52	6,595,267	269	217	1	37,599
	IL-1-7	16,132	32	35	37	32,496	15,062	n/a	1,624	2,302	1,137	9,370,381	138	1,186	1	49,780
	IL-1-8	0	3	5	6	17,985	15,003	n/a	69	71	51	5,756,890	33	36	1	28,713
	IL-1-9	1,591	24	5	93	8,505	15,478	n/a	1,972	2,106	1,453	3,061,088	21	28	3	18,605
	IL-1-10	758	12	19	51	8,294	16,124	n/a	1,032	979	887	3,097,132	23	22	3	18,604
	IL-2-5	15,444	4	5	15	8,241	41,188	n/a	505	518	618	2,970,367	4	9	1	18,430
	IL-2-6	0	3	3	9	14,471	13,276	n/a	56	55	140	5,149,633	1	5	1	27,260
	IL-2-7	1,386	9	12	43	3,600	14,182	n/a	1,428	796	784	358,506	2	1	1	7,430
	IL-2-8	267	6	7	31	18,128	15,261	n/a	519	282	491	5,302,047	5	1	1	28,660
	IL-2-9	171	6	7	30	18,021	16,313	n/a	336	257	509	5,457,976	1	3	1	28,896
	IL-2-10	32	8	10	41	17,718	13,922	n/a	768	565	649	5,131,115	3	1	1	28,812
	IL-3-5	3.34 G	773,619	782,884	—	—	29,590	n/a	105,472,000	112,019,456	—	—	2,794	2,844	—	—
	IL-3-6	3.75 G	1,770,593	2,042,988	—	—	87,525	n/a	263,450,624	284,635,136	—	—	6,583	5,507	—	—
	IL-3-7	584.92 M	1,239,115	1,307,131	—	—	102,971	n/a	72,719,360	99,340,288	—	—	5,264	2,988	—	—
	IL-3-8	19.56 G	6,996,328	7,379,970	—	—	2,068,100	n/a	307,225,600	1,007,030,272	—	—	6,805	7,314	—	—
	IL-3-9	954.41 M	2,254,083	2,452,539	—	—	158,595	n/a	153,052,160	185,125,888	—	—	9,276	1,803	—	—
	IL-3-10	954.41 M	2,090,013	2,154,819	—	—	141,940	n/a	175,571,968	208,282,624	—	—	4,881	2,853	—	—
LUBM-10K	T1	2,528	727	1,017	26,886	13,410	n/a	309,361	4,033	716,588	1,770,962	197,762	37	148	7,238	1,144
	T2	10.80 M	699	975	711	927	n/a	23,685	150,073	151,466	86,683	104,657	144	153	1	154
	T3	0	419	1,149	879	771	n/a	10,368	1,897	1,915	15	466	1	12	1	708
	T4	10	4	4	2	7	n/a	753	51	44	11	115	1	8	1	1
	T5	10	2	2	2	2	n/a	125	18	17	11	63	1	1	1	1
	T6	125	3	3	1	85	n/a	1,914	38	41	4	153	1	1	1	1
	T7	439,994	1,010	14,574	7,709	7,294	n/a	46,123	10,857	1,454,774	166,989	29,592	8	90	815	844
	N1	2,528	1,723	9,263	—	1,755	n/a	n/a	5,487	1,742,285	—	8,154	21	48	—	232
	N2	4.11 M	4,487	32,226	—	23,711	n/a	n/a	138,803	4,973,168	—	184,661	72	280	—	3,501
	N3	2.22 M	920	6,297	—	33,661	n/a	n/a	65,539	1,204,533	—	111,571	24	4	—	6,645

was shown comparable to EAGRE [2], H-RDF-X [9], SHAPE [11], FedX [42], and SPLENDID [43]). We considered PEDA rather than PECA since the former was shown to be more efficient. S2RDF and PEDA were evaluated in the literature in similar settings (S2RDF used 1.9 GHz processors, and PEDA used 16 GB of RAM per server). Based on the published results, AdPart and TriAD clearly outperform S2RDF/PEDA by orders of magnitude on all but the IL-3-n queries, which was confirmed in the recent comparison [19]. Thus, for S2RDF and PEDA we only recapitulate the query evaluation times published in the literature.

Test Setting. All of our tested systems (AdPart, TriAD, and RDFox) were configured to deliver query answers to the coordinator, but not to store or print them. We evaluated each query five times, and in each run we recorded the query evaluation times as reported by each system, the network communication as reported by running `iptraf` on each server, and the total maximum memory use across all servers. RDFox times do not cover query planning as the planner was implemented separately, but this should not affect our analysis as query planning in other systems is very efficient. RDFox servers used five processing and one communication thread, and TriAD determined the number of threads internally. Graph partitioning for RDFox and TriAD was conducted on one server with 256 GB of RAM and two 2.6 GHz Intel Xeon E5-2640v3 CPUs. All queries were run on a cluster of ten m4.2xlarge Amazon EC2

TABLE 3: Three New Queries for LUBM

N1	SELECT ?P1 ?D1 ?S1 ?U1 WHERE { ?D1 ub:subOrganizationOf ?U1 . ?P1 ub:worksFor ?D1 . ?S1 ub:advisor ?P1 . ?S1 ub:undergraduateDegreeFrom ?U1 }
N2	SELECT ?S1 ?C1 ?P1 ?C2 ?S2 ?C3 WHERE { ?S2 ub:teachingAssistantOf ?C2 . ?P1 ub:teacherOf ?C2 . ?S2 ub:takesCourse ?C3 . ?S1 ub:takesCourse ?C3 . ?S1 ub:takesCourse ?C1 . ?P1 ub:teacherOf ?C1 }
N3	SELECT ?S1 WHERE { ?S2 ub:teachingAssistantOf ?C2 . ?P1 ub:teacherOf ?C2 . ?S2 ub:takesCourse ?C1 . ?P1 ub:teacherOf ?C1 . ?S2 ub:takesCourse ?C3 . ?S1 ub:takesCourse ?C1 . ?S1 ub:takesCourse ?C3 }

TABLE 4: Min./Max. Numbers of Triples and Average Number of Resources Per Partition Element (M), and Partitioning Times (s)

	WatDiv-10K				LUBM-10K			
Partitioning Scheme	Min	Max	Res	Time	Min	Max	Res	Time
Weighted, pruning	103.1	113.0	20.9	19,719	126.4	138.2	32.9	15,606
Weighted, no pruning	102.1	113.0	21.6	21,727	123.6	139.8	35.7	30,018
Unweighted, no pruning	22.5	410.7	18.1	24,118	123.7	142.3	36.0	30,396
Subject hashing	109.0	109.3	24.2	6,174	133.3	133.7	52.5	13,142

servers, each having 32 GB RAM and eight virtual cores of 2.4 GHz Intel Xeon E5-2676v3 CPUs, connected by a network with dedicated bandwidth of 1,000 Mbps. AdPart could not load the LUBM dataset on these servers, so we loaded the data on a server with 160 GB of RAM.

Results. For each query, the number of answers, and the query evaluation time, the total network communication (i.e., the total amount of data sent by all servers), and the total memory used for query processing (i.e., the difference between the amount of memory used each server during and before query processing, added over all servers) averaged over five runs are shown in Table 2. We evaluated queries with no timeout; however, AdPart and TriAD crashed due to running out of memory on nine and eight queries, respectively, which are marked with '—'. For S2RDF and PEDA, results that were not available in the literature are marked with 'n/a'. We verified all outputs using a centralised RDF store, confirming that TriAD and RDFox returned correct answers in all cases. In contrast, AdPart returned duplicate answers on ten basic WatDiv queries and produced incorrect results on LUBM queries T1 and T6.

Discussion. All systems outperformed PEDA, which we believe is inherent in its query evaluation strategy. We have studied a closely related idea in our earlier work [21] and have observed that the initial modified evaluation of a query on all servers tends to produce huge numbers of partial answers. Indeed, even the partial query evaluation phase of PEDA (see [15]) is already slower by orders of magnitude than full query evaluation in the other systems.

On queries other than IL-3-n, AdPart, TriAD, and RDFox outperformed S2RDF by up to four orders of magnitude. To understand why, note that Spark partitions data at the level of blocks of a distributed file system, rather than at the level of triples. Triples are thus assigned to servers randomly, and not even triples with the same subject are guaranteed to be collocated; hence, each join requires data exchange, even subject–subject joins. Also, Spark processes query plans synchronously, reducing the potential for parallelism.

On the IL-3-n queries, S2RDF is up to an order of magnitude faster than the two RDFox variants. These queries were designed as a 'torture test' for RDF stores, and, as Table 2 shows, they return several orders of magnitude more answers than the other queries. These results are explained by the fact that query evaluation in Spark materialises the answers in a distributed file system. In contrast, our approach sends all answers to the coordinator, which becomes a major source of overhead for such queries. For example, during the evaluation of IL-3-5 on RDFox$_{GP}$, about 96% of all network traffic (about 107 GB) is used for sending answers, which requires 856 s using the available bandwidth of 1000 Mbps. In practice, RDFox$_{GP}$ evaluates the query in 773 s, which is possible since not all data is sent from the same server. The overhead of sending answers varies across queries (e.g., about 56% of network traffic is used on IL-3-6), but it is substantial in each case. This skews the comparison between RDFox and S2RDF on queries with large answer sets. Finally, if the user does not wish to iterate over all answers, our approach can easily be modified to store answers on the server where they are produced.

The three systems are roughly comparable on the linear (Ln) and star (Sn) WatDiv queries. These queries contain at most one join that is not subject–subject, and since all three systems compute subject–subject joins without any communication, performance variation is due to the implementation of local query evaluation. AdPart is fastest in nine cases, despite the fact that TriAD uses faster merge joins. The snowflake (Fn) queries involve two groups of atoms mostly joined on subjects connected by a subject–object join. Our approach computes most answers using at most one hop between servers, allowing the two RDFox variants to be fastest in four cases. Complex (Cn) queries C1 and C2 exhibit different join patters and AdPart is fastest by a factor of three. Query C3 contains only subject–subject joins, but it is not selective and returns 42.44 M answers. While RDFox and TriAD can process C3 without any problems, AdPart is slower by two orders of magnitude, uses four orders of magnitude more communication, and three orders of magnitude more memory. The RDFox variants outperformed TriAD on the snowflake and complex queries by up to two orders of magnitude (e.g., on F2, F3, and F5), even though TriAD prunes the search space using a summary graph.

The RDFox variants outperformed AdPart and TriAD by between one and four orders of magnitude on the extended WatDiv queries, and on all LUBM queries apart from T4 and T5. The memory use of hash joins in AdPart and TriAD is determined by the sizes of intermediate relations, which, as we argue in Section 3.3, can significantly limit the scalability of distributed systems. This is particularly acute on queries returning large answer sets: both systems ran out of memory on all IL-3-n queries, and AdPart could also not answer any of the Nn queries. In contrast, index nested loop joins coupled with dynamic data exchange and careful flow control limited memory use in the RDFox variants and allowed them to successfully process all queries.

RDFox$_{HP}$ was often slightly faster than RDFox$_{GP}$ on the WatDiv linear and star queries, and their performance was broadly the same on the IL-m-n queries since sending answers is the main source of overhead there. In contrast,

RDFox$_{GP}$ performed much better than RDFox$_{HP}$ on all measures (time, communication, and memory use) on the snowflake and complex WatDiv queries and on LUBM. The uniform structure of LUBM is particularly amenable to graph partitioning, and the benefits of graph partitioning seem most significant there. This confirms our intuition that dynamic data exchange is particularly useful when a substantial proportion of answers can be produced locally. In general, communication overhead seems to be lowest in AdPart, followed by RDFox$_{GP}$, TriAD, and RDFox$_{HP}$. However, RDFox$_{HP}$ is still generally faster than AdPart and TriAD, and it uses much less memory on query T1.

Table 1 shows the average and maximal memory use per server (excluding dictionaries) and the standard deviation across servers after loading the data and before query processing. RDFox does not duplicate data and thus uses about half the memory of TriAD, which hashes triples by subject and object. AdPart uses the most memory.

8.3 Scalability Experiments

To analyse the horizontal scalability of our approach, Figure 2 shows the evaluation times of the basic WatDiv queries on datasets with scale factors 1,000, 3,000, 5,000, 7,000, and 10,000. The times for all linear queries apart from L2, all star and snowflake queries, and complex queries C1 and C2 scale sublinearily with the data sizes. The times for L3, S1, S7, and F5 are constant on all data sets. Finally, times for C3 scale roughly linearly with the data size. Thus, we conclude that our approach scales very well.

8.4 Data Partitioning Experiments

Table 4 shows the triple and resource distributions as well as partitioning times for the data partitioning strategy described in Section 7 (weighted, pruning), the strategy used without the pruning step (weighted, no pruning), vertex graph partitioning (unweighted, no pruning), and subject hashing. All times include writing the partitioned output to files. Subject hashing involves almost no overhead in terms of computing the partition elements, so its time is roughly equal to a fixed overhead that is incurred in all cases.

Subject hashing produces very balanced partitions and is much faster than graph partitioning. Moreover, weighted partitions are much more uniform than unweighted ones, particularly on WatDiv. Due to better clustering of connections, the average number of resources per partition element is much smaller in partitioning-based approaches than for subject hashing. This is in line with our observations from Section 8.2: since local results do not require any communication, RDFox$_{GP}$ often outperforms RDFox$_{HP}$.

Pruning also speeds up partitioning by reducing the size of the graph being partitioned: on LUBM, the performance gain is a factor of two.

9 Conclusion & Outlook

We have presented a new technique for query answering in distributed RDF systems based on dynamic data exchange. A key feature of our technique is that the memory used for query processing can be bounded, which allowed our system to outperform two state of the art systems in terms of response time and memory and network use, often by orders of magnitude. A major line of our future research will involve incorporating bushy join plans, and *factorised* query evaluation [44] is a promising direction due to its strong optimality guarantees. We will also investigate materialising datalog rules over distributed RDF graphs.

Acknowledgements

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

References

[1] F. Goasdoué, Z. Kaoudi, I. Manolescu, J. Quiané-Ruiz, and S. Zampetakis, “CliqueSquare: Flat plans for massively parallel RDF queries,” in *ICDE*, 2015, pp. 771–782.
[2] X. Zhang, L. Chen, Y. Tong, and M. Wang, “EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud,” in *ICDE*, 2013, pp. 565–576.
[3] M. F. Husain, J. P. McGlothlin, M. M. Masud, L. R. Khan, and B. M. Thuraisingham, “Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing,” *IEEE TKDE*, vol. 23, no. 9, pp. 1312–1327, 2011.
[4] K. Rohloff and R. E. Schantz, “Clause-Iteration with MapReduce to Scalably Query Data Graphs in the SHARD Graph-Store,” in *DIDC*, 2011, pp. 35–44.
[5] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen, “S2RDF: RDF Querying with SPARQL on Spark,” *CoRR*, vol. abs/1512.07021, 2015.
[6] A. Schätzle, M. Przyjaciel-Zablocki, T. Berberich, and G. Lausen, “S2X: Graph-Parallel Querying of RDF with GraphX,” in *Big-O(Q) Workshop*, 2015, pp. 155–168.
[7] A. Schätzle, M. Przyjaciel-Zablocki, T. Hornung, and G. Lausen, “PigSPARQL: A SPARQL Query Processing Baseline for Big Data,” in *ISWC (Poster)*, 2013, pp. 241–244.
[8] A. Schätzle, M. Przyjaciel-Zablocki, A. Neu, and G. Lausen, “Sempala: Interactive SPARQL Query Processing on Hadoop,” in *ISWC*, 2014, pp. 164–179.
[9] J. Huang, D. J. Abadi, and K. Ren, “Scalable SPARQL Querying of Large RDF Graphs,” *PVLDB*, vol. 4, no. 11, pp. 1123–1134, 2011.
[10] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris, “H_2RDF+: High-performance Distributed Joins over Large-scale RDF Graphs,” in *Big Data*, 2013, pp. 255–263.
[11] K. Lee and L. Liu, “Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning,” *PVLDB*, vol. 6, no. 14, pp. 1894–1905, 2013.
[12] S. Harris, N. Lamb, and N. Shadbolt, “4store: The Design and Implementation of a Clustered RDF Store,” in *SSWS*, ser. CEUR Workshop Proceedings, vol. 517, 2009, pp. 94–109.
[13] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang, “A Distributed Graph Engine for Web Scale RDF Data,” *PVLDB*, vol. 6, no. 4, pp. 265–276, 2013.
[14] A. Harth, J. Umbrich, A. Hogan, and S. Decker, “YARS2: A Federated Repository for Querying Graph Structured Data from the Web,” in *ISWC*, 2007, pp. 211–224.
[15] P. Peng, L. Zou, M. T. Özsu, L. Chen, and D. Zhao, “Processing SPARQL queries over distributed RDF graphs,” *VLDB Journal*, vol. 25, no. 2, pp. 243–268, 2016.
[16] R. Al-Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, and M. Sahli, “Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning,” *VLDB Journal*, vol. 25, no. 3, pp. 355–380, 2016.
[17] L. Galárraga, K. Hose, and R. Schenkel, “Partout: a distributed engine for efficient RDF processing,” in *WWW*, 2014, pp. 267–268.
[18] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald, “TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing,” in *SIGMOD*, 2014, pp. 289–300.
[19] I. Abdelaziz, R. Harbi, Z. Khayyat, and P. Kalnis, “A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data,” *PVLDB*, vol. 10, no. 13, pp. 2049–2060, 2017.

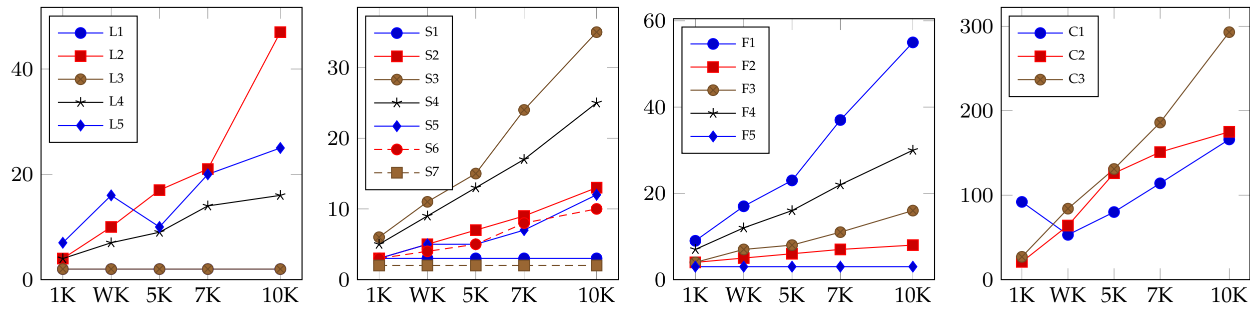

Fig. 2: Scalability on WatDiv-1K, 3K, 5K, 7K, and 10K

[20] O. Polychroniou, R. Sen, and K. A. Ross, “Track Join: Distributed Joins with Minimal Network Traffic,” in *SIGMOD*, 2014, pp. 1483–1494.

[21] A. Potter, B. Motik, and I. Horrocks, “Querying Distributed RDF Graphs: The Effects of Partitioning,” in *SSWS*, ser. CEUR Workshop Proceedings, vol. 1261, 2014, pp. 29–44.

[22] G. Karypis and V. Kumar, “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs,” *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, 1998.

[23] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee, “Diversified Stress Testing of RDF Data Management Systems,” in *ISWC*, 2014, pp. 197–212.

[24] Y. Guo, Z. Pan, and J. Heflin, “LUBM: A benchmark for OWL knowledge base systems,” *J. Web Semantics*, vol. 3, no. 2, pp. 158–182, 2005.

[25] D. Kossmann, “The State of the Art in Distributed Query Processing,” *ACM Comput. Surv.*, vol. 32, no. 4, pp. 422–469, 2000.

[26] G. Graefe and D. L. Davison, “Encapsulation of Parallelism and Architecture-Independence in Extensible Database Query Execution,” *IEEE Trans. Software Eng.*, vol. 19, no. 8, pp. 749–764, 1990.

[27] Y. Kambayashi, M. Yoshikawa, and S. Yajima, “Query Processing for Distributed Databases Using Generalized Semi-Joins,” in *SIGMOD*, 1982, pp. 151–160.

[28] J. K. Mullin, “Optimal Semijoins for Distributed Database Systems,” *IEEE TSE*, vol. 16, no. 5, pp. 558–560, 1990.

[29] B. Wu, Y. Zhou, P. Yuan, H. Jin, and L. Liu, “SemStore: A Semantic-Preserving Distributed RDF Triple Store,” in *CIKM*, 2014, pp. 509–518.

[30] M. Hammoud, D. A. Rabbou, R. Nouri, S. Beheshti, and S. Sakr, “DREAM: Distributed RDF Engine with Adaptive Query Planner and Minimal Communication,” *PVLDB*, vol. 8, no. 6, pp. 654–665, 2015.

[31] K. Hose and R. Schenkel, “WARP: Workload-aware replication and partitioning for RDF,” in *Workshop at ICDE*, 2013, pp. 1–6.

[32] T. Neumann and G. Weikum, “The RDF-3X engine for scalable management of RDF data,” *VLDB Journal*, vol. 19, no. 1, pp. 91–113, 2010.

[33] B. Motik, Y. Nenov, R. Piro, I. Horrocks, and D. Olteanu, “Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems,” in *AAAI*, 2014, pp. 129–137.

[34] P. Yuan, C. Xie, H. Jin, L. Liu, G. Yang, and X. Shi, “Dynamic and fast processing of queries on large-scale RDF data,” *KAIS*, vol. 41, no. 2, pp. 311–334, 2014.

[35] T. Neumann and G. Weikum, “Scalable Join Processing on Very Large RDF Graphs,” in *SIGMOD*, 2009, pp. 627–640.

[36] C. Weiss, P. Karras, and A. Bernstein, “Hexastore: Sextuple Indexing for Semantic Web Data Management,” *VLDB*, vol. 1, no. 1, pp. 1008–1019, 2008.

[37] S. Ganguly, W. Hasan, and R. Krishnamurthy, “Query Optimization for Parallel Execution,” in *SIGMOD*, 1992, pp. 9–18.

[38] M. A. Gallego, J. D. Fernández, M. A. Martinez-Prieto, and P. de la Fuente, “An Empirical Study of Real-World SPARQL Queries,” *CoRR*, vol. abs/1103.5043, 2011.

[39] G. Stefanoni, B. Motik, and E. V. Kostylev, “Estimating the Cardinality of Conjunctive Queries over RDF Data Using Graph Summarisation,” in *WWW*, 2018, pp. 1043–1052.

[40] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler, “Matrix “Bit”loaded: A Scalable Lightweight Join Query Processor for RDF Data,” in *WWW*, 2010, pp. 41–50.

[41] L. Sidirourgos, R. Gonçalves, M. Kersten, N. Nes, and S. Manegold, “Column-Store Support for RDF Data Management: not all swans are white,” *PVLDB*, vol. 1, no. 2, 2008.

[42] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt, “Fedx: Optimization techniques for federated query processing on linked data,” in *ISWC*, 2011, pp. 601–616.

[43] O. Görlitz and S. Staab, “Splendid: SPARQL endpoint federation exploiting void descriptions,” in *Consuming Linked Data*, 2011, pp. 13–24.

[44] D. Olteanu and J. Závodný, “Size Bounds for Factorised Representations of Query Results,” *ACM TODS*, vol. 40, no. 1, pp. 2:1–2:44, 2015.

Anthony Potter is a doctoral student in the Department of Computer Science at the University of Oxford. He received his BSc in Mathematics from Imperial College London in 2011 and his MSc in Computer Science from the University of Oxford in 2013. His research interests include semantic technology, databases, and distributed systems. He has authored several papers on distributed RDF and graph databases.

Boris Motik is a Professor of Computer Science at the University of Oxford. His research interests include knowledge representation and reasoning, databases, ontologies and the Semantic Web, and their applications. He is the lead editor of the OWL 2 standard. He received the 2007 Cor Baayen Award, was one of “AI's 10 to Watch” in IEEE Intelligent Systems in 2007, received an EPSRC Early Career Fellowship in 2012, and won the 2013 Roger Needham Award.

Yavor Nenov is a computer scientist at the Oxford University. He received his MSc degree from Sofia University in Bulgaria in 2008 and his PhD degree from the University of Manchester in the UK in 2011. He has been a postdoctoral researcher at Oxford University since 2011. His research interests focus on knowledge representation and reasoning and databases. He has authored a number of scientific papers in the top AI journals and conferences.

Ian Horrocks is a Professor in the Department of Computer Science at the University of Oxford and a Visiting Professor in the Department of Informatics at the University of Oslo. He was a key developer of the OWL language and the underlying description logic $\mathcal{SROIQ}$. His interests include reasoning algorithms, optimisation techniques, and applications. He is a Fellow of the Royal Society, a member of Academia Europaea, and a fellow of the European AI Society.

APPENDIX
PROOF OF THEOREM 1

Theorem 1. *If Algorithms 1, 2, and 4 are applied to a strict partition P of an RDF graph G distributed over a cluster C of servers where each server has $n+1$ finite message queues,*

1) *servers terminate after sending $\Theta(n|C|^2)$ FIN messages,*
2) *the coordinator for Q correctly outputs $\mathsf{ans}(Q, G)$, and*
3) *at most $O(n^2)$ memory is needed per server thread.*

Proof of Claims 1 and 3: We assume that the servers evolve over discrete time instants $1, 2, \ldots$ where, at each time t, one thread of one server performs an action. We say that server k is *finished* for stage $i \geq 1$ at time t if $D_i = R_i$ and $N_i = |C|$ hold at server k after the action at time t. Also, to unify the case analyses, we define all servers as finished for stage $i = 0$ at each time $t \geq 0$, and we define all servers as not finished for each stage $i \geq 1$ at $t = 0$. When server k becomes finished for stage i at time t, SWAP in line 28 ensures that the server sends precisely one $\mathsf{FIN}[i+1, S_{i+1,\ell}]$ message to each server $\ell \in C$ if $i < n$, or to the coordinator if $i = n$. We next prove that, for each $t \geq 1$,

1) if a server is finished for stage i at time t, then all servers are finished for stage $i-1$ at time $t-1$,
2) a server finished for stage i at time t does not have a queued PAR/ANS message for stage i and it is not processing such a message at time t, and
3) if a server is finished for stage i at time $t-1$, the server remains finished for stage i at time t.

The proof is by induction on t, where the base case and inductive step are the same. Consider a time $t \geq 1$ such that all properties hold for $t-1$. Assume that server k that is finished for stage i at time t. Server k increments its counter N_i in line 24 once for each FIN message received for stage i, and each server sends to server k just one FIN message per stage; hence, $N_i = |C|$ ensures that all servers have sent their FIN messages for stage $i-1$ before time t (if $i \geq 1$), so all servers are finished for stage $i-1$ at time $t-1$, as required for property 1. Moreover, the inductive hypothesis for properties 1 and 3 ensures that all servers are finished for each stage $j < i$ at time $t-1$. Thus, by property 2, no server is sending a PAR/ANS message for stage $j < i$ to server k at time $t-1$, so server k is not retrieving such a message from a message queue at time t. Lines 42 and 25 ensure that counter R_i at server k contains the number of received PAR/ANS messages for stage i, and counter D_i at server k is incremented in line 17 or 21 whenever such a message is processed; hence, $D_i = R_i$ implies that server k has no queued PAR/ANS messages for stage i and is not processing such messages, as required for property 2. Finally, if server k is finished for stage i at time $t-1$, the observations made thus far ensure that counters N_i, R_i, and D_i of server k remain unchanged at time t, so server k is finished for stage i at time t, as required property 3.

To complete the proof, each P_k is finite so, in each call to MATCHATOM, server k passes through the loop in lines 70–85 a finite number of times and thus produces finitely many PAR/ANS messages. Moreover, at each point in time, at least one queue in the cluster contains a message with the highest stage index, and this message is eventually processed either in the message processing thread or in line 40 of Algorithm 2. Thus, at each time, at least one server makes progress, so all messages are processed eventually; hence, all servers will eventually become finished for stage n, and the coordinator will become finished for stage $n+1$. Each server sends $|C|$ FIN messages per stage, so the total number of such messages is $\Theta(n|C|^2)$. Finally, recursion depth of each thread is n, and each recursion level can be implemented using n iterators that explore set E in line 68 'on the fly', so each thread requires $O(n^2)$ memory. □

Proof of Claim 2: Our algorithms are clearly sound (i.e., $\sigma \in \mathsf{ans}(Q, G)$ holds for each $\langle \sigma, m \rangle$ that is output), so we next prove completeness. Let I_X, U_X, B_i, and $\boldsymbol{V}_i$ be as in lines 61–66 of Algorithm 4; let $\boldsymbol{A} = A_1, \ldots, A_n$ be the reordered atoms of $Q(\boldsymbol{X})$ from line 2 of Algorithm 1; for each $1 \leq i \leq n$, let $\boldsymbol{K}_i = \boldsymbol{V}_i \cap \bigcup_{j=1}^{i-1} \mathsf{vars}(A_j)$, let $\boldsymbol{X}_i = \boldsymbol{X} \setminus \boldsymbol{K}_i$, and let $Q_i(\boldsymbol{X}_i) = A_i \wedge \cdots \wedge A_n$; and let $\boldsymbol{\mu} = \mu_s, \mu_p, \mu_o$ be the vector of occurrence mappings from Section 4.1. We say that a vector $\boldsymbol{\lambda} = \lambda_s, \lambda_p, \lambda_o$ of partial occurrence mappings is *consistent* with $\boldsymbol{\mu}$ if $\lambda_\pi \subseteq \mu_\pi$ for each $\pi \in \Pi$ (i.e., each λ_π coincides with μ_π on the common resources). We also say that a server k *can match* an atom A if an assignment ρ exists such that $A\rho \in P_k$. The following property clearly holds for each atom A and vector $\boldsymbol{\lambda}$ consistent with $\boldsymbol{\mu}$:

> $(*)$ $k \in \text{OCCURS}(A, \boldsymbol{\lambda})$ holds for each server $k \in C$ that can match atom A.

Next, we show that, for each $1 \leq i \leq n$, assignment σ with $\mathsf{dom}(\sigma) = \boldsymbol{K}_i$, positive integer m, vector $\boldsymbol{\lambda}$ consistent with $\boldsymbol{\mu}$, and $k \in C$, if $\text{MATCHATOM}(i, \sigma, m, \boldsymbol{\lambda})$ is called on server k and the call returns $v < i-1$, then there exists $j \geq i$ such that $B_j = v$ and $A_j\sigma\rho \notin G$ for all assignments ρ. The proof is by induction on recursion depth. For the base case, assume that v is returned in line 69 and consider an arbitrary assignment ρ. Then, $\text{OCCURS}(A_i\sigma, \boldsymbol{\lambda}) \subseteq \{k\}$ and $(*)$ ensure $A_i\sigma\rho \notin G \setminus P_k$, and $E = \emptyset$ ensures $A_i\sigma\rho \notin P_k$; thus, the claim holds for $j = i$. For the induction step, assume that v is returned in line 85 after a recursive call in line 84 for assignment σ'. Due to $v < i$ and the induction hypothesis, there exists $j \geq i+1 > i$ such that $B_j = v$ and $A_j\sigma'\rho \notin G$ for all assignments ρ. But then, the definition of B_i and I_X ensure $A_j\sigma' = A_j\sigma$, so the inductive claim holds for j.

Finally, we show that, for each $1 \leq i \leq n$, assignment σ with $\mathsf{dom}(\sigma) = \boldsymbol{K}_i$, positive integer m, and vectors $\boldsymbol{\lambda}_k$ for $k \in C$ consistent with $\boldsymbol{\mu}$, if $\text{MATCHATOM}(i, \sigma, m, \boldsymbol{\lambda}_k)$ is called on each server k that can match $A_i\sigma$, then, for each $\nu \in \mathsf{ans}(Q_i\sigma, G)$ with multiplicity w, the coordinator outputs tuples $\langle \sigma \cup \nu, p_1 \rangle, \ldots, \langle \sigma \cup \nu, p_r \rangle$ such that $p_1 + \cdots + p_r = m \cdot w$. This property proves Claim 2 of Theorem 1 since line 4 of Algorithm 1 calls MATCHATOM (via line 16) for $i = 1$, $\sigma = \emptyset$, $m = 1$, and $\lambda_k = \boldsymbol{\emptyset}$. The proof is by 'reverse' induction on i going from n down to 1.

Many observations are the same for the base case and the inductive step, so we consider them first. Let $\rho = \nu|_{\boldsymbol{V}_i}$, for $k \in C$ let c_k be the number of distinct extensions ρ' of ρ where $A_i\sigma\rho' \in P_k$, let c be the number of such ρ' where $A_i\sigma\rho' \in G$, let $\sigma' = \sigma \cup \rho$, and let $\nu' = \nu \setminus \rho$ (i.e., ν' contains the mappings of all variables in ν not covered by ρ). Since P is strict, we have $c = \sum_k c_k$. It should be clear that $\nu' \in \mathsf{ans}(Q_{i+1}\sigma', G)$, and that the multiplicity w' of ν' satisfies $w = w' \cdot c$. Now consider an arbitrary server

$k \in C$ with $c_k \neq 0$; thus, server k can match atom $A_i\sigma$ so MATCHATOM is called there. By the definition of EVALUATE, we have $\langle \rho, c_k \rangle \in E$ in line 68. Thus, $E \neq \emptyset$ so MATCHATOM does not return in line 69. For each variable $X \in \mathsf{dom}(\rho)$ and position $\pi \in U_X$, the definition of U_X ensures that atom A_j exists such that $j > I_X$ and $X = \mathsf{term}_\pi(A_j)$; but then, ν' being an answer to $Q_{i+1}\sigma'$ ensures $\mu_\pi(\rho(X)) \neq \emptyset$; also, $X \in \mathsf{dom}(\rho)$ ensures $\rho(X) \in \mathsf{voc}(P_k)$, so $\mu_{k,\pi}$ is defined and it satisfies $\mu_{k,\pi}(\rho(X)) \neq \emptyset$ and CANPRUNE(ρ) returns *false* in line 70. Finally, if a recursive call in line 84 were to return a number smaller than i, then some $j > i$ would exist such that $A_j\sigma\xi \notin G$ for all assignments ξ, so ν could not be an answer to $Q_i\sigma$. Hence, MATCHATOM does not return in line 85, so ρ, c_k, and σ' are considered at some point in the body of the loop in lines 70–85 on server k. We next show that the servers jointly produce the required answers.

For the base case $i = n$, server k outputs σ' with multiplicity $m \cdot c_k$ in line 75 or via line 77 and lines 19–26 of Algorithm 2. The multiplicity of σ' aggregated over all servers is $\sum_k m \cdot c_k = m \cdot \sum_k c_k = m \cdot w$, as required.

Now consider the induction step. Vector $\boldsymbol{\lambda}'$ is clearly consistent with $\boldsymbol{\mu}$. Property $(*)$ ensures that the set L in line 79 contains each server ℓ that can match $A_{i+1}\sigma'$; server k sends to each such ℓ a partial answer message in line 82; and, by Claim 1, ℓ eventually processes the message in lines 15–17 of Algorithm 2. By the inductive assumption, then σ' is output with aggregated multiplicity $m \cdot c_k \cdot w'$. Finally, MATCHATOM is called on each server k that can match $A_i\sigma$, so the multiplicity of σ' aggregated over all servers is $\sum_k m \cdot c_k \cdot w' = m \cdot (\sum_k c_k) \cdot w' = m \cdot w$, as required. $\square$