Web Semantics: Science, Services and Agents on the World Wide Web 73 (2022) 100711

Contents lists available at ScienceDirect

Web Semantics: Science, Services and Agents on the World Wide Web

journal homepage: www.elsevier.com/locate/websem

Materialisation and data partitioning algorithms for distributed RDF systems

Temitope Ajileye, Boris Motik *

Department of Computer Science, University of Oxford, UK
Sirius Research Centre, Oslo, Norway

ARTICLE INFO

Article history:
Received 20 August 2021
Received in revised form 15 February 2022
Accepted 23 March 2022
Available online 31 March 2022

Keywords:
Datalog materialisation
Distributed reasoning
Streaming partitioning
RDF

ABSTRACT

Many RDF systems support reasoning with Datalog rules via *materialisation*, where all conclusions of RDF data and the rules are precomputed and explicitly stored in a preprocessing step. As the amount of RDF data used in applications keeps increasing, processing large datasets often requires distributing the data in a cluster of shared-nothing servers. While numerous distributed query answering techniques are known, distributed materialisation is less well understood. In this paper, we present several techniques that facilitate scalable materialisation in distributed RDF systems. First, we present a new distributed materialisation algorithm that aims to minimise communication and synchronisation in the cluster. Second, we present two new algorithms for partitioning RDF data, both of which aim to produce tightly connected partitions, but without loading complete datasets into memory. We evaluate our materialisation algorithm against two state-of-the-art distributed Datalog systems and show that our technique offers competitive performance, particularly when the rules are complex. Moreover, we analyse in depth the effects of data partitioning on reasoning performance and show that our techniques offer performance comparable or superior to the state of the art min-cut partitioning, but computing the partitions requires considerably less time and memory.

1. Introduction

The *Resource Description Framework* (RDF) is a popular data format that allows a domain of interest to be represented in terms of entities called *resources*, and labelled relationships between resources called *triples*. An RDF dataset can be seen as a directed graph in which triples correspond to edges between resources. While answering queries over an RDF dataset is the focus of most RDF applications, *reasoning* capabilities of RDF systems have been growing in importance. RDF reasoning systems take as input an RDF dataset and a formal description of an application domain, which is often captured using a prominent rule-based formalism called *Datalog* [1]. A Datalog *rule* expresses an 'if-then' condition specifying how to derive one or more triples from structural patterns in an RDF dataset. When answering queries, RDF reasoning systems take into account not only the explicitly given triples, but also triples that logically follow from a given set of Datalog rules. The computational properties and the expressivity of Datalog are well understood, which has contributed to wide adoption of Datalog in practice. For example, reasoning in the OWL 2 RL profile of the *Web Ontology Language* (OWL) can be supported either by translating an OWL ontology into rules [2], or by using the fixed rule set from the OWL 2 RL specification [3]. Furthermore, application logic is sometimes captured directly in Datalog rules [4–6]. Thus, developing efficient algorithms for Datalog reasoning over RDF datasets is an active research topic. Datalog reasoning is often supported by *materialisation*: all triples that logically follow from a dataset and a set of rules are precomputed and stored in a preprocessing step, so that queries can be evaluated without referring to the rules. Materialisation is typically realised using the *seminaïve algorithm* [1], which ensures the *nonrepetition property*: no rule is applied to the same triples more than once. This was shown to be essential in practice for even moderately sized datasets.

The size of RDF datasets used in applications has been increasing continuously. For example, the UniProt[1] dataset contains over 34 billion triples; moreover, many applications combine several large datasets. This poses significant challenges to RDF systems that centralise processing on a single computer. The answer is often to partition the data in a cluster of shared-nothing servers, but this introduces considerable complexity: related triples may reside on different servers so network communication may be

* Corresponding author at: Department of Computer Science, University of Oxford, UK.
E-mail addresses: temitope.ajileye@cs.ox.ac.uk (T. Ajileye), boris.motik@cs.ox.ac.uk (B. Motik).

[1] https://www.uniprot.org/.

https://doi.org/10.1016/j.websem.2022.100711

needed. In the context of distributed RDF querying, numerous solutions have been presented and incorporated into systems such as YARS2 [7], 4store [8], H-RDF-3X [9], Trinity.RDF [10], SHARD [11], SHAPE [12], Partout [13], AdPart [14], TriAD [15], SemStore [16], DREAM [17], and WARP [18]. Abdelaziz et al. [19] surveyed 22 and evaluated 11 such systems on a variety of data and query loads, showing AdPart [14] and TriAD [15] to be the best performing.

Distributed reasoners face several problems that are not found in distributed query answering systems: freshly derived triples must participate in all relevant inferences, which can interact with mechanisms for distributing and storing derived triples; moreover, it is essential for the nonrepetition property to be preserved. These issues have been addressed in practice in several different ways. Certain systems handle only fixed Datalog rules: systems by Kaoudi et al. [20] and Weaver and Hendler [21] handle RDFS rules; WebPIE [22] and Cichlid [23] support the so-called *ter Horst fragment* [24]; and SPOWL [25] supports OWL 2 RL rules. While tailoring the reasoning algorithms to specific rules simplifies issues such as nonrepetition of derivations, such solutions are limited in their generality. PLogSPARK [26] can handle arbitrary rules, but it does not seem to use seminaïve evaluation. BigDatalog [27] and Cog [28] implement the seminaïve algorithm, but they seem to be able to process only a few linear rules at a time. Distributed SociaLite [29] implements the seminaïve algorithm for arbitrary Datalog rules. The standard techniques for implementing the seminaïve algorithm require maintaining and copying several auxiliary relations, which can be inefficient in a distributed system. Thus, the tradeoffs in developing algorithms for distributed Datalog reasoning do not yet seem to be fully understood.

Another problem in distributed RDF systems is to partition the data in a way that facilitates efficient distributed computation: intuitively, tightly connected clusters of resources should be placed on a single server in order to reduce communication during both rule matching and fact derivation. Very little attention has been devoted to this problem so far. Most distributed RDF systems use a variant of either subject hashing, where the placement of a triple is determined by hashing the triple's subject, or min-cut partitioning [30], where resources are partitioned to minimise the number of triples spanning two partitions. The former technique is simple to implement, but it does not produce tightly connected partitions; in contrast, min-cut partitioning tends to produce tight partitions, but it requires considerable time and memory and may be infeasible on large datasets. Thus, the question of how to partition the data in distributed RDF reasoning systems is still largely open.

In this paper, we present several novel techniques that provide the foundation for scalable distributed RDF reasoning systems. Our contribution is two-fold.

First, we present a new algorithm for distributed materialisation of Datalog rules over RDF datasets. We build on the work by Potter et al. [31] on distributed query answering using *dynamic data exchange*, from which we inherit several important properties. First, inferences that can be made within a single server are made without any communication; coupled with careful data partitioning, this can significantly reduce network communication overheads. Second, rule evaluation is completely asynchronous, which promotes parallelism. This, however, introduces a complication: to ensure nonrepetition of inferences, we must be able to partially order rule derivations across the cluster. We address this problem using *Lamport timestamps* [32], which allows us to support seminaïve evaluation without expensive maintenance of auxiliary relations. Moreover, dynamic data exchange requires careful maintenance of certain indexes as new facts are derived, which introduces considerable technical difficulties due to asynchronous processing. We present our materialisation algorithm in Section 4.

Second, we consider the problem of partitioning RDF data. We draw our inspiration from the extensive literature on *streaming graph partitioning* algorithms that can process large graphs 'on the fly'. Specifically, such algorithms read a suitable encoding of a graph sequentially (possibly more than once), but their memory use is determined by the number of vertices, rather than the number of edges in the graph. A recent survey [33] identified the HDRF [34] algorithm for streaming partitioning of undirected graphs as particularly suitable for graphs with power-law degree distribution. The more recently proposed 2PS [35] algorithm seems to be able to outperform HDRF in some cases. RDF datasets often contain at least an order of magnitude more triples than resources, so one can expect streaming approaches to be particularly suitable to very large RDF datasets. Thus, in Section 5 we present two new algorithms for streaming partitioning of RDF data that adapt the HDRF and 2PS algorithms to the specifics of RDF. Since subject–subject joins are the most common in RDF queries [36], a key challenge is to ensure that our modified algorithms always place all triples with the same subject on one server.

We have implemented our reasoning and partitioning algorithms in a prototype system called DMAT. In Section 6, we present the results of several experiments that we used to evaluate our techniques. First, we analysed how different data partitioning strategies affect the performance of reasoning. Second, to explore the limits of our approach, we investigated how reasoning performance scales with increasing data loads. Third, to evaluate our reasoning approach against the state of the art, we compared the performance of materialisation in DMAT with that of BigDatalog [27] and Cog [28]. Our results show that our data partitioning algorithms are generally very effective in reducing communication during reasoning, and that this often leads to shorter reasoning times. Moreover, DMAT could handle increasing data loads well, and it outperformed the competition on all benchmarks. Thus, our techniques seem to provide a sound foundation for the development of massively scalable distributed RDF reasoners.

2. Preliminaries

To make this paper self-contained, we now recapitulate the definitions that we use in the rest of this paper.

Syntax. A *constant* is an IRI, a blank node, or a literal. A *term* is a constant or a *variable*. An *atom* is an expression of the form $\langle t_s, t_p, t_o \rangle$ over terms t_s (*subject*), t_p (*predicate*), and t_o (*object*). Let $\Pi = \{s, p, o\}$ be the set of *positions*. Then, given an atom $a = \langle t_s, t_p, t_o \rangle$ and a position $\pi \in \Pi$, $a|_\pi$ is the term that occurs in atom a at position π—that is, $a|_\pi = t_\pi$. A *fact* is a variable-free atom. Whenever no explicit qualification is given, we use lowercase letters from the end of the alphabet ($x, y, z, \ldots$) for variables, lowercase letters from the beginning of the alphabet ($a, b, c, \ldots$) for subject and object constants, and uppercase letters from the middle of the alphabet ($R, S, T, \ldots$) for predicate constants.

An *(RDF) dataset* G is a finite set of facts. The *vocabulary* of G is the set of all constants occurring in G. For c a constant, let $G^+(c) = \{\langle s, p, o \rangle \in G \mid s = c\}$ and $G(c) = \{\langle s, p, o \rangle \in G \mid s = c \text{ or } o = c\}$. Then, $|G^+(c)|$ and $|G(c)|$ are the *out-degree* and the *degree* of c, respectively.

A *query* is a conjunction of atoms of the form (1), where $n \geq 1$ and all a_i are atoms. A Datalog *rule* is an implication of the form (2), where h is the *head* atom, all b_i are *body* atoms, $n \geq 1$, and each variable occurring in h also occurs in some b_i. A Datalog

program is a finite set of rules.

$$a_1 \wedge \cdots \wedge a_n \tag{1}$$

$$h \leftarrow b_1 \wedge \cdots \wedge b_n \tag{2}$$

Note that this definition allows for arbitrarily shaped rules over RDF data. In particular, it includes, but is not limited to, the OWL 2 RL/RDF rules [3], or any subset of these rules such as, for example, the ter Horst fragment [24].

In RDF literature, constants are often called *RDF terms*, atoms are called *triple patterns*, facts are called *triples*, and datasets are called *RDF graphs*. In this paper, however, we adopt the terminology commonly used in the literature on Datalog reasoning.

Substitutions. A *substitution* σ is a partial function that maps finitely many variables to constants. For α a term or an atom, $\alpha\sigma$ is the result of replacing with $\sigma(x)$ each occurrence of a variable x in α on which σ is defined.

Semantics. Let G be a dataset. For Q a query of the form $a_1 \wedge \cdots \wedge a_n$, substitution σ is an *answer* to Q on G if σ is defined precisely on all variables occurring in Q, and $a_i\sigma \in G$ holds for each $1 \leq i \leq n$. The result of applying a rule r of the form $h \leftarrow b_1 \wedge \cdots \wedge b_n$ to G is the set of facts

$$r(G) = G \cup \{h\sigma \mid \sigma \text{ is an answer to } b_1 \wedge \cdots \wedge b_n \text{ on } G\}.$$

For P a program, let $P(G) = \bigcup_{r \in R} r(G)$; let $P^0(G) = G$; and let $P^{i+1}(G) = P(P^i(G))$ for $i \geq 0$. Then, the *closure* of P on G is defined as $P^\infty(G) = \bigcup_{i \geq 0} P^i(G)$. When the closure is computed in advance and persisted, we refer to both $P^\infty(G)$ and the process of computing it as *materialisation*.

Seminaïve Evaluation. We can compute $P^\infty(G)$ using the definition just given: we evaluate the body of each rule $r \in P$ as a query over G and instantiate the head of r for each query answer, we eliminate duplicate facts, and we repeat the process until no new facts are derived. However, $P^i(G) \subseteq P^{i+1}(G)$ holds for each $i \geq 0$, so this *naïve* approach derives in each round of rule application all facts from all previous rounds. The *semïnaive strategy* avoids this problem: when matching a rule r in round $i+1$, at least one body atom of r must be matched to a fact derived in round i. This is critical in practice even for very simple rules.

Dataset Partitions. A *partition* $\mathcal{P}$ of an RDF dataset G is a list of RDF datasets $\mathcal{P} = G_1, \ldots, G_\ell$ such that $G_i \cap G_j = \emptyset$ for $1 \leq i < j \leq \ell$ and $G = \bigcup_{i=1}^{\ell} G_i$. We call datasets G_i *partition elements*. The objective of distributed reasoning is to compute $P^\infty(G)$ using a partition where each partition element is stored in a distinct server in a shared-nothing cluster. For convenience, we identify each server in the cluster by an integer between 1 and ℓ.

Partitioning Problem. A key question in distributed RDF processing is to compute a partition of an RDF dataset that facilitates efficient query processing and reasoning. In the graph partitioning literature, the *vertex replication factor* is often used as a measure of partition 'tightness' [33,34]. This notion can be adapted to RDF as follows: for $\mathcal{P} = G_1, \ldots, G_\ell$ a partition of an RDF dataset, the *replication set* of a constant c is defined as $A(c) = \{k \mid G_k \cap G(c) \neq \emptyset\}$, and the *replication factor* of a partition $\mathcal{P}$ is defined as

$$\mathsf{RF}(G, \mathcal{P}) = \frac{1}{|V|} \sum_{c \in V} |A(c)| \tag{3}$$

where V is the vocabulary of G. In other words, the replication factor is the average number of servers that constants occur on. The term 'replication' is sometimes used in the RDF literature to denote the idea of storing the same fact in more than one server, but this is not the intended meaning in the literature on graph partitioning and our work; in fact, partition elements in this paper are all pairwise disjoint.

Given a fixed tolerance parameter $\alpha \geq 1$, the objective of graph partitioning is to compute a partition $\mathcal{P}$ of an RDF dataset G such that $|G_i| \leq \alpha \frac{|G|}{\ell}$ holds for each $1 \leq i \leq \ell$, while minimising the replication factor $\mathsf{RF}(G, \mathcal{P})$. In other words, each G_i should hold roughly the same number of facts, while ensuring that constants are replicated as little as possible. Solving this problem exactly is computationally hard, so the objective is usually weakened in practice. The algorithms we present in this paper will honour the restrictions on the sizes of G_i, and they will aim to make the replication factor small, but without firm minimality guarantees.

3. Related work

Before presenting our contribution, we first discuss the relevant related work. In Section 3.1, we discuss the existing distributed Datalog reasoning systems and approaches. In Section 3.2, we survey the work by Potter et al. [31] on distributed query answering with *dynamic data exchange*, which provides the starting point for our distributed reasoning algorithm. In Section 3.3, we discuss the data partitioning approaches typically used in distributed RDF systems.

3.1. Approaches to distributed reasoning

A number of approaches were developed in the 90 s for materialising Datalog programs when the data is distributed across several processors. These approaches are not specific to RDF, but we shall discuss them in our setting. The key idea is to partition rule applications to processors. For example, to evaluate $\langle x, R, z\rangle \leftarrow \langle x, R, y\rangle \wedge \langle y, R, z\rangle$ on ℓ processors, we let each processor i with $1 \leq i \leq \ell$ evaluate rule

$$\langle x, R, z\rangle \leftarrow \langle x, R, y\rangle \wedge \langle y, R, z\rangle \wedge h(y) = i, \tag{4}$$

where $h(y)$ is a *partition function* that maps values of y to integers between 1 and ℓ. If h is uniform and constants are uniformly distributed across triples, then each processor receives roughly the same fraction of the workload. Practical experience suggests that such an approach can often parallelise computation very effectively. However, since a fact of the form $\langle s, R, o\rangle$ can match either atom in the body of rule (4), each such fact must be replicated to processors $h(s)$ and $h(o)$ to ensure completeness. Based on this idea, Ganguly et al. [37] show how to handle general Datalog; Zhang et al. [38] study different partition functions; Seib and Lausen [39] identify programs and partition functions where no replication of derived facts is needed; Shao et al. [40] further break rules in segments; and Wolfson and Ozeri [41] replicate all facts to all processors in order to increase parallelism. The primary motivation behind these approaches seems to be parallelisation of computation, which explains why the high rates of data replication were deemed acceptable.

Materialisation can also be implemented without any data replication. First, one must select a data partitioning strategy: a common approach is to assign each triple $\langle s, p, o\rangle$ to server $h(s)$ using a suitable hash function h, and another popular option is to use a distributed file system (e.g., HDFS) and thus leverage its partitioning mechanism. Second, the rule bodies are evaluated using a distributed query evaluation algorithm, the newly derived facts are distributed according to the partitioning strategy, and the process is repeated iteratively as long as fresh facts are derived.

These principles were applied to reasoning over RDF datasets. Most systems in this category handle only fixed rule sets (i.e., the rules are hardcoded in the algorithms and cannot be changed). The systems by Weaver and Hendler [21] and Kaoudi et al. [20] support RDFS reasoning. Other systems borrow mechanisms for distributed data storage and query evaluation from big data frameworks such as Hadoop and Spark. In particular, WebPIE [22]

supports the *ter Horst fragment* [24] in Hadoop; Cichlid [23] also supports ter Horst fragment, but in Spark; and SPOWL [25] supports extensions of the OWL 2 RL rules in Spark. Handling only fixed rule sets considerably simplifies the design of reasoning algorithms. For example, seminaïve evaluation is not needed for RDFS reasoning since nonrepetition of inferences can be ensured by evaluating the rules in a particular order. Greater generality is offered by PLogSPARK [26], which handles general Datalog rules over RDF data in Spark. However, this system seems to use naïve rule evaluation, which can be prohibitive when the rules are complex.

Distributed Datalog reasoning has also been studied in the database community. BigDatalog [27] and Cog [28] were implemented on top of Spark and Flink, respectively, but they target a slightly different setting. Both systems accept as input a Datalog program and a query. They then select the rules relevant to the query, compute their materialisation, and evaluate the given query over the materialisation. However, the materialisation is not persisted: it is treated as a temporary object used to just answer the query. Both systems claim to support arbitrary Datalog and seminaïve evaluation, but, as we report in Section 6, we managed to use them with only a handful of linear rules. Yedalog [42] is Google's private implementation of Datalog on top of a proprietary data model. Distributed SociaLite [29] implements distributed seminaïve materialisation for general Datalog, but users must explicitly specify the data distribution strategy and communication patterns. For example, by writing a fact $R(a, b)$ as $R[a](b)$, one specifies that the fact should be stored on server $h(a)$ for some hash function h. Rule (4) can then be written in SociaLite as $R[x](z) \leftarrow R[x](y) \wedge R[y](z)$, specifying that the rule should be evaluated by sending each fact $R[a](b)$ to server $h(b)$, joining such facts with $R[b](c)$, and sending the resulting facts $R[a](c)$ to server $h(a)$. While the evaluation of some of these rules can be parallelised, all servers in a cluster must synchronise after each round of rule application. Yedalog and SociaLite also implement extensions of Datalog, such as monotonic aggregation and builtin predicates.

3.2. Dynamic data exchange

Distributed query answering is a key ingredient of distributed reasoning algorithms. Most distributed query evaluation algorithms can be understood as using a variant of the data exchange operators first introduced in the Volcano system [43]. Such operators send and receive partial query answers during query evaluation, and are introduced into query plans to ensure that all partial answers that may participate in a join are transmitted to one server.

Recently, Potter et al. [31] presented a distributed query evaluation algorithm where data exchange is *dynamic*: communication is guided by the data encountered during query evaluation, rather than being fixed at query compilation time. The objectives of dynamic data exchange are to reduce communication and eliminate synchronisation between servers. To this end, each server k maintains three indexes called *occurrence mappings*. For each constant c occurring in G_k, occurrence mapping $\mu_{k,s}(c)$ contains all servers where c occurs in the subject position, and occurrence mappings $\mu_{k,p}(c)$ and $\mu_{k,o}(c)$ provide analogous information for the predicate and object positions. Occurrence mappings thus allow a server to locate all facts containing a particular constant during query answering; this is reminiscent of decentralised indexes in peer-to-peer RDF systems by Aebeloe et al. [44].

To make these ideas concrete, consider evaluating query (5) over partition elements (6) and (7).

$$Q = \langle x, R, y\rangle \wedge \langle y, R, z\rangle \tag{5}$$

Table 1
Example occurrence mappings on servers 1 and 2.

$\mu_{1,s} = \{$	$a \mapsto \{1\},$	$b \mapsto \{1, 2\},$	$c \mapsto \emptyset,$ $R \mapsto \emptyset$ $\}$
$\mu_{1,p} = \{$	$a \mapsto \emptyset,$	$b \mapsto \emptyset,$	$c \mapsto \emptyset,$ $R \mapsto \{1, 2\}\}$
$\mu_{1,o} = \{$	$a \mapsto \emptyset,$	$b \mapsto \{1\},$	$c \mapsto \{1\},$ $R \mapsto \emptyset$ $\}$
$\mu_{2,s} = \{$	$b \mapsto \{1, 2\},$	$d \mapsto \{2\},$	$e \mapsto \emptyset,$ $R \mapsto \emptyset$ $\}$
$\mu_{2,p} = \{$	$b \mapsto \emptyset,$	$d \mapsto \emptyset,$	$e \mapsto \emptyset,$ $R \mapsto \{1, 2\}\}$
$\mu_{2,o} = \{$	$b \mapsto \{1\},$	$d \mapsto \{2\},$	$e \mapsto \{2\},$ $R \mapsto \emptyset$ $\}$

$$G_1 = \{\langle a, R, b\rangle, \langle b, R, c\rangle\} \tag{6}$$

$$G_2 = \{\langle b, R, d\rangle, \langle d, R, e\rangle\} \tag{7}$$

The occurrence mappings are initialised as shown in Table 1. In particular, the occurrence mappings on each server must cover all constants occurring in this server; thus, $\mu_{1,s}$, $\mu_{1,p}$, and $\mu_{1,o}$ are defined on constants a, b, c, and R, whereas $\mu_{2,s}$, $\mu_{2,p}$, and $\mu_{2,o}$ are defined on constants b, d, e, and R.

Query processing is started by sending the query to the two servers, each storing a partition element. Each server independently evaluates Q over its partition using index nested loop joins. Thus, server 1 evaluates atom $\langle x, R, y\rangle$ over G_1, which produces a *partial* answer $\sigma_1 = \{x \mapsto a, y \mapsto b\}$. Server 1 then evaluates $\langle y, R, z\rangle\sigma_1 = \langle b, R, z\rangle$ over G_1 and thus obtains one full answer $\sigma_2 = \{x \mapsto a, y \mapsto b, z \mapsto c\}$. To see whether $\langle b, R, z\rangle$ can be matched on other servers, server 1 consults its occurrence mappings for all constants in the atom. Since $\mu_{1,s}(b) = \mu_{1,p}(R) = \{1, 2\}$, server 1 sends the partial answer σ_1 to server 2, telling it to continue matching the query from the second atom. After receiving σ_1, server 2 matches atom $\langle b, R, z\rangle$ in G_2 to obtain another full answer $\sigma_3 = \{x \mapsto a, y \mapsto b, z \mapsto d\}$. Server 2 also evaluates $\langle x, R, y\rangle$ over G_2 and obtains partial answer $\sigma_4 = \{x \mapsto b, y \mapsto d\}$, and it consults its occurrences to determine which servers can match $\langle y, R, z\rangle\sigma_4 = \langle d, R, z\rangle$. Since $\mu_{2,s}(d) = \{2\}$, server 2 knows it is the only one that can match this atom, so it proceeds without any communication and computes $\sigma_5 = \{x \mapsto b, y \mapsto d, z \mapsto e\}$.

This strategy has several important benefits. First, all answers that can be produced within a single server, such as σ_5 in our example, are produced without any communication. Second, the location of every constant is explicitly recorded, rather than computed using a fixed rule such as a hash function. An RDF dataset can thus be partitioned based on its structural properties, and highly interconnected constants can all be placed on one server with the aim of reducing network communication. Third, the system is fully asynchronous: when server 1 sends σ_1 to server 2, server 1 does not need to to wait for server 2 to finish, and server 2 can process σ_1 whenever it can. The lack of synchronisation between servers is beneficial to parallelisation.

3.3. Data partitioning

Reducing I/O cost via judicious data partitioning has a long tradition in RDF systems; for example, Aluç et al. [45] proposed a self-tuneable data partitioning scheme that aims to optimise on-disk placement of data. However, while it is intuitive to expect that partitioning the data carefully to minimise communication would improve the performance of distributed systems, the effects of data partitioning remain poorly understood. Janke et al. [46] studied this problem in the context of distributed query processing. Interestingly, they concluded that reducing communication can be detrimental if done at the expense of uneven server workload. However, it is unclear to what extent their conclusions apply to distributed reasoning. Reasoning over large datasets involves evaluating millions of queries and distributing derived facts, both of which can incur much more communication than

in the case of a single query. Moreover, workload imbalances in individual queries could even themselves out when many queries are evaluated.

Existing approaches to data partitioning can be broadly divided into three groups. The first groups consists of systems that store their data in a distributed file system. Data is usually allocated randomly to servers, which makes exploiting data locality during reasoning difficult. The second group consists of hash-based variants, where the location of a fact is determined by hashing one or more of the fact's components (usually subject). The third group consists of variants based on min-cut graph partitioning, which aims to reduce communication by minimising the number of edges between partitions. Since subject–subject joins were shown to be very common [36], most systems in the latter two groups colocate triples with the same subjects.

Distributed RDF systems sometimes use *data replication*, where some or all facts are stored on more than one server. The decision which facts to replicate is typically made during data loading. A prominent approach is *n-hop replication*, where facts are replicated so that all path queries of length n or less can be processed without any communication [47]. Other approaches to replicating data during loading have been considered too [12,15]. A more recent approach aims to analyse the query load and replicate commonly used triples on the fly [14]. While data replication can be effective at reducing communication in a distributed RDF system, it can also significantly increase the storage requirements; for example, it was shown that undirected two-hop replication can increase the storage requirements by a factor of 4.2 [47]. In reasoning systems, data replication can increase the communication needed to store newly derived facts, and it may lead to redundant derivations. Since our main objective is to devise a distributed reasoning algorithm that does not repeat derivations, we do not consider data replication in our work. In fact, we are unaware of any related approach to distributed reasoning that uses data replication.

4. Distributed materialisation algorithm

We now present our distributed materialisation algorithm. Before presenting the technical details, in Section 4.1 we discuss our main technical challenges. Then, in Section 4.2 we discuss data structures used to store facts; in Section 4.3 we introduce the occurrence mappings; in Section 4.4 we discuss the communication infrastructure; in Section 4.5 we discuss how to detect termination; in Section 4.6 we present the algorithm's pseudocode and argue about its correctness; and in Section 4.7 we illustrate various aspects of our algorithm using a simple example.

4.1. Technical challenges

As mentioned in Section 2, seminaïve evaluation is critical to ensuring practicability of materialisation. This algorithm evaluates rule bodies as queries, so it may be tempting to try to adapt seminaïve evaluation to a distributed setting by switching to a distributed query answering algorithm. We discuss the problems of such a straightforward approach by means of the example rule (8).

$$\langle x, R, z\rangle \leftarrow \langle x, R, y\rangle \wedge \langle y, R, z\rangle \tag{8}$$

The objective of seminaïve evaluation is to avoid repeating derivations, where a derivation refers to matching the body of a rule to a particular set of facts. For example, matching rule (8) to facts $\langle a, R, c\rangle$ and $\langle c, R, b\rangle$, and to facts $\langle a, R, d\rangle$ and $\langle d, R, b\rangle$, constitutes two distinct derivations. Seminaïve evaluation ensures that each of these derivations is made just once, but it does not prevent both derivations from producing $\langle a, R, b\rangle$: a separate duplicate elimination step is required to deal with that issue. To this end, rules are applied in rounds, but in each rule application at least one body atom is matched to a fact derived in the previous round. A common way to ensure this is to maintain four sets of facts: the set of 'current' facts, the set of 'old' facts derived before the last round, the 'delta' set of facts that are present in the 'current' but not the 'old' set, and the set of 'new' facts derived in each round of rule application. Rule (8) is then rewritten to use these sets as follows.

$$\langle x, R, z\rangle^{new} \leftarrow \langle x, R, y\rangle^{\Delta} \wedge \langle y, R, z\rangle \tag{9}$$

$$\langle x, R, z\rangle^{new} \leftarrow \langle x, R, y\rangle^{old} \wedge \langle y, R, z\rangle^{\Delta} \tag{10}$$

Thus, rule (9) produces all derivations of rule (8) where atom $\langle x, R, y\rangle$ in the latter rule is matched to a fact derived in the previous round of rule application. Rule (9) handles the second body atom of rule (8) analogously; by matching the first body atom of rule (10) to the 'old' facts, we ensure no repetition of inferences between the two rules. Rules (9) and (10) can be evaluated using any distributed query evaluation strategy. Each round of rule application is followed by a round of updates: the 'delta' facts are added to the 'old' facts, and the difference between the 'new' and the 'current' facts is copied to the 'delta' facts and added to the 'current' facts. This can be inefficient for at least two reasons. First, the update round requires shuffling of data between datasets, which can be costly; this can be particularly acute in systems built on top of Hadoop and Spark, where data storage and distribution is managed automatically by a distributed file system. Second, rounds must be synchronised (i.e., updates cannot begin before rule application finishes and vice versa), which can lead to workload skew.

We address these problems by drawing inspiration from the work by Motik et al. [48] on parallelising Datalog materialisation in centralised, shared memory systems. Instead of applying rules in rounds, this approach considers each fact in the dataset, identifies each rule and body atom that can be matched to the fact, and evaluates the rest of the rule as a query. Moreover, all facts are ordered totally based on the sequence of their derivation; this order can be recovered easily from how facts are stored. To prevent inference repetition, each query is evaluated only against the facts derived *before* the fact being processed. Such an approach does not proceed in rounds and does not need an explicit update step; rather, rules are applied asynchronously, which is beneficial for parallelisation: the number of facts is generally very large, so the materialisation process decomposes naturally into a large number of completely independent steps that can be evaluated in parallel. The approach has been successfully applied to very large datasets [49].

Our distributed materialisation algorithm is based on the same general principle: each server in a cluster matches the rules to locally stored facts, but the resulting queries are evaluated using dynamic data exchange. Our approach thus requires no synchronisation between servers, and communication is reduced in the same way as described in Section 3.2. We thus expect the same benefits as for the query answering algorithm by Potter et al. [31]. However, the lack of synchronisation between servers introduces a complication: since there is no notion of a rule application round, it is not obvious how to guarantee nonrepetition of inferences. A straightforward solution might be to associate each fact with a timestamp recording when the fact was derived so that the order of fact derivation can be recovered; however, this would require maintaining a high coherence of server clocks in the cluster, which is generally impractical. In Section 4.2, we discuss how we solve this problem using Lamport timestamps [32]—a well known, simple way of determining a partial order of events across a cluster.

Another complication arises due to the observation that occurrence mappings may need updating when new facts are derived. The occurrence mappings of all relevant servers must be updated before any rule is applied to such newly derived facts; otherwise, such facts might be skipped during query evaluation, which would jeopardise completeness. Our solution to this problem is again fully asynchronous.

Finally, since no central coordinator keeps track of the state of the computation of different servers, detecting when the system as a whole can terminate is not straightforward. We solve this problem using a well-known termination detection algorithm based on token passing [50].

4.2. Lamport timestamps

To prevent repetition of derivations, we need to arrange all derived facts into a sequence that reflects the derivation order. As mentioned already, maintaining a precise global clock in a distributed system is very challenging, so we instead use Lamport timestamps [32], a general and simple technique for ordering events in a distributed system. In particular, each event is annotated with an integer timestamp in a way that guarantees the following property:

> [$*$] if there is any way for an event A to possibly influence an event B, then the timestamp of A is strictly smaller then the timestamp of B.

To achieve this, each server keeps an integer counter called the *local clock* (even though this counter does not measure time), which is incremented whenever an event of interest occurs; this clearly ensures property ($*$) for events A and B occurring in the same server. Now assume that events A and B occur in servers 1 and 2, respectively; clearly, A can influence B only if server 1 sends a message to server 2, and server 2 processes this message before event B takes place. To ensure property ($*$), server 1 includes its current clock value into the message it sends to server 2; moreover, when processing this message, server 2 updates its local clock to the maximum of the message clock and the local clock, and then increments the local clock. Thus, when event B happens after receiving the message, the timestamp of the event is guaranteed to be larger than the timestamp of event A.

To apply this idea to Datalog materialisation, a derivation of a fact corresponds to the notion of an event, and using a fact to derive another fact corresponds to the 'influences' notion. Thus, we associate facts with integer timestamps.

More precisely, each server k in the cluster maintains an integer C_k called the *local clock*, a dataset G_k of facts stored in the server, and a *timestamp function* $T_k : G_k \to \mathbb{N}$ that associates each fact in the server with a natural number. Before materialisation, C_k is initialised to zero, and all input facts (i.e., the facts given by the user) assigned to server k are loaded into G_k and assigned a zero timestamp.

To capture formally how timestamps are used during query evaluation, we introduce the notion of an *annotated query* as a conjunction of the form

$$Q = a_1^{\bowtie_1} \wedge \cdots \wedge a_n^{\bowtie_n}, \tag{11}$$

where each $a_i^{\bowtie_i}$ is an *annotated atom* consisting of an atom a_i and a symbol $\bowtie_i$ that can be $<$ or $\leq$. Answers to an annotated query are computed with respect to a timestamp. More precisely, a substitution σ is an answer to Q on a dataset G_k and function T_k w.r.t. an integer timestamp τ if

- σ is an answer to the 'ordinary' query $a_1 \wedge \cdots \wedge a_n$ on dataset G_k as usual, and
- $T_k(a_i\sigma) \bowtie_i \tau$ holds for each $1 \leq i \leq n$.

For example, let Q, G, and T be as below, and let $\tau = 2$ be a timestamp.

$$\begin{aligned} Q &= \langle x, R, y\rangle^{<} \wedge \langle y, S, z\rangle^{\leq} \\ G &= \{\langle a, R, b\rangle, \langle b, S, c\rangle, \langle b, S, d\rangle\} \\ T &= \{\langle a, R, b\rangle \mapsto 1,\ \langle b, S, c\rangle \mapsto 2,\ \langle b, S, d\rangle \mapsto 3\} \end{aligned}$$

Then, $\sigma_1 = \{x \mapsto a, y \mapsto b, z \mapsto c\}$ is an answer to Q on G and T w.r.t. τ, but $\sigma_2 = \{x \mapsto a, y \mapsto b, z \mapsto d\}$ is not an answer to Q on G and T w.r.t. τ due to $T(\langle b, S, d\rangle) > 2$.

We assume that each server can evaluate one annotated atom—that is, server k can call Evaluate($a^{\bowtie}, \tau, G_k, T_k, \sigma$), where $a^{\bowtie}$ is an annotated atom, τ is a timestamp, G_k is the dataset stored in the server, T_k provides timestamps for the facts in G_k, and σ is a substitution. The call returns each substitution ρ defined over the variables in a and σ such that $\sigma \subseteq \rho$, $a\rho \in G_k$, T_k is defined on $a\rho$, and $T(a\rho) \bowtie \tau$ holds. In other words, Evaluate matches $a^{\bowtie}$ on G_k, T_k, and τ, and it returns each match that extends σ and satisfies $a^{\bowtie}$ and τ. For efficiency, server k should index the facts in G_k. Any RDF indexing scheme can be used, and index lookup can be modified to skip facts whose timestamps do not match τ.

Finally, we assume that each server implements a function MatchRules(f, P) that takes as input a fact f and a Datalog program P. For each rule $h \leftarrow b_1 \wedge \cdots \wedge b_n$ in P, each body atom b_p with $1 \leq p \leq n$, and each substitution σ over the variables of b_p such that $b_p\sigma = f$, the function returns the 4-tuple (σ, b_p, Q, h) where

$$Q = b_1^{<} \wedge \cdots \wedge b_{p-1}^{<} \wedge b_{p+1}^{\leq} \wedge \cdots \wedge b_n^{\leq}. \tag{12}$$

Intuitively, MatchRules identifies each rule and each *pivot* body atom b_p that can be matched to f via substitution σ. This σ will be extended to all body atoms of the rule by recursively matching all remaining atoms using function Evaluate. The annotations in (12) specify how to match the remaining atoms without repetition: facts matched before the pivot must have timestamps strictly smaller than the timestamp of f, and facts matched after the pivot must have timestamps strictly smaller or equal to the timestamp of f. The atoms of query (12) may need to be reordered to obtain an efficient query plan. This can be achieved using any known query planning technique, and further discussion of this issue is out of scope of this paper.

4.3. Occurrence mappings

As in the query answering approach based on dynamic data exchange [31], each server k must store indexes $\mu_{k,s}$, $\mu_{k,p}$, and $\mu_{k,o}$ called *occurrence mappings*. Each of these maps constants to (possibly empty) sets of integers identifying servers. Intuitively, $\mu_{k,s}(c)$ can be used on server k to identify the servers on which constant c occurs as the subject of a fact. However, requiring each server to track the location of all constants would mean that the servers' memory use is determined by the size of the entire RDF dataset, rather than the size of the partition element stored in the servers. To remedy this, we allow each server to track locations of certain constants; the exact conditions on which constants must be tracked are formalised in Definition 4.1. Roughly speaking, we require each server to keep track only of constants occurring in the partition element stored in the server, so the size of each occurrence mapping is bounded by the size of the corresponding partition element.

Definition 4.1. *Let P be a program, let $\boldsymbol{G} = G_1, \ldots, G_\ell$ be datasets, and let $\boldsymbol{\mu} = \mu_{1,s}, \mu_{1,p}, \mu_{1,o}, \ldots, \mu_{\ell,s}, \mu_{\ell,p}, \mu_{\ell,o}$ be a collection of occurrence mappings.*

A constant c is relevant *for P, $\boldsymbol{G}$, and index $k \in [1, \ell]$ if c occurs in the head of a rule of P or in G_k at any position.*

Occurrence mappings $\boldsymbol{\mu}$ *are* consistent *with* P *and* $\boldsymbol{G}$ *if* $j \in \mu_{k,\pi}(c)$ *holds for all* $k, j \in [1, \ell]$, *each position* $\pi \in \Pi$, *and each constant* c *that is relevant for* P, $\boldsymbol{G}$, *and* k *and that occurs at position* π *of a fact in* G_j.

Occurrence mappings $\boldsymbol{\mu}$ *satisfy the* subject constraint *if, for each constant* c *and each server* k, *set* $\mu_{k,s}(c)$ *contains at most one element, and* $j \in \mu_{k,s}(c)$ *implies that constant* c *occurs in subject position of a fact in* G_j.

Occurrence mappings $\boldsymbol{\mu}$ *are* correct *for* P *and* $\boldsymbol{G}$ *if* $\boldsymbol{\mu}$ *are consistent with* P *and* G *and satisfy the subject constraint.*

To simplify the presentation of our algorithm, we assume that each $\mu_{k,\pi}$ is defined on all constants. In practice, an implementation can choose to explicitly store only the values for the relevant constants, and use the empty set as the default value for all other constants. Moreover, occurrence mappings of one server are shared and possibly updated by multiple threads of execution during materialisation. We assume that mappings are accessed atomically: whenever $\mu_{k,\pi}(c)$ is used in our algorithm, the result is the value of mapping $\mu_{k,\pi}$ on c at a point in time.

We illustrate Definition 4.1 by an example involving the following partition elements G_1 and G_2 and rule r.

$$G_1 = \{\langle a, R, b\rangle, \langle a, R, d\rangle, \langle d, S, c\rangle\} \tag{13}$$

$$G_2 = \{\langle b, S, a\rangle, \langle b, S, c\rangle\} \tag{14}$$

$$r = \langle z, T, x\rangle \leftarrow \langle x, R, y\rangle \wedge \langle y, S, z\rangle \tag{15}$$

Constant b is relevant to server 1 because b occurs in G_1 (at any position). Consistency ensures that server 1 has complete information about the whereabouts of constant b—that is, $1 \in \mu_{1,o}(b)$ and $2 \in \mu_{1,s}(b)$ must hold. Intuitively, since constant b is relevant to server 1, there is a chance that the body of some rule can be matched to a fact containing b, which can instantiate a variable in the rest of the rule body. In our example, atom $\langle x, R, y\rangle$ in rule r can be matched to $\langle a, R, b\rangle$, which instantiates the second atom to $\langle b, S, z\rangle$. To continue matching the rule body, $2 \in \mu_{1,s}(b)$ is needed to instruct server 1 to forward the partial match to server 2 so that the latter server can potentially complete the match. Thus, consistency ensures completeness of rule body evaluation by placing a lower bound on occurrence mappings. Note that, say, $2 \in \mu_{1,o}(d)$ is allowed to hold even though constant d does not occur in G_2: this can only make server 1 send superfluous partial matches to server 2.

The subject constraint consists of two parts. The first part ensures that all facts with the same subject are assigned to one server. In our example, consistency ensures $1 \in \mu_{1,s}(a)$, and the subject constraint ensures $\mu_{1,s}(a) = \{1\}$—that is, all facts where a occurs in the subject position must occur in G_1. Most distributed RDF systems group facts by subject for performance (see Section 3.3), but our approach uses this additionally to send each derived fact to the server containing the fact's subject. For example, when server 2 completes the match from the previous paragraph and derives $\langle a, T, a\rangle$, it knows that the fact should be sent to server 1. The second part of the subject constraint ensures that the subject occurrence mappings do not contain superfluous servers, which is also important for correct placement of derived facts. Note that both servers 1 and 2 can derive $\langle c, T, d\rangle$. Now let us assume that $\mu_{1,s}(c) = \{1\}$ (which is allowed by the consistency condition), so the fact is recorded on server 1. In contrast, let us assume that $\mu_{2,s}(c) = \emptyset$ (which also satisfies consistency). Then, server 2 has no information about where to send $\langle c, T, d\rangle$, so it determines the destination by hashing the fact's subject. Now hashing c can result in the fact being sent to server 2, which would break the first part of the subject constraint. To remedy this, our approach requires the occurrences for the subject position to be exact.

The correctness condition just combines consistency and the subject contrast, and ensuring that this property remains preserved as new facts are derived is a key source of technical difficulty in our approach.

Allowing servers to track the location of relevant constants only introduces one complication: when server k receives a partial match σ from another server, the occurrence mappings stored in server k may not cover all constants in σ. Potter et al. [31] solve this by accompanying each partial match σ with a vector $\boldsymbol{\lambda} = \lambda_s, \lambda_p, \lambda_o$ of *partial occurrences*. Whenever a server extends σ by matching an atom, it also records in $\boldsymbol{\lambda}$ its local occurrences for each constant added to σ that can be used in the rest of the rule body. Occurrences of the matched constants are propagated together with partial matches, which ensures that each server has access to occurrences of constants in atoms that are yet to be matched.

4.4. Communication infrastructure and messages

We assume that each server in the cluster can send a message m to a destination server d by calling SEND(m, d). This function can return immediately, and the receiver can process the message later—that is, communication can be asynchronous. Also, our core algorithm is correct as long as each sent message is processed eventually—that is, messages sent between a pair of servers can be processed in an arbitrary order without affecting correctness. However, the approach used to detect termination (which is largely orthogonal to our core algorithm) can introduce other message types and might impose constraints on the order of message processing; we discuss this in more detail in Section 4.5.

Message PAR$[i, \sigma, Q, h, \tau, \boldsymbol{\lambda}]$ informs a server that σ is a partial match obtained by matching some fact with timestamp τ to the body of a rule with head atom h; moreover, the remaining atoms to be matched are given by an annotated query Q starting from the atom with index i. The partial occurrences of the constants in σ that may be needed when matching the remaining atoms of Q are recorded in $\boldsymbol{\lambda}$.

Message FCT$[f, \tau, \boldsymbol{\lambda}]$ says that f is a new fact to be stored at server processing the message. Timestamp τ reflects when the message was sent, and $\boldsymbol{\lambda}$ records the partial occurrences of the constants in f.

Message OCC$[f, D, k_h, \tau, \boldsymbol{\lambda}]$ says that f is a new fact to be stored at server k_h. Set D identifies servers whose occurrences may need updating before f can be added to G_{k_h}. Timestamp τ reflects when the message was sent, and $\boldsymbol{\lambda}$ records the partial occurrences of the constants in f.

Potter et al. [31] observed that PAR messages correspond to partial join results so a large number of such messages can be produced during query evaluation. For asynchronous processing, the PAR messages may need to be buffered on the receiving server, which can require considerable memory. They also presented a flow control mechanism that can restrict memory consumption at each server without jeopardising completeness. This solution is directly applicable in our approach as well, so we do not discuss it further.

4.5. Ensuring termination

Termination of our approach follows in the same way as in a centralised setting. In particular, the set of facts stored in each server grows monotonically and duplicate facts are removed; since the number of constants in the input dataset is finite, each server can derive at most finitely many facts. Analogously, occurrence mappings grow monotonically as well, so each server can end up containing the maximal occurrence mappings, which are clearly finite as well. Thus, at some point in time, each server will

run out of facts to process, at which point it will not generate any further messages. Once all messages queued in the system have been processed, all servers can terminate. However, no server in our approach keeps track of the progress of other servers, so detecting that all servers are idle is not straightforward. We next summarise a well-known solution to this problem.

When messages between each pair of servers are guaranteed to be processed in the order in which they are sent (as is the case in our implementation), one can use Dijkstra's token ring algorithm [50]. All servers in the cluster are numbered from 1 to ℓ and are arranged in a ring (i.e., server 1 comes after server ℓ). Each server can be black or white, and the servers will pass between them a *token* that can also be black or white. Initially, all servers are white and server 1 has a white token. The algorithm proceeds as follows.

- When server 1 has the token and it becomes idle (i.e., it has no pending work or messages), it sends a white token to the next server in the ring.
- When a server other than 1 has the token and it becomes idle, the server changes the token's colour to black if the server is itself black (and it leaves the token's colour unchanged otherwise); the server forwards the token to the next server in the ring; and the server changes its colour to white.
- A server i turns black whenever it sends a message (i.e., not just a token message) to a server $j < i$.
- All servers can terminate when server 1 receives a white token.

The Dijkstra–Scholten algorithm [51] can be used when messages sent between a pair of servers are not guaranteed to be processed in the order in which they are sent. We do not use this algorithm in our implementation, so we do not discuss the details any further.

4.6. The algorithm

With these definitions in mind, Algorithm 1 presents our approach to distributed Datalog materialisation. Before starting, each server k receives a copy of the program P to be materialised, loads the corresponding partition element into its local store G_k, sets the timestamp of each fact in G_k to zero, initialises its occurrence mappings $\mu_{k,s}$, $\mu_{k,p}$, and $\mu_{k,o}$ to be correct for P and $G_1, \ldots, G_\ell$ as per Definition 4.1, and initialises its local clock C_k to zero. The server then starts an arbitrary number of server threads, each executing the ServerThread function. Each thread repeatedly processes an unprocessed fact f in G_k or an unprocessed message m; if both are available, ties are broken arbitrarily. Otherwise, termination is checked as discussed in Section 4.5.

The core of our approach involves matching body atoms of the rules in the program. Rule matching on server k can commence in one of the following two ways. First, an unprocessed fact f can be extracted from the partition element G_k and passed to the ProcessFact function. To start matching the rules to f, our algorithm calls MatchRules to identify all rules where one pivot atom matches to f via a partial answer σ, and it uses the FinishMatch function to extend σ to a full answer. Second, a PAR message can be received. The message contains a partial answer σ, and it instructs the server to continue matching a rule body from atom i. Thus, the server evaluates atom $a_i^{\bowtie_i}$ in G_k and T_k w.r.t. τ to enumerate all partial answers σ', and for each it uses the FinishMatch function to extend σ' to a full answer.

Function FinishMatch finishes matching atom a_{last} by (i) extending $\boldsymbol{\lambda}$ with the occurrences of all constants that might be relevant for the remaining body atoms or the rule head, and (ii) either matching the next body atom or deriving the rule head. For the former, when variable x is matched to a constant c, the occurrences of c can be needed to match the rest of the rule only if x occurs in the remaining body atoms or in the head atom of the rule being matched. Therefore, the algorithm identifies in line 17 each such variable x and adds the occurrences of $x\sigma$ to λ_π for each position π. Now if Q has been matched completely (line 18), the server also ensures that the partial occurrences are correctly defined for the constants occurring in the rule head (lines 19–20), it identifies the server k_h that should receive the derived fact as described in Section 4.3, and sends the FCT message to k_h. Otherwise, atom $a_{i+i}\sigma$ must be matched next. To determine the set D of servers that could possibly match $a_{i+i}\sigma$, server k intersects the occurrences of each constant from $a_{i+i}\sigma$ (line 26) and sends a PAR message to all servers in D.

A FCT message informs server k that fact f is a newly derived fact that should be added to G_k. The main challenge is to ensure that adding f to G_k does not affect the correctness condition from Definition 4.1. To this end, our algorithm identifies in lines 31–36 the set of servers D whose occurrences might need updating. In particular, if f contains a constant $c = f|_\pi$ at position π and server k does not occur in the local occurrences $\mu_{k,\pi}(c)$, then all servers containing constant c at any position might need updating (line 36); moreover, if c occurs in the head of a rule in P, then all servers need to be informed of the location of c (line 34). Once the set D of candidate servers has been constructed, an OCC message is sent to some server in D; since the occurrences of all servers must be updated before fact f is added to partition element G_k, the message is sent to server k_h only if k_h is the only remaining server in D.

An OCC message informs server k that fact f will be added to G_{k_h} so the occurrences of the constants in f might need updating. Set D lists the remaining servers that must be informed. A key difficulty arises when constant c becomes relevant to several distinct servers at roughly the same time, so several OCC messages referring to the same c are circulating simultaneously. This problem is addressed as follows. Upon receiving an OCC message for a fact f containing a constant c at position π, set $\lambda_\pi(c)$ contains the servers that knew about c when f was derived. Moreover, another intervening OCC message for constant c will have already updated $\mu_{k,\pi}(c)$; thus, $M := \mu_{k,\pi}(c) \setminus \lambda_\pi(c)$ identifies the servers whose occurrences may need additional updating. After computing M, server k updates its $\mu_{k,\pi}(c)$ by adding $\lambda_\pi(c)$; these steps must be performed atomically so that, if two messages concurrently update $\mu_{k,\pi}(c)$, set M computed in the second message contains the servers added by the first message. This set M is then added to D and $\lambda_\pi(c)$ (line 43). If set D is empty at that point, then $k = k_h$ holds due to how servers are extracted from D in lines 37 and 46; consequently, all servers have been updated and fact f can be added to the local partition element G_k (line 44). Otherwise, the OCC message is forwarded to the remaining servers in D ensuring that k_h is processed last (lines 46 and 47).

The following theorem captures the formal properties of our algorithm—that is, the algorithm correctly computes the materialisation and exhibits the nonrepetition property. The proof is given in full in Appendix A.

Theorem 4.1. *Let I be a dataset, let P be a Datalog program, and let $G_1, \ldots, G_\ell$ be the datasets obtained by applying Algorithm 1 to an arbitrary partition of I as specified in this section. Then, $P^\infty(I) = G_1 \cup \cdots \cup G_\ell$, and moreover the algorithm exhibits the nonrepetition property.*

Algorithm 1 Distributed Materialisation Algorithm at Server k in a cluster of ℓ servers

function SERVERTHREAD
  while cannot terminate **do**
    if G_k contains an unprocessed fact f, or a message m is pending **then**
      PROCESSFACT(f, $T_k(f)$) or PROCESSMESSAGE(m), as appropriate
    else if the termination token has been received **then**
      Process the termination token

function PROCESSFACT(f, τ)
  SYNCHRONISE(τ)
  for each $(\sigma, a, Q, h) \in$ MATCHRULES(f, P) **do**
    FINISHMATCH($0, \sigma, a, Q, h, \tau, \emptyset$) ▷ $\emptyset$ is the vector of empty partial occurrences

function PROCESSMESSAGE(PAR[$i, \sigma, Q, h, \tau, \boldsymbol{\lambda}$]) where $Q = a_1^{\bowtie 1} \wedge \cdots \wedge a_n^{\bowtie n}$ and $\boldsymbol{\lambda} = \lambda_s, \lambda_p, \lambda_o$
  SYNCHRONISE(τ)
  for each substitution $\sigma' \in$ EVALUATE($a_i^{\bowtie i}$, τ, G_k, T_k, σ) **do**
    FINISHMATCH($i, \sigma', a_i, Q, h, \tau, \boldsymbol{\lambda}$)

function FINISHMATCH($i, \sigma, a_{last}, Q, h, \tau, \boldsymbol{\lambda}$) where $Q = a_1^{\bowtie 1} \wedge \cdots \wedge a_n^{\bowtie n}$ and $\boldsymbol{\lambda} = \lambda_s, \lambda_p, \lambda_o$
  for each position $\pi \in \Pi$ and each variable x that occurs in a_{last} and either in h or in some a_j with $j > i$ **do**
    if λ_π is undefined on $x\sigma$ **then** Extend λ_π with the mapping $x\sigma \mapsto \mu_{k,\pi}(x\sigma)$
  if $i = n$ **then**
    for each position $\pi \in \Pi$ and each constant c occurring in h **do**
      if λ_π is undefined on c **then** Extend λ_π with the mapping $c \mapsto \mu_{k,\pi}(c)$
    if $\lambda_s(h\sigma|_s) = \emptyset$ **then** $k_h := h\sigma|_s \bmod \ell$
    else $k_h :=$ the singleton element in $\lambda_s(h\sigma|_s)$
    PROCESSORSENDMESSAGE(FCT[$h\sigma, C_k, \boldsymbol{\lambda}$], k_h)
  else
    $D :=$ the set of all servers
    for each position $\pi \in \Pi$ such that $a_{i+1}\sigma|_\pi$ is a constant c **do** $D := D \cap \lambda_\pi(c)$
    for each $d \in D$ **do** PROCESSORSENDMESSAGE(PAR[$i+1, \sigma, Q, h, \tau, \boldsymbol{\lambda}$], d)

function PROCESSMESSAGE(FCT[$f, \tau, \boldsymbol{\lambda}$]) where $\boldsymbol{\lambda} = \lambda_s, \lambda_p, \lambda_o$
  SYNCHRONISE(τ)
  $D := \{k\}$
  for each position $\pi \in \Pi$ and $c = f|_\pi$ such that $k \notin \mu_{k,\pi}(c)$ **do**
    Add k to $\lambda_\pi(c)$
    if c occurs in the head of a rule in P **then**
      Extend D to contain all servers in the cluster
    else
      for each position $\pi' \in \Pi$ **do** Add $\lambda_{\pi'}(c) \cup \mu_{k,\pi'}(c)$ to D
  Remove an element d from D, preferring any element over k if possible
  PROCESSORSENDMESSAGE(OCC[$f, D, k, C_k, \boldsymbol{\lambda}$], d)

function PROCESSMESSAGE(OCC[$f, D, k_h, \tau, \boldsymbol{\lambda}$]) where $\boldsymbol{\lambda} = \lambda_s, \lambda_p, \lambda_o$
  SYNCHRONISE(τ)
  for each position $\pi \in \Pi$ and each constant c in f **do**
    Atomically compute $M := \mu_{k,\pi}(c) \setminus \lambda_\pi(c)$ and then add $\lambda_\pi(c)$ to $\mu_{k,\pi}(c)$
    Add M to both D and $\lambda_\pi(c)$
  if $D = \emptyset$ **then** Atomically check whether $f \notin G_k$, and if so, add f to G_k and set $T_k(f)$ to C_k
  else
    Remove an element d from D preferring any element over k_h if possible
    SEND(OCC[$f, D, k_h, C_k, \boldsymbol{\lambda}$], d)

function SYNCHRONISE(τ)
  Atomically check if $C_k \leq \tau$, and set $C_k := \tau + 1$ if so

function PROCESSORSENDMESSAGE(m, d)
  if $d = k$ **then** PROCESSMESSAGE(m) **else** SEND(m, d)

4.7. Example

To clarify the ideas presented in this paper, we next present a simple example that illustrates the flow of processing in our system. To make the example manageable, we consider the rule r from Section 4.3 and just two servers with partition elements G_1 and G_2, each initially containing juts a single fact as shown below.

$$G_1 = \{\langle a, R, b\rangle\} \tag{16}$$

$$G_2 = \{\langle b, S, c\rangle\} \tag{17}$$

$$r = \langle z, T, x\rangle \leftarrow \langle x, R, y\rangle \wedge \langle y, S, z\rangle \tag{18}$$

The timestamps of both facts are initialised to zero, and the occurrence mappings for the subject and object positions are initialised as follows. To make the example easier to follow, we ignore the occurrence mappings for predicate positions, and we do not consider constants occurring in such positions.

$$\mu_{s,1} = \{a \mapsto \{1\}, b \mapsto \{2\}\} \tag{19}$$

$$\mu_{o,1} = \{a \mapsto \emptyset, b \mapsto \{1\}\} \tag{20}$$

$$\mu_{s,2} = \{b \mapsto \{2\}, c \mapsto \emptyset\} \tag{21}$$

$$\mu_{o,2} = \{b \mapsto \{1\}, c \mapsto \{2\}\} \tag{22}$$

Note that constants a and b are relevant to server 1, so, to satisfy Definition 4.1, $\mu_{s,1}$ and $\mu_{o,1}$ must contain mappings for these two constants. Analogously, $\mu_{s,2}$ and $\mu_{o,2}$ must cover constants b and c since these are relevant to server 2.

4.7.1. Rule matching and message flow

We now illustrate the process of rule matching and the flow of messages in our system. Both servers start matching the rule r in their partition element. Server 1 matches atom $\langle x, R, y\rangle$ to fact $\langle a, R, b\rangle$ in G_1, which produces a partial match $\sigma^1 = \{x \mapsto a, y \mapsto b\}$. The body is not yet fully matched, so server 1 identifies the set D of candidate servers that can finish the match (lines 25–26): $\mu_{s,1}(b) = \{2\}$ identifies server 2 as a viable candidate. Note that server 2 has no information about the occurrences of constant a, but this information will be needed once server 2 derives $\langle c, T, a\rangle$. Thus, server 1 copies its local occurrences for a and b into partial occurrences λ_s and λ_o (lines 16–17), and it then sends the PAR message to server 2 that contains the partial match σ^1, timestamp $\tau^1 = 0$, annotated query $Q^1 = \langle y, S, z\rangle^{\leq}$, and the following partial occurrences:

$$\lambda_s^1 = \{a \mapsto \{1\}, b \mapsto \{2\}\} \tag{23}$$

$$\lambda_o^1 = \{a \mapsto \emptyset, b \mapsto \{1\}\}. \tag{24}$$

Server 2 eventually processes this PAR message (line 11) and attempts to match the partially instantiated second atom $\langle y, S, z\rangle^{\leq}\sigma^1 = \langle b, S, z\rangle^{\leq}$ in G_2 (line 13). Fact $\langle b, S, c\rangle$ is a match since its timestamp is equal to zero, which is less or equal to τ^1. This produces $\sigma^2 = \sigma^1 \cup \{z \mapsto c\}$, and server 2 extends the partial occurrences as follows (lines 16–17):

$$\lambda_s^2 = \{a \mapsto \{1\}, b \mapsto \{2\}, c \mapsto \emptyset\} \tag{25}$$

$$\lambda_o^2 = \{a \mapsto \emptyset, b \mapsto \{1\}, c \mapsto \{2\}\}. \tag{26}$$

The match is now complete (lines 19–20), so server 2 must determine where to store the derived fact $\langle c, T, a\rangle$. Since $\lambda_s^2(c) = \emptyset$, the server knows that no server in the system contains constant c in subject position, so it hashes the subject (line 21). Let us assume that this results in $k_h = 2$—that is, the fact should be stored on server 2. Server 2 thus forwards the FCT message to itself containing fact $\langle c, T, a\rangle$ timestamp τ^1, and partial occurrences $\boldsymbol{\lambda}^2$ (line 23).

Adding this fact to G_2 will change the occurrences as constants a and c will appear on server 2 in object and subject positions, respectively. To maintain the correctness property of Definition 4.1, server 2 must disseminate the new occurrences to the relevant servers before the fact can be added to G_2. The server updates the partial occurrence to reflect that the fact will be added to server 2 (line 32), thus producing the following partial occurrences:

$$\lambda_s^3 = \{a \mapsto \{1\}, b \mapsto \{2\}, c \mapsto \{2\}\} \tag{27}$$

$$\lambda_o^3 = \{a \mapsto \{2\}, b \mapsto \{1\}, c \mapsto \{2\}\}. \tag{28}$$

Moreover, server 2 identifies which servers need to be informed of the change by combining the partial and local occurrences of all constants occurring in the derived fact (lines 31–36). This produces $D = \{1, 2\}$, so server 2 sends an OCC message to server 1 (line 38) containing the derived fact and the updated partial occurrences λ_s^3 and λ_o^3.

This message informs server 1 that fact $\langle c, T, a\rangle$ is about to be added to G_2, so server 1 updates its local occurrences for each position and each constant in the fact (lines 41–43). The resulting occurrences on server 1 are as follows:

$$\mu_{s,1} = \{a \mapsto \{1\}, b \mapsto \{2\}, c \mapsto \{2\}\} \tag{29}$$

$$\mu_{o,1} = \{a \mapsto \{2\}, b \mapsto \{1\}, c \mapsto \{2\}\}. \tag{30}$$

Server 1 then forwards the OCC message further to server 2 (line 47), which eventually adds $\langle c, T, a\rangle$ to G_2 (line 44).

4.7.2. Nonrepetition of derivations

In addition to the steps outlined in Section 4.7.1, server 2 also matches rule r in partition element G_2, which produces the partial match $\sigma^3 = \{y \mapsto b, z \mapsto c\}$. Thus, server 2 will sent a PAR message to server 1 containing σ^3, timestamp $\tau^2 = 0$, and annotated query $Q^2 = \langle x, R, y\rangle^{<}$. When server 1 processes this message, the partially instantiated atom $\langle a, R, y\rangle^{<}$ matches to $\langle a, R, b\rangle$ in terms of its structure, but not in terms of the timestamp. Therefore, the EVALUATE function returns no substitutions and consequently matching stops at this point. As a result of this, server 1 does not derive fact $\langle c, T, a\rangle$—that is, this fact is derived only on server 2.

5. Streaming partitioning of RDF data

We now consider how to partition To this end, in Section 5.1 we review the drawbacks of the existing partitioning schemes and discuss our technical challenges. Then, in Sections 5.2 and 5.3, we present two new techniques that partition RDF data in a *streaming* fashion—that is, without loading the entire dataset in memory.

5.1. Motivation

Approaches to partitioning RDF data are often based on subject hashing. The main benefit of such approaches is its simplicity: just one pass over the dataset is needed, and just one fact must be kept in memory at any point in time. However, subject hashing does not take into account the structure of an RDF dataset, so there is no attempt to ensure locality of subject–object or object–object joins.

A number of partitioning approaches are based on variants of min-cut graph partitioning. Such approaches take the structural properties of an RDF dataset into account and are thus more likely to partition a dataset into tightly connected subsets. However, the time and memory requirements of such methods are often prohibitive. Typically, an entire dataset is loaded into a single server so that one can apply a graph partitioner such as

METIS[2]; however, this defeats the main objective of distributing the data, which is to process large datasets using commodity servers. This drawback can, at least in principle, be addressed by using distributed partitioner such as ParMETIS. Nevertheless, min-cut graph partitioning is NP-hard [52] so, while implementations rarely solve this problem exactly, partitioning often takes a considerable amount of time and memory in practice. Thus, the questions of how to partition RDF data effectively, and how this affects distributed reasoning, are still largely open.

To answer the former question, we draw inspiration the literature on *streaming graph partitioning* [33–35,53–56] methods, where the aim is to produce good partitions while iterating over the graph edges a fixed number of times. The memory usage of these approaches is typically determined by the number of vertices in the graph, which is usually at least an order of magnitude smaller than the number of edges. This results in a much smaller memory footprint for partitioning than with, say, METIS. The HDRF [34] algorithm was recommended as particularly suitable for graphs with power-law degree distribution [33], which is often present in RDF datasets. Moreover, the 2PS [35] algorithm was shown to sometimes outperform HDRF. Thus, we use HDRF and 2PS as the basis for our work.

Streaming partitioning algorithms, however, were designed for general (directed or undirected) graphs, so applying them straightforwardly to RDF is unlikely to produce good results: facts with the same subject would not necessarily be placed on the same server, which, as we explained in Section 3, is critical. To remedy this, we adapt HDRF and 2PS to take the specifics of RDF into account, but without compromising the quality of the produced partitions.

5.2. The HDRF$_3$ algorithm

We now present our HDRF$_3$ algorithm for streaming partitioning of RDF data. We follow the 'high degree replicated first' principle from the HDRF algorithm for general graphs [34]. In Section 5.2.1 we briefly discuss the original idea, and in Section 5.2.2 we adapt these principles to RDF.

5.2.1. The original HDRF algorithm

The HDRF algorithm was developed for *scale-free* undirected graphs, where the distribution of vertex degrees exhibits (or is close to) the power-law distribution. Such graphs contain few high-degree vertices, and many low-degree vertices. HDRF aims to replicate (i.e., assign to more than one server) vertices with higher degrees so that a smaller number of vertices has to be replicated overall. The algorithm processes sequentially the edges of the input graph and assigns them to servers. For each server $k \in \{1, \dots, \ell\}$, the algorithm maintains the number N_k of edges currently assigned to server k; all of N_k are initialised to zero. Moreover, for each vertex v, the algorithm maintains the partial degree $deg(v)$ and the partial replication set $A(v)$ in the subgraph processed thus far. For each vertex v, the degree $deg(v)$ is initialised to zero, and $A(v)$ is initialised to the empty set. To assign an undirected edge $\{v, w\}$, the algorithm first increments $deg(v)$ and $deg(w)$, and then for each candidate server $k \in \{1, \dots, \ell\}$ it computes the score $C(v, w, k)$. Finally, the algorithm sends the edge $\{v, w\}$ to the server k with the highest score $C(v, w, k)$, increments N_k, and updates sets $A(v)$ and $A(w)$ to contain k.

The score $C(v, w, k)$ consists of two parts, $C_{REP}(v, w, k)$ and $C_{BAL}(k)$. The former estimates the impact that placing $\{v, w\}$ on server k has on replication and is computed as

$$C_{REP}(v, w, k) = g(v, w, k) + g(w, v, k),$$

where

$$g(v, w, k) = \begin{cases} 1 + \frac{deg(w)}{deg(v)+deg(w)} & \text{if } k \in A(v), \\ 0 & \text{otherwise.} \end{cases}$$

To understand the intuition behind this formula, assume that vertex v occurs only on server k, vertex w occurs only on server k', and $deg(v) > deg(w)$. We then have $g(v, w, k) < g(w, v, k')$, which ensures that edge $\{v, w\}$ is sent to server k'—that is, vertex v is replicated to server k', in line with our desire to replicate higher-degree vertices. The sum $deg(v) + deg(w)$ in the denominator of the formula for $g(v, w, k)$ normalises the degrees of v and w.

Considering $C_{REP}(v, w, k)$ only would risk producing partitions of unbalanced sizes. Therefore, the second part of the score is used to favour assigning edge $\{v, w\}$ to the currently least loaded server using formula

$$C_{BAL}(k) = \frac{maxsize - N_k}{\epsilon + maxsize - minsize},$$

where *maxsize* and *minsize* are the maximal and minimal, respectively, possible partition sizes.

Scores $C_{REP}(v, w, k)$ and $C_{BAL}(k)$ are finally combined using a fixed weighting factor λ as

$$C(v, w, k) = C_{REP}(v, w, k) + \lambda \cdot C_{BAL}(k)$$

Thus, λ allows us to control how important is balancing partition sizes versus achieving low replication factors.

The version of the algorithm presented above makes just one pass over the graph edges, and $g(v, w, k)$ and $g(w, v, k)$ are computed using the partial vertex degrees (i.e., degrees in the subset of the graph processed thus far). The authors of HDRF also discuss a variant where exact degrees are computed in a preprocessing pass, and they show empirically that this does not substantially affect the partition quality.

5.2.2. Adapting the algorithm to RDF graphs

Several problems need to be addressed to adapt HDRF to RDF. A minor issue is that RDF facts correspond to labelled directed edges, which we address by ignoring the predicate component of facts. A more important problem is to ensure that all facts with the same subject are placed on the same server. To achieve this, we compute the destination for all facts with subject s the first time we see such a fact.

The pseudo-code of HDRF$_3$ is shown in Algorithm 2. It takes as input a parameter α determining the maximal acceptable imbalance in partition element sizes, the balance parameter λ as in HDRF, and another parameter δ that we describe shortly. The algorithm uses a preprocessing pass over G (not shown in the pseudo-code), where it determines the size of the dataset $|G|$, and the out-degree $|G^+(c)|$ and the degree $|G(c)|$ of each constant c in G. The algorithm also maintains (i) the replication set $A(c)$ for each constant, which is initially empty, (ii) a mapping T of constants occurring in subject position to servers, which is initially undefined on all constants, and (iii) the numbers $N_1, \dots, N_\ell$ and $C_1, \dots, C_\ell$ of facts and constants, respectively, assigned to servers thus far, all of which are initially set to zero.

Our algorithm uses the ProcessFact function to assign each fact $\langle s, p, o \rangle \in G$ to a server. Mapping T keeps track of the server that will receive facts with a particular subject. Thus, if $T(s)$ is undefined (line 53), the algorithm sets $T(s)$ to the server with the highest score (line 54) in a way analogous to HDRF. All facts with the same subject encountered later will be assigned to server $T(s)$, so counter $N_{T(s)}$ is updated with the out-degree of s (line 55). Finally, the fact is sent to server $T(s)$ (line 56), and the replication sets of s and o and the number of constants $C_{T(s)}$ on server $T(s)$ are updated if needed (lines 57 and 58).

[2] http://glaros.dtc.umn.edu/gkhome/home-of-metis.

Algorithm 2 HDRF$_3$

Require:
- the tolerance parameter $\alpha > 1$
- the balance parameter λ
- the degree imbalance parameter δ
- the target number of servers ℓ
- $|G|$, $|G^+(c)|$, and $|G(c)|$ for each constant c in G are known
- $A(c) := \emptyset$ for each constant c in G
- Mapping T of constants to servers, initially undefined on all constants
- $N_k := C_k := 0$ for each server $k \in \{1, \ldots, \ell\}$

52: **function** ProcessFact(s, p, o)
53: **if** $T(s)$ is undefined **then**
54: $T(s) := \text{argmax}_{k \in \{1,\ldots,\ell\}}$ Score(s, o, k)
55: $N_{T(s)} := N_{T(s)} + |G^+(s)|$
56: Add (s, p, o) to $G_{T(s)}$
57: **if** $T(s) \notin A(s)$ **then** Add $T(s)$ to $A(s)$ and increment $C_{T(s)}$
58: **if** $T(s) \notin A(o)$ **then** Add $T(s)$ to $A(o)$ and increment $C_{T(s)}$

59: **function** Score(s, o, k)
60: $C_{REP} := 0$
61: **if** $k \in A(s)$ and Deg(k) $\leq \min_{\ell \in \{1,\ldots,\ell\}}$ Deg(ℓ) $+ \delta$ **then** $C_{REP} := C_{REP} + 1 + \frac{|G(o)|}{|G(s)|+|G(o)|}$
62: **if** $k \in A(o)$ and Deg(k) $\leq \min_{\ell \in \{1,\ldots,\ell\}}$ Deg(ℓ) $+ \delta$ **then** $C_{REP} := C_{REP} + 1 + \frac{|G(s)|}{|G(s)|+|G(o)|}$
63: $C_{BAL} := 1 - \ell \frac{N_{k'} + |G^+(s)|}{\alpha |G|}$
64: **return** $C_{REP} + \lambda \frac{\sum_k N_k}{|G|} C_{BAL}$

65: **function** Deg(k)
66: **return** $C_k = 0 \ ? \ 0 \ : \ N_k / C_k$

The score of sending fact $\langle s, p, o \rangle$ to server k is calculated as in HDRF. The replication part C_{REP} of the score is computed in lines 61 and 62. Unlike the original HDRF algorithm, the first time we encounter a constant s, we determine the target server for all facts with subject s in the rest of the input; thus, knowing the degree of s in advance allows us to take into account the impact of all future allocations of facts with subject s to server $T(s)$. Moreover, we observed empirically that reasoning tends to be faster when partition elements have roughly similar average constant degrees. Function Deg estimates the current average degree of constants in server k as a quotient of the currently numbers of facts (N_k) and constants (C_k) assigned to server k. Then, in lines 61 and 62, C_{REP} is updated only if the average degree of server k is close (i.e., within the range defined by the parameter δ) to the minimal average degree.

Line 63 computes the balance factor by observing that a partition element can have at most $\alpha |G|/\ell$ facts.

Finally, C_{REP} and C_{BAL} are combined using λ in line 64. Unlike in the original HDRF algorithm, factor $\sum_k N_k / |G|$ ensures that partition balance grows in importance towards the end of partitioning.

As we mentioned in Section 2, balancing partition sizes while minimising the replication factor is computationally hard, so the minimality requirement is typically dropped. The following result shows that Algorithm 2 honours the balance requirements, provided that α and λ are chosen in a particular way. The proof is given in Appendix B.

Proposition 5.1. *Algorithm 2 produces a partition that satisfies* $|G_k| \leq \alpha \frac{|G|}{\ell}$ *for each* $1 \leq k \leq \ell$ *whenever* α *and* λ *are selected such that*

$$\alpha > 1 + \ell \frac{\max_c |G^+(c)|}{|G|} \quad \text{and} \quad \lambda \geq \frac{4\alpha}{\ell \left(\frac{\alpha - 1}{\ell} - \frac{\max_c |G^+(c)|}{|G|} \right)^2}.$$

5.3. The 2PS$_3$ algorithm

We now present our 2PS$_3$ algorithm for RDF, which adapts the *two-phase streaming* algorithm 2PS [35]. In Section 5.3.1 we discuss the original idea, and in Section 5.3.2 we discuss how we apply these principles to RDF.

5.3.1. The original 2PS algorithm

The 2PS algorithm processes undirected graphs in two phases. In the first phase, the algorithm clusters vertices into communities aiming to place highly connected vertices into a single community. This is achieved by initially assigning each vertex in the graph to a separate community. Then, when processing an edge $\{v, w\}$ in the first phase, the sizes of the current communities of v and w are compared, and the vertex belonging to the smaller community is merged into the larger community. Thus, communities are iteratively coarsened as edges of the input graph are processed in the first phase. The entire first phase can be repeated several times to improve community detection.

After all edges are processed in the first phase, the identified communities are greedily assigned to servers. Then, the graph is processed in the second phase, and edges are assigned to the communities of their vertices.

5.3.2. The algorithm

Just like in the case of HDRF, the main challenge in extending 2PS to RDF is to deal with the directed nature of RDF facts, and to ensure that facts with the same subject are assigned to the same server.

The pseudo-code of 2PS$_3$ is shown in Algorithm 3. As in HDRF$_3$, the algorithm uses a preprocessing phase to determine the size of dataset $|G|$ and the out-degree $|G^+(c)|$ of each constant c. Thus, our 2PS$_3$ algorithm actually uses three phases; however, to stress the relationship with the 2PS algorithm, we call the algorithm 2PS$_3$.

The algorithm maintains a global mapping M of constants to communities—that is, $M(c)$ is the community of each constant c. Thus, two constants c_1 and c_2 are in the same community if $M(c_1) = M(c_2)$. Initially, each constant c is assigned to its own community m_c. As the algorithm progresses, the image of M contains fewer and fewer communities. Once communities are assigned to servers, a fact $\langle s, p, o \rangle$ is assigned to the server of

Algorithm 3 $2PS_3$

Require:
- the tolerance parameter $\alpha > 1$
- the target number of servers ℓ
- $|G|$ and $|G^+(c)|$ for each constant c in G are known
- $M(c) := m_c$ and $S(m_c) := |G^+(c)|$ for each constant r in G, where m_c is a community unique for c

67: **function** ProcessFact-Phase-I(s, p, o)
68: Let $c_{max} := \text{argmax}_{c \in \{s,o\}} S(M(c))$, and let c_{min} be the other constant
69: **if** $S(M(c_{max})) + |G^+(c_{min})| < (\alpha - 1)|G|/\ell$ **then**
70: $S(M(c_{max})) := S(M(c_{max})) + |G^+(c_{min})|$
71: $S(M(c_{min})) := S(M(c_{min})) - |G^+(c_{min})|$
72: $M(c_{min}) := M(c_{max})$

73: **function** AssignCommunities
74: $N_k := 0$ for each server $k \in \{1, \ldots, \ell\}$
75: **for** each community m occurring in the image of the mapping M **do**
76: $T(m) := \text{argmin}_{k \in \{1,\ldots,\ell\}} |N_k|$
77: $N_{T(m)} := N_{T(m)} + S(m)$

78: **function** ProcessFact-Phase-II(s, p, o)
79: Add (s, p, o) to server $T(M(s))$

community $M(s)$; thus, facts with the same subject are assigned to one server.

The algorithm also maintains a global function that, for each community m, keeps track of the number $S(m)$ of facts whose subject is assigned to community m. Thus, $S(m_c)$ is initialised as $|G^+(c)|$ for each constant c.

After initialisation, each fact $\langle s, p, o \rangle \in G$ is processed using function ProcessFact-Phase-I. In line 68, the algorithm compares the sizes $S(M(s))$ and $S(M(o))$ of the communities to which s and o, respectively, are currently assigned. It identifies c_{max} as the constant whose current community size is larger, and c_{min} as the constant whose current community size is smaller (ties are broken arbitrarily). The aim of this is to move c_{min} into the community of c_{max}, but this is done only if, after the move, we can satisfy the requirement on the sizes of partition elements: if each community contains no more than $(\alpha - 1)\frac{|G|}{\ell}$ facts, we can later assign communities to servers greedily and the resulting partition elements will contain fewer than $\alpha\frac{|G|}{\ell}$ facts. This is reflected in the condition in line 68: if satisfied, the algorithm updates the sizes of the communities of c_{max} and c_{min} (lines 70–71), and it moves c_{min} into the community of c_{max} (line 72). If desired, G can be processed several times using function ProcessFact-Phase-I to improve the community structure.

After G is processed, function AssignCommunities assigns communities to servers. To this end, for each server k, the algorithm maintains the number N_k of facts currently assigned to partition element k. Then, the communities from the image of M (i.e., the communities that have 'survived' after shuffling the constants in the first phase) are assigned by greedily preferring the least loaded server. Finally, using function ProcessFact-Phase-II, each fact $\langle s, p, o \rangle \in G$ is assigned to the server of community $M(s)$.

As in $HDRF_3$, our algorithm is not guaranteed to minimise the replication factor. However, the following result shows that the algorithm will honour the restriction on the sizes of partition elements for a suitable choice of α. The proof is given in Appendix C.

Proposition 5.2. *Algorithm 3 produces a partition that satisfies* $|G_k| \leq \alpha\frac{|G|}{\ell}$ *for each* $1 \leq k \leq \ell$ *whenever*

$$\alpha > 1 + \frac{\max_c |G^+(c)|}{|G|}.$$

6. Evaluation

To empirically evaluate the techniques we presented in this paper, we implemented a prototype distributed Datalog reasoner called DMAT. To store and manage triples in RAM, we have reused the storage subsystem of RDFox [48], a state-of-the-art centralised RDF system. The storage subsystem of RDFox maintains exhaustive hash-based indexes as described by Motik et al. [48] to support efficient enumeration of triples matching a single atom; for example, given an atom $\langle x, R, x \rangle$, it can efficiently provide all values for x that instantiate the atom to a triple in the locally stored dataset. On top of this basic data storage facility, we implemented a mechanism for associating triples with timestamp and the Evaluate function by first identifying candidate matches using the functionality provided by RDFox and then filtering the matches according to timestamps. Finally, we implemented our algorithms from scratch (i.e., without reusing any algorithms from RDFox). Our prototype is written in C++. At the moment, DMAT can run our materialisation algorithm on just one thread: the need to synchronise threads on one server introduced considerable complexity to our implementation, so we decided to leave this aspect for future work. DMAT can partition the data using subject hashing, a variant of min-cut partitioning by Potter et al. [31] that we call METIS, and the $HDRF_3$ and $2PS_3$ algorithms described in Sections 5.2 and 5.3, respectively.

We evaluated DMAT by conducting four sets of experiments. First, we investigated how using different partitioning schemes, including $HDRF_3$ and $2PS_3$, affects the performance of materialisation. Second, we investigated the extent to which the performance of our algorithm depends on network speed. Third, we studied how the performance of materialisation changes when the input data and the number of servers increase. Fourth, we compared DMAT with BigDatalog and Cog, two state-of-the-art systems that use static data exchange strategies.

We next present our experimental setting and discuss the results. In particular, we introduce our datasets in Section 6.1, we discuss our test setting in Section 6.2, and then we present the data partitioning tests in Section 6.3, the network speed tests in Section 6.4, the scalability tests in Section 6.5, and the system comparison tests in Section 6.6. Considering data partitioning first will allow us to identify $2PS_3$ as the partitioning strategy that seems to offer the best performance on average, so we use $2PS_3$

Table 2
Program statistics.

Dataset	#rules	# recursive rules	Avg. # of body atoms
LUBM	103	3	1.20
WatDiv	32	2	2.10
MAKG*	15	2	2.20

in the remaining two tests. The DMAT system and all datasets and programs used in our experiments are available online.[3]

6.1. Datasets

In this section we discuss the datasets we used in our experiments. Table 2 summarises some basic information about the programs used. The sizes of the input datasets varied in each test, so we report the data sizes when discussing the relevant experiments. All programs and datasets are available from the mentioned Web page.

The LUBM [57] benchmark has been extensively used to test the performance of RDF systems. We generated datasets of varying sizes using LUBM's data generator, and we used the *extended lower bound* Datalog program by Motik et al. [48] designed to stress-test reasoning systems. The program was obtained by translating the OWL 2 RL portion of the LUBM ontology into Datalog and manually adding several recursive rules that produce many redundant derivations. To the best of our knowledge, this program has not yet been used in the literature to test the performance of distributed RDF reasoners, and it provides us with more insights than the standard, relatively 'easy' lower bound program.

The WatDiv [58] benchmark was developed for testing the performance of query answering in RDF systems. It comes with a data generator that can produce datasets in which the degrees of resources follow a power law distribution. Such datasets are challenging to both query answering and partitioning algorithms, which makes WatDiv highly relevant to our setting. However, WatDiv does not include an ontology or a Datalog program, so we manually created a program consisting of 32 chain, cyclical, and recursive rules.

The *Microsoft Academic Knowledge Graph* [59] is an RDF translation of the Microsoft Academic Graph—a heterogeneous dataset of scientific publication records, citations, authors, institutions, journals, conferences, and fields of study. The original MAKG dataset contains 8 billion triples and includes links to datasets in the Linked Open Data Cloud. A significant portion of the dataset consists of triples with datatype properties providing annotations such names, publication dates, various counts, and so on. Such triples are not interesting for reasoning as they do not define the graph structure, but they increase the memory requirements on servers and data loading times. Thus, to make experimentation practical, we selected 3.67 billion triples with 'structural' properties such as *appearsInJounal*, *hasDiscipline*, and so on; the list of all selected properties is available on the aforementioned Web page. We call the resulting dataset MAKG*. The dataset does not come with an ontology or a Datalog program, so we manually created a program consisting of 15 chain, cyclical, and recursive rules.

As far as we know, this is the first time that WatDiv and MAKG were used to benchmark Datalog reasoning.

[3] https://krr-nas.cs.ox.ac.uk/2021/DMAT/.

6.2. Test setting

We ran our experiments on the Amazon EC2 cloud. For each server in the cluster, we used one `r5.4xlarge` instance equipped with a 3.1 GHz Intel Xeon Platinum 8000 series (Skylake-SP or Cascade Lake) processor, two virtual CPUs, 128 GB of RAM, and running Linux kernel 4.14. The servers were connected by Ethernet that, according to Amazon, can support speeds up to 10 Gbps. For the experiments with DMAT, the disk space was irrelevant since our system stores data in RAM. For the experiments with BigDatalog and Cog, we equipped each server with 1 TB of Amazon Elastic Block Storage (EBS) to be able to run Spark and Flink. Finally, METIS requires loading the entire dataset into memory, so we used one `r5.24xlarge` server with 784 GB of RAM to partition the data in the experiments with METIS.

Two virtual CPUs per server were sufficient for our experiments: as we already mentioned, DMAT can currently run our algorithm only on one thread; thus, to ensure a fair comparison, we configured BigDatalog and Cog to use just one worker thread per server. In our data partitioning and system comparison experiments, we computed the materialisation using DMAT, BigDatalog, and Cog on ten servers. In our scalability experiments, we scaled the number of servers proportional to the input size. In all experiments with DMAT, we used one additional 'master' server whose role was to distribute the facts and the program to the servers and initiate materialisation. We used the 'master' server mainly for convenience: this server never participated in materialisation, so its use in the initialisation phase does not affect our results in any substantial way.

6.3. Data partitioning experiments

The objective of our partitioning experiment was to see how different data partitioning strategies affect the performance of materialisation. To this end, we compared the HDRF_3 and 2PS_3 algorithms from Section 5 with subject hashing and theMETIS variant of min-cut partitioning by Potter et al. [31]. The latter approach uses weighted graph partitioning to balance the number of triples, rather than the number of resources on different servers.

As we mentioned in Section 3.3, several approaches have been proposed in the literature that replicate facts to more than one server [12,14,15,47]. However, the main objective of our work was to avoid repetition of derivations in a distributed setting, which seems incompatible with data replication. As a result, our reasoning algorithm requires all partition elements to be pairwise disjoint, and so we cannot include any data partitioning strategy that involves data replication in our evaluation.

Test Setting. We used the LUBM dataset for 10k universities, the WatDiv-1G dataset provided by the authors of WatDiv, and the MAKG* dataset in this test; for each dataset, Table 3 shows the number of resources, the numbers of input and output triples, and the number of derivations (i.e., the number of facts derived in line 23 before duplicate elimination). To speed up loading, we preprocessed all datasets by replacing all resources with integers. With Hash, HDRF_3, or 2PS_3, the 'master' server processed the data in a streaming fashion and distributed the triples to the ten materialisation servers, and then it started the materialisation by distributing the Datalog program. With METIS, the precomputed partition elements were loaded directly into the materialisation servers, and the 'master' just distributed the Datalog program. To hash the triples' subjects, we simply multiplied the integer subject value by a large prime in order to randomise the distribution. In HDRF_3 and 2PS_3, we used $\alpha = 1.25$. Also, in HDRF_3, we used $\delta = 0.25$ and λ was set to the lowest value satisfying Proposition 5.1; the values of λ thus vary for each dataset and

Table 3
Data partitioning experiments.

Method	Partitioning Stats [n = 10]					Reasoning Stats	
	Min (%)	Max (%)	Med (%)	RF	Time (s)	Time (s)	Nonlocal PAR messages (G)
LUBM-10k [328 M resources, 1.34 G input and 3.32 G output triples, 79.30 G derivations, λ = 819]							
Hash	10.00	10.00	10.00	1.60	**530**	16 990	72.93
METIS	9.24	10.66	9.98	1.19	15 300	9 950	5.75
$HDRF_3$	9.35	10.47	10.00	1.43	590	12 940	44.62
$2PS_3$	9.06	10.35	10.00	**1.04**	700	**5 190**	**0.97**
WatDiv-1G [100 M resources, 1.09 G input and 1.75 G output triples, 2.11 G derivations, λ = 800]							
Hash	10.00	10.00	10.00	2.48	**520**	1 198	7.89
METIS	9.70	10.35	10.00	**2.16**	15 100	1 620	**7.64**
$HDRF_3$	10.00	10.00	10.00	2.48	590	**1 180**	7.73
$2PS_3$	9.92	10.02	10.00	2.40	1 080	1 636	7.61
MAKG* [490 M resources, 3.67 G input and 5.63 G output triples, 17.47 G derivations, λ = 800]							
Hash	10.00	10.00	10.00	1.99	**2 220**	8 200	28.24
METIS	–	–	–	–	–	–	–
$HDRF_3$	10.00	10.00	10.00	**1.66**	3 500	7 050	25.35
$2PS_3$	9.91	10.06	10.00	1.67	3 640	**6 450**	**20.70**

are shown in Table 3. We processed the datasets twice in the first phase of $2PS_3$. We recorded the wall-clock time and the number of PAR messages sent on each server during materialisation. For each test, Table 3 shows the minimum, maximum, and median numbers of triples in partition elements (given as percentages of the number of input triples), the replication factor (see Section 2 for a definition), the partitioning and reasoning times, and the number of nonlocal PAR messages (i.e., the number of messages that were sent over the network).

Partition Times and Balance. All partitioning schemes produced partition elements with sizes within the tolerance parameters: Hash achieves perfect balance if the hash function is uniform; METIS explicitly aims to equalise partition sizes; and our algorithms do so by design and the choice of parameters. For all streaming methods, the partitioning times were not much higher than the time required to read the datasets from disk and send triples to their designated servers. In contrast, METIS partitioning took longer than materialisation on LUBM-10k and WatDiv-1G, and on MAKG* it ran out of memory even though we used a very large server equipped with 784 GB of RAM.

Replication, Communication, and Reasoning. Generally lowest replication factors were achieved with $2PS_3$: only METIS achieved a lower value on WatDiv-1G, and $HDRF_3$ achieved a comparable value on MAKG*. The replication factor of Hash was highest in all cases, closely followed by $HDRF_3$. Moreover, lower replication factors seem to correlate closely with decreased communication overhead; for example, the number of messages was significantly smaller on LUBM-10k and MAKG* with $2PS_3$ than with other schemes. This reduction seems to generally lead to shorter reasoning times: $2PS_3$ was faster than all other schemes on LUBM-10k and MAKG*; for the former, the improvement over Hash is by a factor of 2.25. However, the reasoning times do not always correlate with the replication factor: on WatDiv-1G, METIS and $2PS_3$ were slower than Hash and $HDRF_3$, despite exhibiting smaller replication factors.

Workload Imbalance. To further analyse the results of our experiments, we show in Fig. 1 the numbers of derivations and the total size of partial messages processed by each of the ten servers in the cluster. As one can see, the numbers of derivations and messages per server are quite uniform for Hash and, to an extent, for $HDRF_3$; in contrast, with $2PS_3$ and METIS, certain servers seem to be doing much more work than others, particularly on WatDiv-1G and MAKG*. Thus, reducing communication seems to matter only up to a point. For example, $2PS_3$ reduces communication by about an order of magnitude on LUBM-10k, which, combined with a uniform workload distribution, seems to 'pay off' in terms of reasoning times. On MAKG*, $2PS_3$ reduces communication by about a factor of two, while $HDRF_3$ seems to distribute the workload more evenly. Combined, these factors lead to more modest (yet still significant) improvements in reasoning times for $2PS_3$. Finally, on WatDiv-1G, communication overhead does not appear to be significant with any partitioning strategy, so the reasoning times seem to be determined mainly by the workload imbalance.

Graph Structure. LUBM-10k data is organised into universities, where most triples connect entities within one university; thus, the data seems to naturally decompose into modules of roughly the same sizes. The $2PS_3$ algorithm seems to exploit this modular structure very well, allowing it to reduce the communication overhead by an order of magnitude. In contrast, WatDiv-1G and MAKG* do not seem to decompose into modules as easily. In fact, WatDiv-1G was specifically designed to produce RDF graphs where the vertex degrees follow a power-law distribution; such datasets have irregular structure and highly variable constant degrees, which makes partitioning difficult. The original HDRF algorithm was identified as particularly suitable for such graphs [33]. Our results seem to agree: $HDRF_3$ offers the best materialisation times on WatDiv-1G.

Overall Performance. In general, $2PS_3$ seems to provide a good performance mix: unlike METIS, it can be implemented without placing unrealistic requirements on the servers used for partitioning; it can significantly reduce communication, particularly on highly modular graphs; and the resulting partition can lead to workload imbalances, but these do not appear to be excessive. Thus, $2PS_3$ seems like a good alternative to the thus far dominant hash partitioning, and therefore we use it in our scalability (Section 6.5) and system comparison experiments (Section 6.6). The $HDRF_3$ algorithm seems to be worth considering on graphs with a power-law vertex degree distribution.

6.4. Effects of network speed

Although 10 Gbps network speeds are often found in modern data centres, it is nevertheless interesting to see how our techniques perform on older and slower networks. Since our main objective is to reduce network communication, one can expect performance gains from locality-aware processing to be more pronounced with slower networks. To answer this question, we artificially slowed down the network of our servers using the `tc`

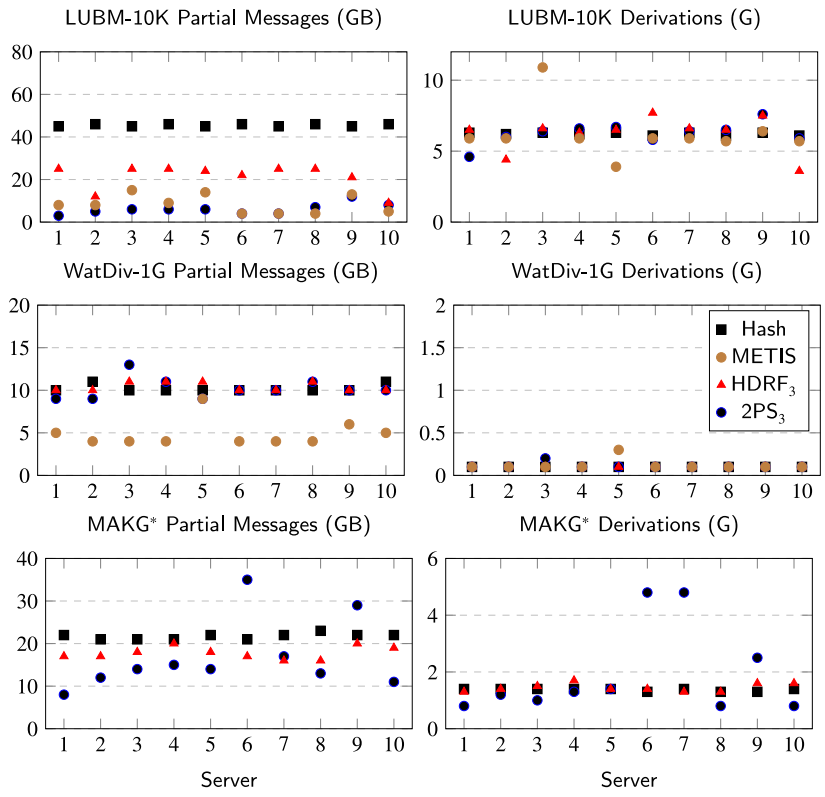

Fig. 1. Input partitioning experiments.

Table 4
Materialisation of WatDiv-1G on networks of varying speeds.

Network speed (Mbps)	Time (s)
10 000	1162
2048	1288
1024	1569
512	2331

Linux command and then materialised the WatDiv-1G dataset partitioned using the $2PS_3$ algorithm. We chose WatDiv-1G-1G because, as we discuss in Section 6.3, the number of local PAR messages during materialisation is lowest on this dataset, so a slower network is more likely to affect reasoning times. Table 4 shows the materialisation times for four network speeds.

As one can see, materialisation times grow as network speed decreases, but the increase is sublinear: if we slow down the network by a factor of 19.5, the materialisation time increases only by a factor of two. This suggests that network communication is an important, but not the only factor determining the performance of our algorithm. Moreover, materialisation times increases only by 35% on still relatively common 1 Gbps network, showing that our techniques are applicable to commodify hardware.

6.5. Scalability experiments

The main objective of data distribution is scalability—that is, the ability to process increasing data loads without a significant increase in processing times by proportionally extending the cluster. Note, however, that the size of the input data is not always representative of the work needed to compute the materialisation. For example, applying the rule $\langle x, R, y\rangle \wedge \langle y, R, z\rangle \rightarrow \langle x, R, z\rangle$ to a dataset consisting of triples $\langle a_1, R, a_2\rangle, \langle a_2, R, a_3\rangle, \ldots, \langle a_n, R, a_1\rangle$ derives n^2 triples, and it requires matching the rule body in n^3 ways; thus, materialisation time is likely to depend cubically on the input size on this example. We therefore analyse the scalability of DMAT in terms of two natural and complementary ways to measure the amount of work.

Work Measures. The number of derivations is a good measure of the amount of work for the following reasons. First, this number is equal to the number of answers obtained by evaluating the bodies of all rules as queries over the materialisation, which is fixed for every dataset—that is, the number of derivations does not depend on the materialisation algorithm. Second, duplicate facts can be eliminated in constant amortised time, so the number of derivations also estimates the amount of work for duplicate elimination. Hence, this is a natural measure for seminaïve evaluation, where each derivation is made exactly once.

In addition, we shall also consider the number of PAR messages produced during materialisation. If most partial answers lead to a derivation (and our experience suggests that this is frequently the case), the number of PAR messages is much smaller than the number of derivations. However, this is not necessarily so; for example, in a chain rule, the join of the initial two atoms can produce many partial answers that do not 'survive' a join with the third atom; thus, computing the partial answers in the join of the first two atom then dominates the performance of reasoning and should be taken into account. The main drawback of measuring the work in terms of the number of PAR messages is that this number is determined not only by the dataset and the rules, but also by the order of atoms in rule bodies.

Test Setting LUBM and WatDiv are ideally suited to this experiment as we can scale the input datasets in a controlled manner. Thus, we generated LUBM datasets for 2k, 4k, 8k, and 10k universities, and WatDiv datasets with roughly 200 M, 400 M, 800 M, and 1 G triples. In contrast, MAKG* is a real-world dataset that cannot be scaled easily, so we did not consider it in this experiment. We used test setting described in Section 6.3, but we scaled the number of servers proportionally to the input size. We conducted all experiments using the $2PS_3$ partitioning strategy.

Table 5
Scalability experiments.

Data	Servers	Res.	Triples (M)		Mat.	Derivations		PAR messages		Reasoning
Size		(M)	Input	Output	time (s)	Total (G)	Rate (k/ss)	Total (G)	Local (%)	Rate (k/ss)
LUBM										
2k	2	66	267	664	4803	15.85	1620	1.43	99	1800
4k	4	131	534	1329	4457	31.73	1750	2.89	99	1942
8k	8	262	1068	2658	5275	63.45	1480	6.90	99	1667
10k	10	328	1335	3322	5198	79.30	1500	7.96	99	1679
WatDiv										
200 M	2	20	218	342	470	0.43	468	0.93	75	1447
400 M	4	40	436	649	810	0.81	250	2.10	46	898
800 M	8	79	873	1278	1285	1.66	160	6.35	32	780
1 G	10	97	1092	1749	1636	2.10	128	9.00	28	678

Results. The results of our scalability experiments are summarised in Table 5. For each dataset, we report the cluster size, number of resources in the input, the numbers of triples in the input and output, the materialisation time, the number of derivations, the derivation rate (i.e., the number of derivations per server per second), the number of PAR messages, the percentage of PAR messages that are local to a server (i.e., that are not sent over the network), and the total reasoning rate (i.e., sum of the numbers of derivations and PAR messages processed per server per second).

Discussion. In the two benchmarks, the amount of work scales differently with the input size. On LUBM, the number of PAR messages is an order of magnitude smaller than the number of derivations; hence, the benchmark is 'well-behaved' in the sense that most partial answers contribute to a derivation. Also, the overwhelming majority of PAR messages are local. This is unsurprising because each university in a LUBM dataset contains roughly the same number of triples, and there are relatively few connections between universities. Thus, the number of derivations scales roughly linearly with the input size, which allows DMAT to exhibit near-constant derivation and reasoning rates.

In contrast, the number of PAR messages on WatDiv is much larger than the number of derivations, it scales super-linearly with the input size, and the percentage of local messages decreases steeply as the input grows. We conjecture that this is because WatDiv datasets exhibit a highly irregular structure, so the difficulty of partition increases with the dataset size. As a result, the derivation rate drops significantly as the dataset increases in size: it is about 3.6 lower on the 1 G dataset than on the 200 M dataset. The reasoning rate also drops, but by a smaller factor of only 2.1. Nevertheless, note that the overall amount of data increases by a factor of five, so the decrease in the performance is still below the increase in overall data size.

To summarise, our reasoning approach seems to scale well when the overall work scales linearly with input size, and increasing the input size does not create a highly connected dataset that is difficult to partition. However, even in the latter case, our approach is still able to materialise large datasets with complex, recursive rule sets.

6.6. System comparison experiments

To see how DMAT compares to the state of the art, we compared it to BigDatalog [27] and Cog [28], which are based on Spark and Flink, respectively. We are grateful to the authors of both systems for their extensive assistance.

Test Systems. We obtained the source code of BigDatalog and Cog from public repositories and compiled it ourselves. Both systems rely on Apache Calcite,[4] an open source framework for building databases and data management systems, to compile logical plans into SQL and recursion operators. This framework, however, does not seem to correctly handle arbitrary recursive Datalog programs, and it also could not process larger programs. After extensive experimentation and discussion with the systems' authors, we were able to compile only small linear programs. To overcome this setback, in this experiment we selected from each Datalog program two rules, one of which was recursive.

As we mentioned in Section 3.1, BigDatalog and Cog are not classical materialisation systems; rather, they take as input a query and then materialise only a part of the program relevant to the query. Thus, when running BigDatalog and Cog, we used $\langle x, p, y\rangle$ as a query, where p is the property computed by the two rules. Moreover, BigDatalog and Cog are based on the standard relational data model, rather than the RDF model. We used the well-known *vertical partitioning* technique [60,61] to transform triples into relations: we introduced a binary relation p for each property p occurring in the input dataset or a rule; we converted each input triple $\langle s, p, o\rangle$ to tuple $\langle s, o\rangle$ in relation p; and we transformed each rule or query atom $\langle t_s, t_p, t_o\rangle$ into a relational atom $t_p(t_s, t_o)$, which was possible because t_p is never a variable. We also eliminated from the input datasets all triples of the form $\langle s, p, o\rangle$ where p does not occur in the body of the two rules; such triples cannot be matched by the rules so they do not contribute to materialisation, and this data reduction made our experiments more practical. Table 6 shows the numbers of input and output triples in the reduced datasets.

Test Setting. We conducted our experiments as follows. For BigDatalog and Cog, we copied the input datasets into the local directories of all ten materialisation servers, we instructed the systems to answer the query (which involved materialising the rules), and we measured the wall-clock time required. For DMAT, we used the test setting described in Section 6.3, but modified to use the two-rule program. In all tests, we measured the total amount of data sent over the network using the `ifconfig` command.

Results. Table 6 shows, for each test, the materialisation time and the total amount of data sent over the network. For DMAT, the table also shows the number of derivations, the number of PAR messages, and the percentage of PAR messages that are local; note that such metrics do not apply to BigDatalog and Cog. On MAKG*, BigDatalog ran out of memory, and Cog could not compile the program.

Discussion. As one can see from the table, DMAT consistently outperformed both systems. The difference is not as pronounced on WatDiv, but it is by a factor of three or more on LUBM. Moreover, even the reduced MAKG* program with just two rules was too complex for other systems: BigDatalog ran out of memory despite each server being equipped with 1 TB of storage that Spark could use for scratch data. While it is hard to be absolutely sure about the cause, we conjecture that this is because Spark

[4] https://calcite.apache.org.

Table 6
Systems comparison experiments.

Dataset	Triples (M)		BigDatalog		Cog		DMAT				
	Input	Output	Time (s)	TX	Time (s)	TX	Time (s)	TX	Der (G)	PAR (G)	Local (%)
LUBM	281	454	1107	90	1441	110	325	0	2.71	7.90	99
WatDiv	339	404	1114	68	1050	80	920	965	0.41	10.82	12
MAKG*	1784	2665	oom	–	Error	–	4081	290	6.37	16.02	84

Note: TX is the total amount of data in GB transmitted over the network in the cluster.

evaluates queries bottom-up and materialises the intermediate results, which can be costly. In contrast, DMAT uses nested index loop joins, and it processes local PAR messages without storing them: only partial answers that need to be sent to another server are 'materialised' in the sense that they are explicitly created and stored (e.g., in buffers of the networking stack). As a result, our rule evaluation strategy can be less memory-intensive when the rules are complex.

Our technique seems to be most effective when data can be partitioned well. As we observed in Section 6.3, the $2PS_3$ algorithm seems to detect the modular structure of LUBM, which considerably reduce communication. Indeed, most PAR messages are local on LUBM, so materialisation introduces very little communication. On WatDiv, $2PS_3$ is not as effective in reducing communication: only 12% of PAR messages are local, which makes reasoning about three times slower than on LUBM, and leads to an order of magnitude more communication than with the other two systems. On MAKG*, our techniques also seem to be very effective at reducing communication. In contrast, data partitioning in BigDatalog and Cog is not locality-aware, which incurs more communication and longer reasoning times; moreover, the difference in performance on LUBM and WatDiv is smaller because neither dataset is partitioned to explore locality.

7. Conclusion and further work

In this paper we have presented a novel approach to Datalog reasoning in distributed RDF systems. Our materialisation algorithm supports arbitrary Datalog rules over RDF data and arbitrary partitions of the input dataset. To the best of our knowledge, no other system has these traits. We have also presented two new streaming partitioning algorithms that enable our reasoner to process large datasets. We have shown experimentally that our techniques can considerably reduce communication, scale well in many cases, and are competitive with regard to the state of the art.

In our future work, we plan to extend our implementation with support for multi-threaded processing in the servers to improve parallelism. We will also aim to further improve materialisation performance by reducing imbalances in the workload among servers. One possibility might be to analyse the Datalog program before partitioning and thus identify workload hotspots. Furthermore, we aim to support more advanced features of Datalog, such as stratified negation and aggregation, which are needed in many practical applications. Another question that, to the best of our knowledge has not been considered in the literature thus far, is to efficiently support distributed incremental reasoning.

CRediT authorship contribution statement

Temitope Ajileye: Conceptualisation, Methodology, Software, Data curation, Writing – original draft. **Boris Motik:** Conceptualisation, Methodology, Writing – original draft, Writing – review & editing, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was supported by the SIRIUS Centre for Scalable Access in the Oil and Gas Domain, and the EPSRC project AnaLOG.

Appendix A. Proofs for Section 4

Theorem 4.1. *Let I be a dataset, let P be a Datalog program, and let $G_1, \dots, G_\ell$ be the datasets obtained by applying Algorithm 1 to an arbitrary partition of I as specified in this section. Then, $P^\infty(I) = G_1 \cup \dots \cup G_\ell$, and moreover the algorithm exhibits the nonrepetition property.*

Throughout this section, we fix an arbitrary Datalog program P, input dataset I, partition $\mathcal{P}$ of I, and a run of Algorithm 1 on $\mathcal{P}$ as specified in Section 4. We assume that the result of the algorithm's execution is equivalent to a run where all lines on all servers are executed in some sequential order—that is, we assume that our computation is sequentially consistent. Thus, each line of the algorithm is executed at some time instant i where $i \geq 0$, and time instant zero refers to the algorithm's start. This allows us to talk about some data structure (e.g., $\mu_{k,s}$ for some k) *at time instant i*, which is the state of the data structure just after the line at time instant i was executed. We next introduce several useful definitions.

We say that a fact f *occurs* at time instant i on some server k if server k contains f at that time instant. A constant c *occurs* at time instant i on server k at position $\pi \in \Pi$ if there exists a fact f that occurs at time instant i on server k such that $f|_\pi = c$. A constant c is *relevant* at time instant i to server k if c occurs in the head of a rule in P or in server k at any position at time instant i. We use 'occur initially' and 'occur eventually' to refer to the time instant zero and the time instant at algorithm's termination, respectively. Fact f is *derived on* server k if f does not occur initially on server k, but it occurs eventually on server k.

For each position $\pi \in \Pi$ and each constant c that occurs eventually on some server, we define the set of servers $L_\pi(c)$. If c occurs in the head of some rule in P, we let

$$L_\pi(c) = \{1, \dots, \ell\} \tag{31}$$

Otherwise, c occurs initially on some server, and we let

$$L_\pi(c) = \bigcap_{k \in R(c)} \mu_{k,\pi}(c), \tag{32}$$

where $R(c)$ is the set of servers for which c is initially relevant, and $\mu_{k,\pi}$ are the occurrence mappings of server k at time instant zero. Furthermore, we let $L(c) = \bigcup_{\pi \in \Pi} L_\pi(c)$. Each $L(c)$ contains the set of servers to which c is initially relevant: this is obvious when c occurs in the head of some rule in P; otherwise, if c occurs initially on some server j at position π, then $j \in \mu_{k,\pi}(c)$ holds for each $k \in R(c)$. Consequently, $L(c)$ is never empty.

We identify five types of events that are of interest in our proof, which refer to particular kinds of time instant.

- An event of type $\mathsf{process}^k(f)$ occurs after server k completes line 8 when processing a fact f.
- An event of type $\mathsf{par}^k(f, r, p, i)$ occurs after server k completes line 12 for a partial answer message with index i that originates from matching the pth atom in the body of rule r to fact f.
- An event of type $\mathsf{fct}^k(f)$ occurs just before server k reaches line 37 while processing a FCT message for fact f.
- An event of type $\mathsf{occ}^k(f, c, \pi, j)$ occurs after server k completes line 42 for constant c, position $\pi \in \Pi$ and a fact f derived on server j. Note that $f|_\pi = c$ does not necessarily hold.
- An event of type $\mathsf{add}^k(f)$ occurs when fact f is not present in server k before the time instant, and f is added to server k in line 44.

An event of the same type can occur more than once in the algorithm's run; for example, a fact f can be derived many times, and each derivation of f gives rise to a distinct event of type $\mathsf{occ}^k(f, c, \pi, j)$. Sometimes, we use a time instant index after the event type name to show both the event's instant and type; for example, $\mathsf{occ}^k_i(f, c, \pi, j)$ means that event at time instant i is of type $\mathsf{occ}^k(f, c, \pi, j)$.

Note that, for each event $\mathsf{add}^k_i(f)$ in the algorithm's run, each constant c in f, and each position $\pi \in \Pi$, the run contains an event $\mathsf{occ}^k_{i'}(f, c, \pi, k)$ such that $i' < i$.

A fact f *introduces* a constant c on server k at position $\pi \in \Pi$ if $f|_\pi = c$, constant c does not occur initially on server k at position π, fact f occurs eventually on server k, so event $\mathsf{occ}^k_i(f, c, \pi, k)$ occurs on server k for some i, and, for each event $\mathsf{occ}^k_j(g, c, \pi, k)$ with $f \neq g$ and $g|_\pi = c$ occurring on server k, we have $i < j$.

We are now ready to proceed with the proof of Theorem 4.1, which we split into several claims for clarity. First, we establish two properties that relate $L(c)$ to Algorithm 1.

Lemma A.1. *For each server k, each position π, each constant c not occurring in the head of a rule in P, each instant in the algorithm's run at which c occurs on server k, and for $\mu_{k,\pi}$ the occurrence mapping for position π at that instant, $L_\pi(c) \subseteq \mu_{k,\pi}(c)$ holds.*

Proof. The proof is by induction on time instants in the algorithm's run. For the base case at time instant zero, the claim follows immediately from the definition of L_π. For the induction step, we consider a time instant $i > 0$ such that claim holds at all instants i' with $i' < i$, and we show that the claim holds at time instant i as well.

If time instant i does not add a fresh constant c to server k, then the claim holds vacuously because the occurrence mappings never become smaller. If there exist a constant c and server k such that c does not occur at instant $i-1$ but occurs at instant i on server k, then instant i corresponds to an event of type $\mathsf{add}^k(f)$, where f is a fact that contains c. Consequently, there exists an event $\mathsf{occ}^k_{i'}(f, c, \pi, k)$ in the run for some instant i' with $i' < i$ and position $\pi \in \Pi$. Let λ_π be the partial occurrence mapping for position π attached to the OCC message at instant i'. The set $\lambda_\pi(c)$ was initialised in line 17 as $\mu_{j,\pi}(c)$ on some server j at some instant before i', and moreover constant c was obtained by matching a fact that occurs on server j at instant i'; thus, c occurs on server j at time instant i'. Consequently, the induction assumption ensures $L_\pi(c) \subseteq \mu_{j,\pi}(c)$ at instant i', which in turn ensures $L_\pi(c) \subseteq \mu_{k,\pi}(c)$ at instant i, as required. □

Lemma A.2. *For all servers k and j, each fact f, each constant c, and all position π and π' such that $j \in L(c)$ and f introduces c at position π on server k, an event of type $\mathsf{occ}^j(f, c, \pi', k)$ occurs on server j during the run.*

Proof. Consider arbitrary k, j, f, c, π, and π' as specified in the lemma. By the definition of 'introduces', fact f does not occur initially on server k, so there exists a time instant i such that event $\mathsf{add}^k_i(f)$ occurs on server k. Because of the algorithm order, event $\mathsf{fct}^k_{i'}(f)$ occurs on server k at some time instant i' with $i' < i$. Let $\mu_{k,\pi}$ be the occurrence mapping at instant i'. Then, we have $k \notin \mu_{k,\pi}(c)$ because f is the fact that introduces c at position π on server k, so the algorithm executes lines 32–36 for position π. We have the following two possibilities.

- If c occurs in the head of a rule in P, then all the servers of the cluster are added to the set D in line 34, which ensures $L(c) \subseteq D$.
- If c does not occurs in the head of a rule in P, then c was matched to a variable x in a body atom of some rule in P on some server, and x also occurs in the rule head. Let λ_s, λ_p, and λ_o be the partial occurrences for the FCT message at instant i'; clearly, D includes the sets $\lambda_s(c)$, $\lambda_p(c)$, and $\lambda_o(c)$. These sets were copied from the occurrence mappings $\mu_{k',\pi''}$ on some server k' at some time instant when c occurs in k', so Lemma A.1 ensures $L(c) \subseteq \bigcup_{\pi'' \in \Pi} \mu_{k',\pi''}(c)$; consequently, $L(c) \subseteq \bigcup_{\pi' \in \Pi} \lambda_{\pi'}(c)$ holds. Set D is extended in line 36 with these partial occurrences, which clearly ensures $L(c) \subseteq D$.

Either way, set D includes $L(c)$; thus, $j \in L(c)$ implies $j \in D$. Lines 38 and 47 ensure that an OCC message for f is sent to every server in D, so line 42 ensures that an event of type $\mathsf{occ}^j(f, c, \pi', k)$ occurs on server j. □

Second, we show that the consistency of occurrence mappings is maintained as the computation progresses. We will later use this to ensure that partial answers are sent to all relevant servers that can possibly match an atom.

Lemma A.3. *At each time instant, the collection of the occurrence mappings of all servers is consistent with P and the datasets stored in the servers at that instant.*

Proof. Let us fix an arbitrary time instant during the run of Algorithm 1. Let $\boldsymbol{G} = G_1, \ldots, G_\ell$ be the datasets and let $\boldsymbol{\mu} = \mu_{1,s}, \mu_{1,p}, \mu_{1,o}, \ldots, \mu_{\ell,s}, \mu_{\ell,p}, \mu_{\ell,o}$ be the occurrence mappings stored in each server at that instant. Moreover, let us fix arbitrary servers k and j, position $\pi \in \Pi$, and constant c that is relevant for P and G_k and that occurs at position π in G_j. We prove $j \in \mu_{k,\pi}(c)$ by considering four cases.

Case 1: Constant c is initially relevant for P and server k, but c does not occur initially on server j. Let f be the fact that introduces c at position π on server j. By the definition of L, we have $k \in L(c)$, so Lemma A.2 ensures that an event of type $\mathsf{occ}^k(f, c, \pi, j)$ occurs on server k. Let i be the time instant of the first event of that type. Since c occurs in G_j and each OCC message for f is sent to server j last, then i precedes the current time instant, so $j \in \mu_{k,\pi}(c)$ holds.

Case 2: Constant c is initially relevant for P and server k and it occurs initially on server j at position π. Occurrence mappings are correct initially, so $j \in \mu_{k,\pi}(c)$ holds.

Case 3: Constant c is not initially relevant for P and server k, and c does not occur initially on server j. Clearly, c does not occur in the head of a rule in P. Let f be the fact that introduces c in position π on server j. Now if $k = j$, then line 42 is executed before line 44 when processing the OCC message for fact f, so $j \in \mu_{k,\pi}(c)$ holds. To complete this case, we assume that $k \neq j$ holds. Let f' be the fact that introduces c in any position on server k and thus causes c to become relevant to k, let π' be a position of c in f' (i.e., we choose one π' if c occurs in f' more than once), and choose arbitrarily some m in $L(c)$ (which is possible due to $L(c) \neq \emptyset$). By Lemma A.2, there exist events $\mathsf{occ}^m_{i_1}(f, c, \pi, j)$ and

$\mathsf{occ}^m_{i_2}(f', c, \pi', k)$ both occurring on server m. These events are distinct because of $k \neq j$, and we can also assume that they are the first events of their respective types. Since c is relevant to k and it occurs in G_j at position π at the current time instant, both i_1 and i_2 precede the present moment. We now consider two possibilities.

- Assume $i_1 < i_2$. By our assumption, c is relevant to k at the current time instant, so there exists an event $\mathsf{occ}^k_{i_3}(f', c, \pi', k)$ occurring on server k before the current point in time; clearly, $i_2 < i_3$. Let λ_s, λ_p, and λ_o be the partial occurrence mappings in the OCC message for f' at point i_3. Then, event $\mathsf{occ}^m_{i_1}(f, c, \pi, j)$ ensures that j is present in the occurrence mappings of server m at instant i_1; this occurs before event $\mathsf{occ}^m_{i_2}(f', c, \pi', k)$, where line 43 ensures $j \in \lambda_\pi(c)$. Thus, when the OCC message for f' is processed on server k at point i_3, line 42 ensures $j \in \mu_{k,\pi}(c)$.
- Assume $i_2 < i_1$. Since c occurs at position π on server j at the current time instant, event $\mathsf{occ}^j_{i_3}(f, c, \pi, j)$ occurring on server j introduces c at position π to server j before the current instant; since the OCC message for f is sent to server j last, we have $i_1 < i_3$. Event $\mathsf{occ}^m_{i_2}(f', c, \pi', k)$ ensures that k is present in the occurrence mappings on server m at instant i_2; this occurs before event $\mathsf{occ}^m_{i_1}(f, c, \pi, j)$, and so line 43 ensures that k is added to the set D. Thus, an event of type $\mathsf{occ}^k(f, c, \pi, j)$ happens on server k before i_3 because each OCC message for f is sent to server j last. Finally, line 32 ensures $j \in \lambda_\pi(c)$ where λ_π is the partial occurrence mapping from the OCC message, so clearly $j \in \mu_{k,\pi}(c)$ holds.

Case 4: Constant c is not initially relevant for P and server k, but c occurs initially on server j at position π. Then, $j \in L_\pi(c)$ holds. Let f be the fact that introduces c in any position on server k. Then, there exist a position π' and an instant before the current one such that at which $\mathsf{occ}^k_i(f, c, \pi', k)$ occurs on server k. Let λ_π be the partial occurrence mapping for position π in the OCC message at instant i'. Set $\lambda_\pi(c)$ was read from some server k' at some time instant when c occurs in k', so Lemma A.1 ensures that $\lambda_\pi(c)$ includes $L_\pi(c)$; thus, line 42 ensures that j present in the occurrence mappings on server k at instant i, and so $j \in \mu_{k,\pi}(c)$ holds. □

Third, we show that the occurrence mappings for the subject position are maintained in a way that ensures that all facts with the same subject are stored on the same server.

Lemma A.4. *For each constant c, at each time instant set $\bigcup_{1 \leq k \leq \ell} \mu_{s,k}(c)$ contains at most one element; moreover, if this set contains a server j, then either $j \in \bigcup_{1 \leq k \leq \ell} \mu_{s,k}(c)$ holds at time instant zero, or $c \bmod \ell = j$.*

Proof. The proof is by induction on the time instants in the algorithm's run. The base case at time instant zero holds because the occurrence mappings are initially correct. For the induction step, we consider a time instant $i > 0$ such that claim holds at all instants i' with $i' < i$, and we show that the claim holds at time instant i as well.

The claim holds vacuously if no occurrence mapping changes at time instant i. Thus, we assume that there exist servers k and j, constant c, and fact f such that f is derived on server j and contains c, and event $\mathsf{occ}^k_i(f, c, s, j)$ occurs at time instant i. Let $\mu_{k,s}$ be the occurrence mapping on server k at instant $i - 1$, and let λ_s be the partial occurrence mapping from the OCC message being processed at instant i. There exist a server k' and time instant m such that $\lambda_s(c)$ is initially set to $\mu_{k',s}(c)$ at instant $m < i$. Note that, after initialisation, $\lambda_s(c)$ can later change only in line 32 or 43.

We assumed that the subject occurrence mappings of server k change at instant i, so $\lambda_s(c) \neq \emptyset$ holds. We next show that $\lambda_s(c)$ has exactly one element. For the sake of a contradiction, assume that $\lambda_s(c)$ contains two or more elements. Now consider an arbitrary $j' \in \lambda_s(c)$ such that $j' \neq c \bmod \ell$. Then, j' is not added to λ_s via lines 21 and 32, so server j' must be present in some subject occurrence mapping for c at time instant zero. Since occurrence mappings are correct at that instant, constant c occurs initially in subject position on some server j', and moreover $L_s(c) = \{j'\}$. Since this holds for arbitrary j' and $L_s(c)$ is uniquely defined, $\lambda_s(c)$ can contain at most two elements: one with value $j' \neq c \bmod \ell$, and another one with value $j'' = c \bmod \ell$; note that $L_s(c) = \{j'\}$ holds, so $j'' \notin L_s(c)$—that is, constant c does not occur initially on server j'' in subject position. Now Lemma A.1 ensures that $L_s(c) \subseteq \mu_{k',s}(c)$ holds at instant m, and the induction assumption holds for $\mu_{k',s}$ at instant m, so $\mu_{k',s}(c) = \{j'\}$ holds at instant m. Thus, when the destination server for fact f is determined in lines 21–22, the partial occurrence for the subject position for c contains exactly j', so the destination server is selected in line 22 as j'. In other words, no new server added to $\lambda_s(c)$ in line 32 when a FCT message for f is processed. If j'' were added at line 43, then there exist an instant $i'' < i$ and server k'' such that $j'' \in \mu_{k'',s}(c)$ at that instant. The induction assumption holds for instant i'', so the set $\bigcup_{1 \leq k \leq \ell} \mu_{s,k}(c)$ at instant i'' contains at most one element; thus, $j'' = j'$, which contradicts our assumption that $\lambda_s(c)$ has two or more elements.

Now if $\mu_{k,s}(c) = \emptyset$ holds, then adding $\lambda_s(c)$ to $\mu_{k,s}(c)$ clearly does not violate the inductive claim. Thus, assume that $\mu_{k,s}(c) \neq \emptyset$. By the induction assumption, $\mu_{k,s}(c)$ contains just one server j''. The inductive claim clearly holds if $j' = j''$. For the sake of a contradiction, assume that $j' \neq j''$. Then either j' or j'' must be different from $c \bmod \ell$. In the same way as in the previous paragraph, we can conclude that c then occurs initially in subject position in both j' and j'', which contradicts our assumption that occurrences are correct at time instant zero. □

Lemma A.4 straightforwardly ensures that $\lambda_s(h\sigma|_s)$ in line 21 contains at most one server, and therefore all facts with the same subject are stored on the same server. Thus, each fact is stored on precisely one server, which ensures that the algorithm's run contains at most one event of the following types for each fact f.

- An event of type $\mathsf{add}^k(f)$ can occur at most once because f is derived on a server uniquely identified by its subject and duplicates are eliminated.
- An event of type $\mathsf{process}^k(f)$ can occur at most once because of the observation in the previous item, and moreover each fact in a server is processed once by the function ProcessFact.

For f a fact, we define $T(f)$ as the value of $T_k(f)$ upon the algorithm's termination where k is the server whose partition element contains fact f at that instant; since each fact is stored and assigned a timestamp on just one server, there is precisely one such k for each fact f.

Fourth, we show that the chronological 'happens-before' relationship on the events agrees with the timestamps assigned to the facts involved in the events.

Lemma A.5. *Relationship $T(f_1) < T(f_2)$ holds for events*

- $\mathsf{par}^k_{i_1}(f_1, r, p, j)$ *and* $\mathsf{add}^k_{i_2}(f_2)$ *such that* $i_1 < i_2$,
- $\mathsf{process}^k_{i_1}(f_1)$ *and* $\mathsf{occ}^k_{i_2}(f_2, c, \pi, j)$ *such that* $i_1 < i_2$ *and there exists no event* $\mathsf{occ}^k_{i_3}(f_2, c, \pi, j)$ *with* $i_3 < i_2$, *and*
- $\mathsf{par}^k_{i_1}(f_1, r, p, i)$ *and* $\mathsf{occ}^k_{i_2}(f_2, \pi, j)$ *such that* $i_1 < i_2$ *and there exists no event* $\mathsf{occ}^k_{i_3}(f_2, c, \pi, j)$ *with* $i_3 < i_2$.

Proof. Consider events $\mathsf{par}^k_{i_1}(f_1, r, p, j)$ and $\mathsf{add}^k_{i_2}(f_2)$ with $i_1 < i_2$. The PAR message from the first event contains a timestamp $\tau = T(f_1)$. Thus, after the call to SYNCHRONISE in line 12, the value of the local clock C_k is strictly larger than $T(f_1)$. Given that events of type $\mathsf{add}^k(f_2)$ do not repeat, fact f_2 cannot be added before instant i_2. Thus, when fact f_2 is later assigned a timestamp, we have $T(f_1) < T(f_2)$.

Consider events $\mathsf{process}^k_{i_1}(f_1)$ and $\mathsf{occ}^k_{i_2}(f_2, c, \pi, j)$ such that $i_1 < i_2$ and there exists no event $\mathsf{occ}^k_{i_3}(f_2, c, \pi, j)$ with $i_3 < i_2$. Clearly, the timestamp τ in first event has the value of $T_k(f_1)$, so, after the call to SYNCHRONISE in line 8, the value of the local clock C_k is strictly larger than $T(f_1)$. If the set of servers that need to be updated by the OCC message for f_2 is empty, then f_2 is added to the partition element of server k and the timestamp of f_2 is set to the current value of C_k in line 44, which ensures $T(f_1) < T(f_2)$. Otherwise, server k attaches the current value of C_k to the new OCC message produced for f_2 in line 47. Before f_2 is added to some server k', server k' calls SYNCHRONISE in line 40, which ensures $T(f_1) < T(f_2)$, as required.

The case of events $\mathsf{par}^k_{i_1}(f_1, r, p, i)$ and $\mathsf{occ}^k_{i_2}(f_2, \pi, j)$ such that $i_1 < i_2$ and there exists no event $\mathsf{occ}^k_{i_3}(f_2, c, \pi, j)$ with $i_3 < i_2$ is analogous to above, but uses line 12 instead of line 8. □

For the rest of this section, we fix $G_1, \dots, G_\ell$ as the datasets computed in each server after the algorithm terminates. We now complete the proof of Theorem 4.1 by considering the soundness, completeness, and nonrepetition properties of the algorithm.

Lemma A.6. *It is the case that* $G_1 \cup \cdots \cup G_\ell \subseteq P^\infty(I)$.

Proof. The proof is by induction on the construction of sets G_i. The argument is straightforward so we just present a sketch: when (σ, a, Q, h) is returned on some server k in line 9, substitution σ satisfies $a\sigma \in G_k$; moreover, as matching of Q progresses, each substitution σ' returned in line 13 satisfies $a_i\sigma' \in G_{k'}$; consequently, each substitution σ in line 23 is an answer to the annotated query Q. Thus, each such σ matches all body atoms of the rule corresponding to (σ, a, Q, h) in $P^\infty(I)$, and so we clearly have $h\sigma \in P^\infty(I)$. □

Lemma A.7. *It is the case that* $P^\infty(I) \subseteq G_1 \cup \cdots \cup G_\ell$.

Proof. Let $P^i(I)$ be the sets from the construction of $P^\infty(I)$ as defined in Section 2. The claim follows from the following property:

[$*$] for each i and each fact $f \in P^i(I)$, there exists k such that $f \in G_k$.

The proof is by induction on i. The base case holds trivially, so we assume that $(*)$ holds for some $i \geq 0$ and show that it also holds for $i+1$. To this end, we consider an arbitrary fact $f \in P^{i+1}(I) \setminus P^i(I)$. This fact is derived by a rule $r := h \leftarrow b_0 \wedge \cdots \wedge b_n \in P$ and substitution σ such that $h\sigma = f$ and $b_j\sigma \in P^i(I)$ for $0 \leq j \leq n$. Now choose p as the smallest integer between 0 and n such that $T(b_{p'}\sigma) \leq T(b_p\sigma)$ holds for each $0 \leq p' \leq n$. Let $a_0, \dots, a_n$ be the body atoms of the rule rearranged so that $a_0 = b_p$ is the pivot atom, and the remaining atoms correspond to the annotated query $Q = a_1^{\bowtie_1} \wedge \cdots \wedge a_n^{\bowtie_n}$ returned by MATCHRULES$(b_p\sigma, P)$ in line 9 on fact $b_p\sigma$. Finally, for each $0 \leq j \leq n$, let σ_j be the substitution σ restricted to all variables occurring in atoms $a_0, \dots, a_j$ and let $\tau_j = T(a_j\sigma)$; moreover, $(*)$ holds for i by the induction assumption, so there exists a server k_j such that $a_j\sigma \in G_{k_j}$ holds. We next prove the following:

[$\diamond$] for each j with $0 \leq j \leq n$, a time instant exists when FINISHMATCH$(j, \sigma_j, a_j, Q, h, \tau_0, \boldsymbol{\lambda}_j)$ is called for some mapping $\boldsymbol{\lambda}_j$.

Property $(\diamond)$ implies our claim because, in lines 18–23, the algorithm then constructs a FCT message for $h\sigma$ and dispatches it to some server k_h, so $h\sigma$ is later added to server k_h in line 44, as required for $(*)$.

We prove $(\diamond)$ by induction on $0 \leq j \leq n$. For the base case, fact $a_0\sigma$ occurs initially in server k_0. Consequently, PROCESSFACT$(a_0\sigma, T_k(a_0\sigma))$ is called on server k_0, so a call to MATCHRULES$(a_0\sigma, P)$ returns (σ_0, a_0, Q, h), after which FINISHMATCH$(0, \sigma_0, a_0, Q, h, \tau_0, \emptyset)$ is called in line 10, as required. For the induction step, we assume that $(\diamond)$ holds for some $0 \leq j < n$, and we show that it holds for $j+1$ as well. By the induction assumption, fact $a_{j+1}\sigma$ occurs eventually in some server k_{j+1}.

Assume now that event $\mathsf{par}^{k_{j+1}}_m(a_0\sigma, r, p, j+1)$ occurs at some time instant m. Server k_{j+1} then calls EVALUATE at line 13 for $a_{j+1}^{\bowtie_{j+1}}$. We next show that server k_{j+1} contains $a_{j+1}\sigma$ at the time instant when line 13 is executed. We have the following possibilities.

- If no event of type $\mathsf{add}^{k_{j+1}}(a_{j+1}\sigma)$ ever happens, then fact $a_{j+1}\sigma$ occurs initially on server k_{j+1}.
- If an event $\mathsf{add}^{k_{j+1}}_{m'}(a_{j+1}\sigma)$ occurs at time instant m' such that $m' < m$, then server k_{j+1} clearly contains fact $a_{j+1}\sigma$ when line 13 is executed.
- Assume now that event $\mathsf{add}^{k_{j+1}}_{m'}(a_{j+1}\sigma)$ happens at time instant m' with $m' > m$. Then, Lemma A.5 implies $T(a_0\sigma) < T(a_{j+1}\sigma)$, which contradicts our assumption that $T(a_{j+1}\sigma) \leq T(a_0\sigma)$.

Moreover, if $T(a_{j+1}\sigma) = T(a_0\sigma)$, since $a_0 = b_p$ was chosen so that p is the least index of a body atom matched to a fact with timestamp $T(a_0\sigma)$, the shape of Q from (12) ensures that $\bowtie_{j+1} = \leq$. Thus, the call to EVALUATE in line 13 on server k_{j+1} returns σ_{j+1}, so the call in line 14 ensures $(\diamond)$.

To complete the proof, we assume that no event of type $\mathsf{par}^{k_{j+1}}(a_0\sigma, r, p, j+1)$ occurs during the algorithm's run (i.e., server k_j never forwards a PAR message to server k_{j+1}) and show that this leads to a contradiction. Under this assumption, there exists a position $\pi \in \Pi$ such that, for $c = a_{j+1}\sigma_j|_\pi$, we have $k_{j+1} \notin \lambda_\pi(c)$ at the time instant when line 26 is executed on server k_j, and so k_{j+1} is removed from D. However, this $\lambda_\pi(c)$ is populated in line 17 when, for some index $0 \leq s \leq j$ of an atom a_s, constant c is matched on server k_s at some position π^s; hence, at that instant we have $k_{j+1} \notin \mu_{k_s,\pi}(c)$. Now if no event of type $\mathsf{add}^{k_{j+1}}(a_{j+1}\sigma)$ ever happens, then $a_{j+1}\sigma$ would occur initially on server k_{j+1}; but then, since c is relevant to k_s, Lemma A.3 implies $k_{j+1} \in \mu_{k_s,\pi}(c)$, which is a contradiction. Consequently, an event of type $\mathsf{add}^{k_{j+1}}(a_{j+1}\sigma)$ occurs on server k_{j+1}. Moreover, let α be the event type $\mathsf{process}^{k_s}(a_0\sigma)$ if $s = 0$, and the event type $\mathsf{par}^{k_s}(a_0\sigma, r, p, s)$ otherwise. By the induction assumption, property $(\diamond)$ holds for s, so an event of type α occurs on server k_s. Clearly, an event of type α can occur only once during the algorithm's run, so let m_1 be the corresponding time instant.

Let g be the fact that introduces c in position π on server k_{j+1}; such a fact exists because $a_{j+1}\sigma$ contains c in position π and it is added to server k_{j+1}. Let g' be the fact that introduces c in any position in k_s, and let π' be the position of c in g'. Such a fact exists because c is matched on k_s. Choose an arbitrary server $k' \in L(c)$. By Lemma A.2, there exist time instants m_2 and m_3 such that $\mathsf{occ}^{k'}_{m_2}(g', c, \pi', k_s)$ and $\mathsf{occ}^{k'}_{m_3}(g, c, \pi, k_{j+1})$ occur on server k'. If $m_3 < m_2$, then k_{j+1} would be added to the occurrence mappings of k' at instant m_3 when the OCC message for g is processed, and then to the partial occurrence mappings in the OCC message for g' at instant m_2. Thus, $k_{j+1} \in \mu_{k_s,\pi}$ would hold at instant m_1, which contradicts our assumption that $k_{j+1} \notin \mu_{k_s,\pi}(c)$ at that instant.

Therefore, we have $m_2 < m_3$. Event $\text{occ}^{k'}_{m_2}(g', c, \pi', k_s)$ then ensures that $k_s \in \mu_{k',\pi}$ holds at instant m_2, and event $\text{occ}^{k'}_{m_3}(g, c, \pi, k_{j+1})$ ensures that $k_s \in D$ holds after line 36 at instant m_3; thus, an event of type $\text{occ}^{k_s}(g, c, \pi, k_{j+1})$ occurs and adds k_{j+1} to $\mu_{k_s,\pi}(c)$. We know that $k_{j+1} \notin \mu_{k_s,\pi}(c)$ holds when c is matched by σ, so the event of type α must happen before all events of type $\text{occ}^{k_s}(g, c, \pi, k_{j+1})$. Consequently, any OCC message for g that is forwarded to server k_{j+1} ensures that $C_{k_{j+1}}$ is set to a value higher than $T(a_0\sigma)$ with the call to SYNCHRONISE in line 40. Because of how g is defined, there exists an event of type $\text{occ}^{k_{j+1}}(g, c, \pi, k_{j+1})$ that precedes all events of type $\text{occ}^{k_{j+1}}(a_{j+1}\sigma, c, \pi, k_{j+1})$, and therefore, when $a_{j+1}\sigma$ is added to server k_{j+1}, we have $T(a_0\sigma) < T(a_{j+1}\sigma)$, which contradicts our assumption that $T(a_{j+1}\sigma) \leq T(a_0\sigma)$ holds. □

Lemma A.8. *No derivations are repeated in the run.*

Proof. Assume that PROCESSFACT considers two facts f_1 and f_2, both of which matched the same rule and produce the same substitution σ. Let b_1 and Q_1 be the pivot atom and the annotated query returned in line 9 when f_1 is processed, and let b_2 and Q_2 be defined analogously. Thus, $b_1\sigma = f_1$ and $b_2\sigma = f_2$. Since each fact is processed only once, atoms b_1 and b_2 are distinct. Now w.l.o.g. let us assume that b_1 occurs before b_2 in the body of the rule; thus, the atom corresponding to b_2 in Q_1 is annotated with $\leq$, and the atom corresponding to b_1 in Q_2 is annotated with $<$. But then, f_2 is not matched by Q_1 if $T(f_1) < T(f_2)$ holds, and f_1 is not matched by Q_2 if $T(f_1) \geq T(f_2)$ holds, which contradicts our assumption that the algorithm repeats inferences. □

Appendix B. Proofs for Section 5.2

To prove Proposition 5.1, we need to reason about the state of the counters N_k from the HDRF$_3$ algorithm. Thus, in the rest of this appendix, we use N^i_k to refer to the value of N_k from Algorithm 2 after processing the ith fact of G.

Lemma B.1. *For $\alpha > 1$ and $\lambda > 0$, in each run of Algorithm 2 on a dataset G,*

$$\max_k N^i_k - \min_k N^i_k < M_\lambda \tag{33}$$

holds after processing the ith fact of G, where

$$M_\lambda = |G|\sqrt{\frac{4\alpha}{\ell\lambda}} + \max_c |G^+(c)|.$$

Proof. We prove the claim by induction on the index i of the fact being processed. For the induction base, the claim is clearly true for $i = 0$. For the induction step, assume that property (33) holds after the ith fact has been processed, and consider processing fact $\langle s_{i+1}, p_{i+1}, o_{i+1}\rangle$. If $T(s_{i+1})$ is defined, then $N^{i+1}_k = N^i_k$ for each server k, so (33) clearly holds. Otherwise, let k_1 and k_2 be the servers such that $N^{i+1}_{k_1}$ and $N^{i+1}_{k_2}$ are minimal and maximal, respectively, among all N^{i+1}_k at step $i+1$. If $N^i_{k_2}$ is also maximal among all N^i_k at step i and fact $\langle s_{i+1}, p_{i+1}, o_{i+1}\rangle$ is sent to a server different from k_2, then property (33) clearly holds at step $i+1$. Thus, the only remaining case is when the fact is sent to server k_2. The scores for k_1 and k_2 are as follows, for $j \in \{1, 2\}$:

$$\text{SCORE}_j = C_{REP,j} + \lambda\frac{\sum_k N^i_k}{|G|}C_{BAL,j}$$

For convenience, let $S = \sum_k N^i_k$. We can bound SCORE_1 as follows:

$$\begin{aligned}\text{SCORE}_1 &= C_{REP,1} + \lambda\frac{S}{|G|}C_{BAL,1}\\ &\geq \lambda\frac{S}{|G|}C_{BAL,1}\\ &= \frac{\lambda S}{|G|}\left(1 - \ell\frac{N^i_{k_1} + |G^+(s_{i+1})|}{\alpha|G|}\right)\end{aligned}$$

Moreover, we can bound SCORE_2 as follows using the fact that the definition of $C_{REP,2}$ clearly ensures $C_{REP,2} < 4$:

$$\begin{aligned}\text{SCORE}_2 &= C_{REP,2} + \frac{\lambda S}{|G|}C_{BAL,2}\\ &< 4 + \frac{\lambda S}{|G|}\left(1 - \ell\frac{N^i_{k_2} + |G^+(s_{i+1})|}{\alpha|G|}\right)\end{aligned}$$

Fact $\langle s_{i+1}, p_{i+1}, o_{i+1}\rangle$ is sent to k_2, so $\text{SCORE}_1 \leq \text{SCORE}_2$. Combined with the above bounds for SCORE_1 and SCORE_2, we make the following observation.

$$\begin{aligned}&\tfrac{\lambda S}{|G|}\left(1 - \ell\tfrac{N^i_{k_1} + |G^+(s_{i+1})|}{\alpha|G|}\right) <\\ &\qquad 4 + \tfrac{\lambda S}{|G|}\left(1 - \ell\tfrac{N^i_{k_2} + |G^+(s_{i+1})|}{\alpha|G|}\right)\end{aligned}$$

This can be rewritten as

$$\frac{\lambda S}{|G|}\left(-\ell\frac{N^i_{k_1}}{\alpha|G|}\right) < 4 + \frac{\lambda S}{|G|}\left(-\ell\frac{N^i_{k_2}}{\alpha|G|}\right),$$

which can further be rewritten as

$$N^i_{k_2} - N^i_{k_1} < 4\frac{\alpha|G|}{\ell\lambda}\frac{|G|}{S}.$$

Now $N^i_{k_2} - N^i_{k_1} < S$ clearly holds at each step i, so

$$N^i_{k_2} - N^i_{k_1} < 4\frac{\alpha|G|}{\ell\lambda}\frac{|G|}{N^i_{k_2} - N^i_{k_1}}.$$

This can be rewritten as follows.

$$\begin{aligned}(N^i_{k_2} - N^i_{k_1})^2 &< 4\frac{\alpha|G|^2}{\ell\lambda}\\ N^i_{k_2} - N^i_{k_1} &< |G|\sqrt{\frac{4\alpha}{\ell\lambda}}\end{aligned}$$

Since $|G^+(s_{i+1})| \leq \max_c |G^+(c)|$, we have

$$N^i_{k_2} + |G^+(s_{i+1})| < N^i_{k_1} + |G|\sqrt{\frac{4\alpha}{\ell\lambda}} + \max_c |G^+(c)|.$$

Furthermore, $N^i_{k_2} + |G^+(s_{i+1})| = N^{i+1}_{k_2}$ and the definition of M_λ ensure that this formula can be rewritten as

$$N^{i+1}_{k_2} < N^i_{k_1} + M_\lambda.$$

Finally, $N^i_{k_1} = N^{i+1}_{k_1}$ holds since the fact is sent to server k_2, so the last observation proves our claim. □

Proposition 5.1. *Algorithm 2 produces a partition that satisfies $|G_k| \leq \alpha\frac{|G|}{\ell}$ for each $1 \leq k \leq \ell$ whenever α and λ are selected such that*

$$\alpha > 1 + \ell\frac{\max_c |G^+(c)|}{|G|} \quad \text{and} \quad \lambda \geq \frac{4\alpha}{\ell\left(\frac{\alpha-1}{\ell} - \frac{\max_c |G^+(c)|}{|G|}\right)^2}.$$

Proof. Let $\alpha > 1$ and λ be as stated in the proposition. Note that the condition on α ensures

$$\frac{\alpha - 1}{\ell} - \frac{\max_c |G^+(c)|}{|G|} > 0.$$

We now show that $M_\lambda \leq (\alpha - 1)\frac{|G|}{\ell}$ holds. Towards this goal, we make the following observations, each obtained from previous one using standard algebraic identities.

$$\lambda \geq \frac{4\alpha}{\ell\left(\frac{\alpha-1}{\ell} - \frac{\max_c |G^+(c)|}{|G|}\right)^2}$$

$$\sqrt{\frac{4\alpha}{\lambda\ell}} \leq \frac{\alpha - 1}{\ell} - \frac{\max_c |G^+(c)|}{|G|}$$

$$|G|\sqrt{\frac{4\alpha}{\lambda\ell}} \leq |G|\frac{\alpha - 1}{\ell} - \max_c |G^+(c)|$$

$$|G|\sqrt{\frac{4\alpha}{\lambda\ell}} + \max_c |G^+(c)| \leq (\alpha - 1)\frac{|G|}{\ell}$$

Using the definition of M_λ from Lemma B.1, the last inequality can be rewritten as

$$M_\lambda \leq (\alpha - 1)\frac{|G|}{\ell}.$$

Let $\mathcal{P} = G_1, \ldots, G_\ell$ be the partition produced by Algorithm 2. Clearly, we have $\min_k |G_k| \leq \frac{|G|}{\ell}$. Now consider an arbitrary server k. Property (33) of Lemma B.1 ensures $|G_k| \leq |G|/\ell + M_\lambda$, which together with the upper bound on M_λ proved above ensures

$$|G_k| \leq \frac{|G|}{\ell} + (\alpha - 1)\frac{|G|}{\ell} = \alpha\frac{|G|}{\ell}.$$

This holds for each server k, which implies our claim. □

Appendix C. Proofs for Section 5.3

Proposition 5.2. *Algorithm 3 produces a partition that satisfies $|G_k| \leq \alpha\frac{|G|}{\ell}$ for each $1 \leq k \leq \ell$ whenever*

$$\alpha > 1 + \frac{\max_c |G^+(c)|}{|G|}.$$

Proof. For each community m, the following property holds at each point during algorithm's execution:

$$S(m) = \sum_{c \text{ with } M(c)=m} |G^+(c)| \tag{34}$$

In particular, S is initialised to $S(m_c) = |G^+(c)|$ for each constant c. Moreover, lines 70 and 71 clearly preserve this property when mapping M is updated in line 72.

We prove by induction that function AssignCommunities ensures the following inequality:

$$\max_k N_k - \min_k N_k \leq (\alpha - 1)\frac{|G|}{\ell}. \tag{35}$$

For the induction base, all N_k are initialised to zero, so Eq. (35) holds after line 74. For the induction step, assume that Eq. (35) holds before line 77 is applied to a community m. Let $k_1 = \arg\min_k N_k$ and $k_2 = \arg\max_k N_k$, and let N'_k be the updated values of N_k after line 77; we clearly have $N'_k = N_k$ for all $k \neq k_1$, $N'_{k_1} = N_{k_1} + S(m)$, and $\min_k N'_k \geq \min_k N_k$. We have two possibilities.

- $N'_{k_1} \leq N_{k_2}$. Then, $\max_k N'_k = N_{k_2}$ and so the following condition holds, where the induction assumption ensures the second inequality:

$$\max_k N'_k - \min_k N'_k \leq \max_k N_k - \min_k N_k \leq (\alpha - 1)\frac{|G|}{\ell}.$$

- $N'_{k_1} > N_{k_2}$. Then, $\max_k N'_k = N_{k_1} + S(m)$, and the requirement on the choice of α in our claim and the condition in line 69 of the algorithm ensure that $S(m) \leq \frac{(\alpha-1)|G|}{\ell}$ holds for each community m at any point in time during an algorithm's run. This, in turn, ensures the following property:

$$\max_k N'_k - \min_k N'_k = S(m) \leq (\alpha - 1)\frac{|G|}{\ell}.$$

Thus, (35) holds. In addition, at the end of function AssignCommunities, so $\min_k N_k \leq \frac{|G|}{\ell}$ because $\sum_k N_k = |G|$. This, in turn, ensures

$$\max_k N_k \leq \min_k N_k + (\alpha - 1)\frac{|G|}{\ell} \leq \alpha\frac{|G|}{\ell}.$$

In the second phase, each triple $\langle s, p, o\rangle$ is assigned to server $T(M(s))$. But then, Eq. (34) clearly ensures $|G_k| = N_k$ for each k, which implies our claim. □

References

[1] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison Wesley, 1995.

[2] B.N. Grosof, I. Horrocks, R. Volz, S. Decker, Description logic programs: Combining logic programs with description logic, in: Proc. of the 12th Int. World Wide Web Conference (WWW 2003), ACM, Budapest, Hungary, 2003, pp. 48–57.

[3] B. Motik, B.C. Grau, I. Horrocks, Z. Wu, A. Fokoue, C. Lutz, OWL 2 web ontology language: Profiles, in: W3C Recommendation, 2009.

[4] R. Piro, Y. Nenov, B. Motik, I. Horrocks, P. Hendler, S. Kimberly, M. Rossman, Semantic technologies for data analysis in health care, in: Proc. of the 15th Int. Semantic Web Conference (ISWC 2016), in: LNCS, vol. 9982, Springer, Kobe, Japan, 2016, pp. 400–417.

[5] B. Luteberget, C. Johansen, Efficient verification of railway infrastructure designs against standard regulations, Form. Methods Syst. Des. 52 (1) (2018) 1–32.

[6] S. Meyer, A. Carter, A. Rodriguez, Liquid: The soul of a new graph database, part 2, 2021, URL: https://engineering.linkedin.com/blog/2020/liquid--the-soul-of-a-new-graph-database--part-2, Accessed on 26/11/2021.

[7] A. Harth, J. Umbrich, A. Hogan, S. Decker, YARS2: A federated repository for querying graph structured data from the web, in: Proc. of the 6th Int. Semantic Web Conference (ISWC 2007), in: LNCS, vol. 4825, Busan, Korea, 2007, pp. 211–224.

[8] S. Harris, N. Lamb, N. Shadbol, 4Store: The design and implementation of a clustered RDF store, in: Proc. of the 5th Int. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2009), in: CEUR Workshop Proceedings, vol. 517, Washington DC, USA, 2009, pp. 94–109.

[9] J. Huang, D.J. Abadi, K. Ren, Scalable SPARQL querying of large RDF graphs, PVLDB 4 (11) (2011) 1123–1134.

[10] K. Zeng, J. Yang, H. Wang, B. hao, Z. Wang, A distributed graph engine for web scale RDF data, PVLDB 6 (4) (2013) 265–276.

[11] K. Rohloff, R.E. Schantz, Clause-iteration with MapReduce to scalably query data graphs in the SHARD graph-store, in: Proc. of the 4th Int. Workshop on Data-Intensive Distributed Computing (DIDC 2011), ACM, San Jose, CA, USA, 2011, pp. 35–44.

[12] K. Lee, L. Liu, Scaling queries over big RDF graphs with semantic hash partitioning, PVLDB 6 (14) (2013) 1894–1905.

[13] L. Galárraga, K. Hose, R. Schenkel, Partout: a distributed engine for efficient RDF processing, in: Proc. of the 23rd Int. World Wide Web Conference (WWW 2014), ACM, Seoul, Korea, 2014, pp. 267–268.

[14] R. Al-Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, M. Sahli, Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning, VLDB J. 25 (3) (2016) 355–380.

[15] S. Gurajada, S. Seufert, I. Miliaraki, M. Theobald, TriAD: A distributed shared-nothing RDF engine based on asynchronous message passing, in: Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD 2014), Snowbird, UT, USA, 2014, pp. 289–300.

[16] B. Wu, Y. Zhou, P. Yuan, H. Jin, L. Liu, SemStore: A semantic-preserving distributed RDF triple store, in: Pro. of the 23rd ACM Int. Conf. on Information and Knowledge Management (CIKM 2014), ACM, Shanghai, China, 2014, pp. 509–518.

[17] M. Hammoud, D.A. Rabbou, R. Nouri, S. Beheshti, S. Sakr, DREAM: Distributed RDF engine with adaptive query planner and minimal communication, PVLDB 8 (6) (2015) 654–665.

[18] K. Hose, R. Schenkel, WARP: Workload-aware replication and partitioning for RDF, in: Workshops Proceedings of the 29th IEEE Int. Conf. on Data Engineering (ICDE 2013), Brisbane, Australia, 2013, pp. 1–6.
[19] I. Abdelaziz, R. Harbi, Z. Khayyat, P. Kalnis, A survey and experimental comparison of distributed SPARQL engines for very large RDF data, PVLDB 10 (13) (2017) 2049–2060.
[20] Z. Kaoudi, I. Miliaraki, M. Koubarakis, RDFS reasoning and query answering on top of DHTs, in: Proc. of the 7th Int. Semantic Web Conference (ISWC 2008), in: LNCS, vol. 5318, Springer, Karlsruhe, Germany, pp. 499–516.
[21] J. Weaver, J.A. Hendler, Parallel materialization of the finite RDFS closure for hundreds of millions of triples, in: Proc. of the 8th Int. Semantic Web Conference (ISWC 2009), in: LNCS, vol. 5823, Chantilly, VA, USA, 2009, pp. 682–697.
[22] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, H.E. Bal, WebPIE: A web-scale parallel inference engine using MapReduce, J. Web Semant. 10 (2012) 59–75.
[23] R. Gu, S. Wang, F. Wang, C. Yuan, Y. Huang, Cichlid: Efficient large scale RDFS/OWL reasoning with spark, in: Proc. of the 29th IEEE Int. Parallel and Distributed Processing Symposium (IPDPS 2015), IEEE Computer Society, Hyderabad, India, 2015, pp. 700–709.
[24] H.J. ter Horst, Completeness, decidability and complexity of entailment for RDF schema and a semantic extension involving the OWL vocabulary, J. Web Semant. 3 (2–3) (2005) 79–115.
[25] Y. Liu, P. McBrien, SPOWL: Spark-based OWL 2 reasoning materialisation, in: Proc. of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and beyond (BeyondMR 2017), Chicago, IL, USA, 2017, pp. 3:1–3:10.
[26] H. Wu, J. Liu, T. Wang, D. Ye, J. Wei, H. Zhong, Parallel materialization of datalog programs with spark for scalable reasoning, in: Proc. of the 17th Int. Conf. on Web Information Systems Engineering (WISE 2016), in: LNCS, vol. 10041, Shanghai, China, 2016, pp. 363–379.
[27] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, C. Zaniolo, Big data analytics with datalog queries on spark, in: Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD 2016), San Francisco, CA, USA, 2016, pp. 1135–1149.
[28] M. Imran, G.E. Gévay, V. Markl, Distributed graph analytics with datalog queries in flink, in: Proc. of the 4th Int. Workshop Software on the Foundations for Data Interoperability and Large Scale Graph Data Analytics (SFDI 2020), in: CCIS, vol. 1281, Springer, Tokyo, Japan, 2020, pp. 70–83.
[29] J. Seo, J. Park, J. Shin, M.S. Lam, Distributed SociaLite: A datalog-based language for large-scale graph analysis, PVLDB 6 (14) (2013) 1906–1917.
[30] G. Karypis, V. Kumar, S. Comput, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Sci. Comput. 20 (1998).
[31] A. Potter, B. Motik, Y. Nenov, I. Horrocks, Dynamic data exchange in distributed RDF stores, IEEE Trans. Knowl. Data Eng. 30 (12) (2018) 2312–2325.
[32] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565.
[33] A. Pacaci, M.T. Özsu, Experimental analysis of streaming algorithms for graph partitioning, in: Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD 2019), ACM, Amsterdam, The Netherlands, 2019, pp. 1375–1392.
[34] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, G. Iacoboni, HDRF: Stream-based partitioning for power-law graphs, in: Proc. of the 24th ACM Int. Conf. on Information and Knowledge Management (CIKM 2015), ACM, Melbourne, VIC, Australia, 2015, pp. 243–252.
[35] R. Mayer, K. Orujzade, H. Jacobsen, 2PS: HIgh-quality edge partitioning with two-phase streaming, 2020, CoRR, abs/2001.07086.
[36] M.A. Gallego, J.D. Fernández, M.A. Martínez-Prieto, P. de la Fuente, An empirical study of real-world SPARQL queries, 2011, CoRR, abs/1103.5043.
[37] S. Ganguly, A. Silberschatz, S. Tsur, Parallel bottom-up processing of datalog queries, J. Log. Program. 14 (1–2) (1992) 101–126.
[38] W. Zhang, K. Wang, S.-C. Chau, Data partition and parallel evaluation of datalog programs, IEEE Trans. Knowl. Data Eng. 7 (1) (1995) 163–176.
[39] J. Seib, G. Lausen, Parallelizing datalog programs by generalized pivoting, in: Proc. of the 10th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1991), Denver, CO, USA, 1991, pp. 241–251.
[40] J. Shao, D.A. Bell, M.E.C. Hull, Combining rule decomposition and data partitioning in parallel datalog program processing, in: Proc. of the 1st Int. Conf. on Parallel and Distributed Information Systems (PDIS 1991), Miami Beach, FL, USA, 1991, pp. 106–115.
[41] O. Wolfson, A. Ozeri, Parallel and distributed processing of rules by data-reduction, IEEE Trans. Knowl. Data Eng. 5 (3) (1993) 523–530.
[42] B. Chin, D. von Dincklage, V. Ercegovac, P. Hawkins, M.S. Miller, F.J. Och, C. Olston, F. Pereira, Yedalog: Exploring knowledge at scale, in: 1st Summit on Advances in Programming Languages (SNAPL 2015), in: LIPIcs, vol. 32, Asilomar, CA, USA, 2015, pp. 63–78.
[43] G. Graefe, D.L. Davison, Encapsulation of parallelism and architecture-independence in extensible database query execution, IEEE Trans. Softw. Eng. 19 (8) (1993) 749–764.
[44] C. Aebeloe, G. Montoya, K. Hose, Decentralized indexing over a network of RDF peers, in: Proc. of the 18th Int. Semantic Web Conf. (ISWC 2019), Auckland, New Zealand.
[45] G. Aluç, M.T. Özsu, K. Daudjee, Building self-clustering RDF databases using tunable-LSH, VLDB J. 28 (2) (2019) 173–195.
[46] D. Janke, S. Staab, M. Thimm, Impact analysis of data placement strategies on query efforts in distributed RDF stores, J. Web Semant. 50 (2018) 21–48.
[47] J. Huang, D.J. Abadi, K. Ren, Scalable SPARQL querying of large RDF graphs, PVLDB 4 (11) (2011) 1123–1134.
[48] B. Motik, Y. Nenov, R. Piro, I. Horrocks, D. Olteanu, Parallel materialisation of datalog programs in centralised, main-memory RDF systems, in: Proc. of the 28th AAAI Conf. on Artificial Intelligence (AAAI 2014), AAAI Press, Québec City, QC, Canada, 2014, pp. 129–137.
[49] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, J. Banerjee, RDFox: A highly-scalable RDF store, in: Proc. of the 14th Int. Semantic Web Conference (ISWC 2015), in: LNCS, vol. 9367, Springer, Bethlehem, PA, USA, 2015, pp. 3–20.
[50] E. Dijkstra, W. Feijen, A. van Gasteren, Derivation of a termination detection algorithm for distributed computations, Inform. Process. Lett. 16 (5) (1983) 217–219.
[51] E.W. Dijkstra, C. Scholten, Termination detection for diffusing computations, Inform. Process. Lett. 11 (1) (1980) 1–4.
[52] T.N. Bui, C. Jones, Finding good approximate vertex and edge partitions is NP-hard, Inform. Process. Lett. 42 (3) (1992) 153–159.
[53] I. Stanton, G. Kliot, Streaming graph partitioning for large distributed graphs, in: Proc. of the 18th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD 2012), ACM, Beijing, China, 2012, pp. 1222–1230.
[54] W. Zhang, Y. Chen, D. Dai, AKIN: A streaming graph partitioning algorithm for distributed graph storage systems, in: Proc. of the 18th IEEE/ACM Int. Symposium on Cluster, Cloud and Grid Computing (CCGRID 2018), IEEE Computer Society, 2018, pp. 183–192.
[55] M. Taimouri, H. Saadatfar, Rbsep: a reassignment and buffer based streaming edge partitioning approach, J. Big Data 6 (2019).
[56] C. Mayer, R. Mayer, M.A. Tariq, H. Geppert, L. Laich, L. Rieger, K. Rothermel, ADWISE: Adaptive window-based streaming edge partitioning for high-speed graph processing, in: Proc. of the 38th IEEE Int. Conf. on Distributed Computing Systems (ICDCS 2018), Vienna, Austria, 2018, pp. 685–695.
[57] Y. Guo, Z. Pan, J. Heflin, LUBM: A benchmark for OWL knowledge base systems, J. Web Semant. 3 (2–3) (2005) 158–182.
[58] G. Aluç, O. Hartig, M.T. Özsu, K. Daudjee, Diversified stress testing of RDF data management systems, in: Proc. of the 13th Int. Semantic Web Conference (ISWC 2014), in: LNCS, vol. 8796, Springer, Riva del Garda, Italy, 2014, pp. 197–212.
[59] M. Färber, The microsoft academic knowledge graph: A linked data source with 8 billion triples of scholarly data, in: Proc. of the 18th Int. Semantic Web Conference (ISWC 2019), in: LNCS, vol. 11779, Auckland, New Zealand, 2019, pp. 113–129.
[60] D.J. Abadi, A. Marcus, S.R. Madden, K. Hollenbach, Scalable semantic web data management using vertical partitioning, in: Proc. of the 33rd Int. Conf. on Very Large Data Bases (VLDB 2007), Vienna, Austria, 2007, pp. 411–422.
[61] S. Álvarez-García, N.R. Brisaboa, J.D. Fernández, M.A. Martínez-Prieto, G. Navarro, Compressed vertical partitioning for efficient RDF management, Knowl. Inf. Syst. 44 (2) (2015) 439–474.