

# Materialisation and Data Partitioning Algorithms for Distributed RDF Systems

Temitope Ajileye

Wolfson College  
University of Oxford

*A thesis submitted for the degree of  
Doctor of Philosophy*

Trinity 2021

## 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. Whereas 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 fewer resources.



# Materialisation and Data Partitioning Algorithms for Distributed RDF Systems



Temitope Ajileye  
Wolfson College  
University of Oxford

A thesis submitted for the degree of  
*Doctor of Philosophy*  
Trinity 2021



# Acknowledgements

I'd like to thank my supervisor Boris Motik for his close guidance, patience and support over the past four years, and my supervisor Ian Horrocks for checking and providing direction at pivotal moments. I am also grateful to the accessors of the initial stages of my thesis for their feedback. My experience was made richer by the friends I made along the way, but my gratitude will fill other pages and spaces.

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



# 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. Whereas 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 fewer resources.





# Contents

<b>I</b>	<b>Foundations</b>	<b>1</b>
<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	A Brief Historical Background . . . . .	3
1.2	Scalable Automated Reasoning with Datalog . . . . .	5
1.3	Summary of Contributions . . . . .	8
<b>2</b>	<b>Preliminaries</b>	<b>11</b>
2.1	RDF, Queries, and Rules . . . . .	11
2.2	Substitutions and Rule Evaluations . . . . .	12
2.3	Seminaïve Evaluation . . . . .	13
2.4	Data Partitioning . . . . .	14
<b>II</b>	<b>Distributed Datalog Materialisation</b>	<b>17</b>
<b>3</b>	<b>Related Work and Technical Challenges</b>	<b>19</b>
3.1	Approaches to Distributed Query Answering . . . . .	19
3.1.1	Distributed Joins . . . . .	20
3.1.2	Indexes and Joins in Distributed RDF Stores . . . . .	22
3.1.3	Dynamic Data Exchange . . . . .	25
3.2	Approaches to Distributed Reasoning . . . . .	27
3.2.1	Rule-Centric Distribution . . . . .	27
3.2.2	Data-Centric Distribution . . . . .	28
3.3	Technical Challenges . . . . .	30
<b>4</b>	<b>Distributed Materialisation Algorithm</b>	<b>35</b>
4.1	Lamport Timestamps . . . . .	35
4.2	Occurrence Mappings . . . . .	38
4.3	Communication Infrastructure and Messages . . . . .	41
4.4	Termination Detection . . . . .	42
4.5	The Algorithm . . . . .	42
4.6	Proof of Correctness . . . . .	46

<b>III</b>	<b>Data Partitioning</b>	<b>61</b>
<b>5</b>	<b>Related Work and Technical Challenges</b>	<b>63</b>
5.1	Data Partitioning in RDF Stores . . . . .	64
5.1.1	Hash-Based Schemes . . . . .	64
5.1.2	Vertical Partitioning Schemes . . . . .	65
5.1.3	Min-Cut Schemes . . . . .	65
5.1.4	Horizontal Containment vs Workload Balance . . . . .	66
5.2	Streaming Graph Partitioning . . . . .	66
<b>6</b>	<b>The HDRF<sub>3</sub> Algorithm</b>	<b>69</b>
6.1	The Original HDRF Algorithm . . . . .	69
6.2	Adapting the Algorithm to RDF Graphs . . . . .	71
6.3	Proof of Proposition 6.2.1 . . . . .	73
<b>7</b>	<b>The 2PS<sub>3</sub> Algorithm</b>	<b>77</b>
7.1	The Original 2PS Algorithm . . . . .	77
7.2	The Algorithm . . . . .	78
7.3	Proof of Proposition 7.2.1 . . . . .	80
<b>IV</b>	<b>Evaluation</b>	<b>83</b>
<b>8</b>	<b>Evaluation</b>	<b>85</b>
8.1	Datasets . . . . .	86
8.2	Data Partitioning Experiments . . . . .	87
8.3	Scalability Experiments . . . . .	92
8.4	System Comparison Experiments . . . . .	96
<b>9</b>	<b>Conclusion and Further Work</b>	<b>99</b>
	<b>Bibliography</b>	<b>101</b>

# List of Algorithms

1	Distributed Materialisation Algorithm at Server $k$ in a cluster of $\ell$ servers . . . . .	43
2	HDRF <sub>3</sub> . . . . .	72
3	2PS <sub>3</sub> . . . . .	78



# Part I

## Foundations



# 1

## Introduction

### Contents

---

<b>1.1</b>	<b>A Brief Historical Background . . . . .</b>	<b>3</b>
<b>1.2</b>	<b>Scalable Automated Reasoning with Datalog . . . . .</b>	<b>5</b>
<b>1.3</b>	<b>Summary of Contributions . . . . .</b>	<b>8</b>

---

### 1.1 A Brief Historical Background

The contributions of this thesis target the intersection of two long-standing efforts within the field of information systems: *deductive databases* and *semantic web* technologies. Deductive databases originate from the coming together of logic and databases, which was the programme of a workshop organised in 1977 by Gallaire, Nicolas, and Minker [17]. In the abstract to the proceedings of the workshop, they note:

The theme of this collection is that mathematical logic provides a conceptual framework for data base systems. The first unit demonstrates this, showing explicitly how data base concepts can be analyzed in terms of formal logic, and provides a characterization of the hypothetical worlds on which data base systems work. The second analyzes knowledge representation and how it relates to the process of deduction; alternate approaches are described and their advantages and limitations are noted. The third considers how logic may be used to express constraints on

a data base and to maintain data base integrity. A similar approach proves useful in dealing with the fourth topic — meaning assigned to answers when negative questions are posed. The last demonstrates how logic provides a unifying framework for query language.

The key-points touched in that abstract, namely logical analysis of database concepts, knowledge representation formalisms, data constraints, and query languages, are still today the object of active research and debate, with the conversation focusing around the datalog language and its extensions (see Abiteboul et al. [3], Miller et al. [44], Bellomarini et al. [6]). Even though implementation of deductive systems were already presented at the 1977 workshop, and the database community has integrated many of the techniques stemming from the new sub-field into mainstream relational databases, commercial deductive systems struggle to scale while retaining their expressiveness.

The idea of the Semantic Web was popularised by Berners-Lee in a 2001 article [7] that advocated a transformation of the internet from a web of linked documents into a web of linked data. This transformation was, in Berners-Lee's vision, a coming of age of the World Wide Web into an information system that transacts machine readable data alongside human-readable documents. The article recognised Knowledge Representation as a foundational component of the proposed new system.

For the semantic web to function, computers must have access to structured collections of information and sets of inference rules that they can use to conduct automated reasoning. Artificial-intelligence researchers have studied such systems since long before the Web was developed. Knowledge representation, as this technology is often called, is currently in a state comparable to that of hypertext before the advent of the Web: it is clearly a good idea, and some very nice demonstrations exist, but it has not yet changed the world. It contains the seeds of important applications, but to realize its full potential it must be linked into a single global system. Traditional knowledge-representation systems typically have been centralized, requiring everyone to share exactly the same definition of common concepts such as "parent" or "vehicle." But central control is stifling, and increasing the size and scope of such a system rapidly becomes unmanageable.



Exactly 20 years have passed since that article, and the semantic-web has yet to be realised. Even though progress has been made<sup>1</sup>, and machines are able today to find and manipulate machine-readable data on the web fuelling applications in search engines, recommendation systems, and fact checkers, these results are confined to application islands: the semantic-web is still limited today by system and organisational bottlenecks. However, the efforts spent towards the semantic web are already yielding dividends at the smaller scale of individual companies, organisations, and fields, where domain specific-knowledge bases have been developed, published and exploited by applications that make use of automated reasoning, such as digital assistants, chatbots, organisation search, and many more. This larger family of tools constitute the semantic web technologies.

Both deductive databases and semantic web technologies deal with knowledge bases, and both are affected by the scalability limits of knowledge base management systems, where the application information is stored. This thesis presents new algorithms that expand those boundaries. The implementations and evaluations provided target knowledge bases developed for the semantic web, but the proposed algorithms can be adapted to general deductive databases. Section 1.2 presents the problem which the contributions of this thesis address, and Section 1.3 presents a summary of those contributions.

## 1.2 Scalable Automated Reasoning with Datalog

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. Such descriptions are often captured using a prominent

---

<sup>1</sup>The Linked Open Data cloud [<https://lod-cloud.net/>] now contains over 1300 datasets

rule-based formalism called *Datalog* [4]. A Datalog *rule* expresses an ‘if-then’ condition specifying how to derive one or more triples from structural patterns in the RDF dataset. When answering queries, RDF systems that support reasoning 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 [21], or by using the fixed rule set from the OWL 2 RL specification [45]. Furthermore, application logic is sometimes captured directly using Datalog rules [51, 41]. 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* [4], 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<sup>2</sup> 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 needed. In the context of distributed RDF querying, numerous solutions have been presented and incorporated into systems such as YARS2 [27], 4store [26], H-RDF-3X [29], Trinity.RDF [72], SHARD [54], SHAPE [38], Partout [16], AdPart [5], TriAD [23], SemStore [69], DREAM [25], and WARP [28].

---

<sup>2</sup><https://www.uniprot.org/>

Abdelaziz et al. [2] surveyed 22 and evaluated 11 such systems on a variety of data and query loads, showing AdPart [5] and TriAD [23] 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. [33] and Weaver and Hendler [67] handle RDFS rules; WebPie [66] and Cichlid [22] support the so-called *ter Horst fragment*; and SPOWL [40] 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 [70] can handle arbitrary rules, but it does not seem to use seminaïve evaluation. BigDatalog [61] and Cog [31] implement the seminaïve algorithm, but they seem to be able to process only a few linear rules at a time. Distributed Socialite [59] implements seminaïve algorithms 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 [34], where resources are partitioned to minimise the number of triples spanning the 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 computational resources and may be infeasible on large RDF datasets. Thus, the question of how to partition the data in distributed RDF reasoning systems is still largely open.

## 1.3 Summary of Contributions

This thesis presents several novel techniques that provide the foundation for scalable distributed RDF reasoning systems. The contributions are two-fold.

Part II presents a new algorithm for distributed materialisation of Datalog rules over RDF datasets. The algorithm builds on the work by Potter et al. [53] 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 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, the algorithm must be able to partially order rule derivations across the cluster. The algorithm address this problem with *Lamport timestamps* [37], which allows it to support seminaïve evaluation without expensive maintenance of auxiliary relations. Another source of difficulty is to correctly maintain certain indexes used by dynamic data exchange when fresh triples are derived. The technical challenges will be reviewed in more detail in Chapter 3, and the materialisation algorithm will be presented in Chapter 4.

Part III presents two new algorithms for partitioning RDF data that produce tight partitions of resources, but without loading entire RDF datasets in memory. Specifically, the algorithms operate in *streaming* mode: they assign triples to servers ‘on the fly’ while reading the dataset sequentially (possibly more than once), and their memory use is determined by the number of resources, rather than triples. RDF datasets often contain at least an order of magnitude more resources than triples, so streaming algorithms are easier to apply to truly large datasets. The algorithms are based on two state-of-the-art 2PS [43] and HDRF [50] schemes for streaming partitioning of general graphs, and the main challenge is to take into

account the specifics of RDF: since subject–subject joins are the most common in RDF queries [18], all triples with the same subject should be placed on the same server to reduce network communication during reasoning. Chapters 6 and 7 discuss how the 2PS and HRDF algorithms can be modified to achieve this important property, but without affecting the quality of the resulting partitions.

To evaluate the aforementioned contributions, we implemented the reasoning and partitioning algorithms in a new prototype system called DMAT. The system is implemented on top of the RDFox system, which provides the mechanisms for storing and matching triples in each server. Chapter 8 presents the results of several experiments that were used to evaluate the algorithms. First, we analysed how different data partitioning strategies affect the performance of reasoning. Second, to explore the limits of my approach, we investigated how reasoning performance scales with increasing data loads. Third, to evaluate the reasoning approach against the state of the art, we compared the performance of materialisation in DMAT with that of BigDatalog [61] and Cog [31]. The 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, the contributions of this thesis seem to provide a sound foundation for the development of truly scalable distributed RDF reasoners.



# 2

## Preliminaries

### Contents

---

<b>2.1</b>	<b>RDF, Queries, and Rules . . . . .</b>	<b>11</b>
<b>2.2</b>	<b>Substitutions and Rule Evaluations . . . . .</b>	<b>12</b>
<b>2.3</b>	<b>Seminaïve Evaluation . . . . .</b>	<b>13</b>
<b>2.4</b>	<b>Data Partitioning . . . . .</b>	<b>14</b>

---

To make this thesis self-contained and precise, this Chapter recapitulates the definitions used in this and subsequent chapters. A small example will also be introduced, and used to illustrate the behaviour of the algorithms presented in Parts II and III.

### 2.1 RDF, Queries, and Rules

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 a fixed set of *positions*. Then, for  $a = \langle t_s, t_p, t_o \rangle$  an atom and  $\pi \in \Pi$  a position,  $a|_\pi$  is the term that occurs in atom  $a$  at position  $\pi$ —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 ( $x, y, z, \dots$ ) for variables, lowercase letters from the beginning of the alphabet ( $a, b, c, \dots$ ) for

subject and object constants, and uppercase letters from the middle of the alphabet ( $R, S, T, \dots$ ) 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 (2.1), where  $n \geq 1$  and all  $a_i$  are atoms. A Datalog *rule* is an implication of the form (2.2), 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.

$$a_1 \wedge \dots \wedge a_n \tag{2.1}$$

$$h \leftarrow b_1 \wedge \dots \wedge b_n \tag{2.2}$$

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 found in the literature on Datalog reasoning.

## 2.2 Substitutions and Rule Evaluations

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

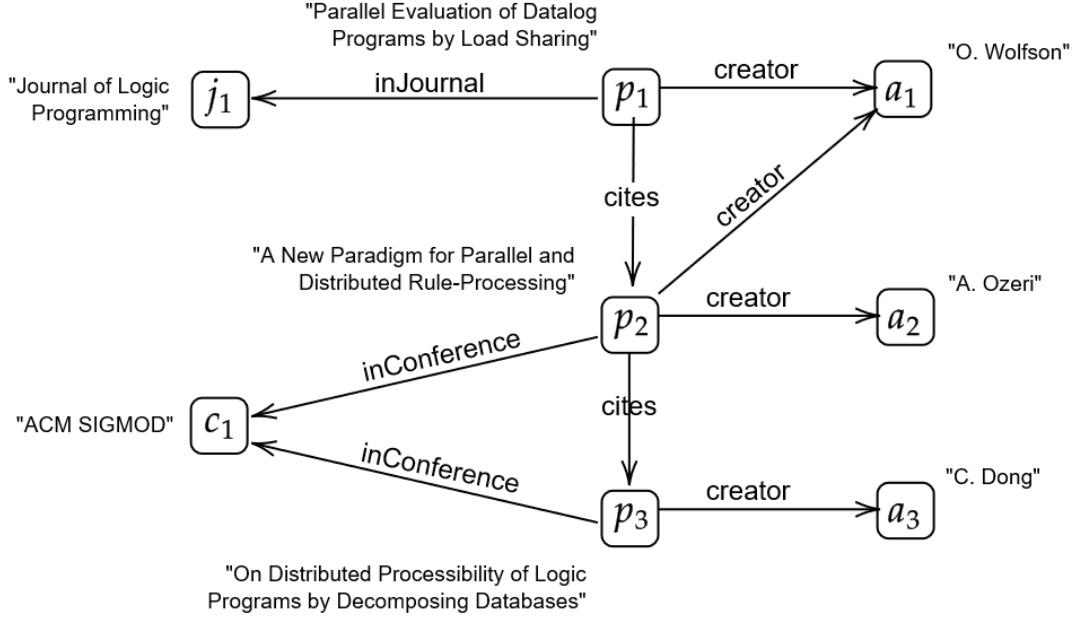
Let  $G$  be a dataset. For  $Q$  a query of the form (2.1), substitution  $\sigma$  is an *answer* to  $Q$  on  $G$  if  $\sigma$  is defined precisely on all variables occurring in  $Q$ , and  $a_i\sigma \in G$  holds for each  $1 \leq i \leq n$ . For  $r$  a rule of the form (2.2), the result of applying  $r$  to  $G$  is the set of facts defined by

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

For  $P$  a program, let  $P(G) = \bigcup_{r \in R} r(G)$ ; and let  $P^{i+1}(G) = P(P^i(G))$  for  $i \geq 0$ , with  $P^0(G) = G$ . Then, the *closure* of  $P$  on  $G$  is defined as  $P^\infty(G) = \bigcup_{i \geq 0} P^i(G)$ .



**Figure 2.1:** A graphical representation of an example dataset,  $Ex_1$ . Labels have been included in this representation for information purposes, but are not included in the dataset. The example is sampled from academic publication records in Computer Science.



**Figure 2.2:** A program,  $P_1$ , for the dataset in Figure 2.1 that computes the transitive closure of the *cites* relationship.

$$\langle x, R, y \rangle \leftarrow \langle x, \text{cites}, y \rangle \quad (2.3)$$

$$\langle x, R, z \rangle \leftarrow \langle x, R, y \rangle \wedge \langle y, R, z \rangle \quad (2.4)$$

When the closure is computed in advance and persisted, we refer to both  $P^\infty(G)$  and the process of computing it as *materialisation*.

## 2.3 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 *seminaïve strategy* avoids this problem: when matching a rule  $r$  in round  $i+1$ , at least one body

**Figure 2.3:** An example run of the seminaïve algorithm on the  $E_1$  dataset and  $P_q$  program. New derivations in each round are marked with boldface, these facts are the basis upon which derivations in subsequent rounds are based. When there are no new derivations, the algorithm terminates. Facts with predicates that do not appear in  $P_1$  are hidden.

$Ex_1$	$\rightarrow P_1(Ex_1)$	$\rightarrow P_1^2(Ex_1)$	$\rightarrow P_1^3(Ex_1)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\langle \mathbf{p_1}, \mathbf{cites}, \mathbf{p_2} \rangle$	$\langle p_1, cites, p_2 \rangle$	$\langle p_1, cites, p_2 \rangle$	$\langle p_1, cites, p_2 \rangle$
$\langle \mathbf{p_2}, \mathbf{cites}, \mathbf{p_3} \rangle$	$\langle p_2, cites, p_3 \rangle$	$\langle p_2, cites, p_3 \rangle$	$\langle p_2, cites, p_3 \rangle$
	$\langle \mathbf{p_1}, \mathbf{R}, \mathbf{p_2} \rangle$	$\langle p_1, R, p_2 \rangle$	$\langle p_1, R, p_2 \rangle$
	$\langle \mathbf{p_2}, \mathbf{R}, \mathbf{p_3} \rangle$	$\langle p_2, R, p_3 \rangle$	$\langle p_2, R, p_3 \rangle$
		$\langle \mathbf{p_1}, \mathbf{R}, \mathbf{p_3} \rangle$	$\langle p_1, R, p_3 \rangle$

atom of  $r$  must be matched to a fact derived in round  $i$ . This is critical in practice even for very simple rules. This scheme will be explained in detail in Section 3.3.

## 2.4 Data Partitioning

A *partition*  $\mathcal{P}$  of an RDF dataset  $G$  is a list of RDF datasets  $\mathcal{P} = G_1, \dots, G_\ell$  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 *replication set* of a constant  $c$  in a partition is defined as  $A(c) = \{k \mid G_k \cap G(c) \neq \emptyset\}$ . The *replication factor* of a partition  $\mathcal{P}$  is defined as

$$\text{RF}(G, \mathcal{P}) = \frac{1}{|V|} \sum_{c \in V} |A(c)|, \quad (2.5)$$

where  $V$  is the vocabulary of  $G$ . 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  $\ell$ .

**Partitioning Problem.** Given a fixed tolerance parameter  $\alpha \geq 1$ , the objective of graph partitioning is to compute a partition  $\mathcal{P}$  of  $G$  such that  $|G_i| \leq \alpha \frac{|G|}{\ell}$  holds for each  $1 \leq i \leq \ell$ , while minimising the replication factor  $\text{RF}(G, \mathcal{P})$ . In other words, each  $G_i$  should hold roughly the same number of facts, while ensuring that constants

**Figure 2.4:** A partition of  $\mathcal{P} = \{Ex_{1,1}, Ex_{1,2}\}$  of  $Ex_1$  in two elements: all facts with  $p_1$  as a subject are included in the first partition element, and all facts with  $p_2$  or  $p_3$  as a subject are included in the second partition element. The replication factor is  $RF(Ex_1, \mathcal{P}) = \frac{1}{8} \left( \sum_{i=1}^3 A(p_i) + \sum_{i=1}^3 A(a_i) + A(j_1) + A(c_1) \right) = \frac{1}{8} (4 + 4 + 1 + 1) = 1.25$

$Ex_{1,1}$	$Ex_{1,2}$
$\langle p_1, inJournal, j_1 \rangle$	$\langle p_2, inConference, c_1 \rangle$
$\langle p_1, creator, a_1 \rangle$	$\langle p_2, creator, a_1 \rangle$
$\langle p_1, cites, p_2 \rangle$	$\langle p_2, creator, a_2 \rangle$
	$\langle p_2, cites, p_3 \rangle$
	$\langle p_3, inConference, c_1 \rangle$
	$\langle p_3, creator, a_3 \rangle$

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.



# Part II

## Distributed Datalog Materialisation



# 3

## Related Work and Technical Challenges

### Contents

---

<b>3.1</b>	<b>Approaches to Distributed Query Answering . . . . .</b>	<b>19</b>
3.1.1	Distributed Joins . . . . .	20
3.1.2	Indexes and Joins in Distributed RDF Stores . . . . .	22
3.1.3	Dynamic Data Exchange . . . . .	25
<b>3.2</b>	<b>Approaches to Distributed Reasoning . . . . .</b>	<b>27</b>
3.2.1	Rule-Centric Distribution . . . . .	27
3.2.2	Data-Centric Distribution . . . . .	28
<b>3.3</b>	<b>Technical Challenges . . . . .</b>	<b>30</b>

---

This chapter discusses the relevant related work to the materialisation algorithm in Chapter 4. Distributed query answering is a key ingredient of distributed reasoning algorithms, therefore Section 3.1 discusses the existing approaches to query answering in distributed RDF stores. Section 3.2 discusses the existing approaches to parallel datalog materialisation, in both centralised and distributed settings.

### 3.1 Approaches to Distributed Query Answering

Most distributed query evaluation algorithms can be understood as using a variant of the data exchange operators first introduced in the Volcano system [20]. Such operators send and receive partial query answers during query evaluation, and are

**Figure 3.1:** Example  $Ex_2$ 

Server 0 - $Ex_{2,1}$	Server 1 - $Ex_{2,2}$
$\langle p_1, R, a_1 \rangle$	$\langle p_2, R, a_1 \rangle$
$\langle p_4, R, a_3 \rangle$	$\langle p_2, R, a_2 \rangle$
$\langle p_1, S, p_2 \rangle$	$\langle p_3, R, a_3 \rangle$
$\langle p_1, S, p_4 \rangle$	$\langle p_2, S, p_3 \rangle$

introduced into query plans to ensure that all partial answers that may participate in a join are transmitted to one server. Section 3.1.1 explores various distributed join strategies found in database literature; Section 3.1.2 looks into how those strategies, in combination with other components, are deployed in distributed RDF stores to answer queries; and Section 3.1.3 focuses on dynamic data exchange, the approach which provided a starting point for the materialisation algorithm presented in Chapter 4.

To make these ideas concrete, consider evaluating query  $Q = \langle x, S, y \rangle \wedge \langle y, R, z \rangle$  over the partition elements in Figure 3.1. This example will be used to illustrate the behaviour of the algorithms presented below. Whenever a hash value is required, assume the hash function being used is as defined below.

$$hash : a_i \mapsto i \quad hash : p_i \mapsto i$$

### 3.1.1 Distributed Joins

In order to join relation  $R$  to relation  $S$  one has to ensure all  $R$ -facts and  $S$ -facts (respectively the facts with predicate  $R$  and  $S$ ) are located in, or sent to a server where they can be matched. One straightforward way to do this requires sending all facts to a server identified by the hashing of their join key. This server will then compute the join and return the results to the client or master server. When  $R$  or  $S$  are large relations, this can lead to a large amount of data being communicated over the network. The remainder of this subsection explores two alternatives that try to minimise data transfer across the network.



**Figure 3.2:** Semi Join

Server 0	Server 1
$\langle p_4, R, a_3 \rangle$	$\langle p_2, R, a_1 \rangle$
	$\langle p_2, R, a_2 \rangle$
	$\langle p_3, R, a_3 \rangle$

**Semi-Join.** The semi-join of  $R$  and  $S$ , denoted with  $R \ltimes S$  identifies the  $R$ -facts that join with some  $S$ -fact. The result of semi-joins can then be used to compute full joins (see Bernstein and Chiu [9], Bernstein et al. [8], Mullin [47]).

In the first phase, relation  $S$  is projected unto its join key and the result is broadcast to all servers, or to a subset identified by some hashing strategy, a sort strategy, or prior knowledge about the location of  $R$ -facts. This information is then used to compute  $R \ltimes S$ , as shown in Figure 3.2. Most join strategies implemented in Spark and similar distributed computing frameworks are based on the complex variants of the semi-join.

In the second phase, the reduced  $R$ -facts are broadcast to appropriate servers to complete the join. In the example only the triple  $\langle p_4, R, a_3 \rangle$  is being left out, but in many cases the number of constants is an order of magnitude less than the number of triples, therefore semi-join strategies can reduce the amount of network communication. In practice, systems use heuristics to determine when the gains of a semi-join strategy offsets the overhead of computing the semi-join.

**2-Phase Track Join.** The 2-phase variant of Track Join, presented here, is the simplest of three variants developed by Polychroniou et al. [52]. It assumes prior determination of which relation needs to be broadcast. In more complex variants, the algorithm decides, on a key-by-key basis, which relation will be broadcast; a complete description is outside the scope of this thesis and can be found in the original paper.

In the first phase, relations  $S$  and  $R$  are projected to their join key and sent over the network, to a server determined by hashing the key (modulo the number of servers). Every server receives and stores the keys alongside the id of the

**Figure 3.3:** Track Join - Phase 1

Server 0	Server 1
$\langle p_4, 0, R \rangle$	$\langle p_1, 0, R \rangle$
$\langle p_2, 0, S \rangle$	$\langle p_3, 1, R \rangle$
$\langle p_4, 0, S \rangle$	$\langle p_3, 1, S \rangle$
$\langle p_2, 1, R \rangle$	

**Figure 3.4:** Track Join - Phase 2A

Server 0	Server 1
$\langle p_4, 0 \rangle$	$\langle p_2, 0 \rangle$
	$\langle p_3, 1 \rangle$

**Figure 3.5:** Track Join - Phase 2B

Server 0	Server 1
$\langle p_1, S, p_4 \rangle \wedge \langle \mathbf{p}_4, \mathbf{R}, \mathbf{a}_3 \rangle$	$\langle p_2, S, p_3 \rangle \wedge \langle \mathbf{p}_3, \mathbf{R}, \mathbf{a}_3 \rangle$
$\langle p_1, S, p_2 \rangle \wedge \langle \mathbf{p}_2, \mathbf{R}, \mathbf{a}_1 \rangle$	
$\langle p_1, S, p_2 \rangle \wedge \langle \mathbf{p}_2, \mathbf{R}, \mathbf{a}_2 \rangle$	

source server and source relation. The tracking information collected after this phase is shown in Figure 3.3.

In the second phase, assuming that relation  $R$  will be broadcast, each server sends matching  $R$  keys to their source servers together with the ids of servers where they can be matched with  $S$ -facts. Then, each servers sends the full  $R$ -facts to the corresponding  $S$  servers where the join will be completed. The broadcast facts are highlighted in bold in Figure 3.5.

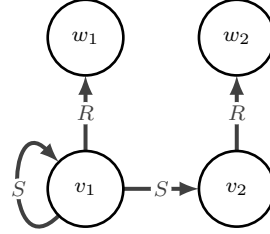
### 3.1.2 Indexes and Joins in Distributed RDF Stores

In addition to an optimal join strategy, database systems use indexing for efficient lookup operations. RDF facts (triples) are especially amenable to indexing because of their fixed number of components. Many RDF stores add each fact to multiple index structures, to speed up access when matching different types of query atoms. A number of distributed RDF stores have been proposed over the past 15 years have adopted this technique. A few notable ones are presented in this subsection.

**YARS2.** YARS2 was one of the first RDF stores to take advantage of multi-indexing. Before answering queries, YARS2 builds an index for each possible triple pattern. These indexes are implemented as blocked and sorted data files, so that

**Figure 3.6:** *pos*-index for  $Ex_2$ 

Server 0		
$R$	$a_1$	$\langle p_1, R, a_1 \rangle$
	$a_3$	$\langle p_4, R, a_3 \rangle$
$S$	$p_2$	$\langle p_1, S, p_2 \rangle$
	$p_4$	$\langle p_1, S, p_4 \rangle$
Server 1		
$R$	$a_1$	$\langle p_2, R, a_1 \rangle$
	$a_2$	$\langle p_2, R, a_2 \rangle$
	$a_3$	$\langle p_3, R, a_3 \rangle$
$S$	$p_3$	$\langle p_2, S, p_3 \rangle$

**Figure 3.7:** A summary graph for  $Ex_2$  with 4 buckets, where  $v_1 = \{p_1, p_2\}$ ,  $v_2 = \{p_3, p_4\}$ ,  $w_1 = \{a_1, a_2\}$ , and  $w_2 = \{a_3\}$ 

a single complete index (for example the *pos*-index), can be used to answer query atoms where the subject is a variable, by using the partial key  $p, o$ . The *pos*-index for  $Ex_2$  is shown in Figure 3.6. To answer a single query atom, each server probes remote indexes (all of them or a subset depending on the data placement strategy) and awaits their responses, which consists of matching facts.

To evaluate query  $Q$ , an *index nested loop join* is used. In the first query atom,  $\langle y, R, z \rangle$ , the predicate is the only non-variable, thus each server probes the  $p$  prefix of the local *pos*-index, and loops over the returned facts. At each step of the loop (for example when server 1 is processing  $\langle p_2, R, a_1 \rangle$ ) the second query atom receives the binding of the variable  $y$  ( $p_2$ ), and probes local and remote indexes for the resulting atom ( $\langle x, S, p_2 \rangle$ ). This operation is blocking and the thread resumes once the remote indexes have returned matching facts ( $\langle p_1, S, p_2 \rangle$  from server 0), after which the join can be completed.

**TriAd.** In addition to an index for each possible triple pattern, TriAd also uses an additional structure to minimise the amount of work needed to evaluate a query, a *graph summary*: a graph whose nodes correspond to buckets of the constants of the input dataset. TriAd runs the Metis graph partitioner to determine these buckets. An example summary graph is shown in Figure 3.7. To obtain the actual

partition elements, summary buckets are assigned to a server via hashing, and each fact is assigned twice to the location of its subject bucket and object bucket. This method aims to preserve the locality information provided by the summary graph. Each server keeps track of the bucket information in its local indexes, so that facts belonging to certain buckets can be skipped during lookup operations.

When TriAd begins processing  $Q$ , it first executes the query against the summary graph, which returns the two results  $\sigma_1 = \{x \mapsto v_1, y \mapsto v_1, z \mapsto w_1\}$  and  $\sigma_2 = \{x \mapsto v_1, y \mapsto v_2, z \mapsto w_2\}$ . In a second stage,  $Q$  is executed against the actual dataset, and the bucket bindings previously obtained are used to remove certain buckets or servers from consideration (pruning). During this stage, TriAd sends and receives partial results asynchronously.

**MapReduce-Based Systems** MapReduce is a family of software frameworks (see Hadoop[62], Spark[71], and Flink[11]) for processing large datasets in a distributed and fault-tolerant setting. Developers compose map functions that split data into key/value pairs, and reduce functions that merge the key/value pair based on the key, to define higher level tasks, while leaving the details of data placement and join strategy to the system.

The main benefit of implementing an RDF store on top of MapReduce is that such systems are easy to setup and scale out. However, defining a query often requires programming and experience in the configuration of the underlying framework for optimal results. For examples of MapReduce-based RDF stores, see [54, 56, 55, 57, 48]. There are two main techniques MapReduce systems use to compute join.

*Reduce-Side Joins* repartition and shuffle the joining relations based on the hash value of the join key as shown in Figure 3.8. A reduce function then computes the join.

*Map-Side Joins* are possible when the joining relations are already sorted by the joining key and partitioned in the same way, or when a large relation is being joined to a smaller one, in which case the smaller relation is copied into the main memory of each server. When either requirement is satisfied, the join can be

**Figure 3.8:** Reduce-Side Join, Map Phase

Server 0			Server 1		
Key	$R$	$S$	Key	$R$	$S$
$p_2$	$\langle p_2, a_1 \rangle$	$\langle p_1, p_2 \rangle$	$p_1$	$\langle p_1, a_1 \rangle$	
	$\langle p_2, a_2 \rangle$		$p_3$	$\langle p_3, a_3 \rangle$	$\langle p_2, p_3 \rangle$
$p_4$	$\langle p_4, a_3 \rangle$	$\langle p_1, p_4 \rangle$			

computed by the mapper function, which has direct access to all the facts matching the keys present in the local server.

### 3.1.3 Dynamic Data Exchange

Recently, Potter et al. [53] 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 govern network communication during query answering.

The occurrence mappings are initialised on each server to cover all constants occurring in this server. Thus, for Example  $Ex_2$ ,  $\mu_{1,s}$ ,  $\mu_{1,p}$ , and  $\mu_{1,o}$  are defined on all constants except  $p_3$  and  $a_2$ , whereas  $\mu_{2,s}$ ,  $\mu_{2,p}$ , and  $\mu_{2,o}$  are defined on all constants except  $p_1$  and  $p_4$ .

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 0 evaluates atom  $\langle x, S, y \rangle$  over  $Ex_{2,1}$ , which produces the *partial* answers  $\sigma_1 = \{x \mapsto p_1, y \mapsto p_2\}$  and  $\sigma_2 = \{x \mapsto p_1, y \mapsto p_4\}$ .

**Figure 3.9:** Occurrence mappings for  $Ex_2$  ( $R$  and  $S$  are mapped jointly in this presentation because of page constraints)

$$\begin{aligned}
\mu_{0,s} &= \{ p_1 \mapsto \{0\}, p_2 \mapsto \{1\}, p_4 \mapsto \{0\}, & a_1 \mapsto \emptyset & \quad a_3 \mapsto \emptyset & \quad R, S \mapsto \emptyset & \} \\
\mu_{0,p} &= \{ p_1 \mapsto \emptyset, \quad p_2 \mapsto \emptyset, \quad p_4 \mapsto \emptyset, & a_1 \mapsto \emptyset & \quad a_3 \mapsto \emptyset & \quad R, S \mapsto \{0, 1\} & \} \\
\mu_{0,o} &= \{ p_1 \mapsto \emptyset, \quad p_2 \mapsto \{0\}, p_4 \mapsto \{0\}, & a_1 \mapsto \{0, 1\} & \quad a_3 \mapsto \{0, 1\} & \quad R, S \mapsto \emptyset & \} \\
\mu_{1,s} &= \{ p_2 \mapsto \{1\}, p_3 \mapsto \{1\}, a_1 \mapsto \emptyset, & a_2 \mapsto \emptyset & \quad a_3 \mapsto \emptyset & \quad R, S \mapsto \emptyset & \} \\
\mu_{1,p} &= \{ p_2 \mapsto \emptyset, \quad p_3 \mapsto \emptyset, \quad a_1 \mapsto \emptyset, & a_2 \mapsto \emptyset & \quad a_3 \mapsto \emptyset & \quad R, S \mapsto \{0, 1\} & \} \\
\mu_{1,o} &= \{ p_2 \mapsto \{0\}, p_3 \mapsto \{1\}, a_1 \mapsto \{0, 1\}, a_2 \mapsto \{1\} & \quad a_3 \mapsto \{0, 1\} & \quad R, S \mapsto \emptyset & \}
\end{aligned}$$

Server 1 then evaluates  $\langle y, R, z \rangle \sigma_1 = \langle p_2, R, z \rangle$  over  $Ex_{2,1}$  and is unable to complete the matching. To see whether  $\langle p_2, R, z \rangle$  can be matched on other servers, server 0 consults its occurrence mappings for all constants in the atom. Since  $\mu_{0,s}(p_2) \cap \mu_{0,p}(R) = \{1\}$ , server 0 sends the partial answer  $\sigma_1$  to server 1, telling it to continue matching the query from the second atom. After receiving  $\sigma_1$ , server 2 matches atom  $\langle p_2, R, z \rangle$  in  $Ex_{2,2}$  to obtain another two full answers  $\sigma'_1 = \{x \mapsto p_1, y \mapsto p_2, z \mapsto a_1\}$  and  $\sigma''_1 = \{x \mapsto p_1, y \mapsto p_2, z \mapsto a_2\}$ . Server 0 also consults its occurrences to determine which servers can match  $\langle y, R, z \rangle \sigma_2 = \langle p_4, R, z \rangle$ . Since  $\mu_{0,s}(p_4) = \{0\}$ , server 0 knows it can proceed without any communication and computes the full answer  $\sigma'_2 = \{x \mapsto p_1, y \mapsto p_2, z \mapsto a_3\}$ .

This strategy has several important benefits. First, all answers that can be produced within a single server, such as  $\sigma_5$  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  $\sigma_1$  to server 2, server 1 does not need to wait for server 2 to finish, and server 2 can process  $\sigma_1$  whenever it can. The lack of synchronisation between servers is beneficial to parallelisation.

## 3.2 Approaches to Distributed Reasoning

Distributed computation can be driven by a necessity of more computational power or more capacity. Due to the number of rule-firings that can take to compute datalog materialisation, the primary motivation behind early approaches was the parallelisation of computation, therefore high rates of data replication were not seen as a problem, as shown in Section 3.2.1. With the growing size of public and enterprise data, there is a need for distributed systems that can cope with partitioned datasets with little or no data replication; some notable examples are surveyed in Section 3.2.2.

### 3.2.1 Rule-Centric Distribution

A number of approaches were developed in the 90s for materialising Datalog programs when the data is distributed across several processors. These approaches are not specific to RDF, but are relevant to 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  $\ell$  processors, we let each processor  $i$  with  $1 \leq i \leq \ell$  evaluate rule

$$\langle x, R, z \rangle \leftarrow \langle x, R, y \rangle \wedge \langle y, R, z \rangle \wedge h(y) = i, \quad (3.1)$$

where  $h(y)$  is a *partition function* that maps values of  $y$  to integers between 1 and  $\ell$ . If  $h$  is uniform, then each processor receives roughly the same fraction of the workload, which benefits parallelisation. However, since a fact of the form  $\langle s, R, o \rangle$  can match either atom in the body of (3.1), each such fact must be replicated to processors  $h(s)$  and  $h(o)$  to ensure completeness. Based on this idea, Ganguly et al. [19] show how to materialise arbitrary Datalog programs with seminaïve evaluation; Zhang et al. [74] study different partition functions; Seib and Lausen [58] identify programs and partition functions where no replication of derived facts is needed; Shao et al. [60] further break rules in segments; and Wolfson and Ozeri [68] replicate all facts to all processors in order to increase parallelism.

### 3.2.2 Data-Centric Distribution

Materialisation can also be implemented without any data replication. First, one must select a data partitioning strategy: a common approach is to assign each  $\langle s, p, o \rangle$  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.

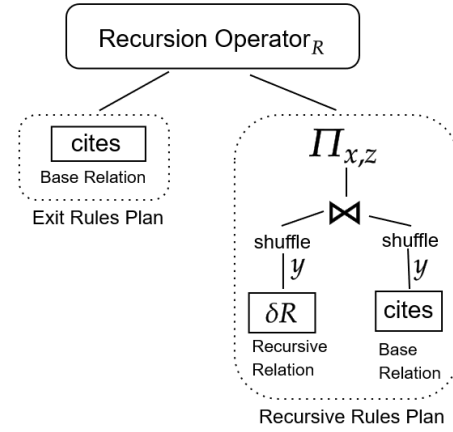
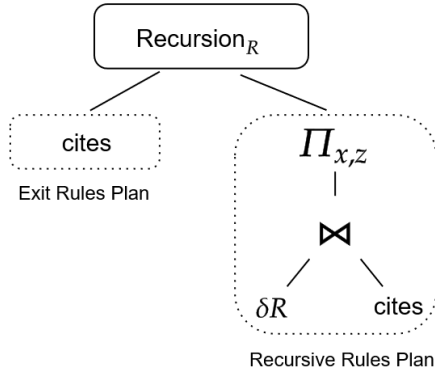
**Fixed Rulesets.** Most systems in this category handle only a fixed number of hardcoded rules. The systems by Weaver and Hendler [67] and Kaoudi et al. [33] support RDFS reasoning. Other systems borrow mechanisms for distributed data storage and query evaluation from big data frameworks such as Hadoop and Spark. In particular, WebPIE [66] supports the *ter Horst rules* [65] in Hadoop; Cichlid [22] supports *ter Horst rules*, but on top of Spark; and SPOWL [40] 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 the rules can be evaluated just once is evaluated in a certain order.

**Full Datalog in Triple Stores.** Greater generality is offered by PLogSPARK [70], a Spark-based system that handles general Datalog rules over RDF data. This system applies two optimisations techniques on top of a straightforward sequential evaluation of rules and joining of results. First, the input dataset is partitioned dynamically during rule evaluation, based on constants present in the program, such as the predicates. Second, rule dependency information is captured via sampling, and is used to improve the scheduling order of rule applications. Despite these optimisations, PlogSPARK uses naïve rule evaluation, which can be prohibitive when the rules are complex.



**Figure 3.10:**  $P_2$ , a linear variant of the program in Figure 2.2

$$\begin{aligned}\langle x, R, y \rangle &\leftarrow \langle x, \text{cites}, y \rangle \\ \langle x, R, z \rangle &\leftarrow \langle x, R, y \rangle \wedge \langle y, \text{cites}, z \rangle\end{aligned}$$

**Figure 3.11:** Logical Program for  $P_2$ **Figure 3.12:** Physical Program for  $P_2$ 

**Datalog and Databases.** Distributed Datalog reasoning has also been studied in the database community. Combining views with recursive queries, which are provided by major relational database vendors, can mimic the expressiveness of Datalog to some extent, limited by the fact that the SQL standard only supports linear recursion.

BigDatalog [61] and Cog [31] 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 just to answer the query.

For example, the program in Figure 3.10 is transformed into the logical plan in Figure 3.11 and then into the physical plan in Figure 3.12. Note that both BigDatalog and Cog partially rely on SQL libraries to construct the logical and physical plans. As a consequence they at times suffer from SQL limitations during rule plan compilation.

Figure 3.12 shows the physical plan for the evaluation of  $P_2$  on BigDatalog (the physical plan for Cog is very similar). The shuffle and join operators are executed internally using one of the methods mentioned in Sections 3.1.1 and 3.1.2. The recursion operator initially executes the exit (non recursive) rule plan, then iteratively executes the recursive rule plan and, removes duplicates, and joins the results with the recursive relations. Furthermore, BigDatalog benefits from an extension to Spark datasets (which are usually read-only) to support these repeated unions, and from an enhanced level of caching to speed up the evaluation of recursive rules.

Both BigDatalog and Cog can perform seminaïve evaluation, but they are limited, as reported in Section 8, to only a handful of linear rules.

**Datalog Extensions.** Yedalog [12] is Google’s private implementation of Datalog on top of a proprietary data model. Distributed Socialite [59] 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 (3.1) 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.

Yedalog and Socialite also implement extensions of Datalog, such as such as monotonic aggregation and builtin predicates (such as input/output predicates). These allows for a wider range of data-analytics pipelines and batch-processing jobs to be fully defined within those languages.

### 3.3 Technical Challenges

As mentioned in Section 2.3, 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. I discuss the problems of such a straightforward approach by means of the example rule (2.4), which is recalled below.

$$\langle x, R, z \rangle \leftarrow \langle x, R, y \rangle \wedge \langle y, R, z \rangle$$

Seminaïve evaluation applies rules in rounds and, to avoid repeating derivations, rules are applied in each round by matching at least one body atom to a fact derived in the previous round. A common way to ensure this is to maintain four sets of facts: the ‘current’ facts, the ‘old’ facts derived before the last round, the ‘delta’ set containing the difference between the ‘current’ and ‘old’ facts, and the ‘new’ facts. Rule (2.4) is then rewritten to use these sets as follows.

$$\langle x, R, z \rangle^{new} \leftarrow \langle x, R, y \rangle^{\Delta} \wedge \langle y, R, z \rangle \quad (3.2)$$

$$\langle x, R, z \rangle^{new} \leftarrow \langle x, R, y \rangle^{old} \wedge \langle y, R, z \rangle^{\Delta} \quad (3.3)$$

Thus, rules (3.2) and (3.3) produce the derivations of rule (2.4) where the first and the second body atom, respectively, are derived in the previous round of rule application; by matching the first body atom of rule (3.3) to ‘old’ facts, we ensure no repetition of inferences between the two rules. Rules (3.2) and (3.3) can be evaluated using any distributed query evaluation strategy. However, each round of rule application is followed by a round of updates: the ‘delta’ facts are added to the ‘old’ facts, and the ‘new’ facts are copied to the ‘delta’ facts and added to the ‘current’ facts. This can be a major source of overhead 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 the management and distribution of datasets 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.

This thesis addresses these problems by drawing inspiration from the work by Motik et al. [46] 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].

The distributed materialisation algorithm proposed in Chapter 4 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 3.1.3. We thus expect the same benefits as for the query answering algorithm by Potter et al. [53]. However, the lack of synchronisation between servers introduces a complication: since there is no notion of a rule application round (unlike, say, in Socialite), 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 derived; however, this would require maintaining a high coherence of server clocks in the cluster which is considered impractical. Section 4.1 discusses how this problem was solved using Lamport timestamps [37]—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. It is critical that all servers are updated before derived facts are used in rule applications; otherwise, the newly derived fact 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. This issue was solved using a well-known termination detection algorithm based on token passing [13].



# 4

## Distributed Materialisation Algorithm

### Contents

---

<b>4.1</b>	<b>Lamport Timestamps . . . . .</b>	<b>35</b>
<b>4.2</b>	<b>Occurrence Mappings . . . . .</b>	<b>38</b>
<b>4.3</b>	<b>Communication Infrastructure and Messages . . . . .</b>	<b>41</b>
<b>4.4</b>	<b>Termination Detection . . . . .</b>	<b>42</b>
<b>4.5</b>	<b>The Algorithm . . . . .</b>	<b>42</b>
<b>4.6</b>	<b>Proof of Correctness . . . . .</b>	<b>46</b>

---

This chapter presents the distributed materialisation algorithm. Section 4.1 discusses data structures used to store facts; Section 4.2 introduces the occurrence mappings; Section 4.3 discusses the communication infrastructure; Section 4.4 discusses the termination detection algorithm; Section 4.5 presents the algorithm’s pseudocode, and Section 4.6 presents the correctness proof. As mentioned in Chapter 1, the algorithm was designed for RDF stores, as is reflected in the details of the pseudocode, but it can be easily extended to suit deductive databases that store relations of arbitrary arity.

### 4.1 Lamport Timestamps

To prevent repetition of derivations, all derived facts need to be arranged into a sequence that reflects the derivation order. As mentioned already, maintaining

a precise global clock in a distributed system is very challenging, so Lamport timestamps are used instead, a general and simple technique for ordering events in a distributed system designed by *Lamport* [37]. 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 than the timestamp of  $B$ .

In other words, the Lamport timestamps are strictly increasing along chains of causally related events. To achieve this, each server keeps an integer counter called the *local clock* (even though it 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, each fact is associated with an integer timestamp.

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*  $T_k : G_k \rightarrow \mathbb{N}$  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

$$Q = a_1^{\boxtimes_1} \wedge \cdots \wedge a_n^{\boxtimes_n}, \quad (4.1)$$



**Figure 4.1:** Timestamps for the dataset  $Ex_1$ 

$T$					
$\langle p_1, cites, p_2 \rangle$	$\mapsto 1$	$\langle p_2, cites, p_3 \rangle$	$\mapsto 2$	$\langle p_1, inJournal, j_1 \rangle$	$\mapsto 0$
$\langle p_1, creator, a_1 \rangle$	$\mapsto 1$	$\langle p_2, creator, a_2 \rangle$	$\mapsto 2$	$\langle p_2, inConference, c_1 \rangle$	$\mapsto 0$
$\langle p_2, creator, a_1 \rangle$	$\mapsto 2$	$\langle p_3, creator, a_3 \rangle$	$\mapsto 0$	$\langle p_3, inConference, c_1 \rangle$	$\mapsto 0$

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  $\sigma$  is an answer to  $Q$  on a dataset  $G_k$  and function  $T_k$  w.r.t. an integer timestamp  $\tau$  if

- $\sigma$  is an answer to the ‘ordinary’ query  $a_1 \wedge \dots \wedge a_n$  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, recall the example dataset  $Ex_1$ ; let  $\tau = 2$ , the query  $Q$  as below, and let  $T$  be as in Figure 4.1.

$$Q = \langle x, cites, y \rangle^< \wedge \langle y, creator, z \rangle^{\leq}$$

Then,  $\sigma_1 = \{x \mapsto p_1, y \mapsto p_2, z \mapsto a_1\}$  is an answer to  $Q$  on  $Ex_1$  and  $T$  w.r.t.  $\tau$ , but  $\sigma_2 = \{x \mapsto p_2, y \mapsto p_3, z \mapsto a_3\}$  is not an answer to  $Q$  on  $G$  and  $T$  w.r.t.  $\tau$  due to  $T(\langle p_2, cites, p_3 \rangle) \geq 2$ .

We assume that each server can evaluate one annotated atom—that is, server  $k$  can call  $\text{EVALUATE}(a^{\bowtie}, \tau, G_k, T_k, \sigma)$ , where  $a^{\bowtie}$  is an annotated atom,  $\tau$  is a timestamp,  $G_k$  is the dataset stored in the server,  $T_k$  provides timestamps for the facts in  $G_k$ , and  $\sigma$  is a substitution. The call returns each substitution  $\rho$  defined over the variables in  $a$  and  $\sigma$  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,  $\text{EVALUATE}$  matches  $a^{\bowtie}$  on  $G_k$ ,  $T_k$ , and  $\tau$ , and it returns each match that extends  $\sigma$  and satisfies  $a^{\bowtie}$  and  $\tau$ . 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  $\tau$ .

Finally, each server is assumed to implement a function  $\text{MATCHRULES}(f, P)$  that takes as input a fact  $f$  and a Datalog program  $P$ . For each rule  $h \leftarrow b_1 \wedge \dots \wedge b_n$  in  $P$ , each body atom  $b_p$  with  $1 \leq p \leq n$ , and each substitution  $\sigma$  over the variables of  $b_p$  such that  $b_p\sigma = f$ , the function returns the 4-tuple  $(\sigma, b_p, Q, h)$  where

$$Q = b_1^< \wedge \dots \wedge b_{p-1}^< \wedge b_{p+1}^{\leq} \wedge \dots \wedge b_n^{\leq}. \quad (4.2)$$

Intuitively,  $\text{MATCHRULES}$  identifies each rule and each *pivot* body atom  $b_p$  that can be matched to  $f$  via substitution  $\sigma$ . This  $\sigma$  will be extended to all body atoms of the rule by recursively matching all remaining atoms using function  $\text{EVALUATE}$ . The annotations in (4.2) 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 (4.2) 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 thesis.

## 4.2 Occurrence Mappings

As in the query answering approach based on dynamic data exchange [53], 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 sever  $k$  to identify the servers on which constant  $c$  occurs as 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 reduce memory consumption, each server is allowed to only track locations of certain constants, as long as occurrence mappings are correct in the following sense.

**Definition 4.2.1.** *Let  $P$  be a program, let  $\mathbf{G} = G_1, \dots, G_\ell$  be datasets, and let  $\boldsymbol{\mu} = \mu_{1,s}, \mu_{1,p}, \mu_{1,o}, \dots, \mu_{\ell,s}, \mu_{\ell,p}, \mu_{\ell,o}$  be a collection of occurrence mappings.*

**Figure 4.2:** An example of correct occurrence mappings for the partition in Figure 2.3. Only non empty sets are shown. Note how  $\mu_{2,o}(p_3)$  includes server 1 despite  $p_3$  not appearing in  $Ex_{1,1}$ . Such 'enlargements' are not possible for subject occurrence mappings.

$\mu_{1,s}$	$\mu_{1,p}$	$\mu_{1,o}$
$p_1 \mapsto \{1\}$	$inJournal \mapsto \{1\}$	$j_1 \mapsto \{1\}$
$p_2 \mapsto \{2\}$	$creator \mapsto \{1, 2\}$	$a_1 \mapsto \{1, 2\}$
	$cites \mapsto \{1, 2\}$	$p_2 \mapsto \{1\}$
$\mu_{2,s}$	$\mu_{2,p}$	$\mu_{2,o}$
$p_2 \mapsto \{2\}$	$inConference \mapsto \{2\}$	$c_1 \mapsto \{2\}$
$p_3 \mapsto \{2\}$	$creator \mapsto \{1, 2\}$	$a_1 \mapsto \{1, 2\}$
	$cites \mapsto \{1, 2\}$	$a_3 \mapsto \{2\}$
		$p_3 \mapsto \{1, 2\}$

A constant  $c$  is relevant for  $P$ ,  $\mathbf{G}$ , 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  $\mu$  are consistent with  $P$  and  $\mathbf{G}$  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  $P$ ,  $\mathbf{G}$ , and  $k$  and that occurs at position  $\pi$  of a fact in  $G_j$ .

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

To simplify the presentation of Algorithm 1, each  $\mu_{k,\pi}$  is assumed to be 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. Furthermore, assume that mappings are accessed atomically: whenever  $\mu_{k,\pi}(c)$  is used in the algorithm, the result is the value of mapping  $\mu_{k,\pi}$  on  $c$  at a point in time.

Definition 4.2.1 places only a lower bound on occurrence mappings:  $j \in \mu_{k,\pi}(c)$  can hold even if  $G_j$  does not contain  $c$  at position  $\pi$ , as illustrated in Figure 4.2. In other words, if  $c$  is relevant for server  $k$  w.r.t.  $P$ , then all servers where  $c$  occurs at position  $\pi$  must be known to server  $k$ ; however, server  $k$  is not required to have precise knowledge about the location of  $c$ .

Note also that constant  $c$  is relevant to server  $k$  as long as it occurs in  $G_k$  at *some* position; however, regardless of the position(s) at which  $c$  occurs in  $G_k$ , sets  $\mu_{k,s}(c)$ ,  $\mu_{k,p}(c)$ , and  $\mu_{k,o}(c)$  must all be correctly defined. This is needed because matching a rule body atom can instantiate arbitrary positions in the rest of the rule body, so information about the the location of all matched constants may be needed to match the remaining atoms. A considerable source of complexity in Algorithm 1 is to ensure that all occurrence mappings remain correct as fresh facts are derived.

Finally, the second item of Definition 4.2.1 essentially requires that all facts with the same subject are assigned to one server. This is common in distributed RDF systems to ensure performance (see Chapter 5), but here it serves an additional purpose: when a fact is derived, it is sent to the server containing the fact's subject. If the subject is currently not located on any server, the fact's destination is determined by hashing the fact's subject.

Allowing servers track the location of relevant constants only introduces a complication: when server  $k$  receives a partial match  $\sigma$  from another server, the occurrence mappings stored in server  $k$  may not cover all constants in  $\sigma$ . Potter et al. [53] solve this by accompanying each partial match  $\sigma$  with a vector  $\lambda = \lambda_s, \lambda_p, \lambda_o$  of *partial occurrences*. Whenever a server extends  $\sigma$  by matching an atom, it also records in  $\lambda$  its local occurrences for each constant added to  $\sigma$  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.

### 4.3 Communication Infrastructure and Messages

In the presentation of the algorithm, assume that each server in the cluster can send a message  $m$  to a destination server  $d$  by calling  $\text{SEND}(m, d)$ . This function can return immediately, and the receiver can process the message later—that is, communication is assumed to be asynchronous. Also, Algorithm 1 is correct as long as each message that is sent is processed eventually—that is, no assumption are made about the order in which messages are processed. This section describes the three message types 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; this is discussed in more detail in Section 4.4.

Message  $\text{PAR}[i, \sigma, Q, h, \tau, \lambda]$  informs a server that  $\sigma$  is a partial match obtained by matching some fact with timestamp  $\tau$  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  $\sigma$  that may be needed when matching the remaining atoms of  $Q$  are recorded in  $\lambda$ .

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

Message  $\text{OCC}[f, D, k_h, \tau, \lambda]$  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  $\tau$  reflects when the message was sent, and  $\lambda$  records the partial occurrences of the constants in  $f$ .

Potter et al. [53] observed that  $\text{PAR}$  messages correspond to partial join results so a large number of such messages can be produced during query evaluation. For asynchronous processing, the  $\text{PAR}$  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 my approach as well, so I do not discuss it further.

## 4.4 Termination Detection

No server keeps track of progress of other servers, so detecting termination is not straightforward. This section summarises a well-known solution that can address this issue.

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 [13]. All servers in the cluster are numbered from 1 to  $\ell$  and are arranged in a ring (i.e., server 1 comes after server  $\ell$ ). 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 to a server  $j < i$ .
- All servers can terminate when server 1 receives a white token.

The Dijkstra–Scholten algorithm [14] can be used when there is no guarantee on the order of message delivery. This variant was not used in the implementation of Algorithm 1, so the details will not be discussed any further.

## 4.5 The Algorithm

With these definitions in mind, Algorithm 1 presents the distributed Datalog materialisation algorithm. Before starting, each server  $k$  receives the copy of the program  $P$  to be materialised, loads the corresponding partition element into its local

---

**Algorithm 1** Distributed Materialisation Algorithm at Server  $k$  in a cluster of  $\ell$  servers

---

```

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

7: function PROCESSFACT( $f, \tau$ )
8:   SYNCHRONISE( $\tau$ )
9:   for each  $(\sigma, a, Q, h) \in \text{MATCHRULES}(\tau, P)$  do
10:    FINISHMATCH( $0, \sigma, a, Q, h, \tau, \emptyset$ )

11: function PROCESSMESSAGE(PAR[ $i, \sigma, Q, h, \tau, \lambda$ ]) where  $Q = a_1^{\bowtie_1} \wedge \dots \wedge a_n^{\bowtie_n}$  and
     $\lambda = \lambda_s, \lambda_p, \lambda_o$ 
12:   SYNCHRONISE( $\tau$ )
13:   for each substitution  $\sigma' \in \text{EVALUATE}(a_i^{\bowtie_i}, \tau, G_k, T_k, \sigma)$  do
14:    FINISHMATCH( $i, \sigma', a_i, Q, h, \tau, \lambda$ )

15: function FINISHMATCH( $i, \sigma, a_{last}, Q, h, \tau, \lambda$ ) where  $Q = a_1^{\bowtie_1} \wedge \dots \wedge a_n^{\bowtie_n}$  and  $\lambda =$ 
     $\lambda_s, \lambda_p, \lambda_o$ 
16:   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
17:     if  $\lambda_\pi$  is undefined on  $x\sigma$  then Extend  $\lambda_\pi$  with the mapping  $x\sigma \mapsto \mu_{k,\pi}(x\sigma)$ 
18:   if  $i = n$  then
19:     for each position  $\pi \in \Pi$  and each constant  $c$  occurring in  $h$  do
20:       if  $\lambda_\pi$  is undefined on  $c$  then Extend  $\lambda_\pi$  with the mapping  $c \mapsto \mu_{k,\pi}(c)$ 
21:       if  $\lambda_s(h\sigma|_s) = \emptyset$  then  $k_h := h\sigma|_s \bmod \ell$ 
22:       else  $k_h :=$  the singleton element in  $\lambda_s(h\sigma|_s)$ 
23:       PROCESSORSENDMESSAGE(FCT[ $h\sigma, C_k, \lambda$ ],  $k_h$ )
24:   else
25:      $D :=$  the set of all servers
26:     for each position  $\pi \in \Pi$  such that  $a_{i+1}\sigma|_\pi$  is a constant  $c$  do  $D := D \cap \lambda_\pi(c)$ 
27:     for each  $d \in D$  do PROCESSORSENDMESSAGE(PAR[ $i + 1, \sigma, Q, h, \tau, \lambda$ ],  $d$ )

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

```

---

---

```

39: function PROCESSMESSAGE(OCC[ $f, D, k_h, \tau, \lambda$ ]) where  $\lambda = \lambda_s, \lambda_p, \lambda_o$ 
40:   SYNCHRONISE( $\tau$ )
41:   for each position  $\pi \in \Pi$  and each constant  $c$  in  $f$  do
42:     Atomically compute  $M := \mu_{k,\pi}(c) \setminus \lambda_\pi(c)$  and then add  $\lambda_\pi(c)$  to  $\mu_{k,\pi}(c)$ 
43:     Add  $M$  to both  $D$  and  $\lambda_\pi(c)$ 
44:   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  $C_k$ 
45:   else
46:     Remove an element  $d$  from  $D$  preferring any element over  $k_h$  if possible
47:     SEND(OCC[ $f, D, k_h, C_k, \lambda$ ],  $d$ )

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

50: function PROCESSORSENDMESSAGE( $m, d$ )
51:   if  $d = k$  then PROCESSMESSAGE( $m$ ) else SEND( $m, d$ )

```

---

store  $G_k$ , sets the timestamp of each fact in  $G_k$  to zero, initialises its occurrence mappings  $\mu_{k,s}, \mu_{k,p}$  to be correct for  $P$  and  $G_1, \dots, G_\ell$  as per Definition 4.2.1, 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 4.4.

The core of the 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  $f$ , Algorithm 1 calls MATCHRULES to identify all rules where one pivot atom matches to  $f$  via a partial answer  $\sigma$ , and it uses the FINISHMATCH function to extend  $\sigma$  to a full answer. Second, a PAR message can be received. The message contains a partial answer  $\sigma$ , 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.  $\tau$  to enumerate all partial answers  $\sigma'$ , and for each it uses the FINISHMATCH function to extend  $\sigma'$  to a full answer.

Function FINISHMATCH finishes matching atom  $a_{last}$  by (i) extending  $\lambda$  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  $\lambda_\pi$  for each position  $\pi$ .

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 4.2, 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 4.2.1. To this end, Algorithm 1 identifies in lines 31–36 the set of servers  $D$  whose occurrences might need updating. In particular, if  $f$  contains a constant  $c = f|_\pi$  position  $\pi$  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  $\pi$ , 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 at that point  $D$  is empty, 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 Algorithm 1—that is, the algorithm correctly computes the materialisation and exhibits the nonrepetition property. The proof is given in full in Section 4.6.

**Theorem 4.5.1.** *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.*

The algorithm shows clearly that it is not just intermediate results that are being communicated over the network, but also new derivations and occurrence updates (through FCT and OCC messages). Ensuring this communication does not become a bottleneck for the system is the main concern of Part III.

## 4.6 Proof of Correctness

Throughout this section, 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 4.5. The result of algorithm’s execution is assumed to be equivalent to a run where all lines on all servers are executed in some sequential order—that is, we can 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

$$L_{\pi}(c) = \{1, \dots, \ell\} \quad (4.3)$$

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

$$L_{\pi}(c) = \bigcap_{k \in R(c)} \mu_{k,\pi}(c), \quad (4.4)$$

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  $\pi$ , 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  $\text{process}^k(f)$  occurs after server  $k$  completes line 8 when processing a fact  $f$ .
- An event of type  $\text{par}^k(f, r, p, i)$  occurs after server  $k$  completes line 12 for a partial answer message with index  $i$  that originates from matching the  $p$ -th atom in the body of rule  $r$  to fact  $f$ .
- An event of type  $\text{fct}^k(f)$  occurs just before server  $k$  reaches line 37 while processing a FCT message for fact  $f$ .
- An event of type  $\text{occ}^k(f, c, \pi, j)$  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  $\text{add}^k(f)$  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  $\text{occ}^k(f, c, \pi, j)$ . Sometimes, we use a time instant index after the event type name to show both the event's instant and type; for example,  $\text{occ}_i^k(f, c, \pi, j)$  means that event at time instant  $i$  is of type  $\text{occ}^k(f, c, \pi, j)$ .

Note that, for each event  $\text{add}_i^k(f)$  in the algorithm's run, each constant  $c$  in  $f$ , and each position  $\pi \in \Pi$ , the run contains an event  $\text{occ}_{i'}^k(f, c, \pi, k)$  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  $\pi$ , fact  $f$  occurs eventually on server  $k$ , so event  $\text{occ}_i^k(f, c, \pi, k)$  occurs on server  $k$  for some  $i$ , and, for each event  $\text{occ}_j^k(g, c, \pi, k)$  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 4.5.1, which we split into several claims for clarity. First, we establish two properties that relate  $L(c)$  to Algorithm 1.

**Lemma 4.6.1.** *For each server  $k$ , each position  $\pi$ , 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  $\pi$  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  $L_\pi$ . 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  $\text{add}_i^k(f)$ , where  $f$  is a fact that contains  $c$ . Consequently, there exists an event  $\text{occ}_{i'}^k(f, c, \pi, k)$  in the run for some instant  $i'$  with  $i' < i$  and position  $\pi \in \Pi$ . Let  $\lambda_\pi$  be the partial occurrence mapping for position  $\pi$  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.  $\square$

**Lemma 4.6.2.** *For all servers  $k$  and  $j$ , each fact  $f$ , each constant  $c$ , and all position  $\pi$  and  $\pi'$  such that  $j \in L(c)$  and  $f$  introduces  $c$  at position  $\pi$  on server  $k$ , an event of type  $\text{occ}^j(f, c, \pi', k)$  occurs on server  $j$  during the run.*

*Proof.* Consider arbitrary  $k, j, f, c, \pi$ , and  $\pi'$  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  $\text{add}_i^k(f)$  occurs on server  $k$ . Because of the algorithm order, event  $\text{fct}_{i'}^k(f)$  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  $\pi$  on server  $k$ , so the algorithm executes lines 32-36 for position  $\pi$ . 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  $\lambda_s$ ,  $\lambda_p$ , and  $\lambda_o$  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 4.6.1 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  $\text{occ}^j(f, c, \pi', k)$  occurs on server  $j$ .  $\square$

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 4.6.3.** *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  $\mathbf{G} = G_1, \dots, G_\ell$  be the datasets and let  $\boldsymbol{\mu} = \mu_{1,s}, \mu_{1,p}, \mu_{1,o}, \dots, \mu_{\ell,s}, \mu_{\ell,p}, \mu_{\ell,o}$  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  $\pi$  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  $\pi$  on server  $j$ .

By the definition of  $L$ , we have  $k \in L(c)$ , so Lemma 4.6.2 ensures that an event of type  $\text{occ}^k(f, c, \pi, j)$  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  $\pi$ .* 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  $\pi$  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  $\pi'$  be a position of  $c$  in  $f'$  (i.e., we choose one  $\pi'$  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 4.6.2, there exist events  $\text{occ}_{i_1}^m(f, c, \pi, j)$  and  $\text{occ}_{i_2}^m(f', c, \pi', k)$  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  $\pi$  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  $\text{occ}_{i_3}^k(f', c, \pi', k)$  occurring on server  $k$  before the current point in time; clearly,  $i_2 < i_3$ . Let  $\lambda_s$ ,  $\lambda_p$ , and  $\lambda_o$  be the partial occurrence mappings in the OCC message for  $f'$  at point  $i_3$ . Then, event  $\text{occ}_{i_1}^m(f, c, \pi, j)$  ensures that  $j$  is present in the occurrence mappings of server  $m$  at instant  $i_1$ ; this occurs before event  $\text{occ}_{i_2}^m(f', c, \pi', k)$ , 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  $\pi$  on server  $j$  at the current time instant, event  $\text{occ}_{i_3}^j(f, c, \pi, j)$  occurring on server  $j$  introduces  $c$  at position  $\pi$  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  $\text{occ}_{i_2}^m(f', c, \pi', k)$  ensures that  $k$  is present in the occurrence mappings on server  $m$  at instant  $i_2$ ; this occurs before event  $\text{occ}_{i_1}^m(f, c, \pi, j)$ , and so line 43 ensures that  $k$  is added to the set  $D$ . Thus, an event of type  $\text{occ}^k(f, c, \pi, j)$  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  $\lambda_\pi$  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  $\pi$ .* 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  $\pi'$  and an instant before the current one such that at which  $\text{occ}_i^k(f, c, \pi', k)$  occurs on server  $k$ . Let  $\lambda_\pi$  be the partial occurrence mapping for position  $\pi$  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 4.6.1 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.  $\square$

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 4.6.4.** *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  $\text{occ}_i^k(f, c, s, j)$  occurs at time instant  $i$ . Let  $\mu_{k,s}$  be the occurrence mapping on server  $k$  at instant  $i - 1$ , and let  $\lambda_s$  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  $\lambda_s$  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 4.6.1 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.  $\square$

Lemma 4.6.4 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  $\text{add}^k(f)$  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  $\text{process}^k(f)$  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 Lamport timestamps capture the causality of fact derivation in this run—that is, chronological ‘happens-before’ relationship between events on facts agrees with the timestamps assigned to the facts.

**Lemma 4.6.5.** *Relationship  $T(f_1) < T(f_2)$  holds for events*

- $\text{par}_{i_1}^k(f_1, r, p, j)$  and  $\text{add}_{i_2}^k(f_2)$  such that  $i_1 < i_2$ ,

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

*Proof.* Consider events  $\text{par}_{i_1}^k(f_1, r, p, j)$  and  $\text{add}_{i_2}^k(f_2)$  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  $\text{add}^k(f_2)$  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  $\text{process}_{i_1}^k(f_1)$  and  $\text{occ}_{i_2}^k(f_2, c, \pi, j)$  such that  $i_1 < i_2$  and there exists no event  $\text{occ}_{i_3}^k(f_2, c, \pi, j)$  with  $i_3 < i_2$ . Clearly, the timestamp  $\tau$  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  $\text{par}_{i_1}^k(f_1, r, p, i)$  and  $\text{occ}_{i_2}^k(f_2, \pi, j)$  such that  $i_1 < i_2$  and there exists no event  $\text{occ}_{i_3}^k(f_2, c, \pi, j)$  with  $i_3 < i_2$  is analogous to above, but uses line 12 instead of line 8.  $\square$

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 4.5.1 by considering the soundness, completeness, and nonrepetition properties of the algorithm.

**Lemma 4.6.6.** *It is the case that  $G_1 \cup \dots \cup G_\ell \subseteq P^\infty(I)$ .*

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

**Lemma 4.6.7.** *It is the case that  $P^\infty(I) \subseteq G_1 \cup \dots \cup G_\ell$ .*

*Proof.* Let  $P^i(I)$  be the sets from the construction of  $P^\infty(I)$  as defined in Chapter 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  $r := h \leftarrow b_0 \wedge \dots \wedge b_n \in P$  and substitution  $\sigma$  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  $Q = a_1^{\bowtie_1} \wedge \dots \wedge a_n^{\bowtie_n}$  returned by  $\text{MATCHRULES}(b_p\sigma, P)$  in line 9 on fact  $b_p\sigma$ . Finally, for each  $0 \leq j \leq n$ , let  $\sigma_j$  be the substitution  $\sigma$  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 the function  $\text{FINISHMATCH}(j, \sigma_j, a_j, Q, h, \tau_0, \lambda_j)$  is called for some mapping  $\lambda_j$ .

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,  $\text{PROCESSFACT}(a_0\sigma, T_k(a_0\sigma))$  is called on server  $k_0$ , so a call to  $\text{MATCHRULES}(a_0\sigma, P)$  returns  $(\sigma_0, a_0, Q, h)$ , after which  $\text{FINISHMATCH}(0, \sigma_0, a_0, Q, h, \tau_0, \emptyset)$  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  $\text{par}_m^{k_{j+1}}(a_0\sigma, r, p, j + 1)$  occurs at some time instant  $m$ . Server  $k_{j+1}$  then calls  $\text{EVALUATE}$  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  $\text{add}^{k_{j+1}}(a_{j+1}\sigma)$  ever happens, then fact  $a_{j+1}\sigma$  occurs initially on server  $k_{j+1}$ .
- If an event  $\text{add}_{m'}^{k_{j+1}}(a_{j+1}\sigma)$  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  $\text{add}_{m'}^{k_{j+1}}(a_{j+1}\sigma)$  happens at time instant  $m'$  with  $m' > m$ . Then, Lemma 4.6.5 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 (4.2) ensures that  $\bowtie_{j+1} = \leq$ . Thus, the call to  $\text{EVALUATE}$  in line 13 on server  $k_{j+1}$  returns  $\sigma_{j+1}$ , so the call in line 14 ensures  $(\Diamond)$ .

To complete the proof, we assume that no event of type  $\text{par}^{k_{j+1}}(a_0\sigma, r, p, j + 1)$  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  $\pi^s$ ; hence, at that instant we have  $k_{j+1} \notin \mu_{k_s, \pi}(c)$ . Now if no event of type  $\text{add}^{k_{j+1}}(a_{j+1}\sigma)$  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 4.6.3 implies  $k_{j+1} \in \mu_{k_s, \pi}(c)$ , which is a contradiction. Consequently, an event of type  $\text{add}^{k_{j+1}}(a_{j+1}\sigma)$  occurs on server  $k_{j+1}$ . Moreover, let  $\alpha$  be the event type  $\text{process}^{k_s}(a_0\sigma)$  if  $s = 0$ , and the event type  $\text{par}^{k_s}(a_0\sigma, r, p, s)$  otherwise. By the induction assumption, property  $(\diamond)$  holds for  $s$ , so an event of type  $\alpha$  occurs on server  $k_s$ . Clearly, an event of type  $\alpha$  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  $\pi$  on server  $k_{j+1}$ ; such a fact exists because  $a_{j+1}\sigma$  contains  $c$  in position  $\pi$  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  $\pi'$  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 4.6.2, there exist time instants  $m_2$  and  $m_3$  such that  $\text{occ}_{m_2}^{k'}(g', c, \pi', k_s)$  and  $\text{occ}_{m_3}^{k'}(g, c, \pi, k_{j+1})$  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,  $k_{j+1} \in \mu_{k_s, \pi}(c)$  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  $\text{occ}_{m_2}^{k'}(g', c, \pi', k_s)$  then ensures that  $k_s \in \mu_{k', \pi}$  holds at instant  $m_2$ , and event  $\text{occ}_{m_3}^{k'}(g, c, \pi, k_{j+1})$  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  $\sigma$ , so the event of type  $\alpha$  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.  $\square$

**Lemma 4.6.8.** *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  $\sigma$ . 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.  $\square$





# Part III

## Data Partitioning



# 5

## Related Work and Technical Challenges

### Contents

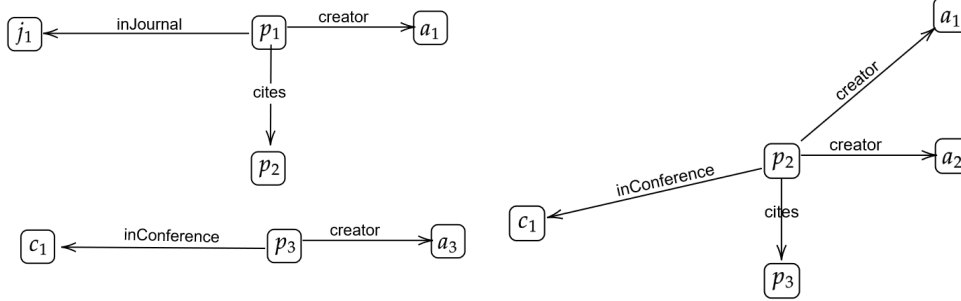
---

<b>5.1</b>	<b>Data Partitioning in RDF Stores . . . . .</b>	<b>64</b>
5.1.1	Hash-Based Schemes . . . . .	64
5.1.2	Vertical Partitioning Schemes . . . . .	65
5.1.3	Min-Cut Schemes . . . . .	65
5.1.4	Horizontal Containment vs Workload Balance . . . . .	66
<b>5.2</b>	<b>Streaming Graph Partitioning . . . . .</b>	<b>66</b>

---

The partitioning scheme applied to the input dataset affects materialisation in two major ways. First, network cycles are slower than CPU cycles, therefore sending messages can become a bottleneck to the system. Second, unbalanced partitions can cause the materialisation algorithm to engage the servers unevenly, with some servers computing more derivations, sending or receiving more messages than others.

This Chapter focuses on the question of how to effectively partition RDF data in a locality-aware way. Towards this goal, Section 5.1 reviews existing partitioning schemes in RDF stores and discusses the inherent technical challenges; Section 5.2 reviews schemes designed to partition undirected graphs in a *streaming* fashion—that is, without loading the entire dataset in memory, and how these can be applied to RDF data. Then, Chapters 6 and 7 present two new techniques that partition RDF data in a streaming fashion.

**Figure 5.1:** A hash partitioning of  $Ex_1$  in two elements; it has four overlapping constants.

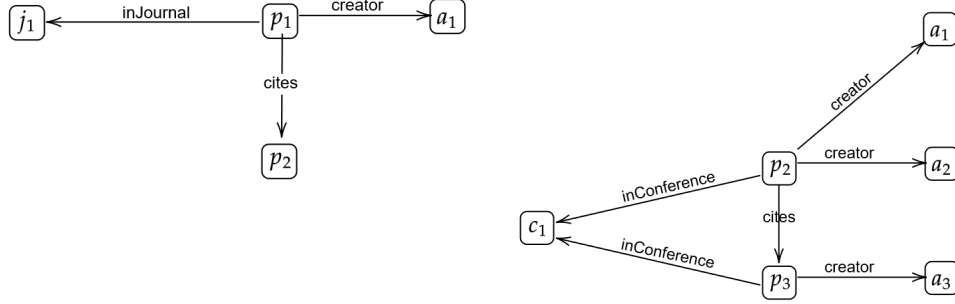
## 5.1 Data Partitioning in RDF Stores

Existing approaches to data partitioning in RDF systems can be broadly divided into three groups: hash-based, vertical partitioning, and min-cut partitioning. Such approaches are sometimes combined with data replication, where a fact is stored on more than one server. Since subject–subject joins were shown to be very common [18], nearly all schemes co-locate triples with the same subjects.

### 5.1.1 Hash-Based Schemes

The first 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 the subject) [72, 27]. In other words, a triple is sent to a server determined by its hash value modulo the number of servers in the cluster. The main benefit of such approaches is very low resource usage: 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. Hierarchical hashing partially addresses this gap: by hashing IRI prefixes in place of full IRI values, facts belonging to the same domain have a higher chance of being co-located. In Lee and Liu [39] the authors combine semantic hashing with hop-based fact replication to increase the share of queries that can be answered locally.

**Figure 5.2:** A min-cut partitioning of  $Ex_1$  in two elements; it has two overlapping constants, fewer than hash partitioning, but a larger size imbalance.



### 5.1.2 Vertical Partitioning Schemes

The second group consists of vertical partitioning schemes, inspired by relational databases, which partition the dataset into chunks of facts with the same predicate (see [35, 36]). Systems that are built on top of distributed computing frameworks such as Hadoop or Spark rely on the built-in partitioning schemes of their distributed file systems, which is usually a vertical partitioning scheme.

### 5.1.3 Min-Cut Schemes

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 [30, 24]. 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 resource 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;<sup>1</sup> 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, graph partitioning is NP-hard [10], so partitioning can take a considerable amount of time regardless of how it is realised.

<sup>1</sup><http://glaros.dtc.umn.edu/gkhome/home-of-metis>

### 5.1.4 Horizontal Containment vs Workload Balance

While it is intuitive to expect that partitioning the data carefully to minimise communication should improve the performance of distributed systems, this is not always the case, and the effects of data partitioning remain poorly understood. Janke et al. [32] studied this problem in the context of distributed query processing and implemented all the afore-mentioned partitioning schemes in an RDF management system whose query engine is independent of the partitioning scheme used. They evaluated queries with a varying number of joins and join patterns on datasets up to two billion triples sampled from a single source dataset, the Billion Triples Challenge 2014 Dataset<sup>2</sup>.

Interestingly, they concluded that reducing communication between servers (horizontal containment) can be detrimental if done at the expense of uneven server workload. However, their experiments are based on a single dataset and 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.

Thus, the questions of how to partition RDF data effectively, and how this affects distributed reasoning, are still largely open. To answer the former, we draw inspiration from recent work on *streaming graph partitioning*, which will be the focus of the next section.

## 5.2 Streaming Graph Partitioning

Streaming graph partitioning schemes were developed to cope with the growing size of modern graph datasets. Data analytics and network analysis tasks over such dataset need to operate on partitioned graphs, and can suffer from performance losses when the partitions cause large communication overheads. On the other hand,

---

<sup>2</sup><http://km.aifb.kit.edu/projects/btc-2014/>

using traditional partitioning schemes can incur in computation and communication costs as high as the task itself.

There are two kind of graph streams: vertex streams, where vertices are presented one at a time to the processor, together with all edges connecting with previously processed vertices, and edge stream, where edges are presented one at a time.

Stanton and Kliot [63] conducted a review of various schemes for vertex streams and found that overall a weighted greedy approach was superior. This scheme consider two contributions when evaluating the affinity of a vertex with a partition element, a positive contribution given by the number of adjacent vertices in that partition element, and a negative contribution given by the size of the partition element. Even though RDF streams are closer to edge streams, the main conclusions of that survey remain valid.

Streaming partitioning schemes such as [50, 73, 64, 42, 43] aim 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, resulting in a much smaller memory footprint for partitioning than with, say, METIS.

These approaches handle general (directed or undirected) graphs, so applying them straightforwardly to RDF is unlikely to be practical: facts with the same subject would not necessarily be placed on the same server, which, as we mentioned in Section 5.1, is critical. We adapt the HDRF [50] and 2PS [43] streaming graph partitioning algorithms to take the specific of RDF into account, but without compromising the quality of the resulting partitions.





# 6

## The HDRF<sub>3</sub> Algorithm

### Contents

---

<b>6.1</b>	<b>The Original HDRF Algorithm . . . . .</b>	<b>69</b>
<b>6.2</b>	<b>Adapting the Algorithm to RDF Graphs . . . . .</b>	<b>71</b>
<b>6.3</b>	<b>Proof of Proposition 6.2.1 . . . . .</b>	<b>73</b>

---

This chapter presents the HDRF<sub>3</sub> algorithm for streaming partitioning of RDF data, which adapts the ‘high degree replicated first’ principle from the HDRF algorithm for general graphs [50]. Section 6.1 briefly discusses the original idea, Section 6.2 adapts these principles to RDF and presents a principled way to choose the parameters of the algorithm, and Section 6.3 proves the main proposition in this chapter.

### 6.1 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 degree  $\deg(v)$  of  $v$  in the subgraph processed thus far, and the replication set  $A(v)$  for  $v$ . For each  $v$ , 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  $C(v, w, k)$ , and it increments  $N_k$ .

The score  $C(v, w, k)$  consists of two parts. The first one estimates the impact of placing  $\{v, w\}$  on server  $k$  would have on replication. It 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 server  $k'$ , and  $\deg(v) > \deg(w)$ . Then  $g(v, w, k) < g(w, v, k')$ , 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  $g(v, w, k)$  is used to normalise 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  $\lambda$  as

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

Thus,  $\lambda$ , 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. The authors show empirically that this does not substantially alter the quality of the partitions that the algorithm produces.

## 6.2 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 is addressed 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, the algorithm computes the destination for all facts with subject  $s$  the first time it sees such a fact.

The pseudo-code of HDRF<sub>3</sub> is shown in Algorithm 2. It takes as input a parameter  $\alpha$  determining the maximal acceptable imbalance in partition element sizes, the balance parameter  $\lambda$  as in HDRF, and another parameter  $\delta$  that will be described 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.

---

**Algorithm 2** HDRF<sub>3</sub>

---

**Require:** • the tolerance parameter  $\alpha > 1$   
• the balance parameter  $\lambda$   
• the degree imbalance parameter  $\delta$   
• the target number of servers  $\ell$   
•  $|G|$ ,  $|G^+(c)|$ , 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  $k \in \{1, \dots, \ell\}$

52: **function** PROCESSFACT( $s, p, o$ )  
53:   **if**  $T(s)$  is undefined **then**  
54:      $T(s) := \arg \max_{k \in \{1, \dots, \ell\}} \text{SCORE}(s, o, k)$   
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  $\text{DEG}(k) \leq \min_{\ell \in \{1, \dots, \ell\}} \text{DEG}(\ell) + \delta$  **then**  $C_{REP} := C_{REP} + 1 + \frac{|G(o)|}{|G(s)| + |G(o)|}$   
62:   **if**  $k \in A(o)$  and  $\text{DEG}(k) \leq \min_{\ell \in \{1, \dots, \ell\}} \text{DEG}(\ell) + \delta$  **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**  $C_k = 0 ? 0 : N_k / C_k$ 

---

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).

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 it encounters a constant  $s$ , it determines 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  $\delta$ ) 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  $\lambda$  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 mentioned in Section 2.4, 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  $\alpha$  and  $\lambda$  are chosen in a particular way. The proof is given in the next section.

**Proposition 6.2.1.** *Algorithm 2 produces a partition that satisfies  $|G_k| \leq \alpha \frac{|G|}{\ell}$  for each  $1 \leq k \leq \ell$  whenever  $\alpha$  and  $\lambda$  are selected such that*

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

### 6.3 Proof of Proposition 6.2.1

To prove Proposition 6.2.1, we need to reason about the state of the counters  $N_k$  from the HDRF<sub>3</sub> algorithm. Thus, in the rest of this appendix, we use  $N_k^i$  to refer to the value of  $N_k$  from Algorithm 2 after processing the  $i$ -th fact of  $G$ .

**Lemma 6.3.1.** *For  $\alpha > 1$  and  $\lambda > 0$ , in each run of Algorithm 2 on a dataset  $G$ ,*

$$\max_k N_k^i - \min_k N_k^i < M_\lambda \quad (6.1)$$

*holds after processing the  $i$ -th 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 (6.1) holds after the  $i$ -th 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  $N_k^{i+1} = N_k^i$  for each server  $k$ , so (6.1) clearly holds. Otherwise, let  $k_1$  and  $k_2$  be the servers such that  $N_{k_1}^{i+1}$  and  $N_{k_2}^{i+1}$  are minimal and maximal, respectively, among all  $N_k^{i+1}$  at step  $i + 1$ . If  $N_{k_2}^i$  is also maximal among all  $N_k^i$  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 (6.1) 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  $j \in \{1, 2\}$ :

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

For convenience, let  $S = \sum_k N_k^i$ . We can bound  $\text{SCORE}_1$  as follows:

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

Moreover, we can bound  $\text{SCORE}_2$  as follows using the fact that the definition of  $C_{\text{REP},2}$  clearly ensures  $C_{\text{REP},2} < 4$ :

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

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  $\text{SCORE}_1$  and  $\text{SCORE}_2$ , we make the following observation.

$$\frac{\lambda S}{|G|} \left( 1 - \ell^{\frac{N_{k_1}^i + |G^+(s_{i+1})|}{\alpha |G|}} \right) < 4 + \frac{\lambda S}{|G|} \left( 1 - \ell^{\frac{N_{k_2}^i + |G^+(s_{i+1})|}{\alpha |G|}} \right)$$

This can be rewritten as

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

which can further be rewritten as

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

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

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

This can be rewritten as follows.

$$(N_{k_2}^i - N_{k_1}^i)^2 < 4 \frac{\alpha |G|^2}{\ell \lambda}$$

$$N_{k_2}^i - N_{k_1}^i < |G| \sqrt{\frac{4\alpha}{\ell \lambda}}$$

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

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

Furthermore,  $N_{k_2}^i + |G^+(s_{i+1})| = N_{k_2}^{i+1}$  and the definition of  $M_\lambda$  ensure that this formula can be rewritten as

$$N_{k_2}^{i+1} < N_{k_1}^i + M_\lambda.$$

Finally,  $N_{k_1}^i = N_{k_1}^{i+1}$  holds since the fact is sent to server  $k_2$ , so the last observation proves our claim.  $\square$

**Proposition 6.2.1.** *Algorithm 2 produces a partition that satisfies  $|G_k| \leq \alpha \frac{|G|}{\ell}$  for each  $1 \leq k \leq \ell$  whenever  $\alpha$  and  $\lambda$  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  $\lambda$  be as stated in the proposition. Note that the condition on  $\alpha$  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.

$$\begin{aligned} \lambda &\geq \frac{4\alpha}{\ell \left( \frac{\alpha-1}{\ell} - \frac{\max_c |G^+(c)|}{|G|} \right)^2} \\ \sqrt{\frac{4\alpha}{\lambda \ell}} &\leq \frac{\alpha - 1}{\ell} - \frac{\max_c |G^+(c)|}{|G|} \\ |G| \sqrt{\frac{4\alpha}{\lambda \ell}} &\leq |G| \frac{\alpha - 1}{\ell} - \max_c |G^+(c)| \\ |G| \sqrt{\frac{4\alpha}{\lambda \ell}} + \max_c |G^+(c)| &\leq (\alpha - 1) \frac{|G|}{\ell} \end{aligned}$$

Using the definition of  $M_\lambda$  from Lemma 6.3.1, the last inequality can be rewritten as

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

Let  $\mathcal{P} = G_1, \dots, G_\ell$  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 (6.1) of Lemma 6.3.1 ensures  $|G_k| \leq |G|/\ell + M_\lambda$ , which together with the upper bound on  $M_\lambda$  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.  $\square$



# 7

## The 2PS<sub>3</sub> Algorithm

### Contents

---

<b>7.1</b>	<b>The Original 2PS Algorithm . . . . .</b>	<b>77</b>
<b>7.2</b>	<b>The Algorithm . . . . .</b>	<b>78</b>
<b>7.3</b>	<b>Proof of Proposition 7.2.1 . . . . .</b>	<b>80</b>

---

This chapter presents the 2PS<sub>3</sub> algorithm for RDF, which adapts the *two-phase streaming* algorithm 2PS [43]. Section 7.1 discusses the original idea, and Section 7.2 discusses the application of these principles to RDF.

### 7.1 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

---

**Algorithm 3** 2PS<sub>3</sub>

---

**Require:** • the tolerance parameter  $\alpha > 1$   
 • the target number of servers  $\ell$   
 •  $|G|$  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  $c_{max} := \arg \max_{c \in \{s, o\}} S(M(c))$ , 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  $k \in \{1, \dots, \ell\}$ 
75:   for each community  $m$  occurring in the image of the mapping  $M$  do
76:      $T(m) := \arg \min_{k \in \{1, \dots, \ell\}} |N_k|$ 
77:      $N_{T(m)} := N_{T(m)} + S(m)$ 

78: function PROCESSFACT-PHASE-II( $s, p, o$ )
79:   Add  $(s, p, o)$  to server  $T(M(s))$ 

```

---

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.

## 7.2 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<sub>3</sub> is shown in Algorithm 3. As in HDRF<sub>3</sub>, the algorithm uses a preprocessing phase to determine the size of dataset  $|G|$  and the out-degree  $|G^+(c)|$  of each constant  $c$ . Thus, the 2PS<sub>3</sub> algorithm actually uses three phases; however, to stress the relationship with the 2PS algorithm, the algorithm was called 2PS<sub>3</sub>.

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

The algorithm also maintains a global function that maps 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 the dataset  $G$  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, the requirement on the sizes of partition elements can be satisfied: if each community contains no more than  $(\alpha - 1) \frac{|G|}{\ell}$  facts, communities can later be assigned 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<sub>3</sub>, the 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  $\alpha$ . The proof is given in Appendix 7.3.

**Proposition 7.2.1.** *Algorithm 3 produces a partition that satisfies  $|G_k| \leq \alpha \frac{|G|}{\ell}$  for each  $1 \leq k \leq \ell$  whenever*

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

### 7.3 Proof of Proposition 7.2.1

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

$$S(m) = \sum_{c \text{ with } M(c)=m} |G^+(c)| \quad (7.1)$$

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:

$$\max_k N_k - \min_k N_k \leq (\alpha - 1) \frac{|G|}{\ell}. \quad (7.2)$$

For the induction base, all  $N_k$  are initialised to zero, so equation (7.2) holds after line 74. For the induction step, assume that equation (7.2) 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,  $\max_k N'_k = N_{k_1} + S(m)$ . Moreover, the requirement on the choice of  $\alpha$  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, (7.2) 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, equation (7.1) clearly ensures  $|G_k| = N_k$  for each  $k$ , which implies our claim.  $\square$



# Part IV

## Evaluation





# 8

## Evaluation

### Contents

---

<b>8.1 Datasets . . . . .</b>	<b>86</b>
<b>8.2 Data Partitioning Experiments . . . . .</b>	<b>87</b>
<b>8.3 Scalability Experiments . . . . .</b>	<b>92</b>
<b>8.4 System Comparison Experiments . . . . .</b>	<b>96</b>

---

To empirically evaluate the techniques presented in this paper, we implemented a prototype distributed Datalog reasoner called DMAT. RDFox, a state-of-the-art centralised RDF system, was used to store and index triples in RAM, on top of which we have implemented a mechanism for associating triples with timestamps. We implemented the EVALUATE function by first answering a query atom using the interfaces of RDFox and then filtering the answers with incorrect time stamps. 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. [53], and the HDRF<sub>3</sub> and 2PS<sub>3</sub> algorithms described in Chapters 6 and 7, respectively.

We evaluated DMAT by conducting three sets of experiments. First, we

investigated how using different partitioning schemes, including HDRF<sub>3</sub> and 2PS<sub>3</sub>, affects the performance of materialisation. Second, we studied how the performance of materialisation changes when the input data and the number of servers increase. Third, we compared DMAT with BigDatalog and Cog, two state-of-the-art systems that use static data exchange strategies.

We ran our experiments on the Amazon EC2 cloud. We used servers of the r5 family, each running Linux kernel 4.14 and equipped with a 2.3 GHz Intel Broadwell processor and connected by 10 Gbps Ethernet. The number of servers, the disk space, and the size of RAM varied across different tests, so we shall specify them for each test separately.

In the rest of this section, we present our experimental setting and discuss the results. In particular, we introduce our datasets in Section 8.1, and then we present the data partitioning tests in Section 8.2, the scalability tests in Section 8.3, and the system comparison tests in Section 8.4. Considering data partitioning first will allow us to identify 2PS<sub>3</sub> as the partitioning strategy that seems to offer the best performance on average, so we use 2PS<sub>3</sub> in the remaining two tests. The DMAT system and all datasets and programs used in our experiments are available online.<sup>1</sup>

## 8.1 Datasets

This section discusses the datasets we used in our experiments. Table 8.1 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.

LUBM has been extensively used to test RDF systems. We generated datasets of varying sizes using LUBM’s data generator. Moreover, we used the *extended lower bound* Datalog program by Motik et al. [46]. The program was constructed to stress-test reasoning systems, and it was obtained by translating the 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,

---

<sup>1</sup><https://krr-nas.cs.ox.ac.uk/2021/DMAT/>

**Table 8.1:** 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

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.

WatDiv was developed for benchmarking 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* [15] 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. To obtain a more manageable ‘real-world’ dataset that we call MAKG\*, we selected 3.67 billion core triples. The original 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.

## 8.2 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, this

section compares the HDRF<sub>3</sub> and 2PS<sub>3</sub> algorithms from Chapters 6 and 7 with subject hashing and the variant of min-cut partitioning by Potter et al. [53] based on METIS. The latter approach uses weighted graph partitioning to balance the number of triples, rather than the number of resources on different servers.

**Test Hardware.** This experiment required ten EC2 servers equipped with 128 GB of RAM for materialisation. Furthermore, to partition and distribute the data and the program, another identical ‘master’ server, which did not participate in the materialisation, was also used. Finally, METIS requires loading the input dataset into memory, so one more server with 784 GB of RAM was used for METIS partitioning.

**Test Setting.** This experiment uses the LUBM dataset for 10K universities, the WatDiv-1B dataset provided by the authors of WatDiv, and the MAKG\* dataset in this test; for each dataset, Table 8.2 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, all datasets were preprocessed by replacing all resources with integers. With Hash, HDRF<sub>3</sub>, or 2PS<sub>3</sub>, the ‘master’ server first processed the data in a streaming fashion and distributed the triples over the network, and then it started the materialisation by distributing the Datalog program. With METIS, the precomputed partition elements were loaded directly into the relevant servers, and the ‘master’ server just distributed the Datalog program. To hash the triples’ subjects, the integer subject value was multiplied by a large prime in order to randomise the distribution. In HDRF<sub>3</sub> and 2PS<sub>3</sub>,  $\alpha$  was set to 1.25. Also, in HDRF<sub>3</sub>,  $\delta$  was set to 0.25 and  $\lambda$  was set to the lowest value satisfying Proposition 6.2.1; the values of  $\lambda$  thus vary for each dataset and are shown in Table 8.2. The datasets were processed twice in the first phase of 2PS<sub>3</sub>. The wall-clock time and the number of PAR messages sent on each server during materialisation were recorded. For each test, Table 8.2 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 2.4

**Table 8.2:** Data Partitioning Experiments

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

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-1B, and on MAKG\* it ran out of memory even though the partitioning was conducted on a very large server equipped with 784 GB of RAM.

**Replication, Communication, and Reasoning.** Generally lowest replication factors were achieved with 2PS<sub>3</sub>: only METIS achieved a lower value on WatDiv-

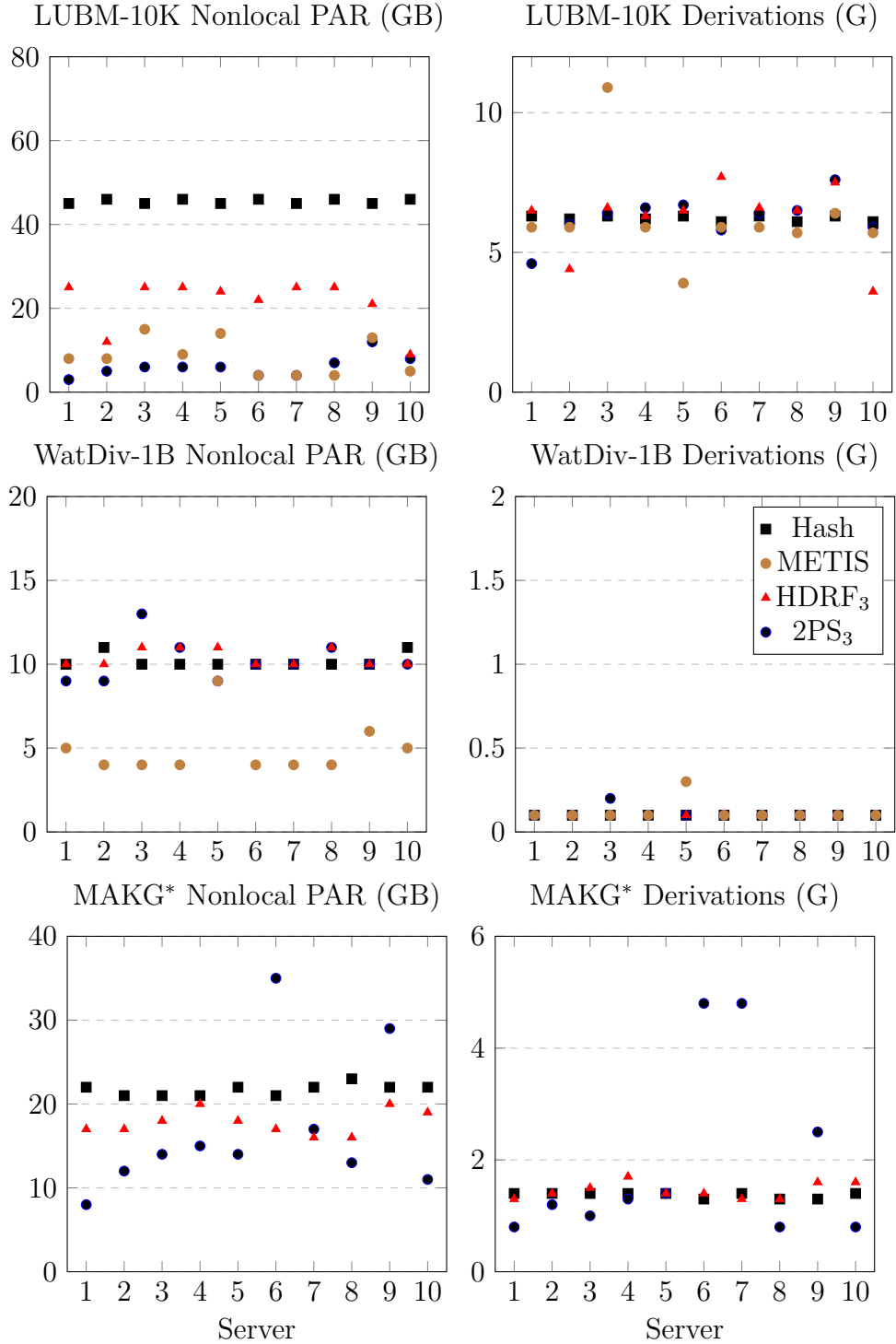
1B, and HDRF<sub>3</sub> achieved a comparable value on MAKG\*. The replication factor of Hash was highest in all cases, closely followed by HDRF<sub>3</sub>. Moreover, lower replication factors seem to correlate closely with decreased communication overhead; for example, the number of messages was significantly smaller on LUBM-10K and MAKG\* with 2PS<sub>3</sub> than with other schemes. This reduction seems to generally lead to a decrease in reasoning times: 2PS<sub>3</sub> 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-1B, METIS and 2PS<sub>3</sub> were slower than Hash and HDRF<sub>3</sub>, despite exhibiting smaller replication factors.

**Workload Imbalance.** To further analyse the results of our experiments, Figure 8.1 shows 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, partitioning the data into strongly connected clusters can introduce a workload imbalance: the numbers of derivations and messages per server are quite uniform for Hash and, to an extent, for HDRF<sub>3</sub>; in contrast, with 2PS<sub>3</sub> and METIS, certain servers seem to be doing much more work than others, particularly on WatDiv-1B and MAKG\*. Thus, reducing communication seems to be important, but only up to a point. For example, 2PS<sub>3</sub> reduces communication drastically on LUBM-10K, and this seems to ‘pay off’ in terms of reasoning times. On MAKG\*, the reduction in communication seems to lead to modest improvements in reasoning times, despite a more pronounced workload imbalance. On WatDiv-1B, however, communication overhead does not appear to be significant with any partitioning strategy, so reasoning times seem to be determined mainly by the workload imbalance.

**Overall Performance.** In general, 2PS<sub>3</sub> 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; and, while this can increase reasoning times due to workload imbalances, such increases do not appear to be excessive. Thus, 2PS<sub>3</sub> seems like a good alternative to the

thus far dominant hash partitioning, and therefore it was used in our scalability (Section 8.3) and system comparison experiments (Section 8.4).

**Figure 8.1:** Input Partitioning Experiments



## 8.3 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  $\langle a_1, R, a_2 \rangle, \langle a_2, R, a_3 \rangle, \dots, \langle a_n, R, a_1 \rangle$  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. This section analysis 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, elimination of duplicate facts can be implemented in  $O(1)$  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 computed exactly once.

In addition, the number of PAR messages produced during materialisation will also be considered. 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 they can be scaled 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 it was not considered in this experiment. The hardware configuration and test setting are identical to those in Section 8.2, but the cluster size was scaled proportionally to the input size. All experiments were conducted using the 2PS<sub>3</sub> partitioning strategy.

**Results.** The results of the scalability experiments are summarised in Table 8.3. For each dataset, the table reports 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. This is because the WatDiv data generator was specifically designed to produce datasets with nonuniform connection patterns. As a result, the derivation rate drops significantly as the input

increases; moreover, the reasoning rate drops as well, but not as steeply as the derivation rate. Specifically, the derivation and reasoning rates are about 3.6 and 2.1 times, respectively, lower on the 1 G dataset than on the 200 M dataset.

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.

**Table 8.3:** Scalability Experiments

Data Size	Servers	Res. (M)	Triples (M)		Mat. Time (s)	Derivations		PAR messages		Reasoning Rate (k/ss)
			Input	Output		Total (G)	Rate (k/ss)	Total (G)	Local (%)	
LUBM										
2K	2	66	267	664	4 803	15.85	1 620	1.43	99	1 800
4K	4	131	534	1 329	4 457	31.73	1 750	2.89	99	1 942
8K	8	262	1 068	2 658	5 275	63.45	1 480	6.90	99	1 667
10K	10	328	1 335	3 322	5 198	79.30	1 500	7.96	99	1 679
WatDiv										
200 M	2	20	218	342	470	0.43	468	0.93	75	1 447
400 M	4	40	436	649	810	0.81	250	2.10	46	898
800 M	8	79	873	1 278	1 285	1.66	160	6.35	32	780
1 G	10	97	1 092	1 749	1 636	2.10	128	9.00	28	678

## 8.4 System Comparison Experiments

To see how DMAT compares to the state of the art, we compared it to BigDatalog [61] and Cog [31], 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,<sup>2</sup> 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 already mentioned in Section 3.2, BigDatalog and Cog are not classical materialisation systems; rather, they take as input a query and then materialise only a portion 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 the standard relational data model, rather than the RDF model. We used the well-known *vertical partitioning* technique [1] to transform triples into relations: for each property  $p$  occurring in the input dataset or a rule, we introduced a binary relation  $p$ ; 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 either 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 8.4 shows the numbers of input and output triples in the reduced datasets.

---

<sup>2</sup><https://calcite.apache.org>

**Table 8.4:** Systems Comparison Experiments

BigDatalog		Cog		DMAT				
Time (s)	TX	Time (s)	TX	Time (s)	TX	Der (G)	PAR (G)	Local (%)
<b>LUBM</b>								
[281 M input and 454 output triples]								
1 107	90	1 441	110	325	0	2.71	7.90	99
<b>WatDiv</b>								
[339 M input and 404 output triples]								
1 114	68	1 050	80	920	965	0.41	10.82	12
<b>MAKG</b>								
[1784 M input and 2665 output triples]								
oom	—	error	—	4 081	290	6.37	16.02	84

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

**Test Hardware.** Just like in Section 8.2, we computed the materialisation on ten EC2 servers equipped with 128 GB of RAM, and we used an additional identical ‘master’ server. For BigDatalog and Cog, we provided each server with 1 TB of disk space. DMAT stores all data in RAM and does not use the disk during materialisation, so disk space is irrelevant.

**Test Setting.** We conducted our experiments as follows. For BigDatalog and Cog, we copied the input datasets into the local directories of all servers, we instructed the systems to answer the query (which involved materialising the rules), and we measured the wall-clock time required. For DMAT, the ‘master’ partitioned the datasets using the 2PS<sub>3</sub> algorithm and streamed the triples, distributed the two-rule program, and measured the wall-clock time of the materialisation. In all tests, we measured the total amount of data sent over the network using the `ifconfig` command.

**Results.** Table 8.4 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 outperforms both systems. The difference is minor 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 disk space 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 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, as long as they can be evaluated with left-deep join plans.

Our technique seems to be most effective when data is partitioned in a way that eliminates most communication. As we already mentioned, LUBM can be partitioned easily, as evidenced by the fact that most PAR messages are local and that materialisation incurs very little communication. In contrast, data partition in BigDatalog and Cog is not locality-aware, which incurs much more communication and reasoning times. On MAKG\*, our techniques also seem to be very effective at reducing communication. In contrast, they do not seem to be as effective on WatDiv: only 12% of PAR messages are local, which leads to an order of magnitude more communication than with the other two systems. Nevertheless, DMAT is still slightly faster.

# 9

## Conclusion and Further Work

This thesis has presented a novel approach to Datalog reasoning in distributed RDF systems. The materialisation algorithm can reason over arbitrary Datalog rules and arbitrary partitions of the input dataset, without repeating derivations. No other system was found to exhibit all these traits, and thus making it a good candidate for applications in the field of semantic web technologies.

The thesis has also presented two new streaming partitioning algorithms that enable the materialisation algorithm to scale out to very large datasets. The experiments conducted indicate that the combined use of the materialisation and partitioning algorithms here presented can considerably reduce communication, scale well in many cases, and is competitive with regard to the state of the art.

In our future work, we will aim to further improve the materialisation performance by developing ways to reduce imbalances in the workload among servers. One possibility might be to analyse the Datalog program before partitioning to take the program structure into account when applying the partitioning scheme. This can be combined with sampling or statistical analysis of the dataset to 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.



# Bibliography

- [1] Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K., 2007. Scalable Semantic Web Data Management Using Vertical Partitioning, in: Proc. of the 33rd Int. Conf. on Very Large Data Bases (VLDB 2007), VLDB Endowment, Vienna, Austria. pp. 411–422.
- [2] Abdelaziz, I., Harbi, R., Khayyat, Z., Kalnis, P., 2017. A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data. PVLDB 10, 2049–2060.
- [3] Abiteboul, S., Bienvenu, M., Galland, A., Rousset, M., 2010. Distributed datalog revisited, in: Datalog.
- [4] Abiteboul, S., Hull, R., Vianu, V., 1995. Foundations of Databases. Addison Wesley.
- [5] Al-Harbi, R., Abdelaziz, I., Kalnis, P., Mamoulis, N., Ebrahim, Y., Sahli, M., 2016. Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. VLDB Journal 25, 355–380.
- [6] Bellomarini, L., Benedetto, D., Gottlob, G., Sallinger, E., 2020. Vadalog: A modern architecture for automated reasoning with large knowledge graphs. Information Systems , 101528.
- [7] Berners-Lee, T., Hendler, J., Lassila, O., 2001. The Semantic Web. Scientific American .
- [8] Bernstein, P., Goodman, N., Wong, E., Reeve, C., Jr, J., 1981. Query processing in a system for distributed databases (sdd-1). ACM Transactions on Database Systems - TODS 6, 602–625. doi:10.1145/319628.319650.
- [9] Bernstein, P.A., Chiu, D.M.W., 1981. Using semi-joins to solve relational queries. J. ACM 28, 25–40. URL: <https://doi.org/10.1145/322234.322238>, doi:10.1145/322234.322238.
- [10] Bui, T.N., Jones, C., 1992. Finding Good Approximate Vertex and Edge Partitions is NP-Hard. Inf. Process. Lett. 42, 153–159.
- [11] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K., 2015. Apache flink™: Stream and batch processing in a single engine. IEEE Data Eng. Bull. 38, 28–38.
- [12] Chin, B., von Dincklage, D., Ercegovac, V., Hawkins, P., Miller, M.S., Och, F.J., Olston, C., Pereira, F., 2015. Yedalog: Exploring Knowledge at Scale, in: Proc. of the 1st Summit on Advances in Programming Languages (SNAPL 2015), Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Asilomar, CA, USA. pp. 63–78.

- [13] Dijkstra, E., Feijen, W., van Gasteren, A., 1983. Derivation of a Termination Detection Algorithm for Distributed Computations. *Inf. Process. Lett.* 16, 217–219.
- [14] Dijkstra, E.W., Scholten, C., 1980. Termination detection for diffusing computations. *Information Processing Letters* 11, 1–4.
- [15] Färber, M., 2019. The Microsoft Academic Knowledge Graph: A Linked Data Source with 8 Billion Triples of Scholarly Data, in: *ISWC*, pp. 113–129.
- [16] Galárraga, L., Hose, K., Schenkel, R., 2014. Partout: a distributed engine for efficient RDF processing, in: *WWW*, pp. 267–268.
- [17] Gallaire, H., Minker, J., 1978. *Logic and Data Bases*. Springer, Boston, MA.
- [18] Gallego, M.A., Fernández, J.D., Martínez-Prieto, M.A., de la Fuente, P., 2011. An Empirical Study of Real-World SPARQL Queries. *CoRR* abs/1103.5043.
- [19] Ganguly, S., Silberschatz, A., Tsur, S., 1992. Parallel Bottom-Up Processing of Datalog Queries. *Journal of Logic Programming* 14, 101–126.
- [20] Graefe, G., Davison, D.L., 1993. Encapsulation of Parallelism and Architecture-Independence in Extensible Database Query Execution. *IEEE Trans. Software Eng.* 19, 749–764.
- [21] Grosz, B.N., Horrocks, I., Volz, R., Decker, S., 2003. Description Logic Programs: Combining Logic Programs with Description Logic, in: *Proc. of the 12th Int. World Wide Web Conference (WWW 2003)*, ACM Press, Budapest, Hungary. pp. 48–57.
- [22] Gu, R., Wang, S., Wang, F., Yuan, C., Huang, Y., 2015. Cichlid: Efficient Large Scale RDFS/OWL Reasoning with Spark, in: *Proc. of the 2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2015)*, IEEE Computer Society, Hyderabad, India. pp. 700–709.
- [23] Gurajada, S., Seufert, S., Miliaraki, I., Theobald, M., 2014a. TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing, in: *SIGMOD*, pp. 289–300.
- [24] Gurajada, S., Seufert, S., Miliaraki, I., Theobald, M., 2014b. TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing, in: *SIGMOD*, pp. 289–300.
- [25] Hammoud, M., Rabbou, D.A., Nouri, R., Beheshti, S., Sakr, S., 2015. DREAM: Distributed RDF Engine with Adaptive Query Planner and Minimal Communication. *PVLDB* 8, 654–665.
- [26] Harris, S., Lamb, N., Shadbol, N., 2009. 4store: The Design and Implementation of a Clustered RDF Store, in: *Proc. of the 5th Int. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, Washington DC, USA. pp. 94–109.
- [27] Harth, A., Umbrich, J., Hogan, A., Decker, S., 2007. YARS2: A Federated Repository for Querying Graph Structured Data from the Web, in: *ISWC*, pp. 211–224.

- [28] Hose, K., Schenkel, R., 2013. WARP: Workload-aware replication and partitioning for RDF, in: Workshop at ICDE, pp. 1–6.
- [29] Huang, J., Abadi, D.J., Ren, K., 2011a. Scalable SPARQL Querying of Large RDF Graphs. PVLDB 4, 1123–1134.
- [30] Huang, J., Abadi, D.J., Ren, K., 2011b. Scalable SPARQL Querying of Large RDF Graphs. PVLDB 4, 1123–1134.
- [31] Imran, M., Gévay, G.E., Markl, V., 2020. Distributed graph analytics with datalog queries in flink, in: Proc. of the 4th Int. Workshop on Software Foundations for Data Interoperability and Large Scale Graph Data Analytics (SFDI 2020), Springer, Tokyo, Japan. pp. 70–83.
- [32] Janke, D., Staab, S., Thimm, M., 2018. Impact analysis of data placement strategies on query efforts in distributed RDF stores. Journal of Web Semantics 50, 21–48.
- [33] Kaoudi, Z., Miliaraki, I., Koubarakis, M., 2008. RDFS Reasoning and Query Answering on Top of DHTs, in: Proc. of the 7th Int. Semantic Web Conf. (ISWC 2008), Springer, Karlsruhe, Germany. pp. 499–516.
- [34] Karypis, G., Kumar, V., Comput, S., 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing 20.
- [35] Khadilkar, V., Kantarcioglu, M., Thuraisingham, B., Castagna, P., 2012a. Jena-hbase: A distributed, scalable and efficient rdf triple store.
- [36] Khadilkar, V., Kantarcioglu, M., Thuraisingham, B., Castagna, P., 2012b. Jena-hbase: A distributed, scalable and efficient rdf triple store, in: International Semantic Web Conference.
- [37] Lamport, L., 1978. Time, Clocks, and the Ordering of Events in a Distributed System. CACM 21, 558–565.
- [38] Lee, K., Liu, L., 2013a. Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. PVLDB 6, 1894–1905.
- [39] Lee, K., Liu, L., 2013b. Scaling queries over big rdf graphs with semantic hash partitioning. Proc. VLDB Endow. 6, 1894–1905. URL: <https://doi.org/10.14778/2556549.2556571>, doi:10.14778/2556549.2556571.
- [40] Liu, Y., McBrien, P., 2017. 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. pp. 3:1–3:10.
- [41] Luteberget, B., Johansen, C., Steffen, M., 2016. Rule-Based Consistency Checking of Railway Infrastructure Designs, in: Proc. of the 12th Int. Conf. on Integrated Formal Methods (IFM 2016), Springer. pp. 491–507.
- [42] Mayer, C., Mayer, R., Tariq, M.A., Geppert, H., Laich, L., Rieger, L., Rothermel, K., 2018. ADWISE: Adaptive Window-Based Streaming Edge Partitioning for High-Speed Graph Processing, in: ICDCS, pp. 685–695.

- [43] Mayer, R., Orujzade, K., Jacobsen, H., 2020. 2ps: High-quality edge partitioning with two-phase streaming. CoRR abs/2001.07086.
- [44] Miller, M.S., Dincklage, D.V., Ercegovac, V., Chin, B., 2017. Uncanny valleys in declarative language design, in: SNAPL.
- [45] Motik, B., Grau, B.C., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C., 2009. OWL 2 Web Ontology Language: Profiles. W3C Recommendation.
- [46] Motik, B., Nenov, Y., Piro, R., Horrocks, I., Olteanu, D., 2014. Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems, in: AAAI, pp. 129–137.
- [47] Mullin, J., 1990. Optimal semijoins for distributed database systems. IEEE Transactions on Software Engineering 16, 558–560. doi:10.1109/32.52778.
- [48] Naacke, H., Amann, B., Curé, O., 2017. Sparql graph pattern processing with apache spark. Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems .
- [49] Nenov, Y., Piro, R., Motik, B., Horrocks, I., Wu, Z., Banerjee, J., 2015. RDFox: A Highly-Scalable RDF Store, in: Proc. of the 14th Int. Semantic Web Conf. (ISWC 2015), Springer, Bethlehem, PA, USA. pp. 3–20.
- [50] Petroni, F., Querzoni, L., Daudjee, K., Kamali, S., Iacoboni, G., 2015. HDRF: Stream-Based Partitioning for Power-Law Graphs, in: CIKM, pp. 243–252.
- [51] Piro, R., Nenov, Y., Motik, B., Horrocks, I., Hendler, P., Kimberly, S., Rossman, M., 2016. Semantic Technologies for Data Analysis in Health Care, in: Proc. of the 15th Int. Semantic Web Conf., Part II (ISWC 2016), Springer. pp. 400–417.
- [52] Polychroniou, O., Sen, R., Ross, K.A., 2014. Track join: distributed joins with minimal network traffic. Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data .
- [53] Potter, A., Motik, B., Nenov, Y., Horrocks, I., 2018. Dynamic Data Exchange in Distributed RDF Stores. IEEE TKDE 30, 2312–2325.
- [54] Rohloff, K., Schantz, R.E., 2011. Clause-Iteration with MapReduce to Scalably Query Data Graphs in the SHARD Graph-Store, in: DIDC, pp. 35–44.
- [55] Schätzle, A., Zablocki, M., Lausen, G., 2011. Pigsparql: mapping sparql to pig latin. doi:10.1145/1999299.1999303.
- [56] Schätzle, A., Zablocki, M., Neu, A., Lausen, G., 2014. Sempala: Interactive sparql query processing on hadoop. doi:10.1007/978-3-319-11964-9\_11.
- [57] Schätzle, A., Zablocki, M., Skilevic, S., Lausen, G., 2015. S2rdf: Rdf querying with sparql on spark. Proceedings of the VLDB Endowment 9. doi:10.14778/2977797.2977806.

- [58] Seib, J., Lausen, G., 1991. Parallelizing Datalog Programs by Generalized Pivoting, in: Proc. of the 10th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1991), ACM Press, Denver, CO, USA. pp. 241–251.
- [59] Seo, J., Park, J., Shin, J., Lam, M.S., 2013. Distributed Socialite: A Datalog-Based Language for Large-Scale Graph Analysis. PVLDB 6, 1906–1917.
- [60] Shao, J., Bell, D.A., Hull, M.E.C., 1991. 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), IEEE Computer Society, Miami Beach, FL, USA. pp. 106–115.
- [61] Shkapsky, A., Yang, M., Interlandi, M., Chiu, H., Condie, T., Zaniolo, C., 2016. 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. pp. 1135–1149.
- [62] Shvachko, K., Kuang, H., Radia, S., Chansler, R., 2010. The hadoop distributed file system, in: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–10. doi:10.1109/MSST.2010.5496972.
- [63] Stanton, I., Kliot, G., 2012. Streaming graph partitioning for large distributed graphs, in: KDD, pp. 1222–1230.
- [64] Taimouri, M., Saadatfar, H., 2019. Rbsep: a reassignment and buffer based streaming edge partitioning approach. Journal of Big Data 6.
- [65] ter Horst, H.J., 2005. Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. Journal of Web Semantics 3, 79–115.
- [66] Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.E., 2012. WebPIE: A Web-scale Parallel Inference Engine using MapReduce. Journal of Web Semantics 10, 59–75.
- [67] Weaver, J., Hendler, J.A., 2009. Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples, in: Proc. of the 8th Int. Semantic Web Conf. (ISWC 2009), Springer, Chantilly, VA, USA. pp. 682–697.
- [68] Wolfson, O., Ozeri, A., 1993. Parallel and Distributed Processing of Rules by Data-Reduction. IEEE Transactions on Knowledge and Data Engineering 5, 523–530.
- [69] Wu, B., Zhou, Y., Yuan, P., Jin, H., Liu, L., 2014. SemStore: A Semantic-Preserving Distributed RDF Triple Store, in: CIKM, pp. 509–518.
- [70] Wu, H., Liu, J., Wang, T., Ye, D., Wei, J., Zhong, H., 2016. Parallel Materialization of Datalog Programs with Spark for Scalable Reasoning, in: Proc. of the 17th Int. Conf. on Web Information Systems Engineering (WISE 2016), Shanghai, China. pp. 363–379.
- [71] Zaharia, M., Chowdhury, M., Franklin, M., Shenker, S., Stoica, I., 2010. Spark: Cluster computing with working sets, in: HotCloud.

- [72] Zeng, K., Yang, J., Wang, H., hao, B., Wang, Z., 2013. A Distributed Graph Engine for Web Scale RDF Data. PVLDB 6, 265–276.
- [73] Zhang, W., Chen, Y., Dai, D., 2018. AKIN: A Streaming Graph Partitioning Algorithm for Distributed Graph Storage Systems, in: CCGRID, pp. 183–192.
- [74] Zhang, W., Wang, K., Chau, S.C., 1995. Data Partition and Parallel Evaluation of Datalog Programs. IEEE Transactions on Knowledge and Data Engineering 7, 163–176.