Datalog Materialisation in Distributed RDF Stores with Dynamic Data Exchange

Temitope Ajileye[0000−0002−3657−7624], Boris Motik[0000−0003−2506−4118], and Ian Horrocks[0000−0002−2685−7462]

Department of Computer Science
University of Oxford
Oxford, United Kingdom

Abstract. Several centralised RDF systems support datalog reasoning by precomputing and storing all logically implied triples using the well-known *seminaïve algorithm*. Large RDF datasets often exceed the capacity of centralised RDF systems, and a common solution is to distribute the datasets in a cluster of shared-nothing servers. While numerous distributed query answering techniques are known, distributed seminaïve evaluation of arbitrary datalog rules is less understood. In fact, most distributed RDF stores either support no reasoning or can handle only limited datalog fragments. In this paper, we extend the *dynamic data exchange* approach for distributed query answering by Potter et al. [12] to a reasoning algorithm that can handle arbitrary rules while preserving important properties such as nonrepetition of inferences. We also show empirically that our algorithm scales well to very large RDF datasets.

1 Introduction

Reasoning with datalog rules over RDF data plays a key role on the Semantic Web. Datalog can capture the structure of an application domain using if-then rules, and OWL 2 RL ontologies can be translated into datalog rules. Datalog reasoning is supported in several RDF management systems such as Oracle's database [7], GraphDB,[1] Amazon Neptune,[2] VLog [17], and RDFox [10].[3] All of these system use a *materialisation* approach to reasoning, where all facts implied by the dataset and the rules are precomputed and stored in a preprocessing step. This is usually done using the *seminaïve algorithm* [2], which ensures the *nonrepetition property*: no rule is applied to the same facts more than once.

Many RDF management systems are *centralised* in that they store and process all data on a single server. To scale to workloads that cannot fit into a single server, it is common to distribute the data in a cluster of interconnected, shared-nothing servers and use a distributed query answering strategy. Abdelaziz et al. [1] present a comprehensive survey of 22 approaches to distributed query

[1] http://graphdb.ontotext.com/
[2] http://aws.amazon.com/neptune/
[3] http://www.cs.ox.ac.uk/isg/tools/RDFox/

answering, and Potter et al. [12] discuss several additional systems. There is considerable variation between these approaches: some use data replication, some compute joins on a dedicated server, others use distributed join algorithms, and many leverage big data frameworks such as Hadoop and Spark for data storage and query processing. In contrast, distributed datalog materialisation is less well understood, and it is more technically challenging. Newly derived facts must be stored so that they can be taken into account in future rule applications, but without repeating derivations. Moreover, synchronisation between rule applications should be reduced to allow parallel computation.

Several theoretical frameworks developed in the 90s aim to address these questions [4, 19, 13, 15, 21]. As we discuss in more detail in Section 3, they constrain the rules so that that each server performs only certain rule applications, and they send the derived facts to all servers where these facts could participate in further rule applications. Thus, the same facts can be stored on more than one server, which can severely limit the scalability of such systems.

The Semantic Web community has recently developed several RDF-specific approaches. A number of them are hardwired to fixed datalog rules, such as RDFS [18, 6] or the so-called *ter Horst fragment* [16, 5]. Focusing on a fixed set of rules considerably simplifies the problem. PLogSPARK [20] and SPOWL [9] handle arbitrary rules, but they do not seem to use seminaïve evaluation. Finally, several probabilistic algorithms aim to handle large datasets [11, 9], but these approaches are approximate and are thus unsuitable for many applications. Distributed SociaLite [14] is the only system we are aware of that provides seminaïve evaluation for arbitrary datalog rules. It uses a custom graph model, but the approach can readily be adapted to RDF. Moreover, its rules must explicitly encode the communication and storage strategy, which increases complexity.

In this paper we present a new technique for distributed materialisation of arbitrary datalog rules. Unlike SociaLite, we do not require any distributed processing hints in the rules. We also do not duplicate any data and thus remove an obstacle to scalability. Our approach is based on the earlier work by Potter et al. [12] on distributed query answering using *dynamic data exchange*, from which it inherits several important properties. First, inferences that can be made within a single server do not require any communication; coupled with careful data partitioning, this can very effectively minimise network communication. 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, which we achieve using *Lamport timestamps* [8]. We discuss the motivation and the novelty in more detail in Section 3, and in Section 4 we present the approach formally.

We have implemented our approach in a new prototype system called DMAT, and in Section 5 we present the results of our empirical evaluation. We compared DMAT with WebPIE [16], investigated how it scales with increasing data loads, and compared it with RDFox to understand the impact of distribution on concurrency. Our results show that DMAT outperforms WebPIE by an order of magnitude (albeit with some differences in the setting), and that it can handle well increasing data loads; moreover, DMAT's performance is comparable to that of RDFox on a single server. Our algorithms are thus a welcome addition to the techniques for implementing truly scalable semantic systems.

2 Preliminaries

We now recapitulate the syntax and the semantics of RDF and datalog. A *constant* is an IRI, a blank node, or a literal. A *term* is a constant or a *variable*. An *atom* a has the form $a = \langle t_s, t_p, t_o \rangle$ over terms t_s (*subject*), t_p (*predicate*), and t_o (*object*). A *fact* is an variable-free atom. A *dataset* is a finite set of facts.

Since the focus of our work is on datalog reasoning, we chose to follow terminology commonly used in datalog literature. Constants are often called *RDF terms* in RDF literature, but we do not use this notion to avoid confusion with datalog terms, which include variables. For the sake of consistency, we then use the datalog notions of atoms, facts, and datasets, instead of the corresponding RDF notions of *triple patterns*, *triples*, and *RDF graphs*, respectively.

We define the set of *positions* as $\Pi = \{s, p, o\}$. Then, for $a = \langle t_s, t_p, t_o \rangle$ and $\pi \in \Pi$, we define $a|_\pi = t_\pi$—that is, $a|_\pi$ is the term that occurs in a at position π. 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.

A *query* Q is a conjunction of atoms $a_1 \wedge \cdots \wedge a_n$. Substitution σ is an *answer* to Q on a dataset I if $a_i\sigma \in I$ holds for each $1 \leq i \leq n$.

A datalog *rule* r is an implication of the form $h \leftarrow b_1 \wedge \cdots \wedge b_n$, where h is the *head* atom, all b_i are *body* atoms, and each variable occurring in h also occurs in some b_i. A datalog *program* is a finite set of rules. Let I be a dataset. The result of applying r to I is $r(I) = I \cup \{h\sigma \mid \sigma \text{ is an answer to } b_1 \wedge \cdots \wedge b_n \text{ on } I\}$. For P a program, let $P(I) = \bigcup_{r \in P} r(I)$; let $P^0(I) = I$; and let $P^{i+1}(I) = P(P^i(I))$ for $i \geq 0$. Then, $P^\infty(I) = \bigcup_{i \geq 0} P^i(I)$ is the *materialisation* of P on I. This paper deals with the problem of computing $P^\infty(I)$ where I is distributed across of a cluster of servers such that each fact is stored in precisely one server.

3 Motivation and Related Work

We can compute $P^\infty(I)$ using the definition in Section 2: we evaluate the body of each rule $r \in P$ as a query over I and instantiate the head of r for each query answer, we eliminate duplicate facts, and we repeat the process until no new facts can be derived. However, since $P^i(I) \subseteq P^{i+1}(I)$ holds for each $i \geq 0$, such a *naïve* approach repeats in each round of rule applications the work from all previous rounds. The *semïnaive strategy* [2] 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. We next discuss now these ideas are implemented in the existing approaches to distributed materialisation, and then we present an overview of our approach and discuss its novelty.

3.1 Related Approaches to Distributed Materialisation

Several approaches to distributed reasoning partition rule applications across servers. For example, to evaluate rule $\langle x, R, z \rangle \leftarrow \langle x, R, y \rangle \wedge \langle y, R, z \rangle$ on ℓ servers, one can let each server i with $1 \leq i \leq \ell$ evaluate rule

$$\langle x, R, z \rangle \leftarrow \langle x, R, y \rangle \wedge \langle y, R, z \rangle \wedge h(y) = i, \tag{1}$$

where $h(y)$ is a *partition function* that maps values of y to integers between 1 and ℓ. If h is uniform, then each server receives roughly the same fraction of the workload, which benefits parallelisation. However, since a triple of the form $\langle s, R, o \rangle$ can match either atom in the body of (1), each such triple must be replicated to servers $h(s)$ and $h(o)$ so they can participate in rule applications. Based on this idea, Ganguly et al. [4] show how to handle general datalog; Zhang et al. [21] study different partition functions; Seib and Lausen [13] identify programs and partition functions where no replication of derived facts is needed; Shao et al. [15] further break rules in segments; and Wolfson and Ozeri [19] replicate all facts to all servers. The primary motivation behind these approaches seems to be parallelisation of computation, which explains why the high rates of data replication were not seen as a problem. However, high replication rates are not acceptable when data distribution is used to increase a system's capacity.

Materialisation can also be implemented without any data replication. First, one must select a triple partitioning strategy: a common approach is to assign each $\langle s, p, o \rangle$ to server $h(s)$ for 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. Then, one can evaluate the rules using a suitable distributed query algorithm and distribute the newly derived triples using the partitioning strategy. These principles were used to realise RDFS reasoning [18, 6], and they are also implicitly present in approaches implemented on top of big data frameworks such as Hadoop [16] and Spark [5, 20, 9]. However, most of these can handle only fixed rule sets, which considerably simplifies algorithm design. For example, seminaïve evaluation is not needed in the RDFS fragment since these nonrepetition of inferences can be ensured by evaluating rules in a particular order [5]. PLogSPARK [20] and SPOWL [9] handle arbitrary rules using the naïve algorithm, which can be detrimental when programs are moderately complex.

Distributed SociaLite [14] is the only system known to us that implements distributed seminaïve evaluation for general datalog. It requires users to explicitly specify the data distribution strategy and communication patterns. For example, by writing a fact $R(a, b)$ as $R[a](b)$, one can specify that the fact is to be stored on server $h(a)$ for some hash function h. Rule (1) 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.

3.2 Dynamic Data Exchange for Query Answering

Before describing our approach to distributed datalog materialisation, we next recapitulate the earlier work by Potter et al. [12] on distributed query answering using *dynamic data exchange*, which provides the foundation for our work.

This approach to query answering assumes that all triples are partitioned into ℓ mutually disjoint datasets $I_1, \dots, I_\ell$, with ℓ being the number of servers. The main objectives of dynamic exchange are to reduce communication and eliminate synchronisation between servers. To achieve the former goal, each server k maintains three *occurrence mappings* $\mu_{k,s}$, $\mu_{k,p}$, and $\mu_{k,o}$. For each constant c occurring in I_k, set $\mu_{k,s}(c)$ contains all servers where c occurs in the subject position, and $\mu_{k,p}(c)$ and $\mu_{k,o}(c)$ provide analogous information for the predicate and object positions. To understand how occurrences are used, consider evaluating $Q = \langle x, R, y \rangle \wedge \langle y, R, z \rangle$ over datasets $I_1 = \{\langle a, R, b \rangle, \langle b, R, c \rangle\}$ and $I_2 = \{\langle b, R, d \rangle, \langle d, R, e \rangle\}$. Both servers evaluate Q using index nested loop joins. Thus, server 1 evaluates $\langle x, R, y \rangle$ over I_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 I_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. After receiving σ_1, server 2 matches atom $\langle b, R, z \rangle$ in I_2 to obtain another full answer $\sigma_3 = \{x \mapsto a, y \mapsto b, z \mapsto d\}$. However, server 2 also evaluates $\langle x, R, y \rangle$ over I_2, obtaining 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. We use this to partition a graph based on its structural properties and thus collocate highly interconnected constants. Combined with the first property, this can significantly reduce network communication. Third, the system is completely 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. This eliminates the need for synchronisation between servers, which is beneficial for parallelisation.

3.3 Our Contribution

In this paper we extend the dynamic data exchange framework to datalog materialisation. We draw inspiration from the work by Motik et al. [10] on parallelising datalog materialisation in centralised, shared memory systems. Intuitively, their algorithm considers each triple in the dataset, identifies each rule and body atom that can be matched to the triple, and evaluates the rest of the rule as a query.

This approach is amenable to parallelisation since distinct processors can simultaneously process distinct triples; since the number of triples is generally very large, the likelihood of workload skew among processors is very low.

Our distributed materialisation algorithm is based on the same general principle: each server matches the rules to locally stored triples, but the resulting queries are evaluated using dynamic data exchange. This approach requires no synchronisation between servers, and it reduces communication in the same way as described in Section 3.2. We thus expect our approach to exhibit the same good properties as the approach to query answering by Potter et al. [12].

The lack of synchronisation between servers introduces a technical complication. Remember that, to avoid repeating derivations, at least one body atom in a rule must be matched to a fact derived in the previous round of rule application. However, due to asynchronous rule application, there is no global notion of a rule application round (unlike, say, in SociaLite). A naïve solution would be to associate each fact with a timestamp recording when the fact has derived in hope that the order of fact derivation could be recovered by comparing timestamps. However, this would require maintaining a high coherence of server clocks in the cluster, which is generally impractical. Instead, we use Lamport timestamps [8], which provide us with a simple way of determining a partial order of events across a cluster. We describe this technique in more detail in Section 4.

Another complication is due to the fact that the occurrence mappings stored in the servers may need to be updated due to the derivation of new triples. For completeness, it is critical that all servers are updated before such triples are used in rule applications. Our solution to this problem is fully asynchronous, which again benefits parallelisation.

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 [3].

4 Distributed Materialisation Algorithm

We now present our distributed materialisation algorithm and prove its correctness. We present the algorithm in steps. In Section 4.1 we discuss data structures that the servers use to store their triples and implement Lamport timestamps. In Section 4.2 we discuss the occurrence mappings. In Section 4.3 we discuss the communication infrastructure and the message types used. In Section 4.4 we present the algorithm's pseudocode. In Section 4.5 we discuss how to detect termination. Finally, in Section 4.6 we argue about the algorithm's correctness.

4.1 Adding Lamport Timestamps to Triples

As already mentioned, to avoid repeating derivations, our algorithm uses Lamport timestamps [8], which is a technique for establishing a causal order of events in a distributed system. If all servers in the system could share a global clock, we could trivially associate each event with a global timestamp, which would allow us to recover the 'happens-before' relationship between events by comparing timestamps. However, maintaining a precise global clock in a distributed system is technically very challenging, and Lamport timestamps provide a much simpler solution. In particular, each event is annotated 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 maintains a local integer clock that is incremented each time an event of interest occurs, which clearly ensures $(*)$ if A and B occur within one server. Now assume that A occurs in server s_1 and B occurs in s_2; clearly, A can influence B only if s_1 sends a message to s_2, and s_2 processes this message before event B takes place. To ensure that property $(*)$ holds in such a case as well, server s_1 includes its current clock value into the message it sends to s_2; moreover, when processing this message, server s_2 updates its local clock to the maximum of the message clock and the local clock, and then increments the local clock. Thus, when B happens after receiving the message, it is guaranteed to have a timestamp that is larger than the timestamp of A.

To map 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 associates facts with integer timestamps.

More precisely, each server k in the cluster maintains an integer C_k called the *local clock*, a set I_k of the derived triples, and a partial function $T_k : I_k \to \mathbb{N}$ that associates triples with natural numbers. Function T_k is partial because timestamps are not assigned to facts upon derivation, but during timestamp synchronisation. Before the algorithm is started, C_k must be initialised to zero, and all input facts (i.e., the facts given by the user) partitioned to server k should be loaded into I_k and assigned a timestamp of zero.

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

$$Q = a_1^{\bowtie_1} \wedge \cdots \wedge a_n^{\bowtie_n}, \tag{2}$$

where each $a_i^{\bowtie_i}$ is called an *annotated atom* and it consists of an atom a_i and a symbol $\bowtie_i$ which can be $<$ or $\leq$. An annotated query requires a timestamp to be evaluated. More precisely, a substitution σ is an answer to Q on I_k and T_k w.r.t. a timestamp τ if (i) σ is an answer to the 'ordinary' query $a_1 \wedge \cdots \wedge a_n$ on I_k, and (ii) for each $1 \leq i \leq n$, the value of T_k is defined for $a_i\sigma$ and it satisfies $T_k(a_i\sigma) \bowtie \tau$. For example, let Q, I, and T be as follows, and let $\tau = 2$.

$$Q = \langle x, R, y\rangle^{<} \wedge \langle y, S, z\rangle^{\leq} \qquad I = \{\langle a, R, b\rangle, \langle b, S, c\rangle, \langle b, S, d\rangle, \langle b, S, e\rangle\}$$
$$T = \{\langle a, R, b\rangle \mapsto 1,\ \langle b, S, c\rangle \mapsto 2,\ \langle b, S, d\rangle \mapsto 3\}$$

Then, $\sigma_1 = \{x \mapsto a, y \mapsto b, z \mapsto c\}$ is an answer to Q on I and T w.r.t. τ. In contrast, $\sigma_2 = \{x \mapsto a, y \mapsto b, z \mapsto d\}$ is not an answer to Q on I and T w.r.t. τ

due to $T(\langle b, S, d\rangle) \geq 2$, and $\sigma_3 = \{x \mapsto a, y \mapsto b, z \mapsto e\}$ is not an answer because the timestamp of $\langle b, S, e\rangle$ is undefined.

To incorporate this notion into our algorithm, we assume that each server can evaluate a single annotated atom. Specifically, given an annotated $a^{\bowtie}$, a timestamp τ, and a substitution σ, server k can call $\textsc{Evaluate}(a^{\bowtie}, \tau, I_k, T_k, \sigma)$. The call returns each substitution ρ defined over the variables in a and σ such that $\sigma \subseteq \rho$ holds, $a\rho \in I_k$ holds, and T_k is defied on $a\rho$ and it satisfies $T(a\rho) \bowtie \tau$. In other words, $\textsc{Evaluate}$ matches $a^{\bowtie}$ in I_k and T_k w.r.t. τ and it returns each extension of σ that agrees with $a^{\bowtie}$ and τ. For efficiency, server k should index the facts in I_k; any RDF indexing scheme can be used, and one can modify index lookup to simply skip over facts whose timestamps do not match τ.

Finally, we describe how rule matching is mapped to answering annotated queries. Let P be a datalog program to be materialised. Given a fact f, function $\textsc{MatchRules}(f, P)$ considers each rule $h \leftarrow b_1 \wedge \cdots \wedge b_n \in P$ and each body atom b_p with $1 \leq p \leq n$, and, for each substitution σ over the variables of b_p where $f = b_p\sigma$, it returns (σ, b_p, Q, h) where Q is the annotated query

$$b_1^{<} \wedge \cdots \wedge b_{p-1}^{<} \wedge b_{p+1}^{\leq} \wedge \cdots \wedge b_n^{\leq}. \tag{3}$$

Intuitively, $\textsc{MatchRules}$ 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 matching all remaining atoms in nested loops using function $\textsc{Evaluate}$. The annotations in (3) specify how to match the remaining atoms without repetition: facts matched to atoms before (resp. after) the pivot must have timestamps strictly smaller (resp. smaller or equal) than the timestamp of f. As is usual in query evaluation, the atoms of (3) may need to be reordered to obtain an efficient query plan. This can be achieved using any known technique, and further discussion of this issue is out of scope of this paper.

4.2 Occurrence Mappings

To decide whether rule matching may need to proceed on other servers, each server k must store indexes $\mu_{k,s}$, $\mu_{k,p}$, and $\mu_{k,o}$, called *occurrence mappings*, that map constants to sets of server IDs. We say that a constant c is *local* to server k is c occurs in I_k at any position. To ensure scalability, $\mu_{k,s}$, $\mu_{k,p}$, and $\mu_{k,o}$ need only to be defined on local constants: if, say, $\mu_{k,s}$ is not defined on constant c, we will assume that c can occur on any server. However, these mappings will need to be correct during algorithm's execution: if a constant c is local to I_k, and if c occurs on some other server j in position π, then $\mu_{k,\pi}$ must be defined on c and it must contain j. Moreover, all servers will have to know the initial locations of all constants occurring in the heads of the rules in P.

Storing only partial occurrences at each server introduces a complication: when a server processes a partial match σ received from another server, its local occurrence mappings may not cover some of the constants in σ. Potter et al. [12] 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 so that this information can be propagated to subsequent servers.

Occurrence mappings are initialised on each server k for each constant that initially occurs in I_k, but they may need to be updated as fresh triples are derived. To ensure that the occurrences correctly reflect the distribution of constants at all times, occurrence mappings of all servers must be updated *before* a triple can be added to the set of derived triples of the target server.

Our algorithm must decide where to store each freshly derived triple. It is common practice in distributed RDF systems to store all triples with the same subject on the same server. This is beneficial since it allows subject–subject joins—the most common type of join in practice—to be answered without any communication. We follow this well-established practice and ensure that the derived triples are grouped by subject. Consequently, we require that $\mu_{k,s}(c)$, whenever it is defined, contains exactly one server. Thus, to decide where to store a derived triple, the server from the subject's occurrences is used, and, if the subject occurrences are unavailable, then a predetermined server is used.

4.3 Communication Infrastructure and Message Types

We assume that the servers can communicate asynchronously by passing messages: each server can call $\textsc{Send}(m, d)$ to send a message m to a destination server d. This function can return immediately, and the receiver can processes the message later. Also, our core algorithm is correct as long as each sent message is processed eventually, regardless of whether the messages are processed in the order in which they are sent between servers. We next describe the two types of message used in our algorithm. The approach used to detect termination can introduce other message types and might place constraints on the order of message delivery; we discuss this in more detail in Section 4.5.

Message $\mathsf{PAR}[i, \sigma, Q, h, \tau, \boldsymbol{\lambda}]$ 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 for all constants mentioned in σ are recorded in $\boldsymbol{\lambda}$.

Message $\mathsf{FCT}[f, D, k_h, \tau, \boldsymbol{\lambda}]$ says that f is a freshly derived fact that should be stored at server k_h. Set D contains servers whose occurrences must be updated due to the addition of f. Timestamp τ corresponds to the time at which the message was sent. Finally, $\boldsymbol{\lambda}$ are the partial occurrences for the constants in f.

Potter et al. [12] already observed PAR messages correspond to partial join results so a large number of such messages can be produced during query evaluation. To facilitate asynchronous processing, the PAR messages may need to be buffered on the receiving server, which can easily require excessive space. They also presented a flow control mechanism that can be used to restrict memory consumption at each server without jeopardising completeness. This solution is directly applicable to our problem as well, so we do not discuss it any further.

4.4 The Algorithm

With these definitions in mind, Algorithms 1 and 2 comprise our approach to distributed datalog materialisation. Before starting, each server k loads its subset of the input RDF graph into I_k, sets the timestamp of each fact in I_k to zero, initialises C_k to zero, and receives the copy of the program P to be materialised. The server then starts an arbitrary number of server threads, each executing the ServerThread function. Each thread repeatedly processes either an unprocessed fact f in I_k or an unprocessed message m; if both are available, they can be processed in arbitrary order. Otherwise, the termination condition is processed as we discuss later in Section 4.5.

Function Synchronise updates the local clock C_k with a timestamp τ. This must be done in a critical section (i.e., two threads should not execute it simultaneously). The local clock is updated if $C_k \leq \tau$ holds; moreover, all facts in I_k without a timestamp are timestamped with C_k since they are derived before the event corresponding to τ. Assigning timestamps to facts in this way reduces the need for synchronising access to C_k between threads.

Function ProcessFact kickstarts the matching of the rules to fact f. After synchronising the clock with the timestamp of f, the function simply calls the MatchRules function to identify all rules where one atom matches to f, and then it calls the FinishMatch function to finish matching the pivot atom.

A PAR message is processed by matching atom $a_i^{\bowtie_i}$ of the annotated query in I_k and T_k w.r.t. τ, and forwarding each match to FinishMatch.

A FCT message informs server k that fact f will be added to the set I_{k_h} of facts derived at server k_h. Set D lists all remaining servers that need to be informed of the addition, and partial occurrences $\boldsymbol{\lambda}$ are guaranteed to correctly reflect the occurrences of each constant in f. Server k updates its $\mu_{k,\pi}(c)$ by appending $\lambda_\pi(c)$ (line 19). Since servers can simultaneously process FCT messages, server k adds to D all servers that might have been added to $\mu_{k,\pi}(c)$ since the point when $\lambda_\pi(c)$ had been constructed (line 18), and it also updates $\lambda_\pi(c)$ (line 19). Finally, the server adds f to I_k if k is the last server (line 20), and otherwise it forwards the message to another server d form D.

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 task, the algorithm identifies in line 30 each variable x in the matched atom that either occurs in the rule head or in a remaining atom, and for each π it adds the occurrences of $x\sigma$ to λ_π. Now if Q has been matched completely (line 31), the server also ensures that the partial occurrences are correctly defined for the constants occurring in the rule head (lines 32–33), it identifies the server k_h that should receive the derived fact as described in Section 4.2, it identifies the set D of the destination servers whose occurrences need to be updated, and it sends the FCT message to one server from D. Otherwise, atom $a_{i+i}\sigma$ must be matched next. To determine the set D of servers that could possibly match this atom, server k intersects the occurrences of each constant from $a_{i+i}\sigma$ (line 44) and sends a PAR message to all servers in D.

Algorithm 1 Distributed Materialisation Algorithm at Server k

```
 1: function ServerThread
 2:     while cannot terminate do
 3:         if $I_k$ contains an unprocessed fact $f$, or a message $m$ is pending then
 4:             ProcessFact($f$) or ProcessMessage($m$), as appropriate
 5:         else if the termination token has been received then
 6:             Process the termination token

 7: function ProcessFact($f$)
 8:     Synchronise($T_k(f)$)
 9:     for each $(\sigma, a, Q, h) \in$ MatchRules($f, P$) do
10:         FinishMatch($0, \sigma, a, Q, h, T_k(f), \boldsymbol{\emptyset}$)

11: function ProcessMessage($\mathsf{PAR}[i, \sigma, Q, h, \tau, \boldsymbol{\lambda}]$) where $Q = a_1^{\bowtie_1} \wedge \cdots \wedge a_n^{\bowtie_n}$
12:     Synchronise($\tau$)
13:     for each substitution $\sigma' \in$ Evaluate($a_i^{\bowtie_i}, \tau, I_k, T_k, \sigma$) do
14:         FinishMatch($i, \sigma', a_i, Q, h, \tau, \boldsymbol{\lambda}$)

15: function ProcessMessage($\mathsf{FCT}[f, D, k_h, \tau, \boldsymbol{\lambda}]$)
16:     Synchronise($\tau$)
17:     for each constant $c$ in $f$ and each position $\pi \in \Pi$ do
18:         $D := D \cup \left[\mu_{k,\pi}(c) \setminus \lambda_\pi(c)\right]$
19:         $\lambda_\pi(c) := \mu_{k,\pi}(c) := \lambda_\pi(c) \cup \mu_{k,\pi}(c)$
20:     if $D = \emptyset$ then Add $f$ to $I_k$
21:     else
22:         Remove an element $d$ from $D$, preferring any element over $k_h$ if possible
23:         Send($\mathsf{FCT}[f, D, k_h, C_k, \boldsymbol{\lambda}], d$)

24: function Synchronise($\tau$) (must be executed in a critical section)
25:     if $C_k \leq \tau$ then
26:         for each fact $f \in I_k$ such that $T_k$ is undefined on $f$ do $T_k(f) := C_k$
27:         $C_k := \tau + 1$
```

4.5 Termination Detection

Since no server has complete information about the progress of any other server, detecting termination is nontrivial; however, we can reuse an existing solution.

When messages between each pair of servers are guaranteed to be delivered in order in which they are sent (as is the case in our implementation), one can use Dijkstra's token ring algorithm [3], which we summarise next. 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.

Algorithm 2 Distributed Materialisation Algorithm at Server k (Continued)

```
28: function FinishMatch($i, \sigma, a_{last}, Q, h, \tau, \boldsymbol{\lambda}$) where $Q = a_1^{\bowtie_1} \wedge \cdots \wedge a_n^{\bowtie_n}$
29:     for each var. $x$ occurring in $a_{last}$ and in $h$ or $a_j$ with $j > i$, and each $\pi \in \Pi$ do
30:         Extend $\lambda_\pi$ with the mapping $x\sigma \mapsto \mu_{k,\pi}(x\sigma)$
31:     if $i = n$ then
32:         for each constant $c$ occurring in $h$ and each $\pi \in \Pi$ do
33:             Extend $\lambda_\pi$ with the mapping $c \mapsto \mu_{k,\pi}(c)$
34:         $k_h :=$ the owner server for the derived fact
35:         $D := \{k_h\}$
36:         for each position $\pi \in \Pi$ and $c = h\sigma|_\pi$ where $k_h \notin \lambda_\pi(c)$ do
37:             Add $k_h$ to $\lambda_\pi(c)$
38:             for each $\pi' \in \Pi$ do Add $\lambda_{\pi'}(c)$ to $D$
39:         Remove an element $d$ from $D$, preferring any element over $k_h$ if possible
40:         if $d = k$ then ProcessMessage(FCT[$h\sigma, D, k_h, C_k, \boldsymbol{\lambda}$])
41:         else Send(FCT[$h\sigma, D, k_h, C_k, \boldsymbol{\lambda}$], $d$)
42:     else
43:         $D :=$ the set of all servers
44:         for each position $\pi \in \Pi$ where $a_{i+1}\sigma|_\pi$ is a constant $c$ do $D := D \cap \lambda_\pi(c)$
45:         for each $d \in D$ do
46:             if $d = k$ then ProcessMessage(PAR[$i+1, \sigma, Q, h, \tau, \boldsymbol{\lambda}$])
47:             else Send(PAR[$i+1, \sigma, Q, h, \tau, \boldsymbol{\lambda}$], $d$)
```

– 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 to a server $j < i$.
– All servers can terminate when server 1 receives a white token.

The Dijkstra–Scholten algorithm extends this approach to the case when the order of message delivery cannot be guaranteed.

4.6 Correctness

We next prove that our algorithm is correct and that it exhibits the nonrepetition property. We present here only an outline of the correctness argument, and give the full proof in the appendix.

Let us fix a run of Algorithms 1 and 2 on some input. First, we show that Lamport timestamps capture the causality of fact derivation in this run. To this end, we introduce four event types relating to an arbitrary fact f. Event $\mathsf{add}_k(f)$ occurs when f is assigned a timestamp on server k in line 26. Event $\mathsf{process}_k(f)$ occurs when server k starts processing a new fact in line 8. Event $\mathsf{PAR}_k(f, i)$ occurs when server k completes line 12 for a PAR message with index i originating from a call to MatchRules on fact f. Finally, event $\mathsf{FCT}_k(f)$ occurs when server k completes line 16 for a FCT message for fact f. We write $e_1 \rightsquigarrow e_2$ if event e_1 occurs chronologically before event e_2; this relation is clearly transitive and irreflexive. Since each fact is stored and assigned a timestamp on just one server, we define $T(f)$ as $T_k(f)$ for the unique server k that satisfies $f \in I_k$. Lemma 1 then essentially says that the 'happens-before' relationship between facts and events on facts agrees with the timestamps assigned to the facts.

Lemma 1. *In each run of the algorithm, for each server k, and all facts f_1 and f_2, we have $T(f_1) < T(f_2)$ whenever one of the following holds:*

- $\mathsf{PAR}_k(f_1, i) \rightsquigarrow \mathsf{add}_k(f_2)$ *for some i,*
- $\mathsf{process}_k(f_1) \rightsquigarrow \mathsf{FCT}_k(f_2)$*, or*
- $\mathsf{PAR}_k(f_1, i) \rightsquigarrow \mathsf{FCT}_k(f_2)$ *for some i.*

Next, we show that then the occurrence mappings $\mu_{k,\pi}$ on each relevant server k are updated whenever a triple is added to some I_j. This condition is formally captured in Lemma 2, and it ensures that partial answers are sent to all relevant servers that can possibly match an atom in a query. Note that the implication in Lemma 2 is the only relevant direction: if $\mu_{k,\pi}(c)$ contains irrelevant servers, we can have redundant PAR messages, but this does not affect correctness.

Lemma 2. *At any point in the algorithm's run, for all servers k and j, each position $\pi \in \Pi$, and each constant c that is local to server k and that occurs in I_j at position π, property $j \in \mu_{k,\pi}(c)$ holds at that point.*

Using Lemmas 1 and 2, we prove our main claim.

Theorem 1. *For $I_1, \ldots, I_\ell$ the sets obtained by applying Algorithms 1 and 2 to an input set of facts I and program P, we have $P^\infty(I) = I_1 \cup \cdots \cup I_\ell$. Moreover, the algorithm exhibits the nonrepetition property.*

5 Evaluation

To evaluate the practical applicability of our approach, we have implemented a prototype distributed datalog reasoned that we call DMAT. We have used RDFox—a state-of-the-art centralised RDF system—to store and index triples in RAM, on top of which we have implemented a mechanism for associating triples with timestamps. To implement the Evaluate function, we use the interface of RDFox for answering individual atoms and then simply filter out the answers whose timestamp does not match the given one. For simplicity, DMAT currently uses only one thread per server, but this limitation will be removed in future.

We have evaluated our system's performance in three different ways, each aimed at analysing a specific aspect of the problem. First, to establish a baseline for the performance of DMAT, as well as to see whether distributing the data can speed up the computation, we compared DMAT with RDFox on a relatively small dataset. Second, to compare our approach with the state of the art, we compared DMAT with WebPIE [16]—a distributed RDF reasoner based on MapReduce.

Third, we studied the scalability of our approach by proportionally increasing the input data and the number of servers.

Few truly large RDF datasets are publicly available, so the evaluation of distributed reasoning is commonly based on the well-known LUBM[4] benchmark (e.g., [16, 9, 20]). Following this well-established practice, in our evaluation we used LUBM datasets of sizes ranging from 134 M to 6.5 G triples. We also used the *lower bound* program, which was obtained by extracting the OWL 2 RL portion of the LUBM ontology and translating it into datalog. The executable of DMAT and the datalog program we used are available online,[5] and the datasets can be reproduced using the LUBM generator.

We conducted all tests with DMAT on the Amazon Elastic Compute Cloud (EC2). We used the *r4.8xlarge* servers,[6] each equipped with a 2.3 GHz Intel Broadwell processors and 244 GB of RAM; such a large amount of RAM was needed since we use RDFox to store triples, and RDFox is RAM-based. An additional, identical server stored the dictionary (i.e., a data structure mapping constants to integers): this server did not participate in materialisation, but was used only to distribute the program and the data to the cluster. The servers were connected using 10 Gbps network. In all tests apart from the ones where we compared DMAT to WebPIE, we partitioned the dataset by using the graph partitioning approach by Potter et al. [12]: this approach aims to place strongly connected constants on the same server and thus reduce communication overhead. Unfortunately, our graph partitioning algorithm ran out of memory on the very large datasets we used to compare DMAT with WebPIE, so in these tests we partitioned the data using subject hashing. For each test, we loaded the input triples and the program into all servers, and computed the materialisation while recording the wall-clock time. Apart from reporting this time, we also report the *reasoning throughput* measured in thousands of triples derived per second and worker (ktps/w). We next discuss the results of our experiments.

Comparison with RDFox. First, we ran RDFox and DMAT on a fixed dataset while increasing the number of threads for RDFox and the numbers of servers for DMAT. Since RDFox requires the materialised dataset to fit into RAM of a single server, we used a small input dataset of just 134 M triples. The results, shown in Table 1, provide us with two insights. First, the comparison on one thread establishes a baseline for the DMAT's performance. In particular, DMAT is slower than RDFox, which is not surprising: RDFox is a mature and tuned system, whereas DMAT is just a prototype. However, DMAT is still competitive with RDFox, suggesting that our approach is free of any overheads that might make it uncompetitive. Second, the comparison on multiple threads shows how effective our approach is at achieving concurrency. RDFox was specifically designed with that goal in mind in a shared-memory setting. However, as one can see from our results, DMAT also parallelises computation well: in some cases the speedup is

[4] http://swat.cse.lehigh.edu/projects/lubm/
[5] http://krr-nas.cs.ox.ac.uk/2019/distributed-materialisation/
[6] http://aws.amazon.com/ec2/instance-types/

Table 1: Comparison of Centralised and Distributed Reasoning

	Threads/Servers							
	1		2		4		8	
	RDFox	DMAT	RDFox	DMAT	RDFox	DMAT	RDFox	DMAT
Times (s)	86	256	56	140	35	82	16	53
Speed-up	1.0x	1.0x	1.5x	1.8x	2.5x	3.1x	5.4x	4.8x
Size	$134M \rightarrow 182M$							

Table 2: Comparison with WebPIE

Dataset	Sizes (G)		WebPIE (64 workers)		DMAT (12 servers)	
	Input	Output	Time (s)	ktps/w	Time (s)	ktps/w
4K	0.5	0.729	1920	4.1	224	85
8K	1	1.457	2100	7.5	461	81
36K	5	6.516	3120	24.9	2087	71

Table 3: Scalability Experiments

Workers	Dataset	Input size (G)	Output size (G)	Time (s)	Rate (ktps/w)
2	4K	0.5	0.73	646	212
6	12K	1.6	2.19	769	173
10	20K	2.65	3.64	887	151

larger than in the case of RDFox. This seems to be the case mainly because data partitioning allows each server to handle an isolated portion of the graph, which can reduce the need for synchronisation.

Comparison with WebPIE. Next, we compared DMAT with WebPIE to see how our approach compares with the state of the art in distributed materialisation. To keep the experimentation effort manageable, we did not rerun WebPIE ourselves; rather, we considered the same input dataset sizes as Urbani et al. [16] and reused their published results. The setting of these experiments thus does not quite match our setting: (i) WebPIE handles only the ter Horst fragment of OWL and thus cannot handle all axioms in the OWL 2 RL subset of the LUBM ontology; (ii) experiments with WebPIE were run on physical (rather than virtualised) servers with only 24 GB of RAM each; and (iii) WebPie used 64 workers, while DMAT used just 12 servers. Nevertheless, as one can see from Table 2, despite using more than five times fewer servers, DMAT is faster by an order of magnitude. Hadoop is a disk-based system so lower performance is to be expected to some extent, but this may not be the only reason: triples in DMAT are partitioned by subject so, unlike WebPIE, DMAT does not perform any communication on subject–subject joins.

Scalability Experiments. Finally, to investigate the scalability of DMAT, we measured how the system's performance changes when the input data and the number of servers increase proportionally. The results are shown in Table 3. As one can see, increasing the size of the input does introduce an overhead for each server. Our analysis suggests that this is mainly because handling a larger dataset requires sending more messages, and communication seems to be the main source of overhead in the system. This, in turn, leads to a moderate reduction in throughout. Nevertheless, the system still exhibits very high inferences rates and clearly scales to very large inputs.

6 Conclusion

In this paper, we have presented a novel approach to datalog reasoning in distributed RDF systems. Our work extends the distributed query answering algorithm by Potter et al. [12], from which it inherits several benefits. First, the servers in our system are asynchronous, which is beneficial for concurrency. Second, dynamic data exchange is effective at reducing network communication, particularly when input data is partitioned so that related triples are co-located. Third, we have shown empirically that our prototype system is an order of magnitude faster than WebPIE [17], and that it scales to increasing data loads.

We see several interesting avenues for our future work. First, we shall extend our evaluation to cover a broader range of systems, datasets, and rule sets. Second, better approaches to partitioning the input data are needed: hash partitioning does not guarantee that joins other than subject–subject ones are processed on one server, and graph partitioning cannot handle large input graphs. Third, supporting more advanced features of datalog, such as stratified negation and aggregation is also needed in many practical applications.

Acknowledgments

This work was supported by the SIRIUS Centre for Scalable Access in the Oil and Gas Domain, and the EPSRC projects AnaLOG and ED3.

Bibliography

[1] Abdelaziz, I., Harbi, R., Khayyat, Z., Kalnis, P.: A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data. PVLDB 10(13), 2049–2060 (2017)

[2] Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)

[3] Dijkstra, E., Feijen, W., van Gasteren, A.: Derivation of a Termination Detection Algorithm for Distributed Computations. Inf. Process. Lett. 16(5), 217–219 (1983)

[4] Ganguly, S., Silberschatz, A., Tsur, S.: Parallel Bottom-Up Processing of Datalog Queries. Journal of Logic Programming 14(1–2), 101–126 (1992)

[5] Gu, R., Wang, S., Wang, F., Yuan, C., Huang, Y.: Cichlid: Efficient Large Scale RDFS/OWL Reasoning with Spark. In: IPDPS. pp. 700–709 (2015)

[6] Kaoudi, Z., Miliaraki, I., Koubarakis, M.: RDFS Reasoning and Query Answering on Top of DHTs. In: ISWC. pp. 499–516 (2008)
[7] Kolovski, V., Wu, Z., Eadon, G.: Optimizing Enterprise-Scale OWL 2 RL Reasoning in a Relational Database System. In: ISWC. pp. 436–452 (2010)
[8] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. CACM 21(7), 558–565 (1978)
[9] Liu, Y., McBrien, P.: SPOWL: Spark-based OWL 2 Reasoning Materialisation. In: BeyondMR@SIGMOD 2017. pp. 3:1–3:10 (2017)
[10] Motik, B., Nenov, Y., Piro, R., Horrocks, I., Olteanu, D.: Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems. In: AAAI. pp. 129–137 (2014)
[11] Oren, E., Kotoulas, S., Anadiotis, G., Siebes, R., ten Teije, A., van Harmelen, F.: Marvin: Distributed reasoning over large-scale Semantic Web data. JWS 7(4), 305–316 (2009)
[12] Potter, A., Motik, B., Nenov, Y., Horrocks, I.: Dynamic Data Exchange in Distributed RDF Stores. IEEE TKDE 30(12), 2312–2325 (2018)
[13] Seib, J., Lausen, G.: Parallelizing Datalog Programs by Generalized Pivoting. In: PODS. pp. 241–251 (1991)
[14] Seo, J., Park, J., Shin, J., Lam, M.: Distributed SociaLite: A Datalog-Based Language for Large-Scale Graph Analysis. PVLDB 6(14), 1906–1917 (2013)
[15] Shao, J., Bell, D., Hull, E.: Combining Rule Decomposition and Data Partitioning in Parallel Datalog Processing. In: PDIS. pp. 106–115 (1991)
[16] Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.: WebPIE: A Web-scale Parallel Inference Engine using MapReduce. JWS 10 (2012)
[17] Urbani, J., Jacobs, C., Krötzsch, M.: Column-Oriented Datalog Materialization for Large Knowledge Graphs. In: AAAI. pp. 258–264 (2016)
[18] Weaver, J., Hendler, J.A.: Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. In: ISWC. pp. 682–697 (2009)
[19] Wolfson, O., Ozeri, A.: Parallel and Distributed Processing of Rules by Data-Reduction. IEEE TKDE 5(3), 523–530 (1993)
[20] Wu, H., Liu, J., Wang, T., Ye, D., Wei, J., Zhong, H.: Parallel Materialization of Datalog Programs with Spark. In: WISE. pp. 363–379 (2016)
[21] Zhang, W., Wang, K., Chau, S.C.: Data Partition and Parallel Evaluation of Datalog Programs. IEEE TKDE 7(1), 163–176 (1995)

A Proofs

Lemma 1. *In each run of the algorithm, for each server k, and all facts f_1 and f_2, we have $T(f_1) < T(f_2)$ whenever one of the following holds:*

- $\mathsf{PAR}_k(f_1, i) \rightsquigarrow \mathsf{add}_k(f_2)$ *for some i,*
- $\mathsf{process}_k(f_1) \rightsquigarrow \mathsf{FCT}_k(f_2)$*, or*
- $\mathsf{PAR}_k(f_1, i) \rightsquigarrow \mathsf{FCT}_k(f_2)$ *for some i.*

Proof. Consider an arbitrary run of Algorithms 1 and 2, arbitrary server k, arbitrary facts f_1 and f_2, and arbitrary events as specified in the lemma.

Assume that $\mathsf{PAR}_k(f_1) \rightsquigarrow \mathsf{add}_k(f_2)$ holds. After the call to SYNCHRONISE in line 12, the local clock of server k has a value that is strictly larger than $T(f_1)$. Thus, when f_2 is assigned a timestamp, $T(f_2) > T(f_1)$ holds.

If $\mathsf{process}_k(f_1) \rightsquigarrow \mathsf{FCT}_k(f_2)$ (resp. $\mathsf{PAR}_k(f_1) \rightsquigarrow \mathsf{FCT}_k(f_2)$) holds, then after the call to SYNCHRONISE in line 8 (resp. 12), the local clock of server k has a value that is strictly larger than $T(f_1)$. Server k reads this value into the FCT message for f_2 in line 41 or line 23. Before f_2 is added to $I_{k'}$ on a destination server k', server k' calls SYNCHRONISE in line 16, which in turn ensures that $T(f_2) > T(f_1)$ holds, as required. □

Before proceeding, for a fact $f \in P^\infty(I)$ and a constant c occurring in f, let $L(f, c)$ be the set of servers defined as follows.

- If $f \in I$ (i.e., f occurs in the input), then $L(f, c) = \bigcup_{\pi \in \Pi} \mu_{j,\pi}(c)$, where j is the server that contains f at algorithm's start, and $\mu_{j,\pi}$ are the initial occurrence mappings.
- If $f \in P^\infty(I) \setminus I$ (i.e., f is derived), then $L(f, c) = \bigcup_{\pi \in \Pi} \lambda_\pi(c)$, where λ_s, λ_p, and λ_o are partial occurrences from lines 36–38 at the point when a FCT message is sent for fact f in line 40 or 41.

Moreover, given facts f and g, we say f *supports* g if f is matched to a body of a rule that derives g in the run of our algorithm. We next show that $L(f, c)$ satisfies the following two auxiliary properties.

Lemma A.1. *For all facts $f, g \in P^\infty(I)$ such that f supports g, and for each constant c that occurs in f and g, property $L(f, c) \subseteq L(g, c)$ holds.*

Proof. Consider arbitrary facts $f, g \in P^\infty(I)$ such that f supports g, and consider arbitrary constant c that occurs in f and g. Furthermore, consider an arbitrary server $k \in L(f, c)$. We consider the following two cases.

Assume $f \in I$. By the definition of $L(f, c)$, there exists $\pi \in \Pi$ such that $k \in \mu_{j,\pi}(c)$, where j is the server that initially contains f, and $\mu_{j,\pi}$ is the initial occurrence mapping. Now if c occurs in the head of the atom that derives g, then k is added to λ_π in line 33, so $k \in L(g, c)$ holds. Otherwise, the rule that derives g contains a variable x that is matched to f and that occurs in the head atom deriving g, so k is added to λ_π in line 30 and $k \in L(g, c)$ holds as well.

Assume $f \in P^\infty(I) \setminus I$. By the definition of $L(f, c)$, there exists $\pi \in \Pi$ such that $k \in \lambda_\pi(c)$, where λ_π is the partial occurrence mapping from lines 36–38 at the point when a FCT message is sent for fact f in line 40 or 41. Fact f is eventually sent to its host server ℓ, where $\lambda_\pi(c)$ is merged into $\mu_{\ell,\pi}(c)$ in line 19. But then, k is added to $L(g, c)$ analogously as in the previous case. □

Lemma A.2. *For all facts $f_1, f_2 \in P^\infty(I)$ and each constant c that occurs in both f_1 and f_2, property $L(f_1, c) \cap L(f_2, c) \neq \emptyset$ holds.*

Proof. We prove the claim by showing that, for each i, all facts $f_1, f_2 \in P^i(I)$, and each constant c that occurs in both f_1 and f_2, property $L(f_1, c) \cap L(f_2, c) \neq \emptyset$ holds. The base case for $i = 0$ is trivial because $L(f_1, c) = L(f_2, c)$ holds.

Now assume that the claim holds for some $i - 1$; we show that the property holds for arbitrary facts $f_1, f_2 \in P^i(I)$. To this end, we define f_1' as follows: if $f_1 \in P^{i-1}(I)$, then let $f_1' = f_1$; otherwise, let f_1' be an arbitrary fact that supports f_1. Either way, these definitions and Lemma A.1 clearly ensure $f_1' \in P^{i-1}(I)$ and $L(f_1', c) \subseteq L(f_1, c)$. We define f_2' analogously; in similar vein, we have $f_2' \in P^{i-1}(I)$ and $L(f_2', c) \subseteq L(f_2, c)$. Moreover, the induction assumption ensures $L(f_1', c) \cap L(f_2', c) \neq \emptyset$. But then, all of these observations clearly ensure $L(f_1, c) \cap L(f_2, c) \neq \emptyset$, as required. □

To prove Lemma 2, we introduce another event in addition to the events from Section 4.6: for f a fact, c a constant in f, and $\pi \in \Pi$ a position, $\mathsf{addOcc}_k(f, c, \pi)$ is the point when $\mu_{k,\pi}(c)$ is updated on server k in line 19 while processing an FCT message for f. We extend relation $\rightsquigarrow$ to such events as well.

Lemma 2. *At any point in the algorithm's run, for all servers k and j, each position $\pi \in \Pi$, and each constant c that is local to server k and that occurs in I_j at position π, property $j \in \mu_{k,\pi}(c)$ holds at that point.*

Proof. Consider arbitrary servers k and j, position π, constant c, and a point during the algorithm's run such that c local to k and it occurs in I_j at position π. We show that $j \in \mu_{k,\pi}(c)$ holds at that point too. If c is local to k and it occurs in I_j at position π when the algorithm is started, this claim holds because the occurrence mappings are initialised consistently. Hence, in the rest of this proof, we assume that either c does not occur in I_j at position π when the algorithm is started, or c is not local to k when the algorithm is started.

(*Case 1*) Assume that neither of the two conditions hold. Let f be the first fact containing c in position π that is added to I_j. Moreover, let f' be the first fact containing c in any position that is added to I_k and thus causes c to become local on k. Let π' be a position of c in f' (i.e., we choose one π' if c occurs in f' more than once). We have the following two possibilities.

If $k = j$, then the order of the operations in our algorithm ensures that $\mathsf{addOcc}_j(f, c, \pi) \rightsquigarrow \mathsf{add}_j(f)$ holds, so the property of the lemma holds. Therefore, assume that $k \neq j$ holds. By Lemma A.2, there exists a server ℓ such that $\ell \in L(f, c) \cap L(f', c)$ holds. Since f is the first fact containing c in position π that is added to I_j, set $L(f, c)$ is included into the set D in lines 36–38 and so event $\mathsf{FCT}_\ell(f)$ occurs on server ℓ. Moreover, f' is the first fact containing c in any position that is added to I_k, so event $\mathsf{FCT}_\ell(f')$ occurs on server ℓ for analogous reasons. Consequently, events $\mathsf{addOcc}_\ell(f, c, \pi)$ and $\mathsf{addOcc}_\ell(f', c, \pi')$ occur on server ℓ as well. We have the following possibilities.

- Assume that $\mathsf{addOcc}_\ell(f', c, \pi') \rightsquigarrow \mathsf{addOcc}_\ell(f, c, \pi)$ holds. Line 37 ensures that k is present in the partial occurrences for f', so line 19 adds k to $\mu_{\ell,\pi'}(c)$. Consequently, when $\mathsf{FCT}_\ell(f)$ is processed, line 18 ensures that k is added to the set D and so f is eventually forwarded to server k. Consequently, event $\mathsf{addOcc}_k(f, c, \pi)$ happens on server k, and so $\mathsf{addOcc}_k(f, c, \pi) \rightsquigarrow \mathsf{add}_j(f)$ holds because f is sent to server j last. Thus, the property of the lemma holds.
- Assume that $\mathsf{addOcc}_\ell(f, c, \pi) \rightsquigarrow \mathsf{addOcc}_\ell(f', c, \pi')$ holds. Line 37 ensures that j is present in the partial occurrences for f, and so line 19 adds j to $\mu_{\ell,\pi}(c)$. Consequently, when $\mathsf{FCT}_\ell(f')$ is processed, line 19 ensures that j is added to $\lambda_\pi(c)$; hence, when f' is eventually forwarded to server k, line 19 adds j to $\mu_{k,\pi}(c)$. This also is the point when c becomes local to k, so the property of the lemma holds.

(*Case 2*) Assume that, when the algorithm starts, c is local to k but it does not occur in I_j at position π. Let f_m be the first facts containing c in position π that is added to I_j. Clearly, there exists a finite sequence of facts $f_0, \ldots, f_m$ such that (i) f_i contains c for each $0 \leq i \leq m$, (ii) $f_0 \in I$ (i.e., f_0 occurs in the input to the algorithm) or f_0 is derived by a rule containing c in the head, and (iii) f_{i-1} supports f_i for each $1 \leq i \leq m$. Since c to local to k at the start, we have $k \in L(f_0, c)$. Moreover, Lemma A.1 ensures $L(f_{i-1}, c) \subseteq L(f_i, c)$ for $1 \leq i \leq m$. Thus, $k \in L(f_m, c)$ holds and, consequently, event $\mathsf{FCT}_k(f)$ happens on server k. Therefore $\mathsf{addOcc}_k(f, c, \pi) \rightsquigarrow \mathsf{add}_j(f)$ holds because f is sent to server j last, so the property of the lemma holds.

(*Case 3*) Assume that, when the algorithm starts, c is not local to k but it occurs in I_j at position π. Let f_m be the first fact containing c in any position that is added to I_k and thus causes c to become local to k. Clearly, there exists a finite sequence of facts $f_0, \ldots, f_m$ such that (i) f_i contains c for each $0 \leq i \leq m$, (ii) $f_0 \in I$ (i.e., f_0 occurs in the input to the algorithm) or f_0 is derived by a rule containing c in the head, and (iii) f_{i-1} supports f_i for each $1 \leq i \leq m$. Since c occurs in I_j at position π at the start, we have $j \in L(f_0, c)$. Moreover, Lemma A.1 ensures $L(f_{i-1}, c) \subseteq L(f_i, c)$ for $1 \leq i \leq m$. Thus, $j \in L(f_m, c)$ holds. Since f_m is the first fact containing c that is added to I_k and thus makes c local, line 19 adds j to $\mu_{k,\pi}(c)$, so the property of the lemma holds. □

Theorem 1. *For $I_1, \ldots, I_\ell$ the sets obtained by applying Algorithms 1 and 2 to an input set of facts I and program P, we have $P^\infty(I) = I_1 \cup \cdots \cup I_\ell$. Moreover, the algorithm exhibits the nonrepetition property.*

Proof (Soundness). Let P be a program, let I be an input dataset, and let $I_1, \ldots, I_\ell$ be the datasets computes after Algorithms 1 and 2 finish on some partition of I to ℓ servers. The proof is by induction on the construction of sets I_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 $a\sigma \in I_k$; moreover, as matching of Q progresses, each substitution σ' returned in line line 13 satisfies $a_i\sigma' \in I_{k'}$; consequently, each substitution σ in line 41 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 $h\sigma \in P^\infty(I)$. $\square$

Proof (Completeness). Let P be a program, let I be an input dataset, and let $I_1, \ldots, I_\ell$ be the datasets computed after Algorithms 1 and 2 finish on some partition of I to ℓ servers. Our claim follows from the following property:

($*$) for each i and each fact $f \in P^i(I)$, a server k exists were $f \in I_k$ holds.

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 $h \leftarrow b_0 \wedge \cdots \wedge b_n \in P$ 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 $a_0, \ldots, a_n$ 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 $\textsc{MatchRules}(b_p\sigma, P)$ 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 $a_0, \ldots, a_j$ 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 $a_j\sigma \in I_{k_j}$ holds. We next prove the following:

($\Diamond$) for each $0 \leq j \leq n$, function $\textsc{FinishMatch}(j, \sigma_j, a_j, Q, h, \tau_0, \boldsymbol{\lambda}_j)$ is called for some partial occurrence mapping $\boldsymbol{\lambda}_j$.

Property ($\Diamond$) implies our claim because in lines 31–41 the algorithm then constructs a FCT message for $h\sigma$ and dispatches it to some server k_h, so $h\sigma$ is eventually added to I_{k_h} in line 20, as required for ($*$).

We next prove ($\Diamond$) by induction on $0 \leq j \leq n$. For the base case, $a_0\sigma \in I_{k_0}$ ensures that $\textsc{ProcessFact}(a_0\sigma)$ is called on server k_0, so $\textsc{MatchRules}(a_0\sigma, P)$ returns (σ_0, a_0, Q, h), and $\textsc{FinishMatch}(0, \sigma_0, a_0, Q, h, \tau_0, \boldsymbol{\emptyset})$ is called in line 10. 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. To this end, we consider several cases.

Assume that event $\mathsf{PAR}_{k_{j+1}}(a_0\sigma, j+1)$ occurs at some point during the algorithm's run. Server k_{j+1} then calls $\textsc{Evaluate}$ at line 13 for $a_{j+1}^{\bowtie_{j+1}}$. Note that $a_{j+1}\sigma \in I_{k_{j+1}}$ holds by induction assumption. We next show that server k_{j+1} contains $a_{j+1}\sigma$ at the point in time when line 13 is executed. We have the following possibilities.

- If event $\mathsf{add}_{k_{j_1}}(a_{j+1}\sigma)$ never happens, then server k_{j+1} contains fact $a_{j+1}\sigma$ since the algorithm's start.
- If $\mathsf{add}_{k_{j_1}}(a_{j+1}\sigma) \rightsquigarrow \mathsf{PAR}_{k_{j+1}}(a_0\sigma, j+1)$ holds, then server k_{j+1} clearly contains fact $a_{j+1}\sigma$ at this point in time.
- If $\mathsf{PAR}_{k_{j+1}}(a_0\sigma, j+1) \rightsquigarrow \mathsf{add}_{k_{j_1}}(a_{j+1}\sigma)$ were to hold, then Lemma 1 implies $T(a_0\sigma) < T(a_{j+1}\sigma)$, contradicting 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 (3) ensures that $\bowtie_{j+1} = \leq$. Consequently, the call to EVALUATE in line 13 on server k_{j+1} returns σ_{j+1}, so the call in line 14 ensures ($\Diamond$).

Now assume that event $\mathsf{PAR}_{k_{j+1}}(a_0\sigma, j+1)$ never occurs during the algorithm's run—that is, server k_j never forwards a PAR message to server k_{j+1}. Then, for some $\pi \in \Pi$ and $c = a_{j+1}\sigma_j|_\pi$, we have $k_{j+1} \notin \lambda_\pi(c)$ at the point in time when line 44 is executed on server k_j, ensuring that k_{j+1} is removed from D. However, this $\lambda_\pi(c)$ is populated in line 30 when, for some index $0 \leq s \leq j$ of an atom a_s, constant c is matched on server k_s, so at that point in time we have $k_{j+1} \notin \mu_{k_s,\pi}(c)$. Now if event $\mathsf{add}_{k_{j+1}}(a_{j+1}\sigma)$ never happens, then server k_{j+1} would contain $a_{j+1}\sigma$ when the algorithm starts; but then, since $\mu_{k_s,\pi}$ is defined on c, Lemma 2 implies $k_{j+1} \in \mu_{k_s,\pi}(c)$, which is a contradiction. Consequently, event $\mathsf{add}_{k_{j+1}}(a_{j+1}\sigma)$ occurs on server k_{j+1}. Moreover, let $\alpha = \mathsf{process}_{k_s}(a_0\sigma)$ if $s = 0$, and otherwise let $\alpha = \mathsf{PAR}_{k_s}(a_0\sigma, j)$ if $s > 0$. By the induction assumption, property ($\Diamond$) holds for s, so event α occurs on server k_s. Let g be the first fact containing c in position π that is added to $I_{k_{j+1}}$; such a fact exists because $a_{j+1}\sigma$ contains c and it is added to $I_{k_{j+1}}$. We have the following two possibilities.

- Assume that $k_s \in L(g, c)$ holds. Event $\mathsf{FCT}_{k_s}(g)$ occurs on server k_s and line 19 adds k_{j+1} to $\mu_{k_s,\pi}(c)$. We know that $k_{j+1} \notin \mu_{k_s,\pi}(c)$ holds at the point in time when c is matched by σ, so $\alpha \rightsquigarrow \mathsf{FCT}_{k_s}(g) \rightsquigarrow \mathsf{FCT}_{k_s}(a_{j+1}\sigma)$ holds. Thus, regardless of how α is defined, Lemma 1 implies $T(a_0\sigma) < T(a_{j+1}\sigma)$, which contradicts our assumption that $T(a_{j+1}\sigma) \leq T(a_0\sigma)$ holds.
- Assume that $k_s \notin L(g, c)$ holds. Let g' be the first fact added to I_{k_s} that contains c in any position. By applying Lemma A.2 to g and g', there exists k' with $k' \in L(g, c) \cap L(g', c)$. Both $\mathsf{FCT}_{k'}(g)$ and $\mathsf{FCT}_{k'}(g')$ occur on k', in one of the following two orders. If $\mathsf{FCT}_{k'}(g) \rightsquigarrow \mathsf{FCT}_{k'}(g')$ holds, then k_{j+1} is added to $\mu_{k',\pi}$ in line 19 when the first message is processed, and then to λ_π in line 19 when the second message is processed. This event is followed by $\mathsf{FCT}_{k_s}(g')$, which adds k_{j+1} to $\mu_{k_s,\pi}(c)$. We know that $k_{j+1} \notin \mu_{k_s,\pi}(c)$ holds at the point in time when c is matched by σ, so $\alpha \rightsquigarrow \mathsf{FCT}_{k_s}(g')$ holds and Lemma 1 ensures $T(a_0\sigma) < T(g')$. But g' is the first appearance of c in server k_s, which implies $T(a_0\sigma) < T(a_s\sigma)$ and thus contradicts our assumption that $T(a_s\sigma) \leq T(a_0\sigma)$ holds. If $\mathsf{FCT}_{k'}(g') \rightsquigarrow \mathsf{FCT}_{k'}(g)$ holds, then k_s is added to $\mu_{k',\pi}$ in line 19 when the first message is processed, and then to D in line 18 when the second message is processed; thus, $\mathsf{FCT}_{k_s}(g)$ eventually happens and we complete the proof as in the first case.

Thus, assuming that $\mathsf{PAR}_{k_{j+1}}(a_0\sigma, j+1)$ never happens leads to a contradiction, so $\mathsf{PAR}_{k_{j+1}}(a_0\sigma, j+1)$ occurs on server k_{j+1}, thus proving our claim. □

Proof (Nonrepetition of Derivations). 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. $\square$