Optimising Datalog Materialisations

Xinyue Zhang
Keble College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy
Hilary 2025

Acknowledgements

The past four years of my DPhil have been incredibly fulfilling and enjoyable, and I am deeply grateful to all those who have supported me throughout this journey.

First and foremost, I would like to express my sincere gratitude to my supervisors, Prof. Ian Horrocks and Prof. Pan Hu, for their invaluable guidance. Their insights have shaped my research, from high-level discussions on research directions to refining the intricate details of algorithms. I am especially grateful to Dr. Yavor Nenov, who was always patient and enthusiastic in discussing technical details, never hesitating to go through even the smallest implementation aspects with me.

I would also like to thank Professor Michael Benedikt for assessing my transfer and confirmation, and for providing invaluable feedback and perspectives from the database field, which have been incredibly enlightening. I am grateful to Dr. Matthias Lanzinger, who assessed my transfer, and Dr. Przemysław Wałęga, who assessed my confirmation - both of whom offered crucial suggestions at key milestones in my PhD. I would like to thank my examiners, Prof. Michael Benedikt and Prof. Thomas Eiter, for carefully reading my thesis and for their valuable comments and suggestions, which have helped improve this work.

My appreciation extends to the other professors in the KRR group. Prof. Bernardo Cuenca Grau provided valuable feedback during my seminar presentations, and Prof. Boris Motik gave me the opportunity to serve as a teaching assistant for the Database Management Implementation course, which greatly enriched my learning experience. I am also thankful to my fellow lab members, Jiaoyan Chen, Shuwen Liu, Yuan He, Dingmin Wang, and Jingchuan Shi, for their support and stimulating discussions.

I am also grateful for the inspiring lecture slides and academic works of Prof. Georg Gottlob and Prof. Dan Olteanu from Oxford, Prof. Moshe Vardi from Rice University, Prof. Paris Koutris from the University of Wisconsin-Madison, as well as Prof. Angela Bonifati from Lyon 1 University. Their materials are exceptionally well-structured and have significantly influenced my PhD studies.

Beyond academia, I would like to express my deepest gratitude to my family. Thanks to my parents, Xuehong and Rongrong, for their unconditional love and unwavering support in all my decisions. To my sister, Xinrao, thank you for always believing in me and standing by my side.

A special thanks to my partner, Bingchen, whose constant support and companionship have made this PhD journey not only unique but also truly enjoyable. Pursuing our PhDs together has been a remarkable experience, filled with shared

challenges, discoveries, and personal growth. I am grateful for your patience, encouragement, and unwavering belief in me, even during times of doubt. As we approach the next chapter, I sincerely hope for your successful graduation and look forward to celebrating all that we have achieved, together.

A warm and heartfelt thanks to my two cats, Pami and Tomi, for coming into my life and bringing warmth, comfort, and unexpected joy during the intense days of viva preparation and my transition from student to professional life.

To my dear friends, Zhengyuan, Yuwen, Shanshan, and Danni, your companionship has made my time in Oxford all the more memorable. Thank you for sharing laughter, encouragement, and countless moments that have made this journey special.

Finally, I would like to conclude with a lyric that both Bingchen and I love:

"It's something unpredictable, but in the end, it's right.
I hope you had the time of your life."

Abstract

Datalog is a prominent knowledge representation language widely used for declarative reasoning, primarily due to its ability to express recursive definitions. Many Datalog-based reasoning systems employ materialisation techniques to precompute all consequences of a given program and dataset, enabling efficient query answering. However, materialisation-based reasoning faces several challenges concerning computational efficiency and storage management for both the initial computation and incremental maintenance of materialisations. Most existing approaches default to general-purpose update strategies without considering optimisations for specific rule structures. From a computational perspective, standard materialisation techniques rely on traditional join plans for rule evaluation, which can be suboptimal for cyclic rules. The redundancy introduced by these plans leads to inefficiencies, especially in recursive reasoning. To address this, we explore the use of hypertree decomposition (HD)-based evaluation techniques to optimise the reasoning process for cyclic rules, including both the initial computation and incremental maintenance. We develop specialised addition, deletion, and rederivation procedures using hypertree decomposition techniques for cyclic rules, which support efficient incremental evaluation and are suitable for integration into the recursive reasoning scenario and incremental maintenance processes. Storage management is another critical challenge, as Datalog rules can significantly expand datasets, making it impractical to store all derived facts explicitly. To mitigate this, we propose a multi-scheme storage framework that integrates customised storage techniques tailored to particular rule derivations into the initial computation and incremental maintenance processes. Specifically, we introduce specialised storage schemes for transitive closure and union rules, allowing compact representation and efficient retrieval while preserving the correctness of reasoning. Our experimental results demonstrate that the proposed optimisations significantly enhance both computational and storage efficiency. The hypertree decomposition-based reasoning method achieves substantial speedups for cyclic rules, while the multi-scheme storage framework effectively reduces space requirements and improves incremental update performance. These contributions provide a scalable and adaptable approach to materialisation-based reasoning, making it more suitable for large-scale and dynamic knowledge-based systems.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Datalog [4] has long served as a foundational framework for Knowledge Representation and Reasoning in multiple fields, including Artificial Intelligence and database systems. However, its materialisation techniques currently face several challenges, particularly regarding computational efficiency for complex rules and storage optimisation for large-scale knowledge bases. These challenges impact both the initial computation and incremental maintenance of materialisations, limiting the practical application of Datalog-based techniques in real-world reasoning systems. In this chapter, we introduce the background of Datalog-based knowledge reasoning (Section 1.1) and discuss the challenges associated with materialisations (Section 1.2). We then present the research questions that guide this thesis in Section 1.3, focusing on computational efficiency and storage optimisation. Following this, thesis contributions and outline are summarised in Section 1.4. Finally, relevant publications resulting from this research are listed.

1.1 Datalog-based Knowledge Reasoning

Datalog is a rule-based formalism with the ability to express recursive dependencies [4], making it a versatile tool across various domains, including data analysis [10] and consistency checking [95]. As a syntactic subset of Prolog with well-defined fixed-point semantics [4], Datalog provides a robust foundation for deductive reasoning, enabling the derivation of implicit facts from explicitly stored knowledge. In the database community, it has been widely adopted as an expressive query language, particularly for recursive query processing [37, 64, 118]. In Artificial Intelligence, Datalog is extensively used as a knowledge representation language [49], leveraging its rule-based structure to encode domain knowledge in the form of 'if-then' rules. If all atoms in the premise of a rule hold, then the conclusion of the rule must also be

true. This logical framework makes Datalog a powerful tool for applications such as Semantic Web Reasoning [61], Knowledge Graph Inference [19], and Ontology-based Data Access [24].

In knowledge-based system applications, Datalog is widely used to manage and infer knowledge by reasoning over *structured facts* and *domain rules*. The *facts* are typically organised using the Resource Description Framework (RDF) [101], a standardised data model by the World Wide Web Consortium (W3C) that provides an intuitive and computationally effective format. RDF represents data as triples, each consisting of a subject, predicate, and object. These triples naturally form a directed, labelled graph, where the subject and object are nodes connected by an edge labelled with the predicate, effectively representing relationships between entities. The *domain rules* are commonly encoded using OWL 2 RL ontologies [103] and SWRL rules [73]. Both the RDF data format and the rule standards in these applications can be naturally captured by a Datalog subset with unary and binary predicates, thereby providing a unified framework for expressive reasoning and efficient inference. This syntactic compatibility facilitates the direct application of Datalog-based reasoning to RDF data, allowing complex domain knowledge to be represented and reasoned over in a structured and declarative manner. Consequently, Datalog-based reasoning has been widely adopted in RDF systems to deduce implicit knowledge. For example, facts such as *locatedIn(EffelTower, Paris)* and *locatedIn(Paris, France)* assert that the Eiffel Tower is located in Paris and Paris is located in France. The domain knowledge captures implicit information: for example, a Datalog rule *locatedIn(x,y), locatedIn(y,z)* $\rightarrow$ *locatedIn(x,z)* declares that if x is located in y and y is located in z, then x is located in z. By applying this rule to existing facts, a Datalog system can automatically infer new relationships, e.g., *locatedIn(EffelTower, France)*, that are not directly stated, thereby enriching the knowledge base. In the database setting, such an enriched RDF knowledge base can be thought of as an edge-labelled graph database enhanced by (possibly recursive) views expressed in a subset of Datalog.

The process of inferring implicit insights from a knowledge base using domain knowledge encoded in Datalog rules is known as logical rule-based knowledge reasoning [39]. This approach offers a distinct advantage in accuracy and reliability, as it strictly adheres to predefined logical constructs, ensuring consistency and predictability in its outcomes. By systematically deriving new facts based on existing ones, logical rule-based reasoning provides a principled framework that is both transparent and verifiable. Therefore, Datalog-based knowledge reasoning is an important technique in the field of Semantic Web and Knowledge Graphs. Recent surveys of

Datalog engines [63, 90, 98] have identified numerous implementations across both open-source and commercial platforms. These include systems such as RDFox [109], VLog [35], Oracle's RDF Store [152], GraphDB[1], Soufflé [85, 124], RecStep [52], SociaLite [125], LogicBlox [14], FVLOG [134], Nemo [80, 81], and Vadalog [18], which demonstrate the widespread adoption of Datalog-based reasoning in practice.

However, existing techniques still face key challenges, particularly in computational efficiency, incremental maintenance, and storage management. In this thesis, we address these challenges by primarily focusing on the reasoning techniques for a Datalog subset with unary and binary predicates, which is sufficient for many practical knowledge-based applications, including RDF data and OWL 2 RL ontologies (possibly extended with SWRL rules). The research presented in this thesis advances knowledge-based systems by enhancing the efficiency and scalability of Datalog-based reasoning, enabling more effective inference and query answering over large-scale, dynamic knowledge graphs. While our focus is on this specific subset, the proposed approach is designed to be general and adaptable in the context of general Datalog, making it potentially applicable to broader cases of Datalog reasoning as well.[2]

In Section 1.2, we will highlight the core technique behind Datalog-based reasoning. Additionally, we outline key challenges associated with computation, scalability, incremental maintenance, and efficient storage, which motivate the optimisations explored in this thesis.

1.2 Materialisations and Open Challenges

Given the explicit facts and Datalog rules, one of the main computational tasks of a knowledge-based system is answering queries over both the given facts and the facts implied by the rules. There are two main approaches to achieving this: *forward chaining* (often called *materialisation*), where all possible outcomes are precomputed and stored (materialised); and *backward chaining* [16, 137, 139], where facts are inferred on demand during query processing. The former allows for fast query evaluation at the expense of higher precomputation and storage requirements; the latter reduces precomputation and storage requirements, but can lead to increases in query evaluation time due to the need for on-the-fly rule evaluation [4, 138]. Given the potentially high cost of on-the-fly evaluation, particularly when complex and/or recursive rules

[1] https://graphdb.ontotext.com/

[2] The adaptability of our proposed approach to broader cases of Datalog, detailing its general applicability and potential limitations, will be clarified in Chapter 6.

are involved, most modern Datalog systems use the materialisation approach, with examples including RDFox [109], VLog [35], Vadalog [18], WebPIE [141], OWLIM [27], Oracle's RDF Store [152], and LogicBlox [14]. Several studies [7, 138] have also demonstrated the advantage of forward chaining over backward chaining. In this thesis, we explore several optimisations of the materialisation technique to enhance its applicability in real-world large-scale and dynamic settings.

Materialised facts (or *materialisations*) can be computed using the *naïve* strategy, which iteratively applies all Datalog rules to the available facts until no new facts can be derived. However, this approach is inefficient due to redundant derivations. The *seminaïve* evaluation strategy [4] optimises this process by ensuring that each rule instance is considered only once in the entire materialisation process, significantly reducing unnecessary recomputations. This process is often referred to as *initial materialisation*, as presented in Figure 1.1. Systems that deploy materialisation techniques usually also consider *incremental maintenance of materialisations* (as highlighted in green in Figure 1.1), in which the materialisations are incrementally updated when the initial facts change; this can be much more efficient than re-computing materialisations from scratch, especially for small changes.[3] As presented in Figure 1.1, in this thesis, we generally refer to the process of computing and maintaining materialisations as *reasoning*, following the convention in knowledge-based systems. This is distinct from *querying*, which in this thesis specifically refers to the evaluation of queries over the materialised fact base. The optimisations considered in this thesis primarily focus on the reasoning process illustrated in Figure 1.1.

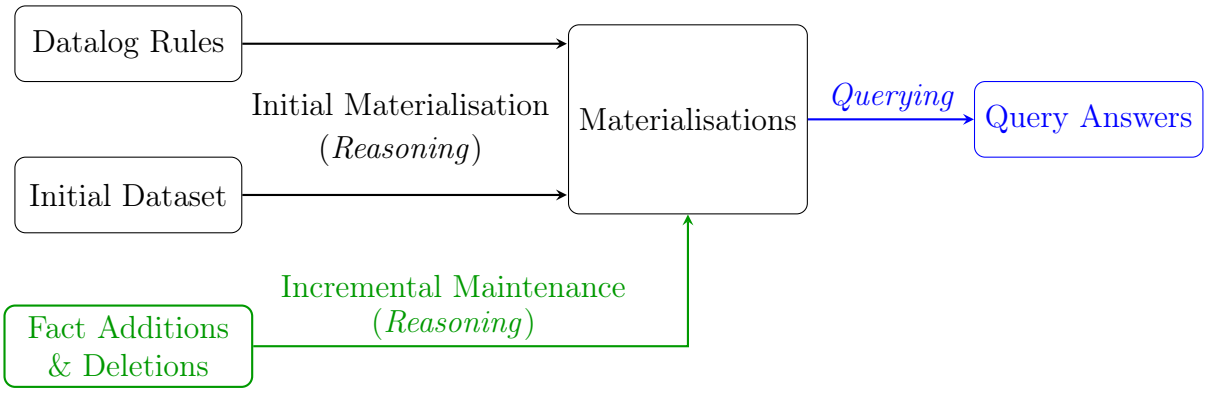


Figure 1.1: Illustration of *Reasoning* and *Querying*.

[3]In this thesis, the term *materialisation* is used to refer to both the computation process and the resulting set of inferences. If the intended meaning is not clear from the context, additional clarification will be provided.

Challenges: Key challenges in the materialisation technique are the efficient computation, maintenance, and storage of materialisations, particularly when dealing with large-scale recursive rules and dynamic updates:

- **Computational Efficiency:** Traditional algorithms for computing materialisations and conducting incremental maintenance typically use a unified iterative evaluation strategy for all rules, disregarding different rule structural information. This one-size-fits-all approach can lead to inefficiencies, as distinct rule shapes may benefit from specialised evaluation strategies. By integrating customised evaluation approaches for different rules into the reasoning process, the efficiency of materialisation and incremental maintenance can be significantly improved. Consequently, it enhances the overall performance of reasoning systems, particularly when dealing with complex rules that would be computationally expensive and inefficient if evaluated using a standard approach. By selectively applying optimised strategies, the system can effectively balance between generality and efficiency. This suggests that specialised evaluation strategies for specific rules could be developed to optimise the reasoning process further. Moreover, exploring how these customised strategies can be seamlessly integrated into the overall reasoning process would provide a flexible and efficient framework for Datalog-based systems.

- **Space Requirement:** Standard approaches typically store materialised facts in a straightforward manner, such as in tables. However, this plain storage strategy demands substantial space since rule applications expand the existing datasets. This space requirement can make materialisation impractical for large datasets and subsequent query-answering infeasible, as queries depend on the precomputed materialised facts. To mitigate these challenges, compressed storage techniques of materialisations can be explored to reduce space consumption while preserving efficient access to inferred facts. One promising direction is to exploit the structural information implied by the rules. For example, transitive closure rules, such as the *LocatedIn* rule mentioned in Section 1.1, imply hierarchical relationships that can be efficiently represented using graph-like data structures. However, integrating such specialised schemes into the reasoning process would be challenging, as it imposes specific requirements on the schemes to support efficient incremental updates: during Datalog reasoning, rules are applied recursively, necessitating that the specialised storage schemes

dynamically reflect newly derived facts and deletions. Moreover, the reasoning algorithms should remain general enough to allow consistent access to the compressed storage across multiple iterations, especially when conducted over multiple storage schemes.

- **Maintenance Challenges:** Incremental maintenance is crucial for Datalog systems under dynamic updates. However, incremental maintenance requires tracking fact dependencies and performing cascading updates when facts are inserted or deleted. With customised application approaches for certain rules and specialised storage schemes, tracking these dependencies becomes significantly more challenging. The recursive nature of Datalog creates complex interdependencies among facts, requiring careful propagation of changes throughout the entire materialised knowledge base to preserve correctness whilst maintaining efficiency.

We now introduce the specific research questions that guide this thesis, focusing on the motivations and objectives in optimising the computation, maintenance, and storage of materialisations.

1.3 Research Questions

As discussed in Section 1.2, our research has been dedicated to optimising the time and space efficiency of the materialisation process, encompassing both the initial computation and the incremental maintenance. In this section, we identify key research gaps and present the specific research questions that this thesis aims to address.

1.3.1 Computational Efficiency for Cyclic Rules

Most general materialisation computation and maintenance algorithms, such as the *counting* algorithm [67], *Delete/Rederive* algorithm [128], and *Backward/Forward* (B/F) algorithm [108], only focus on the rule application workflow and rely on traditional join plans to evaluate rule bodies. While effective in many cases, this approach can be inefficient, especially for cyclic rules [111], leading to excessive intermediate results and redundant computations. This phenomenon can be observed in real-life applications, for example where rules are used to model complex systems, which may include the evaluation of numerical expressions.[4]

[4]https://2021-eu.semantics.cc/graph-based-reasoning-scaling-energy-audits-many-customers

Hypertree decomposition (HD)-based approaches [62] have shown significant benefits in one-time static query evaluation by decomposing complex joins into tree-like structures that allow more efficient processing [2, 136]. However, adapting HD-based evaluation for the computation and maintenance of materialisations remains challenging. Unlike queries, which are evaluated once, materialisation involves iterative rule applications, requiring careful handling of redundant derivations and efficient tracking of fact derivations through incremental updates. This leads to the following research question:

RQ 1 *Considering the evaluation of a single Datalog rule, how can the hypertree decomposition-based approach be adapted to efficiently support the initial materialisation and incremental maintenance processes?*

Such an optimisation is specifically designed for cyclic rules, leveraging hypertree decomposition to improve their evaluation by minimising redundant join operations. However, applying this method uniformly to all rules may introduce additional overhead for non-cyclic rules or simpler cases, where the benefits of hypertree decomposition are less pronounced. This suggests the need for a hybrid approach that selectively applies optimisations based on rule shapes. For example, hypertree decomposition could be used for cyclic rules to enhance computational efficiency by reducing intermediate results, while standard seminaïve evaluation could be applied to acyclic rules for faster processing without unnecessary overhead. This hybrid strategy aims to maximise performance by balancing the strengths of both evaluation techniques, ensuring efficient reasoning across diverse rule structures. This consideration leads to the following research question:

RQ 2 *How can one integrate a hypertree decomposition-based approach for certain rules with standard evaluation methods for others to maximise performance?*

This research question aims to develop a reasoning algorithm that efficiently supports both initial materialisation and incremental maintenance by strategically integrating HD-based optimisations for complex cyclic rules while retaining standard evaluation techniques for simpler rules. By effectively combining different rule processing strategies, this research aims to achieve good reasoning performance while maintaining accuracy and consistency.

1.3.2 Storage Optimisation with Multiple Storage Schemes

Materialisation-based reasoning requires storing all inferred facts, often leading to significant storage overhead, particularly when handling large-scale datasets and recursive rules. Traditional approaches typically use standard table-like storage structures, which are efficient for querying but may be suboptimal for incremental updates and reasoning tasks. Maintaining an efficient and scalable storage mechanism becomes a critical challenge as the dataset grows and new inferences are derived. The failure of materialisation prevents the precomputation of inferences, rendering subsequent query answering infeasible due to the absence of necessary derived facts.

Optimised storage of materialisations has so far been limited to handling equality relations [106] and leveraging columnar storage [35], with existing approaches largely ignoring the structural properties of rules that could lead to significant storage and retrieval optimisations. A promising direction is the customisation of storage schemes tailored to different types of rule derivations, allowing for more efficient representation and access patterns. Certain rules exhibit structural properties that can be exploited for more efficient storage and retrieval. For instance, graph-based storage can be beneficial for derivations of transitive closure rules, while virtual representations may be suitable for union rules that generate highly redundant inferences. As an example, the following rule (R1) declares R as a transitive relation, which can be regarded as a reachability problem in a graph constructed from existing R facts. Therefore, the derivations from (R1) correspond to reachable pairs in the graph, and storing these reachable pairs could benefit from compressed graph-like storage. In contrast, rules (R2)–(R4) define the relation U as the union of relations A, B, and C, thereby categorising them as union rules. Consequences from these rules can be omitted from explicit storage and instead be referenced through their underlying relations (A, B, and C), thereby reducing memory consumption while preserving query efficiency.

$$R(x, z) \leftarrow R(x, y), R(y, z) \tag{R1}$$

$$U(x, y) \leftarrow A(x, y) \tag{R2}$$

$$U(x, y) \leftarrow B(x, y) \tag{R3}$$

$$U(x, y) \leftarrow C(x, y) \tag{R4}$$

Adopting multiple storage schemes tailored to different types of rules could significantly enhance both reasoning efficiency and storage scalability.

However, integrating multiple storage schemes into the materialisation and incremental reasoning workflow is non-trivial. A major challenge is ensuring seamless

interaction between different storage mechanisms while maintaining efficient retrieval, updates, and consistency. Moreover, reasoning engines typically assume a unified storage model, making it difficult to decouple storage from reasoning logic in a way that allows for flexible integration of customised schemes. These challenges lead to the following research question:

RQ 3 *How can a general multi-scheme framework be designed to effectively decouple storage from reasoning processes, enabling the integration of, for example, standard table-like storage schemes and customised optimisations?*

Efficient storage of materialised facts is a key challenge in Datalog-based reasoning, especially for rules that produce large, redundant, or highly repetitive inferences. Transitive closure and union rules are particularly prone to significant storage overhead, necessitating careful storage design. Developing storage schemes that integrate into a multi-scheme framework requires the ability to efficiently manage insertions and deletions of explicit facts while supporting the identification and retrieval of different fact subsets—whether pre-existing, newly derived, or pending deletion. Achieving this balance is challenging, as it demands both structural optimisation and efficient update mechanisms. Therefore, we proposed the following research question:

RQ 4 *How can specialised storage schemes be developed for the derivations of transitive closure and union rules that meet the general requirements of a multi-scheme framework, ensuring efficient integration into the materialisation and incremental reasoning process while maintaining scalability and retrieval efficiency?*

This section has outlined the key research questions that have shaped this PhD research. These questions focus on developing and evaluating advanced optimisations for challenging rule types and their seamless integration into a general reasoning procedure that supports both the initial materialisation and the incremental maintenance. The ultimate goal is to enhance materialisation techniques for efficient computation, storage, and maintenance of large knowledge graphs.

1.4 Thesis Contributions and Outline

This thesis explores optimisations for Datalog-based reasoning, focusing on efficient computation, incremental maintenance, and storage optimisation. Specifically, we

address the challenges posed by cyclic rules, scalable incremental updates, and multi-scheme storage, introducing novel techniques that enhance reasoning performance and adaptability. This thesis is organised as follows.

In Chapter 2, we introduce fundamental notations of Datalog and present classical materialisation computation algorithms, including the naïve and seminaïve algorithms. We highlight the key derivation step using the seminaïve operator, which effectively eliminates redundant derivations and serves as a foundational technique in more advanced materialisation algorithms. Additionally, we present the DRed algorithm, a general incremental maintenance approach that supports both the initial computation and ongoing maintenance of materialisations. This algorithm extends materialisation techniques by efficiently handling fact insertions and deletions, making it a widely used method for reasoning over dynamic datasets. The algorithms discussed in this chapter are conceptual, providing a theoretical foundation for understanding materialisation workflows and incremental maintenance strategies. Later chapters build upon these concepts, integrating them into more sophisticated frameworks that further optimise computation and storage efficiency. Furthermore, in Section 2.4, we introduce key notations in the context of knowledge representation and establish the connection between RDF and Datalog. We also discuss how Datalog-based techniques can be effectively applied to knowledge-based systems, highlighting opportunities for optimising storage techniques to enhance reasoning efficiency.

In Chapter 3, we review existing research on the computation, maintenance, and storage management of the materialisation process. Our analysis highlights two key gaps that motivate the research questions addressed in this thesis:

- First, existing rule evaluation strategies predominantly rely on traditional techniques, which fail to leverage the structural properties of complex rules, such as cyclic rules. This inefficiency motivates our research questions RQ1-RQ2, which explore how hypertree decomposition can be integrated into Datalog reasoning to improve computational efficiency.

- Second, existing studies on storage management primarily focus on unified storage strategies that treat all facts equally, overlooking the structural properties of facts derived from different types of rules. Such an approach neglects opportunities for more specialised and efficient storage representations that can better support materialisation and incremental maintenance. This limitation leads to our research questions RQ3-RQ4, which investigate a multi-scheme storage framework and tailored storage schemes for specific rule patterns.

Through this literature review, we establish the foundation for our proposed optimisations, demonstrating the necessity of both computational and storage-based improvements to enhance the scalability of materialisation-based reasoning.

In Chapter 4, we propose a hypertree decomposition (HD)-based evaluation strategy to optimise the processing of cyclic rules, which are traditionally computationally expensive due to redundant derivations and inefficient join operations inherent in standard seminaïve evaluation. Conventional Datalog reasoning typically relies on join plans that result in excessive intermediate results, especially for complex recursive rules. Although hypertree decomposition has been widely adopted in query answering to generate more efficient execution plans and reduce redundancy, its applicability to materialisation and incremental reasoning remains largely unexplored. To bridge this gap, we develop algorithms that integrate hypertree decomposition into Datalog materialisation and incremental maintenance, thereby enabling more efficient evaluation of cyclic rules. Specifically:

- We introduce three HD-based evaluation algorithms tailored to different phases of the reasoning process - namely, overdeletion, rederivation, and addition. These algorithms enable more efficient handling of cyclic rules by reducing unnecessary recomputations and enhancing rule application efficiency. By integrating these tailored evaluation techniques into the reasoning algorithms that support both initial computation and incremental maintenance, we significantly improve overall performance and scalability in dynamic knowledge-based systems, especially when complex cyclic rules are involved. **This addresses Research Question RQ1.**

However, hypertree decomposition introduces additional data structures, which may impose runtime and memory overhead. To mitigate this:

- We employ a modular framework that selectively applies HD-based reasoning for complex rules while retaining standard evaluation techniques for simpler cases. This hybrid approach ensures that the benefits of hypertree decomposition are leveraged where they are most impactful while avoiding unnecessary computational overhead for straightforward rules. **This addresses Research Question RQ2.**

Our empirical evaluation demonstrates that, particularly for programs with cyclic rules, the combined approach significantly outperforms standard evaluation techniques, achieving substantial speedups while maintaining efficient resource usage.

In Chapter 5, we investigate customised storage strategies for different rule derivations, leading to the development of a multi-scheme storage framework. During materialisation, when rules derive a large number of facts, traditional materialisation approaches may become impractical due to excessive space requirements. To address this challenge:

- We propose a general-purpose framework that decouples storage from reasoning, enabling the integration of specialised storage schemes into standard Datalog materialisation and incremental maintenance algorithms. We formally define the concept of storage schemes and establish the requirements they must satisfy to ensure correctness and efficiency in the materialisation and incremental maintenance process. These requirements specify how schemes should support fact retrieval, incremental updates, and integration with reasoning procedures while preserving the dependencies between derived facts. By structuring storage around these principles, our framework enables seamless incorporation of both standard and specialised storage techniques, facilitating scalable and efficient reasoning over large knowledge bases. **This addresses Research Question RQ3.**

This framework provides a flexible and extensible approach to managing different types of rule derivations by leveraging customised storage structures tailored to specific rule patterns. Additionally:

- We introduce two specialised schemes for transitive closure rules and union rules. The transitive closure scheme employs a graph-based representation to compactly encode reachability relationships, reducing redundancy and optimising incremental updates. Meanwhile, the union rule scheme utilises a virtual storage format to efficiently manage large sets of derived facts, minimising storage overhead. By integrating these schemes into our multi-scheme framework, we ensure that materialisation and incremental reasoning can be performed efficiently while maintaining correctness. **This addresses Research Question RQ4.**

By tailoring storage schemes to specific rule patterns, our framework improves scalability while maintaining correctness in materialisation-based reasoning. Our experiments demonstrate that the proposed approach successfully computes materialisations even in cases where standard storage methods fail due to space limitations. Furthermore, the multi-scheme framework significantly enhances both storage efficiency and

reasoning performance, making it a practical solution for large-scale knowledge-based systems.

Finally, in Chapter 6, we provide a conclusion for this work and discuss the potential applicability of our proposed methods to broader Datalog cases in detail. Specifically, the hypertree decomposition-based optimisations in Chapter 4, the multi-scheme framework, and the union scheme in Chapter 5 are designed within the general Datalog setting and are not limited to the RDF subset. Additionally, we highlight several promising research directions that can further extend and enhance the effectiveness of these approaches.

1.5 Publications

This thesis is based on my publications listed below [156, 157]:

1. Xinyue Zhang, Pan Hu, Yavor Nenov, Ian Horrocks. Optimised Storage for Datalog Reasoning. In *The 38th Annual AAAI Conference on Artificial Intelligence (AAAI)*, pages 10748 - 10755, 2024.

2. Xinyue Zhang, Pan Hu, Yavor Nenov, Ian Horrocks. Enhancing Datalog Reasoning with Hypertree Decompositions. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 3383–3393, 2023.

An extended version of the AAAI-24 paper [157] is planned for submission to the *Artificial Intelligence Journal*.

Chapter 2

Preliminaries

This chapter introduces the preliminaries of Datalog (Section 2.1), as well as basic Datalog materialisation computation (Section 2.2) and maintenance algorithms (Section 2.3), with a focus on the core seminaïve evaluation technique, which serves as a fundamental operation in more advanced approaches. We also establish the connection between RDF and Datalog, discussing how general Datalog techniques can be applied to materialise RDF knowledge bases in Section 2.4.

2.1 Datalog

Datalog is a widely used rule-based language in various domains, including Database Management [4, 57], Data Analysis [10, 95, 125], Logic Programming [45], and Knowledge Representation [34, 140, 144]. In this section, we summarise the formal definitions of the syntax and semantics of Datalog.

2.1.1 Syntax

Let $\mathcal{C}$ be a finite set of constants, $\mathcal{V}$ be a finite set of variables, and $\mathcal{P}$ be a finite set of predicate symbols, each associated with a fixed arity. A *term* is either a constant in $\mathcal{C}$ or a variable in $\mathcal{V}$. An *atom* has the form $P(t_1, \ldots, t_k)$, where $P \in \mathcal{P}$ is a predicate of arity k, and each t_i is a term. A *fact* is a ground atom, i.e., an atom in which all terms are constants; a *dataset* is a finite set of facts. Formally, a Datalog *rule* is an expression with the form:

$$H \leftarrow B_0 \wedge \cdots \wedge B_n, \tag{2.1}$$

where H and B_i with $0 \leq i \leq n$, $n \geq 0$ are atoms. The head of a rule r is denoted as $\mathsf{h}(r) = H$, and its body atoms are $\mathsf{b}(r) = \{B_0, \ldots, B_n\}$. For S an atom or a set

of atoms, $\mathsf{var}(S)$ is the set of variables appearing in S. For a rule r to be *safe*, each variable occurring in its head must also occur in at least one of its body atoms, i.e., $\mathsf{var}(\mathsf{h}(r)) \subseteq \mathsf{var}(\mathsf{b}(r))$. Similarly, $\mathsf{var}(r)$ represents all the variables occurring in the rule r.

Datalog Program: A *program* Π is a finite set of safe rules. Please note that in Equation (2.1), the number of body atoms n can be zero ($n = 0$); in this case, the rule reduces to a fact. That is, a fact can be viewed as a special case of a rule with an empty body. Therefore, a program may contain both rules with non-empty bodies and facts (rules with $n = 0$).

Following standard terminology, we distinguish between *extensional database* (EDB) predicates, whose facts are explicitly provided as input, and *intensional database* (IDB) predicates, whose facts are derived through rules. We denote the set of input facts in a program Π by $E = \mathsf{edb}(\Pi)$. For convenience, in the rest of this thesis, this is referred to as the *input database*, or simply as the explicitly given facts or data. Also, in the rest of this thesis, "rules" refers to rules with non-empty bodies (i.e., with at least one body atom).

Instances: A *substitution* σ is a finite function $\sigma : \mathcal{V} \to \mathcal{C}$ that maps variables to constants. The application of a substitution σ to an expression α, which can be a term, an atom, a rule, or a set of such constructs, yields an expression $\alpha\sigma$ that replaces each occurrence of variable x in α with $\sigma(x)$ if x is in the domain of σ. If a substitution σ maps all variables occurring in a rule r to constants, then $r\sigma$ is an *instance* of r.

2.1.2 Logical Semantics

The meaning of a Datalog program can be formally characterised in terms of *model-theoretic semantics* [4]. In this view, we define the space of possible interpretations of the program and specify which interpretations satisfy the program rules. We now introduce the Herbrand base, Herbrand interpretations, models, and the notion of a least model.

Let $\mathcal{H}$ be the *Herbrand base* of Π over $\mathcal{C}$, i.e., the set of all ground atoms that can be formed from predicate symbols in $\mathcal{P}$ and constants in $\mathcal{C}$. A *Herbrand interpretation* $I \subseteq \mathcal{H}$ is a set of ground atoms. Given a rule r, an *instance* of r is any expression $r\sigma$ obtained by applying a substitution σ that maps all variables in r to constants. An instance $r\sigma$ of the form $H\sigma \leftarrow B_1\sigma \wedge \cdots \wedge B_n\sigma$ is *satisfied* by I if $B_1\sigma, \ldots, B_n\sigma \in I$ implies $H\sigma \in I$. An interpretation I is a *model* of Π if it satisfies all ground instances

of all rules in Π. Among all models of Π, there exists a unique *least Herbrand model* M_Π, which is the smallest $I \subseteq \mathcal{H}$ that is a model of Π.

2.1.3 Fixpoint Semantics

An alternative but equivalent way to characterise the semantics of a Datalog program is through *fixpoint semantics* [4], which provides an operational view of how the program's conclusions can be computed. This perspective is based on the *immediate consequence operator*, whose least fixpoint corresponds exactly to the program's least model. Moreover, fixpoint semantics underlie the practical computation of *materialisation*, which is central to Datalog engine implementations.

Given a rule r and a dataset $I \subseteq \mathcal{H}$, we define the result of applying r to I as:

$$r[I] = \{\mathsf{h}(r\sigma) \mid \sigma : \mathcal{V} \to \mathcal{C} \text{ is a substitution, } \mathsf{b}(r\sigma) \subseteq I\}, \tag{2.2}$$

where:

- σ is a substitution mapping variables to constants,
- $\mathsf{b}(r\sigma)$ represents the instantiation of the rule's body with the substitution σ,
- The condition $\mathsf{b}(r\sigma) \subseteq I$ ensures that all literals in the body of the rule must already exist in I for the head to be inferred.

The result of applying a Datalog program Π to I is defined as:[1]

$$T_\Pi(I) = \Pi[I] = \bigcup_{r \in \Pi} r[I], \tag{2.3}$$

where $T_\Pi : 2^{\mathcal{H}} \to 2^{\mathcal{H}}$ is the *immediate consequence operator*, a standard term used in the fixpoint semantics of Datalog. The operator T_Π is monotone and continuous on the lattice $(2^{\mathcal{H}}, \subseteq)$, and therefore admits a *unique least fixpoint*. Operationally, this fixpoint can be computed by iteratively applying T_Π (or equivalently $\Pi[\cdot]$) starting from the input dataset $E \subseteq \mathcal{H}$: we initialise $I_0 = E = \mathsf{edb}(\Pi)$, and repeatedly compute $I_{i+1} = T_\Pi(I_i)$, until a fixpoint is reached (i.e., $I_{i+1} = I_i$). It is well-known that the resulting fixpoint corresponds to the least Herbrand model M_Π of the Datalog program Π.

[1] For consistency with this formal fixpoint semantics, we introduce the immediate consequence operator T_Π here; however, in the rest of this thesis, we will use the notation $\Pi[I]$ to denote the result of applying the program Π to a dataset I.

2.1.4 Materialisation

The iterative fixpoint computation described above provides the foundation for what is known as *materialisation* in practical Datalog systems. The term "materialisation" refers both to the process of iteratively applying a rule set Π to input facts E, and to the resulting set of facts derived by applying Π to E. Concretely, the materialisation process corresponds to computing the least fixpoint of the immediate consequence operator T_Π starting from E, and can be viewed as the forward-chaining closure of E under Π.

Formally, given an explicit *input dataset* $E \subseteq \mathcal{H}$, the *materialisation* of Π with respect to E is computed by iterative application of T_Π starting from E:

$$I_0 = E, \qquad I_{i+1} = I_i \cup T_\Pi(I_i) = I_i \cup \Pi[I_i], \qquad I_\infty = \bigcup_{i \geq 0} I_i.$$

I_∞ is the least fixpoint of T_Π extending E, representing the materialisation of E under Π. Moreover, let $\Pi^i[E] = \Pi^i[I_0] = I_i$ denote the result of iteratively applying rules in Π to the initial facts I_0 for i times.

2.2 The Computation of Datalog Materialisations

In this section, we introduce fundamental approaches for computing Datalog materialisations: the naïve and semi-naïve algorithms. The evaluation operators for these approaches, formally defined in Equations 2.2 and 2.5, provide the foundational mechanisms for deriving the consequences of Datalog rules. These techniques not only underpin the standard materialisation process but also form the basis for numerous incremental maintenance strategies, which will be reviewed in Chapter 3.

2.2.1 Naïve Algorithm

As discussed in Section 2.1.3, the semantics of Datalog directly follows the operational semantics for Datalog programs stemming from fixpoint theory [4]. The evaluation of a Datalog program can be understood as an iterative process that starts with an initial set of explicitly given facts and applies inference rules repeatedly until a fixpoint is reached. The operator $\Pi[I]$ defined in Section 2.1 can formally capture this process, which applies Π to the current fact base I and derives new facts at each iteration. Using this formalism, we can introduce the naïve algorithm for Datalog

materialisation presented in Algorithm 1. At each iteration i, the entire dataset is expanded by applying all rules exhaustively, leading to:

$$I_i = I_{i-1} \cup \Pi[I_{i-1}]. \tag{2.4}$$

The algorithm halts when the fixpoint condition $I_i = I_{i-1}$ is satisfied, meaning no additional facts are inferred.

This algorithm guarantees completeness by exploring all derivable facts but suffers from inefficiencies due to redundant computations. Specifically, at each iteration, the entire dataset is reprocessed (as highlighted by $\Pi[I_i]$ in line 4), leading to redundant computations and unnecessary rule applications on facts that were already considered in previous iterations. This redundancy can be particularly costly in scenarios where the number of derived facts grows rapidly, making the naïve approach impractical for large-scale datasets. To address these inefficiencies, the seminaïve algorithm introduces an optimisation by leveraging *delta-based computation*, which focuses on newly derived facts at each iteration rather than re-evaluating the entire fact set. By maintaining and propagating only the incremental changes, the seminaïve approach significantly reduces redundant operations, ensuring a more efficient fixpoint computation.

2.2.2 Seminaïve Algorithm

The seminaïve algorithm [4] realises non-repetitive reasoning by identifying newly derived facts in each round of rule application. Let I represent the current fact base and $\Delta \subseteq I$ the newly derived facts at each iteration; the application of a single rule r to these sets is denoted by the seminaïve operator:

$$r[I, \Delta] = \{\mathsf{h}(r\sigma) \mid \sigma \text{ is a substitution s.t. } \mathsf{b}(r\sigma) \subseteq I \text{ and } \mathsf{b}(r\sigma) \cap \Delta \neq \emptyset\}, \tag{2.5}$$

where:

Algorithm 1 Naïve Datalog Materialisation

Require: Initial dataset E, Datalog program Π
Ensure: Materialised fact base $I_\infty = \Pi^\infty[E]$

1: $I_0 \leftarrow E$ ▷ Initialise with explicitly given facts
2: $i \leftarrow 0$
3: **repeat**
4: $\quad I_{i+1} \leftarrow I_i \cup \Pi[I_i]$ ▷ Apply all rules to current fact set
5: $\quad i \leftarrow i + 1$
6: **until** $I_i = I_{i-1}$ ▷ Termination when fixpoint is reached
7: **return** $I_\infty \leftarrow I_i$

- $\mathsf{h}(r\sigma)$ represents the head of the rule instance derived using a substitution σ,
- $\mathsf{b}(r\sigma)$ denotes the set of facts required to satisfy the body of the rule,
- The condition $\mathsf{b}(r\sigma) \cap \Delta \neq \emptyset$ ensures that at least one fact in the rule body is newly derived.

The seminaïve operator can be extended to a Datalog program Π as follows:

$$\Pi[I, \Delta] = \bigcup_{r \in \Pi} \{r[I, \Delta]\}. \tag{2.6}$$

The definition of $\Pi[I, \Delta]$ ensures that the algorithm will only consider rule instances that have not been considered before. In practice, $r[I, \Delta]$ can be efficiently implemented by evaluating the rule body $n + 1$ times [108]. Specifically, for the ith evaluation, $0 \leq i \leq n$, the body is evaluated by:

$$B_0^{I \setminus \Delta} \wedge \cdots \wedge B_{i-1}^{I \setminus \Delta} \wedge B_i^{\Delta} \wedge B_{i+1}^{I} \wedge \cdots \wedge B_n^{I}, \tag{2.7}$$

in which the superscript identifies the set of facts where each atom is matched.

The pseudo-code presented in Algorithm 2 outlines the seminaïve evaluation process. The algorithm initialises the fact base with the initial dataset E and iteratively applies the rule set Π, updating the materialised fact base incrementally until a fixpoint is reached, i.e., when no new facts are derived. Each iteration consists of the following key steps:

1. Updating the fact base: newly derived facts from the previous iteration are added to the current fact set.
2. Incremental computation: new facts are inferred using the seminaïve operator, which ensures that at least one newly derived fact is involved in each rule instantiation.
3. Termination check: The process halts when no further facts can be inferred.

The seminaïve algorithm guarantees completeness while significantly improving the performance of Datalog materialisation by minimising redundant computations.

Algorithm 2 Seminaïve Datalog Materialisation

Require: Initial dataset E, rule set Π
Ensure: Materialised fact base $I_\infty = \Pi^\infty[E]$

```
I_0 ← ∅                                          ▷ Initialise with initial facts
Δ_0 ← E                                          ▷ Track newly derived facts
i ← 0
repeat
    I_{i+1} ← I_i ∪ Δ_i                          ▷ Update fact base with new facts
    Δ_{i+1} ← Π[I_{i+1}, Δ_i] \ I_{i+1}          ▷ Compute new facts using seminaïve operator
    i ← i + 1
until Δ_i = ∅                                    ▷ Termination when no new facts are derived
return I_∞ ← I_i
```

2.3 Incremental Maintenance of Materialisations

Incremental maintenance of materialised Datalog programs is a crucial technique to efficiently handle updates to the underlying fact base without recomputing the entire materialisation from scratch. In dynamic environments where facts are frequently inserted or deleted, a full recomputation can be computationally expensive and impractical. Incremental maintenance aims to update the materialisation incrementally by propagating changes while preserving correctness and efficiency. Formally, given a fully materialised fact base $\Pi^\infty[E]$ computed from an initial fact set E, and an update consisting of newly added facts E^+ and deleted facts E^-, the goal of incremental maintenance is to compute the updated materialisation $\Pi^\infty[E \setminus E^- \cup E^+]$.

The seminaïve algorithm is well-suited for incremental addition of facts in a Datalog program, as it naturally accommodates new facts by iteratively applying rules to derive further conclusions based on the newly added facts. Specifically, incremental addition can be achieved by Algorithm 2 by initialising I_0 as $\Pi^\infty[E]$ and Δ_0 as E^+. However, the seminaïve algorithm is not directly applicable to incremental deletion, as it does not inherently track dependencies between facts. When facts are removed from the fact base, it becomes necessary to identify and remove all derived facts that were dependent on the deleted facts and do not have remaining proofs after the deletion, a task that the seminaïve algorithm cannot efficiently handle.

To tackle the challenge of incremental deletions, the *Delete and Re-derive (DRed)* algorithm [36, 67, 108, 128] has been proposed as a fundamental and effective solution. In this section, we introduce a generalised conceptual DRed algorithm, presented in Algorithm 3, to illustrate the key aspects of the deletion and re-derivation process in Datalog materialisation. A comprehensive review of related works and advancements

Algorithm 3 DRed Algorithm

Require: Initial materialised fact set I, rule set Π, deletions E^-, additions E^+
Ensure: materialisation I is updated from $\Pi^\infty[E]$ to $\Pi^\infty[E \backslash E^- \cup E^+]$

1: OVERDELETE, REDERIVE, ADD
2: **procedure** OVERDELETE
3: $\quad \Delta_{\text{del}} \leftarrow E^- \cap E$
4: $\quad D \leftarrow \Delta_{\text{del}}$ ▷ Collect overdeleted facts
5: $\quad$ **repeat**
6: $\quad\quad \Delta_{\text{del}} \leftarrow \Pi[I, \Delta_{\text{del}}] \cap (I \backslash \Delta_{\text{del}})$ ▷ Apply semi-naïve operator
7: $\quad\quad I \leftarrow I \setminus D$
8: $\quad\quad D \leftarrow D \cup \Delta_{\text{del}}$ ▷ Collect overdeleted facts
9: $\quad$ **until** $\Delta_{\text{del}} = \emptyset$
10: **procedure** REDERIVE
11: $\quad \Delta_{\text{rederived}} \leftarrow \Pi[I] \cap D$
12: **procedure** ADD
13: $\quad \Delta_{\text{add}} \leftarrow (E^+ \cup \Delta_{\text{rederived}}) \setminus I$
14: $\quad$ **while** $\Delta_{\text{add}} \neq \emptyset$ **do**
15: $\quad\quad I \leftarrow I \cup \Delta_{\text{add}}$
16: $\quad\quad \Delta_{\text{add}} \leftarrow \Pi[I, \Delta_{\text{add}}] \setminus I$ ▷ Apply semi-naïve operator
17: **return** I

in incremental maintenance techniques will be provided in Section 3.2.

The DRed algorithm is designed to efficiently handle fact deletions and additions while minimising redundant computations. It follows a three-phase workflow consisting of *over-deletion*, *re-derivation*, and *addition*, as presented in line 1 of Algorithm 3, leveraging the seminaïve evaluation strategy to optimise incremental updates.

The first phase, **Overdelete**, presented in lines 2-9 of Algorithm 3, begins by identifying the facts directly affected by deletions, denoted as E^-, and iteratively propagating their effects using the seminaïve evaluation operator $\Pi[I, \Delta_{\text{del}}]$ in line 6. Overdeletion ensures that all facts derived from the removed facts are properly retracted, and the collected overdeleted facts are stored in the set D.

The second phase, **Rederive**, as shown in lines 10-11, attempts to restore any facts that were incorrectly deleted in the previous phase. This is achieved by checking whether they can still be derived from the remaining fact base I. Specifically, the facts in D are re-evaluated using the remaining facts in I through the rule set Π, as presented in line 11.

The final phase, **Add**, presented in lines 11-16, incorporates both the newly added facts E^+ and the re-derived facts $\Delta_{\text{rederived}}$ into the materialisation. The seminaïve operator $\Pi[I, \Delta_{\text{add}}]$ is applied iteratively (line 16) to propagate the additions through

the rule set until a fixpoint is reached. This procedure is analogous to the seminaïve algorithm, as both recursively apply rules until a fixed point is reached, ensuring that all possible derivations are computed.

The algorithm ensures that the materialisation I is correctly updated from $\Pi^\infty[E]$ to $\Pi^\infty[E \backslash E^- \cup E^+]$, maintaining correctness and completeness while optimising computational efficiency. This is achieved by structuring the workflow into distinct phases and leveraging the seminaïve operator in the overdeletion and addition process. The sections in Chapter 3 will discuss further optimisations and practical considerations for implementing the DRed algorithm in large-scale reasoning systems.

2.4 RDF and Datalog

In this section, we introduce some foundational definitions in the context of knowledge representation. Section 2.4.1 presents the RDF data model and its syntactic structure, while Section 2.4.2 provides an overview of ontologies and their role in reasoning within knowledge-based systems. We also discuss how these syntactic structures can be naturally captured by a Datalog subset with unary and binary predicates, enabling expressive reasoning over RDF data. Section 2.4.3 explores the considerations when applying general Datalog-based approaches to the RDF subset, highlighting the unique challenges and opportunities for optimising storage solutions tailored for RDF reasoning.

2.4.1 RDF Data Model

RDF Syntax: The Resource Description Framework (RDF) is a standardised data model used to represent structured information on the web [101]. It organises data into *triples*, each consisting of a *subject*, *predicate*, and *object*, forming a *statement* about resources. An *RDF triple* has the form:

$$\langle \text{subject}, \text{predicate}, \text{object} \rangle$$

where:

- **Subject** refers to the resource being described.
- **Predicate** denotes a property or relationship.
- **Object** can be either a literal value or another resource.

For example:

$$\langle \texttt{Alice}, \texttt{knows}, \texttt{Bob} \rangle$$

This can be interpreted as "Alice knows Bob," where `Alice` is the subject, `knows` is the predicate (property), and `Bob` is the object. RDF triples can be understood as a graph because each triple represents a directed edge $s \xrightarrow{p} o$ from the subject (s) to the object (o), labelled by the predicate (p), thereby forming a graph structure where nodes correspond to entities and edges denote relationships between them.

Properties and Class Types in RDF: In RDF, *properties* define relationships between resources or between a resource and a literal value. For example:

$$\langle \texttt{Bob}, \texttt{age}, 25 \rangle$$

This statement asserts that Bob's age is 25.

RDF also supports the definition of *class types* using the predicate `rdf:type`, allowing resources to be classified under specific categories. For example:

$$\langle \texttt{Alice}, \texttt{rdf:type}, \texttt{Person} \rangle$$

This indicates that Alice is of type Person.

Mapping to Datalog Predicates: These RDF constructs can be naturally captured using a subset of Datalog with unary and binary predicates:

- **Class Types** can be represented using *unary predicates*. For instance,

  $$\texttt{Person}(\texttt{Alice})$$

  indicates that Alice is a Person.

- **Properties** can be represented using *binary predicates*. For example,

  $$\texttt{knows}(\texttt{Alice}, \texttt{Bob})$$

  represents the statement that Alice knows Bob.

2.4.2 Ontologies

In knowledge-based systems, ontologies are typically defined for RDF data to provide a formal schema that organises and constrains the relationships between entities, enabling consistent interpretation, reasoning, and inference over the data. The ontologies expressed by OWL 2 RL [103] and SWRL [73] can be naturally captured

by Datalog rules. We now briefly present an illustrative example using OWL 2 RL. OWL 2 RL provides a rule-based way to express logical relationships and constraints over RDF data. These rules can be naturally translated into Datalog rules, leveraging Datalog's support for unary predicates (class membership) and binary predicates (property relationships). Below, we present two examples to illustrate this conversion.

- **Class Membership:**

  OWL 2 RL allows defining class hierarchies. For example, the axiom stating that all humans are animals can be represented as:

  ```
  Class: Human
    SubClassOf: Animal
  ```

  This can be captured in Datalog as:

  $$\texttt{Animal}(x) \leftarrow \texttt{Human}(x)$$

  This rule states that if x is a member of the class `Human`, then x is also a member of the class `Animal`.

- **Property Relationship:**

  OWL 2 RL supports property hierarchies. For instance, if we define the property `hasParent` as a subproperty of `hasAncestor`, the axiom would be:

  ```
  ObjectProperty: hasParent
    SubPropertyOf: hasAncestor
  ```

  In Datalog, this translates to:

  $$\texttt{hasAncestor}(x, y) \leftarrow \texttt{hasParent}(x, y)$$

  This rule means that if x has a parent y, then x also has y as an ancestor.

These ontologies are essential for providing domain knowledge and enabling reasoning over RDF data, facilitating advanced knowledge inference.

2.4.3 Considerations on Applying Datalog Techniques to RDF

As discussed above, in a knowledge-based system, RDF triples can be represented as unary and binary facts, while ontologies expressed using OWL 2 RL extended with SWRL rules can be captured as Datalog rules. This compatibility enables the direct application of Datalog-based reasoning techniques to RDF knowledge bases. For example, most materialisation computation and maintenance algorithms introduced in Sections 2.2-2.3, as well as the more advanced techniques reviewed in Sections 3.1-3.2, are directly applicable to RDF knowledge bases. Moreover, our hypertree decomposition-based optimisations proposed in Chapter 4, the multi-scheme framework, and the union scheme introduced in Chapter 5 are proposed in the general Datalog setting, which are suitable to be applied over the RDF subset.

For computation and maintenance algorithms, it is sufficient to refer to general Datalog techniques since they primarily focus on rule application and reasoning processes, which are largely independent of the underlying data format. However, storage techniques are closely tied to the underlying data model, as different data representations inherently require distinct access patterns and retrieval strategies. In the context of Datalog reasoning, general-purpose storage solutions are typically designed to support a wide variety of rule evaluations and queries. In contrast, many storage techniques have been developed specifically for the RDF subset, addressing the unique challenges posed by its inherent graph-based model. In RDF, triples form a directed, labelled graph where subjects and objects are nodes, and predicates define the edges connecting them. This graph-like structure introduces complex access patterns and many-to-many relationships, necessitating optimised storage solutions tailored to RDF's distinctive characteristics.

To optimise query efficiency for RDF data, advanced storage techniques have been developed that leverage its structural properties and cater to diverse access patterns. These include multi-indexing strategies [104], graph-based approaches [71], and property tables [99] that efficiently support common query patterns and efficient join processing. However, most of these techniques are primarily designed for query answering, with limited exploration into optimising storage for reasoning tasks, such as materialisation and incremental maintenance.

Therefore, in Section 3.4.1, we review existing storage techniques specifically for the RDF subset, focusing on their relevance to reasoning efficiency. This review also highlights the research gap and motivates our investigation into specialised storage solutions tailored for derivations of particular Datalog rules.

2.5 Mathematical Notations

Table 2.1 summarises the notations used throughout this thesis. In the following chapters, additional notations specific to particular chapters will be introduced as necessary to facilitate the discussion of more specialised topics.

Notation	Description
r	A single Datalog rule
Π	A set of Datalog rules
$\mathsf{h}(r)$	The head of a rule r
$\mathsf{b}(r)$	The body of a rule r
B_i	one of the body atoms
$\mathsf{var}(r)$	variables appearing in r
σ	A substitution mapping variables to constants
E	A set of initial facts (input database)
I	A set of materialised facts
$r[I]$	The set of facts derived by applying rule r to fact set I
$\Pi[I]$	The set of facts derived by applying all rules in Π to fact set I
Δ	The set of newly derived facts in the current iteration
$r[I, \Delta]$	Semi-naïve application of rule r using newly derived facts Δ
$\Pi[I, \Delta]$	Semi-naïve application of all rules using Δ
E^+	The set of newly added facts in an update
E^-	The set of deleted facts in an update
$\Pi^\infty[E]$	The fully materialised fact set derived from the initial fact set E
$\Pi^\infty[E \backslash E^- \cup E^+]$	The updated materialised fact set after applying updates

Table 2.1: Summary of Mathematical Notations

Chapter 3

Literature Review

This chapter provides a literature review on materialisation computation and storage techniques relevant to Datalog reasoning systems. We begin by examining existing approaches for the computation of materialisations in Section 3.1, focusing on methods that efficiently derive consequences from Datalog rules. We then explore incremental maintenance algorithms in Section 3.2, which update materialisations in response to changes in the dataset, ensuring consistency without redundant recomputation. Next, we investigate customised rule applications in Section 3.3 and discuss how our research questions RQ1-RQ2 contribute to this area by introducing specialised evaluation techniques for complex cyclic rules and exploring their integration with standard reasoning strategies. The chapter then delves into storage of materialisations in Section 3.4, highlighting research gaps in storage techniques for reasoning efficiency. We present how our research questions RQ3-RQ4 aim to address these gaps by proposing the multi-scheme framework and optimisations tailored to transitive closure and union rules. Finally, Section 3.5 reviews several representative modern Datalog engines and their associated reasoning techniques.

3.1 Computation of Materialisations

Research on optimising Datalog materialisation has primarily focused on improving efficiency through strategies such as reducing redundant computations [15, 127, 135], exploiting rule ordering [117], and parallel execution [104]. Bancilhon [15] introduced the seminaïve-based materialisation approach for recursively-defined relations. Ramakrishnan et al. [117] proposed a technique to reduce the number of rule applications by exploiting the monotonicity of rules (rule orderings). Motik et al. [104] proposed a parallel materialisation approach based on the seminaïve evaluation strategy, featuring an efficient workload distribution algorithm and a lock-free indexing structure.

Sun et al. [134] investigated the use of modern datacentre GPUs, such as the NVIDIA H100, which support high thread concurrency and high memory bandwidth, to accelerate Datalog materialisation.

More recent studies [127, 135] constructed a schedule to guide rule execution and minimise redundant computations during materialisation. Tsamoura et al. [135] introduced the concept of *Trigger Graphs* (TGs) to guide the execution of the rules, which can potentially avoid redundant computations during materialisation. Singh et al. [127] model the materialisation of recursive rules in a Datalog program as a scheduling problem on a directed acyclic graph (DAG), where the nodes represent tasks to be executed and the edges denote dependencies between these tasks. This approach effectively captures the intricate dependencies inherent in recursive Datalog rules, enabling a more structured and efficient execution order. By treating the materialisation process as a DAG-based scheduling problem, this method enhances the handling of rule dependencies, thus improving overall performance. LogicBlox [14], a commercial Datalog engine, employed a similar scheduler-based approach.

3.2 Incremental Maintenance Algorithms

In this section, we focus on the workflow of incremental maintenance approaches for Datalog materialisation, which can be broadly classified into counting-based and DRed-based methods. These approaches aimed to efficiently handle updates by minimising redundant computations and ensuring the correctness of the materialised fact base.

Counting-based Algorithms: As discussed in Section 2.3, maintaining materialisations under incremental deletion presents significant challenges due to the need to identify dependency chains among facts to ensure that no invalid facts remain in the materialisation. This process is computationally intensive, as it necessitates tracking each fact's provenance to determine its validity. To address this complexity, the *counting* algorithm was introduced, which maintains a count of the number of derivations for each fact. When a fact's count drops to zero, it indicates that the fact is no longer supported by the current dataset and can be safely removed. Based on this idea, Nicolas and Yazdanian [112], and Gupta et al. [67] proposed counting-based approaches for non-recursive rules, but not supporting recursive programs. Dewan et al. [47], Gupta et al. [66], Motik et al. [108] proposed several variants of the counting algorithm to accommodate recursive rules, extending the applicability of the approach to more complex Datalog programs. The approach proposed by Dewan et al. [47] is

based on the naïve evaluation strategy, which is inherently inefficient, as discussed in Section 2.2. The method described by Motik et al. [108] adapts the counting approach to a more efficient seminaïve evaluation strategy. While counting-based algorithms require additional space to store the counters, they have been shown to be the most efficient solution for handling non-recursive rules [108], as the number associated with each fact can accurately reflect its remaining proofs.

DRed-based Algorithms: Another class of incremental maintenance algorithms is based on the Delete and Re-derive (DRed) procedure, which first deletes all consequences of the deleted facts, and then rederives facts that are still derivable after the update. Following this idea, Ceri and Widom [36] proposed a maintenance algorithm specifically for non-recursive Datalog programs, laying the foundation for handling updates through deletion and re-derivation. Subsequently, Gupta et al. [67] introduced a general DRed algorithm capable of handling both recursive and non-recursive Datalog programs. Staudt and Jarke [128] presented a fully declarative formulation of the DRed approach, where a maintenance Datalog program is developed to compute the updates to the materialisation. However, a key limitation of this formalisation is its inability to support non-repetitive reasoning. This leads to inefficiencies in scenarios where redundant computations could otherwise be avoided. Motik et al. [108] improved this formalisation by supporting non-repetitive evaluation and extending to programs with negation. However, DRed algorithms can be very inefficient if alternate derivations of a fact exist, as a fact (and all its consequences) will be deleted and re-derived during the maintenance. To address this limitation, Motik et al. [107] proposed the Backward/Forward (B/F) algorithm which checks for extra proofs of deleted facts using a combination of backward and forward chaining, instead of overdeleting all the derived consequences. Motik et al. [108] further extended the B/F algorithm to the *Forward/Backward/Forward* (FBF) algorithm, which uses a strategy to determine the balance between overdeletion and backward chaining.

Combination of DRed and Counting: Counting-based approaches utilise maintained counters to track the number of derivations supporting each fact, allowing for efficient determination of whether a fact should be retained or removed when its supporting proofs are modified. In contrast, DRed-based approaches generally employ a 'backward' evaluation strategy to determine surviving proofs of deleted facts. For example, in line 16 of Algorithm 3, the computation of $D \cap \Pi[I]$ is typically performed by matching each deleted fact $t \in D$ to the head of applicable rules in Π, followed by evaluating the partially instantiated rule body as a query over the

remaining fact set I. Hu et al. [75] highlighted that the 'backward' evaluation used in traditional DRed-based approaches can be inefficient in certain scenarios and proposed the DRed^c and B/F^c algorithms to mitigate this issue by avoiding the need for backward evaluation during incremental maintenance. These algorithms combine the idea of counting and DRed approaches, leveraging the advantages of each. The key idea behind their approach is to maintain additional counters that track the number of recursive derivations for each fact. This information allows the system to efficiently determine whether a fact remains valid during the re-derivation phase, thereby reducing unnecessary computations and improving overall performance.

3.3 Customised Rule Application

This section explores customised rule application strategies tailored for different types of rules, aiming to enhance efficiency by leveraging the specific characteristics of each rule type during the materialisation process.

Subercaze et al. [131] employed customised graph-based algorithms to efficiently handle the materialisation of transitive and symmetric properties in RDFS-Plus. Similarly, Dong et al. [48] proposed an approach for incrementally updating query answers in regular chain Datalog programs by evaluating an equivalent non-recursive Datalog program transformed from the query. These approaches offer efficient solutions tailored to specific rule sets; however, they do not address how such specialised applications can be seamlessly integrated with arbitrary rule sets in a general materialisation framework.

In contrast, Motik et al. [105, 106] employed the *rewriting* technique to handle rules representing equality efficiently, and integrated this technique into both the materialisation and incremental maintenance processes, optimising the evaluation of arbitrary Datalog programs with equality. Hu et al. [77] introduced specialised algorithms tailored for transitive closure rules, symmetric transitive closure rules, and regular chain rules. They also proposed a general modular framework that enables the integration of different algorithms for various subsets of rules, enhancing flexibility and efficiency in the materialisation and incremental maintenance process. Their approach generalises the DRed^c workflow [75], which combines the benefits of counting-based and DRed techniques to improve incremental maintenance.

Our research question RQ1 addresses customised rule application, investigating specialised evaluation techniques for complex cyclic rules to support efficient materi-

alisation and incremental maintenance. Meanwhile, RQ2 explores the integration of this proposed technique with standard evaluation strategies during reasoning.

3.4 Storage of Materialisations

Efficient storage of materialised facts is essential for the scalability of Datalog-based reasoning systems. While general-purpose approaches have been explored [90], Section 3.4.1 focuses on storage strategies for RDF subsets, which are prevalent in Knowledge Graph applications. Due to their unique structural properties and access patterns, RDF data enables tailored optimisations for efficient storage and retrieval. Section 3.4.1 reviews advanced RDF storage solutions, assessing their suitability for reasoning tasks. Section 3.4.2 discusses challenges in storing and maintaining materialisations, including incremental updates, and examines advanced Datalog storage techniques to support efficient reasoning in dynamic knowledge graph environments.

3.4.1 RDF Storage Solutions

Many conventional RDF storage solutions are primarily designed to optimise query performance, focusing on efficient retrieval and indexing mechanisms to accelerate data access. We present several key studies in this area; interested readers can refer to [8, 38] for a more comprehensive overview of storage and indexing techniques for RDF data aimed at efficient query processing.

Triple Table: One of the most common approaches is the *triple table*, also known as the *statement table*, which is the most straightforward way to map RDF data to relational databases [20]. In this format, RDF data is stored in a single table with three columns—subject, predicate, and object—where each row represents an individual RDF statement, as shown in Table 3.1. Systems like Sesame [31] and 3-Store [70]

Subject	**Predicate**	**Object**
Alice	rdf:type	Person
Bob	rdf:type	Person
Alice	age	26
Bob	age	21
Alice	knows	Bob
Bob	knows	Alice

Table 3.1: Example Triple Table Representation.

employed this approach. While this schema provides simplicity and compatibility

with traditional database systems, it often results in inefficient join operations and scalability issues in complex queries [38]. Additionally, as the number of triples grows, the table size increases rapidly, affecting querying and reasoning performance.

Property Table: To address some of the limitations of the triple table, systems such as Jena [99, 119, 151] utilised the *property table* (PT) approach. This approach stores triples in wide horizontal tables with n-ary columns, grouping subjects with similar properties into the same table. For example, Table 3.2 organises the triples from Table 3.1 into a property table for the *Person* type, where each property (*age* and *knows*) is represented as a column. This structure avoids redundancy by storing properties as columns, reducing table size; and enables more efficient retrieval for entities of the same type with consistent attributes. For example, for subjects of the type *Person*, they typically has properties such as *age* and *knows*. Retrieving all persons with their properties specified is more efficient in this structure, as each property is stored as a column, enabling fast access to various properties for a given subject. However, the table can be sparse with multiple null values if different entities have varying properties, leading to wasted storage space. In summary, these

Subject	**Age**	**Knows**
Alice	26	Bob
Bob	21	Alice

Table 3.2: Example Property Table Representation for the *Person* class.

approaches improve query efficiency by leveraging schema knowledge, but they still face challenges in handling dynamic updates, as schema changes and fact insertions or deletions can lead to costly modifications and reorganisation of data. This limitation reduces the suitability of the property table approach for reasoning tasks, particularly when dynamic schema evolution or complex rule evaluations are required.

Vertical Partitioning: Another widely adopted approach is *vertical partitioning* (VP) [1], which divides RDF statements into multiple tables based on their predicates. Each table stores subject-object pairs for a specific predicate, enabling efficient lookups for queries involving a single or few predicates. For example, Table 3.3a and Table 3.3b organises triples in Table 3.1 in two tables for the predicate *age* and *knows*, respectively. Vertical partitioning is particularly efficient for answering queries in which the property resource is bound [1]. The subjects can be ordered within each table, and additional indexes can be maintained to optimise query performance for various query patterns in which the subject and/or the object is bound. However,

Subject	Object
Alice	26
Bob	21

(a) Vertical Partitioning: *age* Table

Subject	Object
Alice	Bob
Bob	Alice

(b) Vertical Partitioning: *knows* Table

Table 3.3: Example of Vertical Partitioning

vertical partitioning encounters similar challenges to property tables when dealing with fact changes in dynamic scenarios. In particular, updating or deleting facts can be costly, especially when the subjects are ordered or when extra indexes are maintained to support efficient access. These maintenance costs can significantly impact performance, limiting the applicability of vertical partitioning in reasoning scenarios where frequent updates are common.

Multiple-Indexing: Several *multiple-indexing* solutions have been proposed to enhance the efficiency of RDF data storage and retrieval of various access patterns. Harth and Decker [72] proposed an RDF storage approach that leverages multiple indexing strategies to enhance query performance. Specifically, they introduced six distinct indices to accommodate all possible access patterns, ensuring efficient retrieval regardless of whether the subject (s), predicate (p), or object (o) is specified or treated as a variable. Abadi et al. [1] suggested that an optional index can be constructed to the object columns in the vertical partitioning scheme (e.g., using an unclustered B+ tree) to support access patterns in which the object is specified. Weiss et al. [150] proposed six-fold sorted indexes based on the vertical partitioning scheme for a high degree of data compression and efficient merge joins. Similarly, Neumann and Weikum [110] proposed a storage approach that utilises six indexes built on the triple table, which is stored in a (compressed) clustered B-tree.

Graph-based Storage: More advanced storage techniques employ *graph-based storage* to store RDF data efficiently. As highlighted in Section 2.4, RDF data naturally represents a graph structure where triples form labelled edges between nodes. Graph-based storage systems take advantage of the local repetitions and structural regularities often found in graphs, allowing for effective compression using adjacency lists. Examples of such systems include gStore [159], SpiderStore [26], Trinity.RDF [154], and GRaSS [96]. These systems leverage graph structures to enable sub-graph matching, supporting advanced query processing over RDF data.

While the approaches discussed above demonstrate high efficiency in static data scenarios, they are less effective in dynamic environments, such as those encountered during materialisation and incremental maintenance. In these dynamic scenarios,

data is continuously updated, making it costly to maintain sorted data structures and indexes, which are typically optimised for static datasets.

3.4.2 Datalog Storage Solutions for Materialisation Efficiency

The materialisation process introduces unique challenges for storage solutions, as it requires efficient management of both data retrieval and the insertion of newly derived facts. As materialisation progresses, the storage system must support frequent read and write operations, ensuring that derived facts are efficiently integrated into the existing fact base. Moreover, the seminaïve evaluation, a widely used optimisation technique introduced in Section 2.2, further complicates storage requirements by necessitating selective access to different portions of the data at each iteration. This involves tracking newly derived facts separately from previously materialised facts to avoid redundant computations and to enable incremental updates. Consequently, an effective storage solution must provide fast lookups, efficient insertions, and mechanisms to distinguish between old and new facts while maintaining scalability and consistency throughout the materialisation process.

Incremental maintenance presents even greater challenges, as it requires not only efficient insertion of new facts but also the ability to handle deletions and updates with minimal recomputation. Managing incremental deletions is particularly complex, as it necessitates tracking dependencies between facts to ensure that deletions do not inadvertently invalidate facts that can still be derived from other sources.

Several investigations have explored storage techniques to enhance reasoning efficiency. Motik et al. [104] proposed a hash-based indexing scheme organised as lists, which optimises insertion and deletion operations over row-oriented tuple storage. This design effectively supports index nested loop joins, facilitating efficient random memory access and enabling parallel execution. Their study demonstrated that this storage architecture significantly improves the performance of seminaïve evaluation by enabling fast lookups and efficient join operations, making it particularly well-suited for large-scale materialisation tasks. This technique is implemented in RDFox [109], a highly scalable in-memory RDF store and Datalog reasoning engine known for its parallel reasoning capabilities. Soufflé [85], a high-performance Datalog engine designed for static program analysis, also employs a similar row-based storage approach.

To enhance join efficiency and overall reasoning performance, Urbani et al. [142] proposed the use of column-based data structures to support efficient in-memory seminaïve evaluations. Hu et al. [76] introduced optimisations to join operations within columnar storage and proposed a technique called *structure sharing*. This

method reduces memory consumption by storing common parts of derived facts only once, thereby enhancing the efficiency of materialisation and minimising redundant data storage. While conceptually similar to RETE-style structure sharing [55], this method is applied at the storage level rather than for rule matching. Column-oriented storage has been successfully adopted in high-performance reasoning engines such as VLog [35] and Nemo [80], demonstrating its effectiveness in supporting complex join operations and large-scale knowledge graph reasoning.

Further advancements in Datalog storage have been achieved by leveraging modern hardware architectures. Shovon et al. [126] and Sun et al. [133] explore row-oriented Datalog storage optimised for GPU acceleration, demonstrating how efficient memory access patterns and parallelism can significantly improve join and materialisation performance. Extending this approach, FVLOG [134] introduces a column-oriented GPU data structure that efficiently supports essential relational algebra operations, including projection and selection on GPUs. This design capitalises on the high memory bandwidth and massive parallelism of modern GPUs, achieving substantial performance gains over traditional CPU-based storage solutions.

However, these works primarily focus on general storage solutions applicable to all data, whereas our research question RQ4 emphasises customised storage solutions tailored to the derivations of specific rule sets. By leveraging the inherent properties of Datalog programs, our approach has the potential to offer more efficient and optimised storage strategies. Furthermore, existing studies are largely confined to materialisation and do not explore how different data structures can be utilised to support incremental maintenance procedures effectively. In contrast, our research question RQ3 focuses on a comprehensive framework that integrates multiple storage schemes, accommodating both the materialisation and incremental maintenance processes to achieve greater flexibility and efficiency.

3.5 Modern Datalog Engines

In this section, we briefly review several representative modern Datalog systems and highlight their core techniques for reasoning over Datalog programs. For a comprehensive survey of Datalog engines and their optimisation techniques, we refer the reader to [90]. Modern Datalog systems adopt diverse reasoning strategies to address different application scenarios. For example, systems such as Soufflé [85] are widely used for static program analysis, where high-performance batch materialisation is crucial. Engines like DDlog [122] target low-latency data analysis and dynamic

streaming scenarios, leveraging differential dataflow to efficiently handle incremental updates. Systems such as RDFox [109] focus on semantic reasoning and knowledge graph processing, where advanced materialisation-based strategies and support for deletions are essential. Engines like GDlog [133] focus on GPU-based optimisation for Datalog reasoning, enabling high-throughput deductive analytics on modern hardware. Similarly, SociaLite [125] is designed for large-scale distributed graph analysis, providing high-level abstractions and parallel execution support. In the following, we review several representative reasoning techniques and introduce corresponding Datalog engines that exemplify these approaches.

Seminaïve-based: A widely adopted materialisation strategy in modern Datalog systems is based on the seminaïve algorithm. As introduced in Section 2.2.2, this approach incrementally computes the fixpoint of Datalog programs by maintaining and propagating only the delta - the newly derived facts in each iteration - thereby avoiding redundant derivations and improving performance over naïve evaluation. The seminaïve evaluation remains the foundation of several high-performance Datalog engines, often combined with system-level optimisations to further enhance scalability.

Soufflé [85] is a prominent example of a seminaïve-based system, designed for static program analysis. It compiles Datalog programs into optimised `C++` code, enabling high-performance batch reasoning. Key optimisations include parallel compilation [87], specialised data structures [86], automatic index selection [13], and feedback-directed join optimisation [12], which collectively contribute to its efficiency on large codebases. While Soufflé supports incremental additions, it doesn't currently support incremental deletions, making it primarily suited for static analysis tasks.

GDlog [133] extends the seminaïve approach to GPU-accelerated reasoning. It leverages SIMD/CUDA/HIP-based GPU acceleration and employs advanced in-memory data structures, such as Hash-Indexed Sorted Arrays (HISA) and temporarily materialised joins, to support high-throughput deductive analytics on GPUs. These features enable GDlog to efficiently process large-scale reasoning tasks that benefit from the parallelism and memory bandwidth of modern GPUs. Similar to Soufflé, GDlog does not support incremental deletions, and is designed mainly for batch-oriented, high-throughput reasoning tasks.

Differential Dataflow: Another strategy is based on *Differential Dataflow* [100], which supports efficient incremental computation over dynamic data. Instead of recomputing entire materialisations upon updates, Differential Dataflow propagates differences (deltas) through the dataflow graph, enabling minimal recomputation and

low-latency responsiveness. This model is particularly well-suited for applications where frequent updates occur, such as streaming or interactive data analysis.

DDlog [122] is a modern Datalog engine built on top of Differential Dataflow. The DDlog compiler translates DDlog programs to Differential Dataflow programs. It employs automated incremental maintenance for both insertions and deletions, and represents relations as in-memory collections (sets, vectors, maps). DDlog integrates closely with the Rust ecosystem, benefiting from its performance and safety guarantees. The engine provides high-throughput and low-latency reasoning capabilities for dynamic data, and is often used to implement embedded deductive databases in data-centric systems.

However, while DDlog offers efficient incremental reasoning for dynamic data, it also presents several limitations. Its memory footprint can grow significantly due to the maintenance of differential state, making it less suited for large-scale static datasets. Furthermore, DDlog's support for advanced Datalog features such as complex aggregates or rich ontology reasoning is currently limited. The engine is designed for single-machine execution and lacks built-in support for distributed reasoning, in contrast to systems like RDFox or LogicBlox. Finally, as a relatively young project, DDlog's ecosystem and tooling are still maturing compared to more established Datalog engines.

Materialisation Maintenance-based: Another type of Datalog engine adopts advanced materialisation computation and maintenance strategies, where the entire set of derived facts is explicitly materialised and maintained. This approach can provide efficient query answering for workloads where repeated queries over largely static or moderately changing data are expected. To support dynamic updates, modern materialisation-based systems often implement advanced incremental maintenance techniques and optimised join processing.

RDFox [109] is a high-performance in-memory engine designed for knowledge graph reasoning. It employs a parallel materialisation strategy with a variant of the DRed algorithm to efficiently handle both insertions and deletions [104]. RDFox utilises shared-memory parallelism and lock-free updates, with additional optimisations such as compact hash-based indexes and equality handling [106]. The engine is widely used for semantic reasoning over RDF and OWL 2 RL knowledge bases.

LogicBlox [14] integrates materialisation-based reasoning with advanced enterprise analytics. Its core language, *LogiQL*, extends Datalog with features such as stratified negation, aggregation, and a rich module system. LogicBlox implements advanced incremental maintenance, including a scheduling approach [127], and novel

join strategies such as *Leapfrog Triejoin* [146]. The system supports enterprise-scale applications, particularly in prescriptive and predictive analytics, where fast reasoning over evolving data is required.

This line of work typically focuses on developing advanced incremental maintenance techniques and optimised evaluation algorithms to improve the efficiency of Datalog reasoning. This thesis is designed to enhance materialisation-based Datalog engines by investigating efficient approaches for handling cyclic rules (Research Questions RQ1–RQ2) and by exploring specialised storage solutions (Research Questions RQ3–RQ4).

3.6 Summary

This chapter reviewed existing research on Datalog materialisation and incremental maintenance in Sections 3.1-3.2, focusing on customised rule applications in Section 3.3 and storage solutions in Section 3.4. Customised rule application techniques, such as graph-based algorithms for transitive and symmetric properties, were discussed. Conventional RDF storage methods, such as triple tables, property tables, and vertical partitioning, were examined, highlighting their efficiency for query processing but limitations in handling dynamic updates. Advanced storage solutions tailored for materialisation are often based on columnar storage techniques, which offer high query performance but are generally not well-suited for incremental maintenance, especially when handling deletions or updates.

In the context of optimising Datalog materialisations in terms of computation, storage, and maintenance, significant challenges remain in seamlessly integrating diverse storage schemes and rule-specific optimisations within a unified framework that effectively supports both materialisation and incremental updates. This thesis seeks to address these challenges by proposing a flexible multi-scheme framework that decouples storage from reasoning, enabling efficient, scalable, and adaptable Datalog processing.

Chapter 4

Hypertree Decomposition-based Optimisation for Cyclic Rules

This chapter addresses research questions RQ1 and RQ2, focusing on the development of hypertree decomposition-based evaluation techniques and their integration within a broader reasoning framework. Based on our publication [156], this chapter presents methods for efficiently handling cyclic rules using hypertree decomposition and explores how these techniques can be combined with standard evaluation strategies to optimise materialisation and incremental reasoning.

4.1 Introduction

As highlighted in Section 3.2, while (incremental) materialisation research has largely focused on general workflows, existing algorithms often implicitly rely on traditional join plans for evaluating rule bodies. However, such join plans can be suboptimal in many cases [62, 111], resulting in inefficiencies that hinder the overall performance of materialisation processes, especially in the case of cyclic rules. This can lead to a blow-up in the number of intermediate results and a corresponding degradation in performance (as we will demonstrate in Section 4.6). This phenomenon can be observed in real-life applications, for example where rules are used to model complex systems, which may include the evaluation of numerical expressions.[1] The resulting rules are often cyclic and have large numbers of body atoms.

One promising solution from the database literature deals with such cyclic dependencies using *hypertree decomposition* (HD) [62], a method for structuring a hypergraph into a tree-like form. Hypertree decomposition is able to decompose cyclic

[1] https://2021-eu.semantics.cc/graph-based-reasoning-scaling-energy-audits-many-customers

queries, and Yannakakis's algorithm [153] can then be used to achieve efficient evaluation over the decomposition [62]. This method has been well-investigated with its effectiveness shown in many empirical experiments for query evaluation [2, 136].

It is unclear, however, whether the hypertree decomposition approach can benefit rule evaluation in Datalog reasoning. Unlike query answering, which requires only a single evaluation via decomposition, rules in a Datalog program are applied multiple times until no new data can be derived. In this setting, it is important to avoid repetitive derivations, but this is not easy to achieve when hypertree decomposition is used for rule evaluation. Moreover, incremental materialisation usually depends on efficiently tracking fact derivations, and it is unclear how to achieve this when such derivations depend on hypertree decomposition. Finally, hypertree decomposition introduces some additional overhead, and this may degrade performance on simple rules. A combined approach that selectively applies hypertree decomposition to complex rules while retaining standard evaluation methods for simpler cases may be necessary to balance efficiency and overhead.

Overview: In this chapter, we introduce a Datalog reasoning algorithm that leverages hypertree decomposition to enable efficient (incremental) reasoning for recursive programs. Specifically, Section 4.2 defines the key notations and concepts of cyclic rules and the hypertree decomposition, providing the foundational understanding needed for subsequent discussions. Section 4.3 demonstrates the advantages of hypertree decomposition in rule evaluations through a detailed example and highlights the challenges of adapting HD-based approaches to ensure non-repetitive reasoning and support incremental maintenance. The core hypertree decomposition-based evaluation algorithms, designed based on the seminaïve evaluation strategy, are presented in Section 4.4. Specifically, this section discusses how efficient incremental evaluation and principled tracking of fact derivations are enabled by algorithms designed to evaluate over the structural decomposition provided by the hypertree decomposition. In addition, Section 4.5 introduces a DRed algorithm enhanced with HD-based evaluations, showing how it integrates seamlessly with the standard seminaïve evaluation in a modular framework. This integration avoids unnecessary overhead for simple rules to maintain efficiency. Our empirical evaluation, detailed in Section 4.6, demonstrates that this combined approach significantly outperforms the standard method, achieving improvements by orders of magnitude. Finally, Section 4.7 discusses applications of hypertree decomposition in related fields. For reproducibility, our test system and

data are publicly available online.[2] Additional proofs are provided in Appendix A.

4.2 Preliminaries

In this section, we introduce the necessary notations and concepts that form the foundation for the techniques discussed in this chapter. First, in Section 4.2.1, we formally define cyclic rules by analysing their variable dependency hypergraph and identifying cycles that complicate their evaluation. Next, in Section 4.2.2, we present the concept of hypertree decomposition, a critical tool for transforming cyclic structures into acyclic representations that facilitate efficient reasoning. Section 4.2.3 then describes how typical query evaluation can be performed using hypertree decomposition, focusing on its ability to minimise redundant computations.

4.2.1 Cyclic Rules

The limitations discussed in Section 4.1 primarily concern cyclic rules, which are particularly challenging in Datalog reasoning. Therefore, we formally present the definition of cyclic rules as follows to provide a clear foundation for the techniques discussed in this chapter. We first define the variable dependency hypergraph of a rule r and the join tree of a hypergraph.

Definition 1 *The variable dependency hypergraph of r, denoted as $\mathcal{H}_{\mathsf{b}(r)} = (X, E)$, as follows:*

- *X is the set of vertices, corresponding to the variables in r.*
- *E is the set of hyperedges, where each hyperedge corresponds to the variables appearing in a single atom of r (i.e., for each atom $B_i \in \mathsf{b}(r)$, there is a hyperedge containing the variables in B_i).*

A *join tree* $JT(\mathcal{H})$ for a hypergraph $\mathcal{H}$ is a tree whose vertices correspond to the hyperedges of $\mathcal{H}$, such that for any variable $x \in X$ appearing in two hyperedges E_1, E_2 of $\mathcal{H}$, x must also appear in every vertex along the unique path connecting E_1 and E_2 in $JT(\mathcal{H})$. In other words, the set of vertices in which the variable x occurs induces a connected subtree of the join tree $JT(\mathcal{H})$. Finally, we present the definition of cyclic rules as follows.

[2] https://xinyuezhang.xyz/HDReasoning/

Definition 2 *A Datalog rule r is said to be cyclic if its variable dependency hypergraph $\mathcal{H}_{\mathsf{b}(r)} = (X, E)$ does not admit any join tree.*

This definition refers to the α-acyclicity, as defined in [17, 22, 97], which states that a hypergraph $\mathcal{H}_{\mathsf{b}(r)} = (X, E)$ is α-acyclic if and only if it has a join tree. The α-acyclicity offers a flexible way to capture variable dependencies among rule atoms in hypergraphs. It naturally extends the concept of cycles from graphs to hypergraphs, making it a fitting choice in this context. Readers interested in exploring other definitions of cycles in hypergraphs, such as simple cycles or conformal cycles, may refer to [88] for a comprehensive overview.

4.2.2 Hypertree Decomposition

We now define the hypertree decomposition for Datalog rules, following the definition of hypertree decomposition for conjunctive queries [60].

Definition 3 *For a Datalog rule r, a hypertree decomposition is a hypertree $HD = \langle T, \chi, \lambda \rangle$ in which $T = \langle N, E \rangle$ is a rooted tree, and χ associates each vertex $p \in N$ with a set of variables that appear in r, i.e., in $\mathsf{var}(r)$; whereas λ associates p with a set of atoms in the body of the rule r, i.e., in $\mathsf{b}(r)$.*

This hypertree T satisfies all the following conditions:

1. for each body atom $B_i \in \mathsf{b}(r)$, there exists $p \in N$ such that $\mathsf{var}(B_i) \subseteq \chi(p)$;
2. for each variable $v \in \mathsf{var}(r)$, the set $\{p \in N \mid v \in \chi(p)\}$ induces a connected subtree of T.
3. for each vertex $p \in N$, $\chi(p) \subseteq \mathsf{var}(\lambda(p))$.
4. for each vertex $p \in N$, $\mathsf{var}(\lambda(p)) \cap \chi(T_p) \subseteq \chi(p)$ in which T_p is the subtree of T rooted at p.

Hypertree width measures the degree of cyclicity of hypergraphs [123]. The *width* of the hypertree decomposition is defined as $max_{p \in N}|\lambda(p)|$. The *hypertree-width hw(r)* of r is the minimum width over all possible hypertree decompositions of the rule r. Please note that the acyclic hypergraphs are precisely those hypergraphs having a hypertree with its hypertree-width as one [123]. Therefore, a cyclic rule can also be defined as a rule whose hypertree width $hw(r)$ is greater than 1.

4.2.3 Query Evaluation via HD

We will now introduce how query evaluation works using a decomposition as join plan. Query evaluation via hypertree decomposition is a well-investigated problem in the database literature [54, 62], and such a process typically consists of *in-node evaluation* and *cross-node evaluation*. During in-node evaluation, each node p in the decomposition joins the body atoms that are assigned to it (i.e., $\lambda(p)$) and stores the join results for later use.

Then, cross-node evaluation applies the *Yannakakis* algorithm [153] to the join results for each node p, using the hypertree decomposition T as the join tree. The Yannakakis algorithm provides an efficient evaluation strategy for acyclic conjunctive queries, running in polynomial time in combined complexity. In the cyclic setting, the hypertree decomposition yields an acyclic structural representation of cyclic rules, and the Yannakakis algorithm provides an efficient evaluation strategy over this acyclic structure. The standard Yannakakis algorithm has two steps. The *full reducer* stage applies a sequence of bottom-up left semi-joins through the tree, followed by a sequence of top-down left semi-joins using the same fixed root of the tree [21]. This removes dangling data that will not be needed in the second stage and decreases the join result size for each node. The *cross-node join* stage joins the nodes bottom-up, and it projects to the output variables, i.e., $\mathsf{var}(\mathsf{h}(r))$, to obtain the final answers.

Overall, the (combined) complexity of query evaluation via a decomposition tree is known to be $O(v \cdot (m^k + s) \cdot log(m + s))$ [62] where v is the number of variables in the query, m is the cardinality of the largest relation in the data, k is the hypertree width of r, and s is the output size.

However, it is important to note the limitations of applying the Yannakakis algorithm to purely acyclic cases. Although the algorithm was originally developed for acyclic conjunctive queries and offers strong theoretical runtime guarantees, it is rarely used in practice due to a large hidden constant factor [149]. In our setting, Yannakakis serves as an effective evaluation strategy when operating over the acyclic decompositions of cyclic rules. However, it is not necessarily optimal for purely acyclic cases and may introduce additional overhead. This motivates a combined approach that applies the HD-based evaluation to cyclic rules, while using the standard approach for non-cyclic ones. We will discuss this design choice in more detail in Section 4.5, and the experimental results in Section 4.6 will empirically validate this analysis.

Notation	Description
$\mathcal{H}_{\mathsf{b}(r)} = (X, E)$	A hypergraph formed from body atoms of the rule r
$T =< N, E >$	A rooted tree with vertices N and edges E.
$HD =< T, \chi, \lambda >$	A hypertree decomposition, in which T is a rooted tree, χ and λ maps each vertex in T to a set of variables and body atoms, respectively.
p	A vertex in the rooted tree $T =< N, E >$, and $p \in N$.
$\chi(p)$	A set of variables assigned to p.
$\lambda(p)$	A set of atoms assigned to p.
$hw(r)$	The hypertree-width of the rule r.
$\mathsf{Add}^r(I, \Delta^+)$	HD-based incremental addition evaluation, computing $r[I, \Delta^+] \setminus I$.
$\mathsf{Del}^r(I, \Delta^-)$	HD-based incremental deletion evaluation, computing $r[I, \Delta^-] \cap (I \setminus \Delta^-)$.
$\mathsf{Red}^r(I, \Delta)$	HD-based rederivation evaluation, computing $r[I] \cap \Delta$.
$\Pi_p[I, \Delta]$	Incremental evaluator of a node p in the decomposition.
inst_p	A set of instantiations of p, with various superscripts indicating specific roles.

Table 4.1: Additional notation introduced in Chapter 4.

Finally, we present an additional notation table that introduces key symbols and terminologies used throughout this chapter. These notations supplement the preliminaries discussed earlier and are specifically tailored to the concepts and algorithms related to hypertree decomposition and cyclic rule evaluation. Please note that some of the notations in Table 4.1 will be formally defined and elaborated upon in the text in subsequent sections for a more detailed understanding.

4.3 Motivation and Challenges

In this section, we use an example to explain how hypertree decompositions could benefit rule evaluation and provide some intuitions as to how they can be exploited in the evaluation of recursive Datalog rules.

Example 1 *Consider the following rule r, in which* CA, CW *and* PC *represent* Coauthor, Coworker, *and* PossibleCollaborator *respectively:*

$$\mathsf{PC}(x, y) \leftarrow \mathsf{CW}(x, z_1), \mathsf{CA}(x, z_2), \mathsf{PC}(z_1, y), \mathsf{PC}(z_2, y).$$

Moreover, consider the dataset E as specified below, where n and k are constants.

Refer to Figure 4.1 for a (partial) illustration of the dataset and the joins.

$$\{\mathsf{CW}(a_i, b_{i\cdot k+j}) \mid 0 \leq i < n, \quad 1 \leq j \leq k\} \cup$$
$$\{\mathsf{CA}(a_i, c_{i\cdot k+j}) \mid 0 \leq i < n, \quad 1 \leq j \leq k\} \cup$$
$$\{\mathsf{CW}(a_n, a_2), \quad \mathsf{CA}(a_n, a_3)\} \cup$$
$$\{\mathsf{PC}(b_{i\cdot k+j}, d_j) \mid 0 \leq i < n, \quad 1 \leq j \leq k\} \cup$$
$$\{\mathsf{PC}(c_{i\cdot k+j}, d_j) \mid 0 \leq i < n, \quad 1 \leq j \leq k\}$$

Each relation above contains $O(n \cdot k)$ facts, and the materialisation will additionally derive $n \cdot k + k$ facts, i.e., $\{\mathsf{PC}(a_i, d_j) \mid 0 \leq i \leq n, \quad 1 \leq j \leq k\}$.

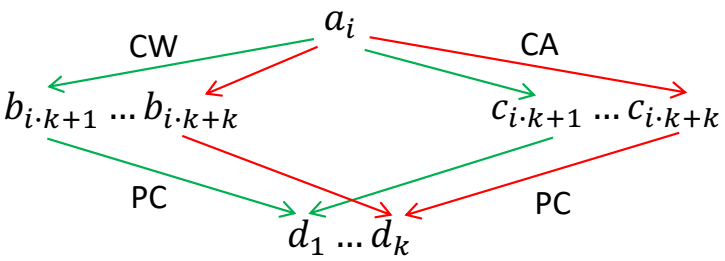


Figure 4.1: Illustration of dataset in Example 1, depicting joins related to a particular a_i $(0 \leq i < n)$ in the first round of rule evaluation.

Now consider the first round of rule evaluation, and assume that the rule body of r, which corresponds to a conjunctive query, is evaluated left-to-right. Then, matching the first three atoms involves considering $O(n \cdot k^2)$ different substitutions for variables x, y, z_1, and z_2; only $O(n \cdot k)$ of them will match the last atom and eventually lead to successful derivations. Please refer to Figure 4.2 for an illustration of generated substitutions of a_1 using this strategy. In fact, one can verify that no matter how we reorder the body atoms of r, it will result in similar behaviour.

Using hypertree decompositions could help process the query more efficiently. Consider decomposition T of the above query consisting of two nodes p_1 and p_2, where p_1 is the parent node of p_2. Furthermore, function χ is defined as: $\chi(p_1) = \{x, z_1, y\}$, $\chi(p_2) = \{x, z_2, y\}$, and function λ is defined as: $\lambda(p_1) = \{\mathsf{CW}(x, z_1), \mathsf{PC}(z_1, y)\}$, $\lambda(p_2) = \{\mathsf{CA}(x, z_2), \mathsf{PC}(z_2, y)\}$. Recall the steps of decomposition-based query evaluation we introduced in Section 4.2.3. During the *in-node evaluation* stage, each node in the decomposition will consider $O(n \cdot k)$ substitutions; the *full reducer* will consider $O(n \cdot k)$ substitutions and find out that nothing needs to be reduced; lastly, the *cross-node evaluation* joining p_1 and p_2 also considers $O(n \cdot k)$ substitutions. Compared with the left-to-right evaluation of the query, the overall cost of this approach

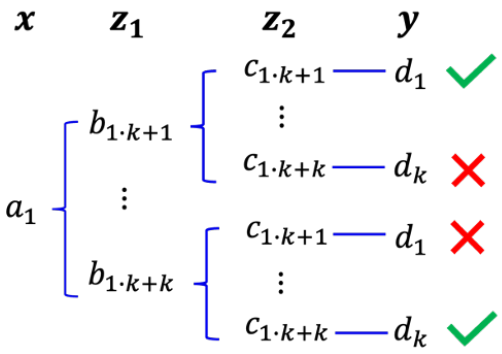


Figure 4.2: Illustration of substitutions in Example 1, depicting the k^2 generated substitutions for a_1 during the first round of rule evaluation using a left-to-right join strategy, where only k substitutions remain valid after evaluating the last atom.

is $O(n \cdot k)$, as opposed to $O(n \cdot k^2)$. For every a_i $(0 \leq i < n)$, the first round of rule application will introduce additional PC relations between a_i and d_1 to d_k ($n \cdot k$ in total).

Notice that rule r is recursive, so the facts produced by the first round of rule evaluation could potentially lead to further derivations of the same rule. This is indeed the case in our example: the first round derives all $\mathsf{PC}(a_i, d_j)$ facts with $0 \leq i < n$ and $1 \leq j \leq k$; combined with $\{\mathsf{CW}(a_n, a_2), \mathsf{CA}(a_n, a_3)\}$ this will additionally derive $\mathsf{PC}(a_n, d_j)$ with $1 \leq j \leq k$. If we used the hypertree decomposition-based technique discussed above, then a naïve implementation would just add all the facts derived in the first round to the corresponding nodes and run the decomposition-based query evaluation again. However, this is unlikely to be very efficient as it would have to repeat all the work performed in the first round of rule evaluation. Ideally, we would like to make the decomposition-based query evaluation algorithm 'incremental', in the sense that the algorithm minimises the amount of repeated work between different rounds of rule evaluation. As we shall see in Section 4.4, this requires nontrivial adaptation of in-node evaluation, as well as the two stages of the Yannakakis algorithm. Handling incremental deletion presents another challenge, which we address following the well-known DRed algorithm.

4.4 Hypertree Decomposition-based Rule Evaluation Algorithms

Recall that in Section 2.2, the seminaïve computation of materialisations relies on the seminaïve operator $\Pi[I, \Delta]$, which takes the current materialised fact set I and

the incremental changes Δ to propagate updates efficiently. The more generalised incremental maintenance algorithm, DRed, introduced in Section 2.3, also relies on such operators to handle deletions and re-derivations systematically.

Inspired by the seminaïve operator and the way DRed employs this operator, in this section, we introduce hypertree decomposition-based evaluation algorithms Del^r, Add^r, and Red^r for a cyclic rule r, which will be used in the reasoning procedure. More specifically:

- Add^r takes as input the current materialisation I and a set of newly added facts Δ^+. It computes $r[I, \Delta^+] \setminus I$, ensuring that only newly derived facts are added to the materialisation without reintroducing existing ones. This operator incrementally extends I by propagating the effects of Δ^+ through the rule r.

- Del^r takes as input the current materialisation I and a set of deleted facts Δ^-. It computes $r[I, \Delta^-] \cap (I \setminus \Delta^-)$, identifying the over-deleted facts that require further consideration. This operator determines which facts, originally derived using Δ^-, may no longer hold and need to be removed.

- Red^r takes as input the current materialisation I and a set of over-deleted facts Δ. It computes $r[I] \cap \Delta$, capturing the facts that can still be derived from the remaining materialisation. This operator is essential for rederiving facts that were mistakenly removed due to dependencies on deleted data.

These operators form the foundation for efficiently handling cyclic rules within the proposed reasoning framework. We will use DRed as the backbone of our algorithm (details presented in Section 4.5), but instead of standard reasoning algorithms with plan-based rule evaluation, we will use our hypertree decomposition-based functions Del^r, Add^r, and Red^r. For each rule r in Π, we assume its hypertree decomposition $\langle T^r, \chi^r, \lambda^r \rangle$ with $T^r = \langle N^r, E^r \rangle$ has already been computed, and t^r is the root of the decomposition tree T^r. Our reasoning algorithms are independent of decomposition methods.

4.4.1 Notation

First, analogously to expression (2.5), for each node $p \in N^r$ we define operator $\Pi_p[I, \Delta]$, in which I and Δ are sets of facts with $\Delta \subseteq I$.

$$\Pi_p[I, \Delta] = \{\chi^r(p)\sigma \mid \lambda^r(p)\sigma \subseteq I \text{ and } \lambda^r(p)\sigma \cap \Delta \nsubseteq \emptyset\},$$

Intuitively, this operator is intended to compute for a node p all the instantiations influenced by the incremental update Δ. Additionally, for each node $p \in N^r$, we define the following sets, which will be used in the presentation of our algorithms. These sets are initially empty before the first materialisation computation and are continuously maintained throughout the incremental maintenance procedure.

1. inst_p^I contains the join result of in-node evaluation for p under the current materialisation I, and it is represented as tuples for variables $\chi^r(p)$. Since cross-node evaluation builds upon such join results, to facilitate incremental evaluation and to avoid computing inst_p^I every time from scratch, inst_p^I has to be correctly maintained between different executions of DRed.

2. $\mathsf{inst}_p^{I \dot{:} \Delta^+}$ represents the set of instantiations that should be added to inst_p^I given a set of newly added facts Δ^+. This set can be obtained using the operator Π_p.

3. $\mathsf{inst}_p^{I \dot{:} \Delta^-}$ represents the set of instantiations that no longer hold after removing Δ^- from I; these instantiations should then be deleted from inst_p^I. Similarly as above, this set can be computed using Π_p.

4. inst_p^{ac} represents the currently active instantiations that will participate in the cross-node evaluation.

5. inst_p^{re} represents the instantiations that will need to be checked during the rederivation phase.

4.4.2 Addition

As discussed in Section 4.3, the decomposition-based query evaluation should be made incremental. To this end, Algorithm 4, which is responsible for addition, needs to distinguish between old instantiations and the new ones added due to changes in the explicitly given data. This is achieved by executing the *in-node evaluation* for each node $p \in N^r$ in line 3 using the Π_p operator.

Then, the *cross-node evaluation* (line 4) is performed in a way similar to the evaluation of $r[I, \Delta]$ outlined in Section 2.2, treating each node in T^r as a body atom. Specifically, as shown in Algorithm 5, we will evaluate the tree $|N^r|$ times. Assume that there is a fixed order among all the tree nodes for r, and let p_i, $1 \leq i \leq |N^r|$, denote the ith node in this ordering. Then, in the ith iteration of the loop of lines 3–9, node p_i is chosen in line 3, and the label of each node will be determined by the

Algorithm 4 HD-based Incremental Addition Evaluation: $\mathsf{Add}^r[I, \Delta^+]$

1: **Input:** Current materialisation I, and fresh additions Δ^+
2: **for** $p \in N^r$ **do**: ▷ in-node evaluation
3: $\quad \mathsf{inst}_p^{I \dot{:} \Delta^+} := \Pi_p[I, \Delta^+] \setminus \mathsf{inst}_p^I$
4: $\Delta_A := \mathsf{CrossNodeEvaluation}^r(\mathsf{L}^+)$ ▷ cross-node evaluation
5: **for** $p \in N^r$ **do** ▷ updating instantiations for each node
6: $\quad \mathsf{inst}_p^I := \mathsf{inst}_p^I \cup \mathsf{inst}_p^{I \dot{:} \Delta^+}$
7: $\quad \mathsf{inst}_p^{I \dot{:} \Delta^+} := \emptyset$
8: **return** $\Delta_A \backslash I$

Algorithm 5 Cross-Node Evaluation of Decomposition T^r: $\mathsf{CrossNodeEvaluation}^r(\mathsf{L})$

1: **Input**: A labelling function L
2: $\Delta := \emptyset$
3: **for** $p_i \in N^r$ **do** /* the Δ node */
4: $\quad$ **for** $p_j \in N^r$ **do**
5: $\quad\quad$ label p_j with the output of $\mathsf{L}(p_i, p_j)$
6: $\quad\quad$ set $\mathsf{inst}_{p_j}^{ac}$ according to the label
7: $\quad \mathsf{TopDownLSJ}(p_i)$
8: $\quad \mathsf{BottomUpLSJ}(t^r)$; $\mathsf{TopDownLSJ}(t^r)$
9: $\quad \Delta := \Delta \cup \pi_{\mathsf{var}(\mathsf{h}(r))}(\mathsf{CrossNodeJoin}(t^r))$
10: **return** Δ

labelling function L^+ as specified below. In particular, node p_i will be labelled Δ^+; nodes preceding and succeeding p_i will be labelled I and $I \cup \Delta^+$, respectively.

$$\mathsf{L}^+(p_i, p_j) = \begin{cases} I \cup \Delta^+, & p_j \prec p_i \\ \Delta^+, & p_j = p_i \\ I, & p_j \succ p_i \end{cases} . \tag{4.1}$$

Based on the labels assigned in line 5, we will set p_j^{ac}, the active instantiations that will participate in the subsequent evaluation, as follows. Note that the last two cases will be used later for deletion.

$$\mathsf{inst}_p^{ac} = \begin{cases} \mathsf{inst}_p^I & I \\ \mathsf{inst}_p^{I \dot{:} \Delta^+} & \Delta^+ \\ \mathsf{inst}_p^I \cup \mathsf{inst}_p^{I \dot{:} \Delta^+} & I \cup \Delta^+ \\ \mathsf{inst}_p^{I \dot{:} \Delta^-} & \Delta^- \\ \mathsf{inst}_p^I \backslash \mathsf{inst}_p^{I \dot{:} \Delta^-} & I \backslash \Delta^- \end{cases} . \tag{4.2}$$

After fixing the active instantiations, Algorithm 5 proceeds with an adapted version of the Yannakakis algorithm: lines 7–8 complete the *full reducer* stage whereas

line 9 performs the *cross-node join*. By performing left semi-joins between nodes, the full reducer stage aims at deactivating instantiations that do not join and keep only the relevant ones. The standard full reducer does not consider incremental updates so adaptations are required. In particular, our incremental version of the full reducer traverses the tree three times. The first traversal in line 7 consists of a sequence of top-down left semi-joins with p_i (the node labelled with Δ^+) as the root. As Δ^+ is typically smaller than the materialisation I, starting from p_i could potentially reduce the numbers of active instantiations for the other nodes to a large extent. The second and the third traversal (line 8) involves applying the standard bottom-up and top-down left semi-join sequences, respectively, using the root of the decomposition tree t^r as the root for the evaluation. Then, the cross-node join in line 9 evaluates the decomposition tree T^r bottom-up: for each node $p \in N^r$, it joins active instantiations in p with those in its children, and then projects the result to variables $\chi^r(p) \cup \mathsf{var}(\mathsf{h}(r))$. The join result obtained at the root t^r is projected to the output variables $\mathsf{var}(\mathsf{h}(r))$ to compute the derived facts, which are then returned to the Add^r function.

Lastly, lines 5–7 of Algorithm 4 update the instantiations inst_p^I for each node p and empty $\mathsf{inst}_p^{I \dot{:} \Delta^+}$ for later use.

By applying the principles of seminaïve evaluation to both the in-node evaluation and the cross-node evaluation, Add^r avoids repeatedly reasoning over the same facts or instantiations. Lemma 1 states that the algorithm is correct. The proof is provided in Appendix A.

Lemma 1 *The* $\mathsf{Add}^r[I, \Delta^+]$ *function shown in Algorithm 4 computes* $r[I, \Delta^+] \backslash I$.

To further elucidate the algorithmic process, we will build upon the examples presented in Section 4.3 to demonstrate our algorithm's recursive application in a step-by-step manner.

Example 2 *Following the initial round of rule application as detailed in Section 4.3, the instantiations in* $\mathsf{inst}_{p_1}^{I \dot{:} \Delta^+}$ *and* $\mathsf{inst}_{p_2}^{I \dot{:} \Delta^+}$ *are derived in line 3 of Algorithm 4 and then merged into* $\mathsf{inst}_{p_1}^I$ *and* $\mathsf{inst}_{p_2}^I$ *in line 6, respectively, before being cleared in line 7. Therefore, we have instantiations* $\mathsf{inst}_{p_1}^I = \{(a_i, b_{i \cdot k+j}, d_j) \mid 0 \leq i < n,\ 1 \leq j \leq k\}$, *and* $\mathsf{inst}_{p_2}^I = \{(a_i, c_{i \cdot k+j}, d_j) \mid 0 \leq i < n,\ 1 \leq j \leq k\}$. *Additionally, the cross-node evaluation in line 4 derives facts* $\{\mathsf{PC}(a_i, d_j) \mid 0 \leq i < n,\ 1 \leq j \leq k\}$, *which are all returned in line 8 of the algorithm.*

In the second round of application, the facts derived in the first round, i.e., $\mathsf{PC}(a_i, d_j)$ *with* $0 \leq i < n$ *and* $1 \leq j \leq k$, *are passed to the* Add^r *function as* Δ^+.

Then, line 3 identifies for p_1 the new instantiations involving facts in Δ^+; specifically, $\{(a_n, a_2, d_j) \mid 1 \leq j \leq k\}$ are assigned to $\mathsf{inst}_{p_1}^{I \dot{:} \Delta^+}$. Similarly, we have $\mathsf{inst}_{p_2}^{I \dot{:} \Delta^+} = \{(a_n, a_3, d_j) \mid 1 \leq j \leq k\}$. For an illustration of the related joins, please refer to figure 4.3 (1). Then, during the cross-node evaluation, lines 3–6 ensure that when node p_1 is labeled with Δ^+, node p_2 is labeled with I, and so $\mathsf{inst}_{p_1}^{I \dot{:} \Delta^+}$ is joined with $\mathsf{inst}_{p_2}^{I}$, deriving no new fact. In contrast, when node p_2 is labeled with Δ^+, node p_1 is labeled with $I \cup \Delta^+$, and so $\mathsf{inst}_{p_2}^{I \dot{:} \Delta^+}$ is joined with $\mathsf{inst}_{p_1}^{I} \cup \mathsf{inst}_{p_1}^{I \dot{:} \Delta^+}$, deriving $\mathsf{PC}(a_n, d_j)$ with $1 \leq j \leq k$. As one can readily see, the second round of rule application does not repeat work already carried out in the first round.

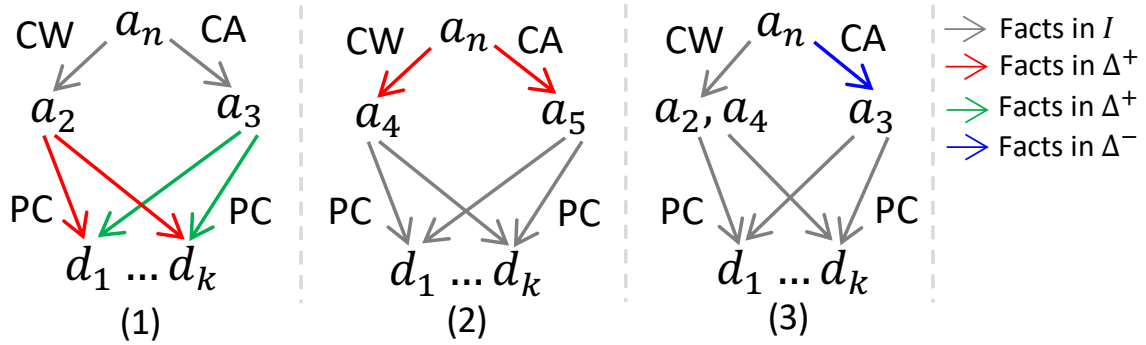


Figure 4.3: Illustrations of Example 2 and 3, depicting joins in three scenarios: (1) the second round of rule evaluation; (2) the incremental rule evaluation in response to the addition of $\mathsf{CW}(a_n, a_4)$ and $\mathsf{CA}(a_n, a_5)$; and (3) the incremental rule evaluation in response to the removal of $\mathsf{CA}(a_n, a_3)$. From these joins, one can easily compute the corresponding instantiations for nodes p_1 and p_2.

The above example demonstrates the process of initial materialisation. Now consider adding $\{\mathsf{CW}(a_n, a_4), \mathsf{CA}(a_n, a_5)\}$ to the explicitly given data, i.e., by setting E^+ to the above set of facts in the DRed algorithm. In this case, Δ^+ in Add^r will consist of $\mathsf{CW}(a_n, a_4)$ and $\mathsf{CA}(a_n, a_5)$. The addition of $\{\mathsf{CW}(a_n, a_4), \mathsf{CA}(a_n, a_5)\}$ introduces the following new rule instantiations:

$$\mathsf{CW}(a_n, a_4), \mathsf{CA}(a_n, a_5), \mathsf{PC}(a_4, d_j), \mathsf{PC}(a_5, d_j) \rightarrow \mathsf{PC}(a_n, d_j), \quad 1 \leq j \leq k.$$

The Add^r function processes new instantiations by updating the instantiations for the nodes p_1 and p_2 in the decomposition and incrementally joining the node instantiations. Specifically, in line 3 of Algorithm 4, we clearly have $\mathsf{inst}_{p_1}^{I \dot{:} \Delta^+} = \{(a_n, a_4, d_j) \mid 1 \leq j \leq k\}$, $\mathsf{inst}_{p_2}^{I \dot{:} \Delta^+} = \{(a_n, a_5, d_j) \mid 1 \leq j \leq k\}$, as illustrated by figure 4.3 (2). Then, facts $\mathsf{PC}(a_n, d_j)$ with $1 \leq j \leq k$ are derived in line 9 of Algorithm 4, but are then removed by "$\setminus I$" in line 8 since these facts are already in I. As a result, no

new fact is derived. This example demonstrates that the Add^r *function, following the seminaïve strategy, can evaluate rules incrementally with the given fact additions* Δ^+*, ensuring that each new instantiation is considered only once without redundantly reprocessing previously evaluated instantiations.*

4.4.3 Deletion

The Del^r algorithm shown in Algorithm 6 is analogous to Add^r, and it identifies consequences of r that are affected by the deletion of Δ^-. The algorithm first computes the overdeletion $\mathsf{inst}_p^{I \dot{:} \Delta^-}$ using the operator Π_p in lines 2–3. In addition, the instantiations that have been overdeleted are also added to inst_p^{re} so that they can be checked and potentially recovered during rederivation.

The *cross-node evaluation* in line 6 is similar to that of Add^r, except that a different labelling function L^- is used:

$$\mathsf{L}^-(p_i, p_j) = \begin{cases} I, & p_j \prec p_i \\ \Delta^-, & p_j = p_i \\ I \backslash \Delta^-, & p_j \succ p_i \end{cases} . \tag{4.3}$$

Note that the initialisation of $\mathsf{inst}_{p_j}^{ac}$ follows Equation (4.2). Finally, for each node p, the set of instantiations inst_p^I is updated in line 8 to reflect the change, and $\mathsf{inst}_p^{I \dot{:} \Delta^-}$ is emptied in line 9 for later use. Similarly as in Add^r, our Del^r function exploits the idea of seminaïve evaluation to avoid repeated reasoning. Lemma 2 states that the algorithm is correct. The proof is provided in Appendix A.

Lemma 2 *The* $\mathsf{Del}^r[I, \Delta^-]$ *function shown in Algorithm 6 computes* $r[I \dot{:} \Delta^-] \cap (I \backslash \Delta^-)$.

The following example illustrates overdeletion using our customised algorithms.

Example 3 *Assume that* E^- *is set as* $\{\mathsf{CA}(a_n, a_3)\}$ *in Algorithm 3. During the overdeletion phase,* E^- *is passed to* Del^r *as* Δ^-*. After the execution of line 3 in Algorithm 6, we have* $\mathsf{inst}_{p_2}^{I \dot{:} \Delta^-} = \{(a_n, a_3, d_j) \mid 1 \leq j \leq k\}$ *and* $\mathsf{inst}_{p_1}^{I \dot{:} \Delta^-} = \emptyset$*, as can be seen from figure 4.3 (3). Then, the cross-node evaluation will derive* $\mathsf{PC}(a_n, d_j)$*,* $1 \leq j \leq k$*. These facts are temporarily overdeleted, and the rederivation stage will check whether they have alternative derivations from the remaining facts.*

4.4.4 Rederivation

The rederivation step described in Algorithm 7 aims at recovering facts that are overdeleted but are one-step rederivable from the remaining facts using rule r. In the

Algorithm 6 HD-based Incremental Deletion Evaluation: $\mathsf{Del}^r[I, \Delta^-]$

1: **Input**: Current materialisations I, and fresh deletions Δ^-
2: **for** $p \in N^r$ **do**: ▷ in-node: overdelete
3: $\quad \mathsf{inst}_p^{I \dot{:} \Delta^-} := \Pi_p[I, \Delta^-] \cap \mathsf{inst}_p^I$
4: $\quad \mathsf{inst}_p^{re} := \mathsf{inst}_p^{re} \cup \mathsf{inst}_p^{I \dot{:} \Delta^-}$
5: /* cross-node: overdelete */
6: $\Delta_D := \mathsf{CrossNodeEvaluation}^r(\mathsf{L}^-)$ ▷ cross-node: overdelete
7: **for** $p \in N^r$ **do** ▷ updating instantiations for each node
8: $\quad \mathsf{inst}_p^I := \mathsf{inst}_p^I \backslash \mathsf{inst}_p^{I \dot{:} \Delta^-}$
9: $\quad \mathsf{inst}_p^{I \dot{:} \Delta^-} := \emptyset$
10: **return** $\Delta_D \cap (I \backslash \Delta^-)$

presentation of the algorithm we take advantage of an *oracle function* O which serves the purpose of encapsulation. The oracle function can be implemented arbitrarily, as long as it satisfies the following requirement: given a fact/tuple f, the oracle function returns true if f has a one-step derivation from the remaining facts/tuples, and it returns false otherwise.

In practice, there are several ways to implement such an oracle function. A straightforward way is through query evaluation. For example, to check whether a tuple $f \in \mathsf{inst}_p^{re}$ is one-step rederivable, one can construct a query using atoms in $\lambda(p)$, instantiate the query with the corresponding constants in f, and then evaluate the partially instantiated query over the remaining facts. A more advanced approach is through tracking derivation counts [75]: each tuple is associated with a number that indicates how many times it is derived; during reasoning, this count is incremented if a new derivation is identified, and it is decremented if a derivation no longer holds. Then, the oracle function can be realised with a simple check on the derivation count of the relevant tuple. We have adopted the second approach in this chapter.

Algorithm 7 proceeds as follows. First, lines 2–3 perform rederivation for in-node evaluation using the oracle. Recall that rule evaluation is decomposed into in-node evaluation and cross-node evaluation stages, so changes in the join results stored in the tree nodes have to be propagated through the decomposition tree, and this is achieved through line 4. Then, lines 5–7 update the join results and clear temporal variables. Finally, line 8 performs rederivation for cross-node evaluation and returns all the rederived facts. Lemma 3 states that the algorithm is correct. Together with Theorem 1 and Lemmas 1 and 2, this ensures the correctness of our approach. The proofs of Lemmas 1-3 are provided in Appendix A.

Algorithm 7 HD-based Rederivation Evaluation: $\mathsf{Red}^r[I, \Delta]$

1: **Input**: Current materialisation I, and over-deleted facts Δ to be examined
2: **for** $p \in N^r$ **do**
3: $\quad \mathsf{inst}_p^{I \dot{:} \Delta^+} := \{f \in \mathsf{inst}_p^{re} \mid \mathsf{O}[f] = \mathsf{true}\}$
4: $\Delta_R := \mathsf{CrossNodeEvaluation}^r(\mathsf{L}^+) \cap \Delta$
5: **for** $p \in N^r$ **do**
6: $\quad \mathsf{inst}_p^I := \mathsf{inst}_p^I \cup \mathsf{inst}_p^{I \dot{:} \Delta^+}$
7: $\quad \mathsf{inst}_p^{re} := \mathsf{inst}_p^{I \dot{:} \Delta^+} := \emptyset$
8: **return** $\Delta_R \cup \{f \in \Delta \mid \mathsf{O}[f] = \mathsf{true}\}$

Lemma 3 *The* $\mathsf{Red}^r[I, \Delta]$ *function shown in Algorithm 7 computes* $r[I] \cap \Delta$.

Below we continue with our running example and focus on the rederivation stage.

Example 4 *After the overdeletion in Example 3, we have* $\mathsf{inst}_{p_2}^{re} = \{(a_n, a_3, d_j) \mid 1 \leq j \leq k\}$. *These instantiations will not be recovered in line 3 of Algorithm 7 since the oracle O will find out that they have no alternative derivation from the remaining data. In contrast, the overdeleted facts* $\mathsf{PC}(a_n, d_j)$ *with* $1 \leq j \leq k$ *are recovered in line 8. This is so since each* $\mathsf{PC}(a_n, d_j)$ *can be rederived using instantiation* (a_n, a_2, d_j) *from* $\mathsf{inst}_{p_1}^I$ *and instantiation* (a_n, a_5, d_j) *from* $\mathsf{inst}_{p_2}^I$. *These rederived triples are passed on to* Add^r *as* Δ^+, *but no new fact will be derived. Overall, the removal of* $\{\mathsf{CA}(a_n, a_3)\}$ *do not affect the materialisation.*

4.5 Using HD-based Evaluations in Reasoning

In this section, we present an enhanced DRed procedure in Algorithm 8, which incorporates the HD-based evaluation algorithms introduced in Section 4.4. This version extends the non-repetitive and generalised DRed algorithm proposed by Hu et al. [77], enabling modular reasoning. Specifically, it allows different parts of the program to be evaluated using customised algorithms, making it more adaptable and suitable for the discussions below. In particular, as shown in lines 11, 16, and 25, rules in the program are evaluated on a per-rule basis, allowing for fine-grained control over the evaluation process. This design facilitates the use of specialised algorithms where needed while maintaining efficiency in standard cases.

The DRed algorithm is shown in Algorithm 8 where input arguments Π and E represent the program and the original set of explicitly given facts, I is the materialisation of Π w.r.t. E, and E^+ and E^- are the sets of facts that are to be added to

and deleted from E, respectively. As shown in line 2, the main idea behind DRed is to first *overdelete* all possible derivations that depend on E^-; and then the algorithm tries to *rederive* facts that have alternative proofs using the remaining facts; lastly, to *add* to the materialisation, the algorithm computes the consequences of E^+ as well as the rederived facts. This version of DRed incrementally computes the set of facts to be deleted, D, and the set of facts to be added, A, throughout the procedure. The materialisation I is then updated accordingly at the end, as shown in line 3.

Specifically, overdeletion involves recursively finding all the consequences derived by Π and E^-, directly or indirectly, as shown in lines 5–12. The function Del^r called in line 11 is intended to compute the facts that are directly affected by the deletion of Δ^-. More precisely, $\mathsf{Del}^r(I, \Delta^-)$ should compute $r[I, \Delta^-] \cap (I \backslash \Delta^-)$. Note that in line 11 the first argument of the call is $I \backslash D$, so it should compute $r[I', \Delta^-] \cap (I' \backslash \Delta^-)$ with $I' = I \backslash D$; the same clarification applies to Red^r and Add^r, so we will not reiterate.

The rederivation step recovers the facts that are overdeleted but are one-step provable from the remaining facts. Formally, function $\mathsf{Red}^r(I, \Delta)$ should compute $r[I] \cap \Delta$. Finally, during addition, the added set N_A is initialised in line 19, and then from line 20 to 25 the rules are iteratively applied, similarly as in the seminaïve algorithm. In this case, function $\mathsf{Add}^r(I, \Delta^+)$ is required to compute $r[I, \Delta^+] \backslash I$.

The correctness of Algorithm 8 is established by Theorem 1, which follows directly from the correctness of the modular update algorithm by Hu et al. [77]. This holds as our functions Del^r, Red^r, and Add^r produce valid derivations that conform to the bounds defined by Hu et al. [77].

Theorem 1 *Algorithm 8 correctly updates the materialisation I_∞ of Π w.r.t. E to I'_∞ of Π w.r.t. E' where $E' = (E \backslash E^-) \cup E^+$, provided that $\mathsf{Del}^r(I, \Delta^-)$, $\mathsf{Red}^r(I, \Delta)$, and $\mathsf{Add}^r(I, \Delta^+)$ compute $r[I, \Delta^-] \cap (I \backslash \Delta^-)$, $r[I] \cap \Delta$, and $r[I, \Delta^+] \backslash I$, respectively.*

Please note that the DRed algorithm could be used for the initial materialisation as well. To achieve this, we can set E, I and E^- as empty sets, and pass the set of explicitly given facts as E^+ to the algorithm. Therefore, both the initial materialisation and the incremental maintenance of materialisations can be achieved through multiple executions of the DRed algorithm, ensuring efficient updates while maintaining consistency in the fact base.

Combined Approach: With the modular nature of Algorithm 8, this approach can be easily adapted to employ a hybrid fact evaluation strategy: using the customised evaluation algorithms Add^r, Del^r, and Red^r introduced in Section 4.4 for cyclic rules,

Algorithm 8 Enhanced DRed Algorithm with HD-based Evaluations

Require: Datalog program Π, initial dataset E, fact additions E^+ and deletions E^-, materialisation I

Ensure: Update materialisation I from $\Pi^\infty[E]$ to $\Pi^\infty[(E \backslash E^-) \cup E^+]$

1: $D := A := \emptyset$, $E^- := (E^- \cap E) \backslash E^+$, $E^+ := E^+ \backslash E$
2: OVERDELETE , REDERIVE, ADD
3: $E := (E \backslash E^-) \cup E^+$, $I := (I \backslash D) \cup A$
4: **procedure** OVERDELETE
5: $N_D := E^-$
6: **loop**
7: $\Delta^- := N_D \backslash D$
8: **if** $\Delta^- = \emptyset$ **then break**
9: $N_D := \emptyset$
10: **for** $r \in \Pi$ **do**
11: $N_D := N_D \cup \mathsf{Del}^r(I \backslash D, \Delta^-)$
12: $D := D \cup \Delta^-$
13: **procedure** REDERIVE
14: $\Delta := \emptyset$
15: **for** $r \in \Pi$ **do**
16: $\Delta := \Delta \cup \mathsf{Red}^r(I \backslash D, D)$
17: $\Delta := \Delta \cup ((E \backslash E^-) \cap D)$
18: **procedure** ADD
19: $N_A := (\Delta \cup E^+) \backslash (I \backslash D)$
20: **loop**
21: $\Delta^+ := N_A \backslash ((I \backslash D) \cup A)$
22: **if** $\Delta^+ = \emptyset$ **then break**
23: $A := A \cup \Delta^+$, $N_A := \emptyset$
24: **for** $r \in \Pi$ **do**
25: $N_A := N_A \cup \mathsf{Add}^r((I \backslash D) \cup A, \Delta^+)$

while applying standard seminaïve evaluation for other normal rules. We have compared a fully HD-based evaluation, where all rules use hypertree decomposition-based approaches, against a combined version that selectively applies HD-based techniques only for cyclic rules. The performance evaluation of these two approaches will be presented in Section 4.6.

4.6 Implementation and Evaluation

To evaluate our algorithms, we developed a proof-of-concept implementation and conducted a series of experiments. Section 4.6.1 describes how hypertree decompositions

are computed in our implementation, and Section 4.6.2 outlines the approaches used for comparison. The experimental hypotheses are presented in Section 4.6.3, and the benchmark datasets are detailed in Section 4.6.4. The experimental setup is introduced in Section 4.6.5. Finally, Section 4.6.6 analyses the results in light of the stated hypotheses, discussing both expected and unexpected outcomes.

4.6.1 Implementation

The algorithms presented in Section 4.4 are independent of the choice of decompositions; however, different hypertree decompositions will lead to very different performance even if they share the same hypertree width. This is because the decomposition method only considers structural information of the queries and ignores quantitative information of the data. To address this problem, Scarcello et al. [123] introduced an algorithm that chooses the optimal decomposition w.r.t. a given cost model. We adopt this algorithm with a cost model consisting of two parts: (1) an estimate of the cost of intra-node evaluation, i.e., the joins among $\lambda(p)$; and (2) an estimate of the cost of inter-node evaluation, i.e., the joins between nodes. In our implementation, for (1), we use the standard textbook cardinality estimator described in Chapter 16.4 of the book [57] to estimate the cardinality of $\bowtie_{B_i \in \lambda(p)} B_i$ for a node p; for (2), we use $2 * (|\hat{p_i}| + |\hat{p_j}|)$ to estimate the cost of performing semi-joins between nodes p_i and p_j, where $|\hat{p_i}|$ and $|\hat{p_j}|$ represent the estimated node size.

Moreover, the extra step of full reducer we introduced in Algorithm 5 (line 7) is more suitable for small updates, in which the node with the smallest size helps reduce other large nodes. If the size of all the nodes is comparable, then this step would be unnecessary. Therefore, in practice, we only perform this optimisation if the number of active instantiations in the Δ node (i.e., p_i) is more than three times smaller than the maximum number of active instantiations in each node.

4.6.2 Comparisons

We considered three different approaches. The *standard* approach uses the seminaïve algorithm for materialisation and an optimised variant of DRed for incremental maintenance. The *HD* approach uses our hypertree decomposition based algorithms. The *combined* approach applies HD algorithms to complex rules and standard algorithms to the remaining rules. To ensure fairness, all three approaches are implemented on

Benchmarks	$\lvert E\rvert$	$\lvert I\rvert$	$\lvert\Pi\rvert$	$\lvert\Pi_s\rvert$	$\lvert\Pi_c\rvert$
LUBM L	66,751,196	91,128,727	98	98	0
LUBM L+C	66,751,196	99,361,809	114	98	16
Exp	3,362,280	6,440,280	3	0	3
YAGO	58,276,868	59,755,990	23	0	23

Table 4.2: Benchmark Statistics used in Chapter 4, in which $|\Pi_s|$ and $|\Pi_c|$ refer to the numbers of simple (non-cyclic) and cyclic rules, respectively.

top of the same code base obtained from the authors of the modular reasoning framework [77]. The framework allows us to partition a program into modules and apply custom algorithms to each module as required.

4.6.3 Hypotheses

The subsequent experiments are designed to test the following hypotheses:

1. The HD-based approach yields significant runtime improvements over the standard seminaïve evaluation for both initial and incremental materialisation in the presence of cyclic rules.

2. For non-cyclic rules, the HD-based approach may incur additional runtime overhead compared to the standard approach, as discussed in Section 4.2.3.

3. The combined approach generally performs best by applying the HD-based evaluation to cyclic rules and the standard evaluation to non-cyclic ones.

4. The HD-based algorithms require storing and maintaining intermediate instantiations for each node in the decomposition, which may introduce additional memory overhead.

To test the above hypotheses, we select datasets that exhibit varying structural properties: pure cyclic, pure non-cyclic, and mixed rule sets. These benchmarks are introduced in Section 4.6.4. Finally, Section 4.6.6 presents an analysis of the results, relating them back to the initial hypotheses and discussing both expected and unexpected findings.

4.6.4 Benchmarks

The selected benchmarks cover pure cyclic case (Exp), pure acyclic case (LUBM L), and mixed rule sets (LUBM L+C and YAGO), as summarised in Table 4.2, where

$|E|$ is the number of initial facts, $|I|$ is the number of facts in the full materialisation, and $|\Pi|$, $|\Pi_s|$, $|\Pi_c|$ are the numbers of rules, simple (non-cyclic) rules, and cyclic rules, respectively. The following provides a detailed introduction to these benchmarks. [3]

LUBM [65] models the university domain and includes a data generator capable of producing datasets of varying sizes. We use the LUBM-500 dataset, which contains data for 500 universities; this dataset serves as the set of facts E. For the rule set, since the original LUBM ontology is not fully expressible in OWL 2 RL, we adopt the LUBM L variant created by Zhou et al. [158]. The LUBM L rules are very simple, so we added 16 rules that capture more complex but semantically reasonable relations in the domain; some of these rules are rewritten from the cyclic queries used by Stefanoni et al. [130]; We call the resulting rule set LUBM L+C. One example rule is:

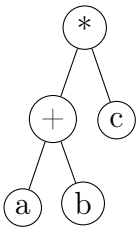
$$\mathsf{SA}(?p_1, ?p_2) \leftarrow \mathsf{HA}(?o, ?p_1), \mathsf{AD}(?p_1, ?ad), \mathsf{HA}(?o, ?p_2), \mathsf{AD}(?p_2, ?ad),$$

in which HA and AD represent predicates hasAlumnus and hasAdvisor respectively, while SA in the head represents a new predicate haveSameAdvisor that links pairs of students $?p_1$ and $?p_2$ (not necessarily distinct) who are from the same university $?o$ and share the same advisor $?ad$. The complete list of cyclic rules used in LUBM L+C is provided in Appendix B.1.

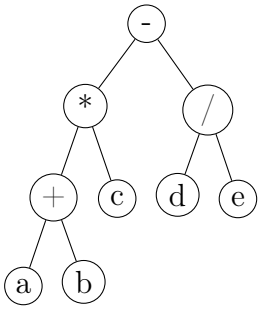
YAGO [132] is a real-world RDF dataset with general knowledge about movies, people, organisations, cities, and countries. We adapted 23 cyclic queries with different topologies (i.e., cycle, clique, petal, flower) from Park et al. [113] into 19 non-recursive rules and 4 recursive rules. More specifically, we use the original query structure as the rule body and manually define the rule head to form complete rules. These rules are helpful to evaluate the performance of our algorithm on topologies that are frequently observed in real-world graph queries [28]. The complete rule set is provided in Appendix B.2.

Exp (*expressions*): As mentioned in Section 4.1, realistic applications often involve complex rules. One example is the use of rules to evaluate numerical expressions, and our Exp benchmark has been created to simulate such cases. Specifically, Exp applies Datalog rules to evaluate expression trees of various depths. An expression tree represents the hierarchical structure of an expression, where the depth indicates the longest path from the root (final result) to a leaf (operand). For example, the expression tree of $(a + b) * c$ has a depth of 2, as illustrated follows:

[3] The test datasets and Datalog programs used in our experiments are publicly available at `https://github.com/xyzBubble/HDReasoning-Benchmarks/tree/main`, and the complete cyclic rule sets are provided in Appendix B.



In contrast, $((a + b) * c) - (d/e)$ has a depth of 3:



The leaves in the expression tree represent parameters in the expression, and in realistic cases, multiple values can be given as input to these parameters. These values propagate through the tree according to the operations defined at each internal node. For example, in the expression tree for $(a + b) * c$, the leaves a and b correspond to input values that are first combined using addition, and the result is then multiplied by c. This structure naturally supports batch computation, where different sets of values for a, b, and c can be processed in parallel. Our Exp benchmark contains three recursive rules capturing the arithmetical operations *addition*, *subtraction* and *multiplication*, which are provided in Appendix B.3. Each of these rules is cyclic and contains 9 body atoms. A generator is used to create random data for a given number of expressions, sets of values and maximum depth. In our evaluation, we generated 300 expressions, each with 300 set of values and a maximum depth of 5.

4.6.5 Test Setup

All of our experiments are conducted on a Dell PowerEdge R730 server with 512GB RAM and 2 Intel Xeon E5-2640 2.60GHz processors, running Fedora 33, kernel version 5.10.8. We evaluate the running time of *materialisation* (the initial materialisation with all the explicitly given facts inserted as E^+ in Algorithm 3), *small deletions* (randomly selecting 1,000 facts from the dataset as E^- in Algorithm 3), *large deletions* (randomly selecting 25% of the dataset as E^-), and *small additions* (adding 1,000 facts as E^+ into the dataset). Materialisation can be regarded as a *large addition*.

Method	materialisation				small deletions			
	L+C	L	Exp	YAGO	L+C	L	Exp	YAGO
standard	29,577.90	95.73	7,039.87	155,022.00	**0.92**	**0.03**	37.60	20.06
HD	1,168.83	740.81	**56.83**	367.59	4.00	3.70	0.47	**0.18**
combined	**554.00**	**75.50**	57.01	**366.03**	1.06	0.04	**0.45**	0.20
Method	large deletions				small additions			
	L+C	L	Exp	YAGO	L+C	L	Exp	YAGO
standard	15,193.70	27.09	4,006.44	126,562.00	0.97	**0.02**	40.23	20.42
HD	812.32	558.90	30.93	168.34	1.04	0.45	0.57	**0.17**
combined	**195.51**	**21.71**	**28.62**	**159.43**	**0.73**	0.06	**0.53**	**0.17**

Table 4.3: Performance Evaluation of HD-based approaches in Chapter 4, reporting materialisation and incremental reasoning time (in seconds).

All the experiments were done as follows: first rules and the explicitly given data were imported, followed by an initial materialisation; then 1,000 tuples were deleted and inserted, corresponding to small deletion and small addition, respectively; lastly, 25% tuples were deleted, this is, the large deletion.

4.6.6 Analysis

The experimental results about running time are shown in Table 4.3 in which L and L+C are short for LUBM L and LUBM L+C respectively. The computation of decompositions takes place during initial materialisation only and the time taken is included in the materialisation time reported in Table 4.3; it takes less than 0.05 seconds in all cases. Table 4.4 and 4.5 show the detailed peak and static memory usage, respectively. And the last line of both tables illustrates the ratio of space consumed by the combined method to that consumed by the standard approach. We measured both the peak memory usage during the materialisation and the static memory usage consumed by the triple tables and indexes used in standard approaches, as well as auxiliary data structures used in HD modules. We now discuss the experimental results organised by hypothesis.

Hypothesis 1: For datasets containing only pure cyclic rules (i.e., Exp and YAGO), our HD-based approach outperforms the standard approach across all tasks, with speedups ranging from 70x to 750x. This result supports Hypothesis 1, which posits that HD-based evaluation is beneficial for cyclic rule evaluation, particularly in recursive settings and in handling complex incremental maintenance. For the mixed rule set (i.e., L+C), the HD-based approach outperforms the standard approach in initial materialisation and large deletion tasks by approximately 20x. However, for

small incremental tasks (e.g., small additions and small deletions), the HD-based approach performs slightly worse, likely due to the overhead introduced when evaluating non-cyclic rules using HD-based evaluation, which will be discussed in more detail below.

Hypothesis 2: For the dataset containing only acyclic rules (i.e., L), the HD-based approach performs worse than the standard approach across all tasks, demonstrating the high constant factor associated with the Yannakakis algorithm, as discussed in Section 4.2.3. This result is consistent with Hypothesis 2.

Hypothesis 3: As can be seen, the combined approach outperforms the other approaches in most cases, sometimes by a large factor, and it is slower than the standard approach only for some of the small update tasks on LUBM L and L+C where processing time is generally small. In contrast, the standard approach performs poorly when complex rules are included (i.e., L+C, YAGO, and Exp), while the HD approach performs poorly on the simple rules in LUBM L. In particular, our combined approach is 75-139x faster than the standard approach for all the tasks on Exp; on YAGO, it is 100-793x faster. Moreover, for the materialisation and large deletion tasks on LUBM L+C, the combined approach is about 53x and 77x faster than the standard approach, respectively. Furthermore, for the small deletion and addition tasks on LUBM L+C and all the tasks on LUBM L, our combined method achieves a comparable result with the standard approach. The combined approach performs similarly to the standard approach on LUBM L, as the HD module is never invoked (there are no cyclic rules), and it performs similarly to the HD approach on Exp and YAGO, as the HD module is always invoked (all rules are cyclic). Our evaluation illustrates the benefit of the hypertree decomposition-based algorithms when processing complex rules, and it shows that by combining HD algorithms with standard reasoning algorithms in a modular framework we can enjoy this benefit without degrading performance in cases where some or all of the rules are relatively simple. This result provides evidence in support of Hypothesis 3.

Hypothesis 4: Finally, the HD algorithms have to maintain auxiliary data structures for rule evaluation, which incurs some space overhead when the HD module is invoked. Specifically, our combined method consumes up to 2.3 times the memory of the standard algorithm. However, this space overhead remains tolerable given the performance benefits achieved, and it is a reasonable trade-off for improved computational efficiency.

Benchmarks	L	L+C	Exp	YAGO
standard	7.2	8.4	0.6	5.2
HD	20.1	29.4	1.2	9.5
combined	7.2	16.5	1.2	9.5
ratio (combined / standard)	1.00	1.96	2.25	1.85

Table 4.4: Performance Evaluation of HD-based approaches in Chapter 4, reporting peak memory usage (in GB).

Benchmarks	L	L+C	Exp	YAGO
standard	5.5	6.3	0.5	3.8
HD	18.7	28.1	1.1	4.8
combined	5.5	14.6	1.1	4.8
ratio (combined / standard)	1.00	2.30	1.77	1.27

Table 4.5: Performance Evaluation of HD-based approaches in Chapter 4, reporting static memory usage (in GB).

4.7 Related Work

4.7.1 HD in Query Answering

The HD methods have been used in database systems to optimise the performance of query answering. For RDF workload, Aberger et al. [2] evaluated empirically the use of HD combined with worst-case optimal join algorithms, showing up to 6x performance advantage on bottleneck cyclic RDF queries. Also, in the EmptyHeaded [3] relational engine, a query compiler has been implemented to choose the order of attributes in multiway joins based on a decomposition. This line of work focuses on optimising the evaluation of a single query, while our work focuses on evaluating recursive Datalog rules. For a more comprehensive review of HD techniques for query answering, please refer to [62].

4.7.2 HD in Datalog-based Answer Set Programming

Jakl et al. [83] applied HD techniques to the evaluation of propositional answer set programs. Assuming that the treewidth of a program is fixed as a constant, they devise fixed-parameter tractable algorithms for key ASP problems including consistency checking, counting the number of answer sets of a program, and enumerating such answers. In contrast to our work, their research focuses on propositional answer

set programs.

For ASP in the non-ground setting, a program is usually grounded first, and then a solver deals with the ground instances. The usage of (hyper)tree decomposition has been investigated to decrease the size of generated ground rules in the grounding phase [23, 33, 102]. Bichler et al. [23] used hypertree decomposition as a guide to rewrite a larger rule into several smaller rules, thus reducing the number of considered groundings; Calimeri et al. [33] studied several heuristics that could predict in advance whether a decomposition is beneficial. In contrast, our work focuses on the (incremental) evaluation directly over the decomposition since the decomposition solely cannot avoid the potential blowup during the evaluation of the smaller rules.

One possible approach to handling cyclic rules in Datalog reasoning is to adopt the rewriting technique, in which a cyclic rule is decomposed into several smaller rules, each corresponding to a component (or bag) of its decomposition. This process would introduce a new predicate containing answer variables for each bag in the decomposition, as Datalog rules can be viewed as conjunctive queries with answer variables as the variables in the head atom. By rewriting the program in this manner, standard reasoning techniques can be applied to achieve both initial materialisation and incremental reasoning. While this is a viable strategy, we anticipate that it may introduce certain inefficiencies compared to our method, which is based on Yannakakis's algorithm for non-Boolean conjunctive queries [153]. Specifically, adding output variables to every bag could potentially increase the width of the decomposition, possibly leading to larger intermediate relations. This may, in turn, negatively affect both memory consumption and runtime performance. Additionally, while Yannakakis's algorithm runs in log-linear time relative to the size of the decomposition plus the size of the output, a rewriting approach that relies on standard join algorithms would likely operate on larger decompositions (as discussed above) and may not always guarantee log-linear evaluation.

However, an empirical comparison between the rewriting approach and our method would be a valuable direction for future work. Further investigation could provide deeper insights into the trade-offs between the two approaches and help determine under which conditions one method may be preferable over the other.

4.8 Conclusion

In this chapter, we introduced a hypertree decomposition-based reasoning algorithm, which supports rule evaluation, incremental reasoning, and recursive reasoning. We

implemented our algorithm in a modular framework such that the overhead caused by using decomposition is incurred only for complex rules, and demonstrate empirically that this approach is effective on simple, complex and mixed rule sets.

Despite the promising results, we see many opportunities for further improving the performance of the presented algorithms. Firstly, our decomposition remains unchanged once it is fixed. However, as the input data and the materialisation change over time, the initial decomposition may no longer be optimal for rule evaluation. It would be beneficial if the maintenance could be done with the underlying decomposition changing. However, this would be challenging since the data structure in each decomposition node is maintained based on the previous decomposition, and changing the decomposition would require transferring information from the old node to the new one.

Secondly, although the memory usage has been optimised to some extent, intermediate results still take up a significant amount of space. This problem could be mitigated by incrementally computing the final join result without explicitly storing the intermediate results, or by storing only "useful" intermediate results.

Finally, it would be interesting to adapt our work to Datalog extensions, such as Datalog$^{\pm}$ [32] and DatalogMTL [148]. This would require introducing mechanisms to process the relevant additional features, such as the existential quantifier in Datalog$^{\pm}$ and the use of intervals in DatalogMTL.

Chapter 5

Multi-Scheme Framework

This chapter addresses Research Questions RQ3-RQ4 by introducing a multi-scheme framework that enables the integration of specialised storage schemes into Datalog materialisation and incremental maintenance, based on our publication [157]. The framework decouples storage from reasoning, allowing different rule derivations to be managed using optimised data structures while ensuring correctness and efficiency. We define the formal requirements for storage schemes and propose tailored schemes for transitive closure and union rules, leveraging their structural properties to improve storage efficiency and retrieval performance. This framework provides a flexible and extensible foundation for scalable materialisation-based reasoning in large knowledge bases.

5.1 Introduction

As discussed in Section 1.3 and 3.4, the cost of storing and maintaining materialisations can be impractical for large datasets and rules that can expand the dataset significantly. For example, given initial facts $\{R(a_i, a_{i+1}) \mid 1 \leq i \leq n\}$ and a rule $R(x, y), R(y, z) \rightarrow R(x, z)$ that declares R as a transitive relation, just like the *locatedIn* relation mentioned in Section 1.1, there are $O(n^2)$ materialised facts. Moreover, in DBpedia [94], there are 8.5 billion facts in the transitive closure of just the *broader* relation, which would require an estimated 510 GB of memory to store. In such cases, the standard approach may fail to complete the materialisation process due to excessive space requirements, making subsequent query-answering infeasible. This limitation significantly restricts the applicability of materialisation-based systems, particularly in large-scale and dynamic reasoning environments.

In this chapter, we address this issue by proposing a general-purpose multi-scheme framework that enables the integration of specialised storage schemes into standard

Datalog materialisation and incremental maintenance algorithms. Our goal is to optimise the memory consumption required for computing and storing materialised facts while ensuring efficient maintenance during incremental reasoning and enabling effective query answering over a compact materialisation. Specifically, our framework preserves the core reasoning process of the *Delete/Rederive (DRed)* algorithm [108] while introducing *schemes* that leverage different underlying data structures for managing materialised facts. The framework defines a set of interfaces corresponding to key functionalities within the DRed algorithm, ensuring modular integration of customised storage strategies. The modified algorithm maintains the correctness of materialisation computation, provided that the integrated schemes conform to these interfaces and satisfy their required properties. This framework directly addresses Research Question RQ3 by providing a flexible and extensible approach to integrating specialised storage schemes into materialisation and incremental maintenance, enhancing storage efficiency while retaining the robustness of existing reasoning workflows.

Moreover, we develop specialised schemes that provide the framework-required interfaces for storing facts derived from *transitive rules* and *union rules*, as these two rule types frequently occur in practical applications. We demonstrate that existing studies generally do not meet the necessary requirements for integration into the Datalog reasoning process, highlighting the need for tailored storage solutions. Our experimental results show that the proposed approach successfully computes and maintains materialisations even in cases where the standard approach fails due to excessive resource consumption. Furthermore, it significantly improves reasoning performance in both space and time efficiency. Additionally, query-answering over compact materialisations remains acceptable, indicating that the proposed framework offers a promising solution for scalable Datalog reasoning in large-scale knowledge-based systems.

Overview: Section 5.2 introduces an example comparing the standard approach with customised storage, highlighting key challenges in integrating specialised storage schemes into the reasoning process. Section 5.3 presents our proposed multi-scheme framework, which enables the integration of different storage schemes while maintaining reasoning efficiency. It also introduces the general requirements that storage schemes must satisfy to be compatible with the framework. Sections 5.4 and 5.5 introduce two customised storage schemes tailored for transitive closure rules and union rules, respectively, demonstrating how specialised storage structures can

enhance materialisation efficiency. Section 5.6 explores further optimisations by incorporating counting-based techniques to improve incremental maintenance and reduce redundant derivations. Section 5.7 presents experimental results, evaluating the performance of the proposed framework in terms of computational efficiency, storage overhead, and query performance. Finally, Section 5.9 summarises the contributions and findings of this chapter and discusses future research directions.

5.2 Motivation

Example 5 *Let the dataset be* $E' = \{A(a_i, a_{i-1}) \mid 2 \leq i \leq n\} \cup \{S(a_3, a_1)\}$*, and consider program* Π' *consisting of following rules:*

$$A(x, y) \rightarrow R(x, y), \tag{R5}$$

$$R(x, y), R(y, z) \rightarrow R(x, z), \tag{R6}$$

$$R(x, y), S(x, y) \rightarrow A(y, x). \tag{R7}$$

We compare the initial materialisation process by applying Π' *to* E' *using the standard seminaïve algorithm and examine how introducing a customised storage scheme for derivations of the transitive rule* (R6) *would modify the materialisation process. This analysis leads to a discussion on the potential requirements for such a storage scheme.*

Interpretation: The rules in Π' can be understood as a formalisation of reachability and connectivity in a graph. Relation A declares directed edges in the graph, representing direct connections between nodes. Rule (R5) states that direct adjacency of nodes leads to initial reachability relations declared by the relation R. Specifically, if there is a direct edge from node x to node y, then x can reach y. Rule (R6) defines transitive reachability. If a node x can reach y, and y can reach z, then x can also reach z. This captures the transitive closure of reachability in the graph. Rule (R7) models bidirectional reachability. If node x can reach y and the connection is undirected and thus bidirectional, as indicated by relation S, then the reverse edge $A(y, x)$ is also inferred. Here, S denotes a symmetric connection. This ensures that reachability respects the symmetric nature of undirected edges. Together, these rules provide a logical framework for modelling directed and undirected reachability within a graph.

The initial dataset E' in Example 5 defines a set of edges in the graph by declaring $A(a_i, a_{i-1})$ for $2 \leq i \leq n$, and a bidirectional connection between a_3 and a_1 by defining $S(a_3, a_1)$. By applying Π' to E', the resulting facts with relation R will

Algorithm 9 Refined Seminaïve Algorithm: A Practical Approach

Require: program Π, initial dataset E, fact additions E^+, materialised facts I.
Ensure: update I from $\Pi^\infty[E]$ to $\Pi^\infty[E \cup E^+]$

1: $M = \Delta = E^+ \backslash E, I = I \cup \Delta$, $E = E \cup E^+$
2: **loop**
3: $\quad \Delta := \Pi[I, M] \setminus I$ ▷ Populate new facts Δ
4: $\quad$ **if** $\Delta = \emptyset$ **then break**
5: $\quad M := \Delta$ ▷ Obtain M used in next round
6: $\quad I := I \cup \Delta$ ▷ Merge new facts Δ to I

reflect the reachability between all connected nodes, including the transitive closure along the directed edges specified by A and the symmetric reachability implied by S. In particular, the bidirectional connection from $S(a_3, a_1)$ will lead to reachability in both directions, ensuring that both $R(a_3, a_1)$ and $R(a_1, a_3)$ are included in the result.

Practical Seminaïve Algorithm: In Algorithm 2, we present a conceptual seminaïve algorithm that uses Δ with different subscripts to distinguish new facts generated at each round of rule application. Similarly, different subscripts are used for I to represent the evolving materialisation. However, in practical implementations, maintaining multiple distinct Δ and I sets is inefficient and unnecessary. To address this, we introduce a practical seminaïve algorithm in Algorithm 9 that employs M to track fact changes used in the seminaïve operator while using Δ to determine the newly derived set of facts, as shown in line 3.[1] The materialisation process consistently updates the same I rather than maintaining separate versions. Please note that the usage of M here is to distinguish between the derived facts Δ in each round, which will be helpful for the discussion below. Additionally, note that Algorithm 9 is presented as an incremental insertion algorithm, but it naturally captures initial materialisation: one can simply set inputs I and E as empty, and E^+ as the dataset to be materialised. Such a practical presentation is more suitable for our discussion, as it aligns with real-world implementations. Below, we illustrate the materialisation process using this practical version of the seminaïve algorithm.

Materialisation of Example 5 using Seminaïve Algorithm: The initial materialisation process of Example 5 using the seminaïve algorithm is performed by initialising the input program as $\Pi = \Pi'$ and the fact additions as $E^+ = E'$, corresponding to the program and dataset presented in Example 5, while setting the other sets, I and E, to be empty. Table 5.1 depicts the initial materialisation obtained

[1]While the usage of M may seem redundant and could be replaced by Δ, we retain it here to distinguish between the derived facts Δ in each round, which will be useful for the discussion below.

by executing Algorithm 9 to apply Π' to the given dataset E'. Specifically, the first column refers to the rule application round corresponding to an iteration of the loop of lines 2–6; the second column is the value of set M in line 3 of the current round; the third column depicts the rules that successfully derive consequences in line 3; finally, the last column is the set of facts derived in line 3.

Round	M	Rules	Δ
1	$\{A(a_i, a_{i-1}) \mid 2 \leq i \leq n\}$, $S(a_3, a_1)$	R5	$\{R(a_i, a_{i-1}) \mid 2 \leq i \leq n\}$
2	$\{R(a_i, a_{i-1}) \mid 2 \leq i \leq n\}$	R6	$\{R(a_i, a_{i-2}) \mid 3 \leq i \leq n\}$
3	$\{R(a_i, a_{i-2}) \mid 3 \leq i \leq n\}$	R6 R7	$\{R(a_i, a_{i-3}) \mid 4 \leq i \leq n\}$ $\{R(a_i, a_{i-4}) \mid 5 \leq i \leq n\}$ $A(a_1, a_3)$
⋮	⋮	⋮	⋮

Table 5.1: Materialisation Process of Example 5 using Seminaïve Algorithm.

During the materialisation process, the rule (R5) first gets applied to obtain initial reachable pairs from the edges declared by the relation A, as shown in round 1 of Table 5.1. The rule (R6) axiomatises the R relation as transitive, and it is applied multiple times to obtain the full transitive closure $\{R(a_i, a_j) \mid 1 \leq j < i \leq n\}$. Each fact $R(a_i, a_j)$ is derived $i - j - 1$ times by the seminaïve algorithm using distinct rule instances $R(a_i, a_k), R(a_k, a_j) \rightarrow R(a_i, a_j)$ for k with $j < k < i$. Thus, computing the transitive closure for the R relation would require $O(n^3)$ time and storing the materialisation requires $O(n^2)$ space, as illustrated in Figure 5.2. Additionally, the rule (R7) derives $A(a_1, a_3)$ (presented in round 3 of Table 5.1) using the rule instance $R(a_3, a_1), S(a_3, a_1) \rightarrow A(a_1, a_3)$ as the symmetric relation S implies bidirectional connection. This allows the reverse edge $A(a_1, a_3)$ to be introduced. The rule (R7) interacts recursively with rules (R5) and (R6): it derives $A(a_1, a_3)$, which in turn derives $R(a_1, a_3)$. Consequently, more rule instances of (R6) will be considered to complete the transitive closure of the reachability relation.

Materialisation of Example 5 using a Customised Storage Scheme: Next, we discuss how integrating a compact storage scheme for the transitive relation into the reasoning process would significantly improve the time and space efficiency. Our optimisation draws inspiration from the work by Agrawal et al. [6], which proposed an interval-based approach to the compact representation of transitive relations. More specifically, observe that each $R(a_i, a_j)$ fact can be seen as a directed edge from node

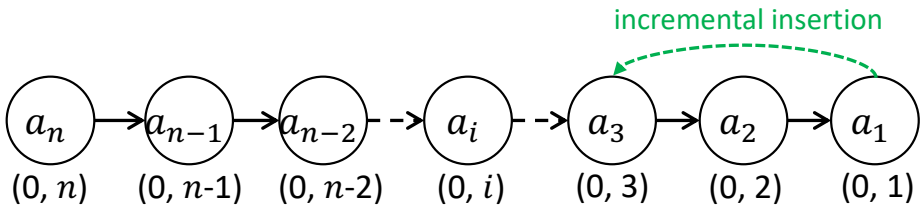


Figure 5.1: Illustration of the data structure constructed in the materialisation process of Example 5: The Interval Graph constructed in Round 2.

The facts table $O(n^2)$		
a_n	R	a_{n-1}
a_n	R	a_{n-2}
⋮	R	⋮
a_n	R	a_1
a_{n-1}	R	a_{n-2}
⋮	R	⋮
a_{n-1}	R	a_1
⋮	R	⋮
a_2	R	a_1

Interval-based Storage $O(n)$		
node	*id*	*Interval*
a_1	1	$(0, 1)$
⋮	⋮	⋮
a_n	n	$(0, n)$

Figure 5.2: Illustration of storage schemes for facts of the relation R in Example 5, using a plain table or the interval-based compact storage. Please note that the fact $A(a_1, a_3)$ derived in Round 3 of Table 5.2 would require incremental updates of this compact storage.

a_i to node a_j, and so the R relation forms a directed graph. Then, if we ignore the impact of rule (R7) for the moment, the interval-based data structure D_R can be constructed as illustrated in Figure 5.1 (excluding the green edge), where each node v in the graph is associated with an index i_v that identifies v (here $i_{a_j} = j$ for each j with $1 \leq j \leq n$), and an interval $\mathcal{I}_v$ that covers indexes of nodes that v can reach in the graph. The constructed storage, illustrated in Figure 5.2, requires $O(n)$ space. Then for each node a_i, facts in the closure with a_i as the first argument can be conveniently retrieved as $I_{a_i} = \{R(a_i, a_j) \mid i_{a_j} \in \mathcal{I}_{a_i}\}$, and so the transitive closure is in essence encoded by $\{I_{a_i} \mid 1 \leq i \leq n\}$ in a compact fashion.

The above compact representation can be efficiently computed in $O(n)$ time and stored in $O(n)$ space for this particular example (as shown in Figure 5.2), as opposed to $O(n^3)$ time and $O(n^2)$ space required by the standard materialisation approach. Note that in general, for an arbitrary graph with $|V|$ number of vertices and $|E|$

number of edges, the construction takes $O(|V| + |E|)$ time, and requires $O(|V|^2)$ space in worst-case scenarios.[2] In practice, space consumption can be optimised by selecting an optimal tree cover for the graph [6], an approach that proves beneficial in our evaluation. The construction details of this data structure are presented in Section 5.4.1. Additionally, this compact representation supports efficient query answering: checking whether $R(a_i, a_j)$ holds can be efficiently implemented as checking if the index of a_j is included in the interval of a_i, an operation that requires $O(1)$ time. Consequently, introducing a customised storage scheme into the materialisation process has the potential of saving time and space while still allowing for efficient query answering.

However, existing compact storage schemes that only target specific types of relations would still require non-trivial adaptions to be suitable for integration with standard Datalog reasoning algorithms, because, in practice, Datalog programs often involve complex interactions between rules. Our example program involves recursive interaction between rule (R7) with rules (R5) and (R6), and so we use it to outline how our proposed framework deals with such interactions in Table 5.2. The first two columns of the table depict the round of reasoning and the set of facts newly derived in the previous reasoning round, respectively; the last column describes the reasoning work performed in the corresponding round. For the sake of simplicity we omit the first round of reasoning as it coincides with that of Table 5.1.

Round	M	Work Performed
2	$\{R(a_i, a_{i-1}) \mid 2 \leq i \leq n\}$	Construct D_R as shown in Figure 5.1.
3	Δ returned by D_R, $\{R(a_i, a_j) \mid 1 \leq j < i \leq n\}$	(R7) derives $A(a_1, a_3)$
4	$A(a_1, a_3)$	(R5) derives $R(a_1, a_3)$
5	$R(a_1, a_3)$	Incrementally insert $R(a_1, a_3)$ to D_R.
6	Δ returned by D_R, $\{R(a_i, a_j) \mid 1 \leq i \leq j \leq 3\}$	Nothing can be derived.

Table 5.2: Materialisation Process of Example 5 with Compact Storage.

In this example, the compact storage scheme is responsible for the reasoning of rule (R6) and the storage of the R facts. Note that R facts could potentially trigger rule (R7). Therefore, after the data structure is initially constructed in Round 2, it

[2]The discussion for the construction of such a data structure will be presented in Section 5.4.

is important that new facts derived by the compact structure should be efficiently identified and passed on to other rules, which are evaluated using standard seminaïve evaluation. Specifically, $A(a_1, a_3)$ is derived by applying (R7) in Round 3, which in Round 4 introduces a new fact $R(a_1, a_3)$. This corresponds to an added edge (a_1, a_3) to the compact storage scheme for R, as shown in the highlighted arrow of Figure 5.1, and so in Round 5, the scheme should be able to handle the incremental insertion of this new edge, to update the intervals to reflect the updated reachability relations, and to identify freshly derived facts that will be used for reasoning in later rounds. Nothing can be derived using these facts in Round 6, and so the materialisation terminates. Analogously, our framework should allow for each storage scheme to deal with incremental deletion, where the scheme is required to efficiently update its underlying data structure and identify affected facts in response to deletions.

In Section 5.3, we will introduce general interfaces required for each scheme, enabling customised storage optimisations to be seamlessly integrated into the materialisation and incremental maintenance process. We also formalise the reasoning process using these scheme interfaces and provide a precise definition of a valid scheme to ensure the correctness of reasoning within the proposed framework. In Sections 5.4 and 5.5, We will present details of two advanced storage schemes tailored for transitive and union rules. These schemes not only meet the requirements of the proposed framework but also include significant improvements, such as a non-trivial enhancement of the interval-based approach for transitive reasoning. Together, these contributions establish a flexible and robust foundation for supporting diverse storage schemes in modular reasoning systems.

5.3 Multi-Scheme Framework

In this section, we first introduce key notations (Section 5.3.1) and functions (Section 5.3.2), along with the required properties for storage schemes, which serve as fundamental building blocks for the proposed framework. We then present our framework in Section 5.3.3, which extends the DRed Algorithm, with a particular focus on how materialisation computation and incremental maintenance are achieved through the integration of scheme functions.

5.3.1 Storage Scheme Notations

The multi-scheme framework is designed to accommodate multiple storage schemes, each managing a disjoint subset of facts. Specifically, each storage scheme T is re-

	Explanation
P_T	The relation that T manages.
Π_T	Rules that derive facts of the relation P_T.
B_T	Relations that appear in the body of rules in Π_T.
$\mathcal{R}_d^T$	Rules that are applied when manipulating the data structure, $\mathcal{R}_d^T \subseteq \Pi_T$.
$\mathcal{R}_n^T$	Rules that are applied in a standard mannaer, $\mathcal{R}_n^T \subseteq \Pi_T$, $\mathcal{R}_n^T \cup \mathcal{R}_d^T = \Pi_T$.
Δ_n^T	Facts scheduled to be encapsulated/removed from the data structure.
M_a^T, M_d^T	Fact changes tracked to be used in seminaïve-like evaluation.
I^T, Δ^T, D^T	Facts serialised in different domains from the data structure.

Table 5.3: Notations defined for a Storage Scheme T.

sponsible for facts of the relation P_T. For instance, in Example 5, a storage scheme T can be constructed to store facts of the relation R. This scheme manages all rules that derive facts for this relation, i.e., rules (R5) and (R6). A specialised data structure tailored to the transitive property implied by the rule (R6) can be constructed to compactly store the R facts, such as the one illustrated in Figure 5.1. Thus, the construction and incremental updates to the data structure are performed while preserving the transitive property. Consequently, manipulating the data structure corresponds to applying the rule (R6). For the rule (R5), which also derives facts for R, its application is carried out in the standard manner. However, the resulting facts of (R5) are then inserted and encapsulated within the compact representation, ensuring consistency and efficient storage. Formally, we denote the set of rules assigned to T as Π_T, where $\Pi_T \subseteq \Pi$ and the predicate P_T appears in the head of each rule. Among the rules in Π_T, those managed by the storage scheme T are denoted as $\mathcal{R}_d^T$, while the remaining rules, $\mathcal{R}_n^T$, are applied using the standard approach. Thus, we have $\Pi_T = \mathcal{R}_d^T \cup \mathcal{R}_n^T$. All access to facts managed by T is handled through the specialised data structure, ensuring efficient retrieval and encapsulation of stored facts.

This definition of schemes also accommodates the case where no storage optimisation can be applied. In this scenario, all rules in Π_T are executed using the standard algorithm: $\mathcal{R}_d^T = \emptyset, \mathcal{R}_n^T = \Pi_T$. All facts within the scheme are stored in a default tabular format. We refer to this as the *default scheme*. The details of the default scheme will be discussed in Section 5.3.4.

Table 5.3 summarises the notations defined for each scheme. In addition to P_T and Π_T, which have already been introduced, the set Δ_n^T records facts of the relation P_T that are to be encapsulated in or removed from the underlying data structure of T. Furthermore, the sets B_T, M_a^T, and M_d^T facilitate the application of rules in Π_T.

Schemes	P_T	Π_T	B_T	$\mathcal{R}_d^T$	$\mathcal{R}_n^T$
A	A	R7: $R(x,y), S(x,y) \rightarrow A(y,x)$	R, S	$\emptyset$	R7
S	S	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
R	R	R5: $A(x,y) \rightarrow R(x,y)$ R6: $R(x,y), R(y,z) \rightarrow R(x,z)$	A, R	R6	R5

Table 5.4: One Suitsble Scheme Initialisation for Example 5.

The set B_T contains the relations used in the body of rules in Π_T, while M_a^T and M_d^T store newly added and newly deleted facts, respectively, for predicates in B_T. Therefore, the facts stored in M_a^T and M_d^T assist in the seminaïve-like evaluation of the rules in Π_T that are managed within the scheme T. Additionally, we refer to the collection of active storage schemes as $\mathbb{T}$.

Finally, I^T, Δ^T, and D^T will be used to denote sets of facts produced by the scheme T in domains 'I', 'Δ', and 'D', respectively. Intuitively, they refer to the currently materialised facts, the newly deleted or inserted facts, and the accumulated set of deleted facts. Currently available facts I is defined as union of I^T for every scheme $T \in \mathbb{T}$. Formally,

$$I = \bigcup_{T \in \mathbb{T}} \{I^T\}. \tag{5.1}$$

Determining how schemes are initialised can be non-trivial and depends on the syntax and semantics of the rules in Π. For now, we focus on establishing the correctness of the algorithm assuming a suitable initialisation of storage schemes. Specifically, we abstractly describe how the algorithm operates given that appropriate schemes are in place, without detailing how these schemes are constructed or selected. We defer the detailed discussion on scheme selection to Section 5.7.1. Next, we present a suitable scheme initialisation for Example 5, explaining the notations in detail to provide better understanding.

Revisiting Example 5 in Section 5.2: *One suitable initialisation of schemes for Example 5 is to have three schemes for facts with predicates A, S, and R, respectively, as shown in Table 5.4. For the scheme that manages R facts, the rules assigned to this scheme Π_T are* (R5) *and* (R6)*. If the scheme employs the interval-based technique to handle the transitive relation axiomatised by rule* (R6)*, then we have $\mathcal{R}_d^T = \{R6\}$ and $\mathcal{R}_n^T = \{R5\}$. This indicates that the construction of the data structure is equivalent to encapsulating the derivations of transitive closure rules within it.*

Meanwhile, rule (R5) *must be applied using standard seminaïve evaluation, and its derived consequences are subsequently inserted into the data structure for encapsulation. Therefore, scheme R uses specialised storage maintaining the transitive closure property implied by rule* (R6)*, while schemes A and S use the default storage scheme. Table 5.4 presents the schemes initialised in Example 5. Other notations introduced in Table 5.3 are preserved throughout the reasoning process and are omitted from this table for clarity. The relations marked in B_T determine which newly derived facts will trigger rule applications with fresh instantiations. For example, newly derived A facts lead to additional rule applications of* (R5) *within scheme R. These new A facts are recorded in M_a^T and M_d^T of scheme R to be processed in the next evaluation step.*

5.3.2 Scheme Interfaces and Properties

In this section, we first introduce a few scheme interface functions in Section 5.3.2.1. Then, similar to how the DRed algorithm employs the seminaïve operator to perform the derivation step, we present how such a derivation step is achieved within a scheme in Section 5.3.2.2. Additionally, we define the bounds of the resulting set of facts (or derivations) for each derivation step. Finally, in Section 5.3.2.3, we formalise the properties that a scheme must satisfy to ensure the correctness of the derivation step throughout various phases.

5.3.2.1 Scheme Functions

For a storage scheme to be fully integrated into the reasoning algorithm, it must implement the scheme functions specified in Table 5.5. This ensures that the scheme can participate in all required computation steps and maintain the correctness of the overall evaluation process. The third column of the table provides an intuitive explanation of the expected functionality of each implementation. In later sections, we will discuss in detail how these functions are implemented by both the default scheme and the customised schemes.

The table categorises the functions for storage schemes into three main types:

- **Incremental Update of Data Structure** – Functions *insert* and *delete* handle the incorporation and removal of facts, identifying fresh derivations or deletions that can be serialised.

Type	Function	Explanation
Incremental Update of Data Structure	*insert*	Incorporate facts in Δ_n^T into the data structure. Fresh derivations $\Delta^T \nsubseteq I^T$ are identified and can be serialised.
	delete	Delete facts in Δ_n^T from the data structure. Fresh deletions $\Delta^T \subseteq I^T$ are identified and can be serialised.
Merge Change via Data Structure Manipulation	*merge*	Update the data structure so that serialised fresh additions Δ^T are merged into I^T. Formally, $I^T = I^T \cup \Delta^T$, $\Delta^T = \emptyset$.
	remove	Update the data structure so that serialised fresh deletions Δ^T are removed from I^T and added to serialised deleted facts D^T. Formally, $I^T = I^T \backslash \Delta^T$, $D^T = D^T \cup \Delta^T$, $\Delta^T = \emptyset$.
Derivation Procedures (same for various schemes)	*deriveForAddition*	Within the context of M_a^T and the current materialised facts I, the rules in Π_T are applied, and the fresh derivations Δ^T are encapsulated within the data structure and can be serialised. Specifically, the rules in $\mathcal{R}_d^T$ are applied by invoking the *insert* function, while the rules in $\mathcal{R}_n^T$ are processed using the seminaïve operator.
	deriveForDeletion	Similar to *deriveForAddition*, but within the context of M_d^T, and the rules in $\mathcal{R}_d^T$ are applied by invoking the *delete* function.
	rederive	Given the overdeleted facts D^T serialised from the data structure and the current materialisation I, compute the set of facts that still have valid proofs within I under the rule set Π.
Schedule	*insertable*	Process derivations during addition and determine if a given fact needs to be inserted by checking if it already exists.
	deletable	Process derivations during overdeletion and determine if a given fact can be deleted from the scheme.
	rederivable	Returns true if the given fact can be derived using remaining facts.

Table 5.5: Interfaces of Storage Schemes

- **Merge Change via Data Structure Manipulation** – Functions *merge* and *remove* update the stored facts, ensuring that fresh additions are merged and deletions are properly removed and tracked.
- **Derivation Procedures** – Functions such as *deriveForAddition*, *deriveForDeletion*, and *rederive* establish derivation procedures for rules within the scheme Π_T, that are utilised in different phases of the reasoning process. These functions operate by invoking internal data structure functions (*insert* and *delete*) while applying the standard seminaïve evaluation to normal rules in $\mathcal{R}_n^T$.
- **Schedule** functions (*insertable*, *deletable*, *rederivable*) determine whether facts should be inserted, deleted, or rederived.

This categorisation ensures efficient fact management and seamless integration into the multi-scheme framework.

5.3.2.2 Derivation Procedure within Schemes

We now present the derivation procedure for a storage scheme. The core derivation step in standard reasoning algorithms, such as line 3 of the seminaïve algorithm

presented in Algorithm 9, computes fresh derivations Δ using the seminaïve operator $\Pi[I, M]$, where I represents the current materialisation and M tracks fact changes. Similarly, in the context of storage schemes, derivations are also computed, but in the context $I = \bigcup_{T \in \mathbb{T}} I^T$ and M_a^T that is tracked for the specific scheme T. Moreover, unlike the standard approach, the generated Δ^T are not explicitly stored; instead, they are encapsulated within the data structure of the corresponding scheme.

Revisiting Example 5: *In round 2, the storage scheme derives facts for the transitive closure rule* (R6) *by constructing the data structure illustrated in Figure 5.1. The resulting reachable pairs* $\Delta^T = \{I_{a_i} \mid 1 \leq i \leq n\}$ *in which* $I_{a_i} = \{R(a_i, a_j) \mid i_{a_j} \in \mathcal{I}_{a_i}\}$ *are encoded as intervals, compactly encapsulated in the data structure.*

Similarly, in a general incremental maintenance algorithm such as the DRed algorithm presented in Algorithm 3, the core derivation steps occur during the overdeletion, rederivation, and addition phases, corresponding to lines 6, 11, and 16, respectively. Table 5.6 summarises the computations performed during the derivation step in each phase of the DRed algorithm.

DRed Procedure	Context of Derivation Step	Computes
Overdeletion	I, Δ_{del}	$\Pi[I, \Delta_{del}] \cap (I \backslash \Delta_{del})$
Rederivation	I, D	$\Pi[I] \cap D$
Addition	I, Δ_{add}	$\Pi[I, \Delta_{add}] \setminus I$

Table 5.6: Summary of Derivation Computations in DRed Procedures

For storage schemes, similar derivation computations are performed using dedicated functions in the corresponding procedures. Specifically, the *overdelete*, *rederive*, and *add* procedures utilise the functions *deriveForDeletion*, *rederive*, and *deriveForAddition*, respectively. Invoking the derive functions ensures that the resulting derivations are encapsulated within the scheme's internal data structure, specifically in the domain Δ. To provide a more flexible and adaptable framework, we define *lower and upper bounds* for each derivation computation, as presented in Table 5.7. These bounds are established based on the modular framework proposed by Hu et al. [77], ensuring compatibility with various storage schemes.

Explanation of Bounds: For *deriveForAddition*, the lower bound corresponds to the seminaïve evaluation of rules in Π_T, while any fact included in the final updated materialisation $\Pi^\infty[E']$, where $E' = (E \backslash E^-) \cup E^+$, can be derived without affecting the correctness of reasoning. Moreover, to ensure the termination of the algorithm,

Derivation Procedures	lower bound	upper bound
deriveForAddition in context of M_a^T and I, $M_a^T \subseteq I$	$\Pi_T[I, M_a^T] \setminus I$	$\Pi^\infty[E'] \setminus I$
rederive in context of D^T and I, $D^T \nsubseteq I$	$\Pi_T[I] \cap D^T$	$\Pi^\infty[E'] \cap D^T$
deriveForDeletion in context of M_d^T and I, $M_d^T \nsubseteq I$	$(I \cap \Pi_T[I \cup M_d^T, M_d^T]) \setminus \Pi^\infty[E']$	I

Table 5.7: The Bounds of Derivation Procedures for Storage Schemes, in which $E' = (E \backslash E^-) \cup E^+$, and $\Pi^\infty[E']$ represents the materialisations after incrementally inserting facts E^+ and deleting E^-.

deriveForAddition encapsulates only those facts that are not already present in I into Δ^T.

Similarly, *deriveForDeletion* computes the consequences potentially affected by the deletion of M_d^T, which tracks fresh deletions during the *overdeletion* procedure. The bounds of Δ^T after invoking *deriveForDeletion* are defined in the context of the rule set Π_T, the currently existing facts I, and the tracked changes M_d^T. Any fact that can be derived from a one-step seminaïve evaluation using Π_T, I, and M_d^T, but does not hold in the final updated materialisation $\Pi^\infty[E']$, can be deleted, as indicated by the lower bound. Meanwhile, to accommodate different implementations, *deriveForDeletion* is permitted to overdelete any currently existing fact in I, as specified in the upper bound.

Finally, *rederive* is used to examine whether overdeleted facts D^T have alternative proofs using remaining facts I. It can recover any fact in D^T that is one-step rederivable by applying Π_T to I, as shown in the lower bound. Moreover, it can also rederive any fact that holds in the final updated materialisation $\Pi^\infty[E']$, as presented in the upper bound.

Derivation Procedures: We now present the derivation procedures that are consistent across different schemes, as outlined in Algorithm 10. As discussed in Section 5.3.1, rules within a scheme T are partitioned into $\Pi_T = \mathcal{R}_d^T \cup \mathcal{R}_n^T$. The rules in $\mathcal{R}_d^T$ are applied by invoking the incremental update functions of the data structure, such as the *insert* and *delete* functions in Table 5.5, while the rules in $\mathcal{R}_n^T$ are processed using the seminaïve operator.

Specifically, the *deriveForAddition* and *deriveForDeletion* function apply $\mathcal{R}_d^T$ and $\mathcal{R}_n^T$ separately. Take *deriveForAddition* function as an example, rules in $\mathcal{R}_n^T$ are first applied using the standard seminaïve evaluation strategy in line 3, and derivations of these rules are processed by the *schedule* function in line 4. The *schedule* function shown in Algorithm 11 uses the scheme's internal function *insertable* and *deletable* to determine whether an input fact can be scheduled for addition and overdeletion,

respectively. It then adds schedulable facts to Δ_n^T before incorporating them into the data structure by calling the *insert* function. The construction and maintenance of the data structure (line 5) is equivalent to applying rules in $\mathcal{R}_d^T$. Fresh derivations $\Delta^T \nsubseteq I^T$ can be identified by the data structure, in which I^T and Δ^T denote facts serialised from the data structure in domain 'I' and 'Δ', respectively. The *deriveForDeletion* function follows a similar procedure, so we will not reiterate here. The *rederive* function checks every overdeleted fact and examines whether an alternative proof exists using the scheme's internal function *rederivable*.

Algorithm 10 Derivation Procedures of Storage Schemes

1: **Result**: Derive facts during various procedures of the DRed algorithm.
2: **procedure** T.DERIVEFORADDITION()
3: $A := \mathcal{R}_n^T[I, M_a^T]$ ▷ Apply normal rules $\mathcal{R}_n^T$.
4: *schedule*(A) ▷ Populate Δ_n^T.
5: T.*insert*() ▷ $\mathcal{R}_d^T$ is applied implicitly.
6: $M_a^T := \emptyset$
7: **procedure** T.DERIVEFORDELETION()
8: $A := \mathcal{R}_n^T[I \cup M_d^T, M_d^T]$ ▷ Apply normal rules $\mathcal{R}_n^T$.
9: *schedule*(A) ▷ Populate Δ_n^T.
10: T.*delete*() ▷ $\mathcal{R}_d^T$ is applied implicitly.
11: $M_d^T := \emptyset$
12: **procedure** T.REDERIVE()
13: **return** $\{t \mid t \in D^T, T.\textit{rederivable}(t)$ is true$\}$, empty D^T

Algorithm 11 Global *schedule* Function in the Multi-scheme Framework

1: **Inputs:** The fact t to be scheduled.
2: **for** scheme $T \in \mathbb{T}$, $t.p = P_T$ **do**
3: **if** in addition & $T.\textit{insertable}(t)$ **then** $\Delta_n^T := \Delta_n^T \cup \{t\}$
4: **if** in overdeletion & $T.\textit{deletable}(t)$ **then** $\Delta_n^T := \Delta_n^T \cup \{t\}$

5.3.2.3 Scheme Properties

We now demonstrate how the functions of storage schemes should be implemented to ensure that the derivation procedure adheres to the bounds established in Table 5.7. Table 5.5 shows a set of functions for storage schemes. The derivation procedure for storage schemes follows a consistent approach, as discussed in Section 5.3.2.2. However, the remaining functions in Table 5.5 should be customised to accommodate

different data structures and reasoning requirements. The following definition formally introduces the role of a scheme T in handling facts associated with the relation P_T, specifying how facts are organised, serialised, and updated. Furthermore, it outlines the key properties that must be preserved during incremental updates and rule applications.

Definition 4 *A scheme T utilises a customised internal data structure to manage and organise facts of the relation P_T. The functions designed to operate on this data structure are listed in the first part of Table 5.5. Facts serialised from the data structure across different domains are denoted as I^T, Δ^T, and D^T. The data structure handles the application of rules defined in $\mathcal{R}_d^T$ by preserving the properties implied by these rules. The incremental update and merge change functions adhere to the following properties:*

1. *The* `insert` *function updates the data structure such that the encapsulated facts in the domain 'Δ' satisfy*

$$\big(\Delta_n^T \cup \mathcal{R}_d^T[I \cup \Delta_n^T]\big) \setminus I^T \;\subseteq\; \Delta^T \;\subseteq\; (\mathcal{R}_d^T)^\infty[I \cup \Delta_n^T] \setminus I^T. \tag{5.2}$$

   Meanwhile, the encapsulated facts in domain 'I' remain unchanged following the invocation of the `insert` *function, and facts in domain 'D' are not affected and remain empty.*

2. *The* `merge` *function updates the data structure such that encapsulated facts in domains 'I' and 'Δ' are updated as follows: $I^T = I^T \cup \Delta^T$, $\Delta^T = \emptyset$, while facts in domain 'D' remain empty.*

3. *The* `delete` *function updates the data structure such that the encapsulated facts in the domain 'Δ' satisfy*

$$\big[\Delta_n^T \cup (I \cap X)\big] \setminus \Pi^\infty[E'] \;\subseteq\; \Delta^T \;\subseteq\; I^T, \tag{5.3}$$

   in which $X = \mathcal{R}_d^T[I \cup M_d^T, M_d^T \cup \Delta_n^T]$ representing direct consequences of deleting M_d^T and Δ_n^T with regard to rules $\mathcal{R}_d^T$. Meanwhile, I^T and D^T remain unchanged.

4. *The* `remove` *function updates the data structure such that $I^T = I^T \setminus \Delta^T$, $D^T = D^T \cup \Delta^T$, $\Delta^T = \emptyset$.*

Additionally, the schedule functions highlighted at the bottom of Table 5.5, should adhere to the following properties:

1. *The* `insertable` *function returns* `true` *if and only if the input fact* t *belongs to the relation* P_T *and satisfies* $t \notin I^T \cup \Delta^T$.

2. *The* `deletable` *function returns* `true` *if and only if the input fact* t *belongs to the relation* P_T *and satisfies* $t \in I^T \setminus \Delta^T$.

3. *The* `rederivable` *function returns* `true` *if and only if the input fact* $t \in D^T$ *and satisfies at least one of the following conditions:*

   - $t \in \Pi_T[I]$
   - $t \in \Pi^\infty[E']$

The following theorem establishes the correctness of derivation procedures for a storage scheme that adheres to Definition 4. Specifically, it guarantees that the facts encapsulated in Δ^T after invoking the *deriveForDeletion* and *deriveForAddition* functions, as well as the facts returned by the *rederive* function, remain within the prescribed lower and upper bounds defined in Table 5.7. This guarantees that each function correctly captures the necessary fact modifications. As a result, the scheme maintains soundness and completeness within the multi-scheme framework, ensuring that all derivations align with the expected reasoning semantics. The proof is provided in Appendix C.2.

Theorem 2 *For a scheme that satisfies the invariants in Definition 4, the derivations encapsulated in* Δ^T *after invoking the* `deriveForDeletion` *and* `deriveForAddition` *functions, as well as the facts returned by* `rederive` *function, adhere to the lower and upper bounds specified in Table 5.7.*

5.3.3 DRed with Multiple Schemes

Finally, in this section, we present the DRed algorithm adapted to support multiple schemes, enabling both materialisation and incremental maintenance through the defined scheme interfaces, as shown in Algorithm 12. It performs the procedure of overdeletion, rederivation, and addition, following principles of the DRed algorithm introduced in Section 2.3, but additionally manages the (possibly compact) representation of derivations for different parts of the program. We first introduce the addition procedure (line 16 to 23) by aligning it with the same procedure of the DRed algorithm, which, as we discussed earlier, coincides with steps presented in Algorithm 9. Initial fresh additions are set as $E^+ \setminus E$ in line 1 of Algorithm 9; similarly, E^+ (and rederived facts A) are scheduled in corresponding schemes in line 17 while whether it

is new is determined by scheme's internal *insertable* function. The *deriveForAddition* in line 19 applies rules in Π_T for each scheme T in the current stratum, like line 3 in Algorithm 9. Fresh additions determined inside the scheme are then serialised and marked as tracked changes (M_a^T) in corresponding schemes in line 22, the same as line 5 in Algorithm 2. This is achieved by the *flagChanges* function presented in Algorithm 13. In the end, function *merge* is called to update the data structure so that serialised Δ^T is added to I^T and then emptied, as line 6 in Algorithm 9.

The overdeletion procedure is very similar to the addition procedure, as it recursively applies rules to obtain deleted facts, while addition recursively applies rules to obtain new derivations. The overdeletion process is achieved by the global *deriveForDeletion* and scheme's internal *remove* function, with the former applies rules in Π_T and derivations are encapsulated in Δ^T, and the latter merges Δ^T to I^T and then empties Δ^T. The rederivation procedure is achieved by invoking the *rederive* function for each scheme in the current stratum. The rederived facts A and E^+ are then initialised as fresh additions during the addition procedure.

The final updated materialisation after executing the DRed reasoning algorithm can be obtained by aggregating facts in the 'I' domain from all schemes $T \in \mathbb{T}$, formally $I = \bigcup_{T \in \mathbb{T}} \{I^T\}$. Theorem 2 establishes the resulting derivation facts produced by the scheme interfaces. Therefore, it is easy to verify the following theorem. The proof is provided in Appendix C.7.

Theorem 3 *The DRed algorithm with multiple schemes presented in Algorithm 12 is correct if all schemes satisfy Definition 4.*

Example 5 Continued: *We show how the initial materialisation is performed by Algorithm 12 using the same example as in Section 5.2. We denote the scheme that manages facts with the predicate R as 'scheme R' for short. For initial computation, only the addition procedure is used. The dataset is given to E^+, which is added to Δ_n^T of corresponding schemes when the schedule function is called in line 17. When the deriveForAddition function is called in line 19, in scheme A, scheduled facts $\Delta_n^T = \{A(a_i, a_{i-1}\} \mid 2 \leq i \leq n\}$ are added to scheme's underlying plain table with associated label 'Δ'; similarly, for scheme S, $\Delta_n^T = \{S(a_3, a_1)\}$ is added to the table with label 'Δ'. Lastly, the scheme that manages R does nothing since $\Delta_n^T = \emptyset$ as there are no R facts scheduled for this scheme. Then, in line 22, all these A facts with the label 'Δ' are accessed and flagged in the M_a^T of scheme R, while $S(a_3, a_1)$ with label 'Δ' is flagged in scheme A. The labels of these facts with the label 'Δ' are updated to label 'I' when calling the merge function.*

Algorithm 12 Adapted DRed Algorithm within the Multi-Scheme Framework

Require: program Π, initial dataset E, fact additions E^+, materialised facts I.
Ensure: update I from $\Pi^\infty[E]$ to $\Pi^\infty[(E \backslash E^-) \cup E^+]$

1: **if** $I = \emptyset$, $E = \emptyset$ **then initialise** schemes $\mathbb{T}$ ▷ Initialise schemes
2: $E^- := (E^- \cap E) \backslash E^+$, $E^+ := E^+ \backslash E$
3: OVERDELETE, REDERIVE, ADD
4: $E := (E \backslash E^-) \cup E^+$
5: **procedure** OVERDELETE
6: *schedule*(E^-)
7: **loop**
8: **for** scheme $T \in \mathbb{T}$ **do** T.*deriveForDeletion()* ▷ Δ^T is ready
9: **if** nothing has been derived **then break**
10: **for** scheme $T \in \mathbb{T}$ **do**
11: **for** $t \in \Delta^T$ **do** *flagChanges*(t) ▷ Label M_d^T to be used in next round
12: T.*remove*() ▷ $I^T = I^T \backslash \Delta^T$, $D^T = D^T \cup \Delta^T$, $\Delta^T = \emptyset$
13: **procedure** REDERIVE
14: $A := \emptyset$
15: **for** scheme $T \in \mathbb{T}$ **do** $A := A \cup T$.*rederive()*
16: **procedure** ADD
17: *schedule*$(A \cup (E \backslash E^-) \cup E^+)$
18: **loop**
19: **for** scheme $T \in \mathbb{T}$ **do** T.*deriveForAddition()* ▷ Δ^T is ready
20: **if** nothing has been derived **then break**
21: **for** scheme $T \in \mathbb{T}$ **do**
22: **for** $t \in \Delta^T$ **do** *flagChanges*(t) ▷ Label M_a^T to be used in next round
23: T.*merge*() ▷ $I^T = I^T \cup \Delta^T$, $\Delta^T = \emptyset$

In the derivation of the second round, scheme R applies $\mathcal{R}_n^T = \{R5\}$ in a standard way to derive $\{R(a_i, a_{i-1}) \mid 2 \leq i \leq n\}$, which are then used to construct the data structure that captures the full transitive closure $\{R(a_i, a_j) \mid 1 \leq j < i \leq n\}$. The serialised facts in Δ^T include this closure, which is flagged in scheme A for the application of the next round, while $I^T = \emptyset$. This step is highlighted in the first row of Table 5.8.

The merge function of scheme R merges the closure to domain I. Then $A(a_1, a_3)$ is derived by (R7) *in the third round with $R(a_3, a_1) \in M_a^T, S(a_3, a_1) \in I$. The fact $A(a_1, a_3)$ is flagged for scheme R for further applications.*

Finally $R(a_1, a_3)$ is derived by applying (R5) *and then incrementally incorporated into the data structure. New reachable pairs $\{R(a_i, a_j) \mid 1 \leq i \leq j \leq 3\}$ are determined internally and can be serialised in Δ^T. This step is illustrated in the second*

Algorithm 13 Global *flagChanges* Function in the Multi-scheme Framework

Inputs: The fact t to be used in the next round of application.
for scheme $T \in \mathbb{T}$ and $t.p \in B_T$ **do**
  if during addition **then**
    if $t \in M_d^T$ **then** $M_d^T := M_d^T \backslash \{t\}$, **else** $M_a^T := M_a^T \cup \{t\}$
  if during overdeletion **then** $M_d^T := M_d^T \cup \{t\}$

Context	Work Performed by *deriveForAddition*	resulting facts in various domains
$M_a^T = \{A(a_i, a_{i-1}\} \mid 2 \leq i \leq n\}$	(1) Applies $\mathcal{R}_n^T = \{R5\}$, derives $\Delta_n^T = \{R(a_i, a_{i-1}) \mid 2 \leq i \leq n\}$; (2) Calling *insert*: Construct data structure using Δ_n^T (apply $\mathcal{R}_d^T = \{R6\}$)	$I^T = \emptyset$ $\Delta^T = \{R(a_i, a_j) \mid 1 \leq j < i \leq n\}$
$M_a^T = \{A(a_1, a_3)\}$	(1) Applies $\mathcal{R}_n^T = \{R5\}$, derives $R(a_1, a_3)$; (2) Calling *insert*: incrementally insert $R(a_1, a_3)$ (apply $\mathcal{R}_d^T = \{R6\}$)	$I^T = \{R(a_i, a_j) \mid 1 \leq j < i \leq n\}$ $\Delta^T = \{R(a_i, a_j) \mid 1 \leq i \leq j \leq 3\}$

Table 5.8: Work Performed by *deriveForAddition* for Scheme R in Example 5.

row of Table 5.8. The next round derives nothing and the materialisation is completed with scheme A contains $I^T = \{A(a_i, a_{i-1}) \mid 2 \leq i \leq n\} \cup \{A(a_1, a_3)\}$ *and scheme S contains* $I^T = \{S(a_3, a_1)\}$*, which are stored in scheme's underlying plain table; while scheme R contains* $I^T = \{R(a_i, a_j) \mid 1 \leq j < i \leq n\} \cup \{R(a_i, a_j) \mid 1 \leq i \leq j \leq 3\}$ *that is represented compactly.*

5.3.4 Default Scheme

Developing schemes with data structures that satisfy Definition 4 is non-trivial, as it requires addressing challenges such as efficient storage management, incremental updates, adherence to correctness bounds, distinguishing between old and new facts, and implementing appropriate serialisation methods for different domains. To begin, we first introduce a default scheme below.

Relations that cannot be handled by any customised storage schemes are managed by the default scheme, such as the A and S predicates in Example 5 presented in Section 5.2. Thus, all rules in the default scheme are applied by the standard seminaïve evaluation, and none of them requires to be handled by the data structure: $\mathcal{R}_n^T = \Pi_T, \mathcal{R}_d^T = \emptyset$. While any tabular-like storage that supports functionality such as insertion, deletion, and retrieval of facts across different domains and access patterns can be used for the default scheme, we adopt the implementation of Motik et al. [104] for the fact table and indexing structures. This choice ensures efficient support for

all access patterns required during reasoning. The details are summarised below.

Facts in the default scheme are managed by a plain table that stores facts faithfully. Each fact in the table is associated with one or more labels to distinguish between facts in different domains. As presented in Algorithm 14, the *insert* function adds facts in Δ_n^T and associates the label 'Δ' to newly added facts, while the *merge* function updates the label 'Δ' to 'I'. Similarly, the *delete* function associates label 'Δ' to facts in Δ_n^T, while the *remove* function deletes the label 'I' and 'Δ' for facts with the label 'Δ', and adds an extra label 'D' to mark deleted facts. Serialised facts Δ^T, I^T, and D^T are then defined intuitively as facts with the corresponding label. The function *insertable* checks whether the input fact is already present in $I^T \cup \Delta^T$, and the *deletable* function checks whether the input fact is in $I^T \backslash \Delta^T$. The function *rederivable* uses backward evaluation to check whether the remaining facts I can derive the deleted fact. Specifically, for each deleted fact $t \in D^T$, it is matched to the head of rules in Π_T, and then verify whether I can match the body atoms of these rules. Then D^T is then emptied by deleting the label 'D'. The following lemma states that the default scheme with functions presented in Algorithm 14 satisfies the invariants in Definition 4, and the complete proof is provided in Appendix C.3.

Lemma 4 *The default scheme satisfies the invariants in Definition 4.*

Algorithm 14 Functions of Default Scheme

Require: T, default scheme. L_T, a plain table managing facts of relation P_T.

procedure COMPUTE I^T, Δ^T, D^T:
    return facts in L_T with the label 'I', 'Δ', and 'D', respectively.
procedure T.*INSERTABLE*(t):
    if $t \notin I^T \cup \Delta^T$ **then return** true; **else return** false.
procedure T.*INSERT()*:
    for fact $t \in \Delta_n^T$ **do**: **add** t to L_T with the label 'Δ'
procedure T.*MERGE*:
    mark facts in L_T with the label 'Δ' as 'I'
procedure T.*DELETABLE*(t):
    if $t \in I^T \backslash \Delta^T$ **then return** true; **else return** false.
procedure T.*DELETE()*:
    for fact $t \in \Delta_n^T$ **do**: **add** label 'Δ' to the fact t
procedure T.*REMOVE()*:
    for fact $t \in \Delta^T$ **do**: **delete** label 'I' and 'Δ', **add** label 'D' to the fact t
procedure T.*REDERIVABLE*(t):
    if $t \in \Pi_T[I]$ **then return** true; **else return** false.

5.3.5 Standard Rule Evaluations through Multiple Schemes

Finally, we briefly discuss how standard rule evaluation is handled within our multi-scheme framework. As highlighted in Section 5.3.2, specialised rules in a scheme Π_T^d are managed through data structure manipulation, while normal rules still require evaluation via the traditional seminaïve algorithm, as shown in line 3 of Algorithm 10.

This evaluation follows the procedure discussed in Section 2.2 and is formally defined in Equation 2.7. Specifically, rule bodies are evaluated multiple times, with each of their body atoms serving as a pivot for join processing. The standard textbook approach to join evaluation is applied, ensuring correctness without requiring modifications to the underlying storage schemes. This is feasible as long as each scheme provides data access across different domains (I, Δ, and $I \backslash \Delta$), supporting arbitrary access patterns, regardless of whether variables are bound or free. For example, an evaluation plan may determine that the relation R is being evaluated with the required domain I, where the first variable is bound. In this case, a specific iterator is constructed from the scheme to facilitate the evaluation. This approach ensures that, even when data is compactly encapsulated within the underlying data structure, it can still be efficiently accessed and utilised during evaluation.

A similar evaluation strategy is employed for query answering, although it is not the primary focus of this work. While further optimisations to the evaluation plan could be explored if schemes provide statistical insights, we leave this discussion for future perspectives in Chapter 6.

5.4 TC Scheme

This section introduces a specialised Transitive Closure (TC) scheme designed to efficiently handle transitively closed relations. As discussed in Section 5.2, this scheme extends existing approaches to support the multi-scheme framework by providing a more comprehensive set of procedures for incremental updates and serialisation of facts across various domains. We begin with an introduction to the general data structures used in our approach in Section 5.4.1. The detailed interface design of the scheme is outlined in Sections 5.4.2–5.4.4, covering how additions, deletions, and rederivations are managed. Further discussions, including complexity analysis, are provided in the Sections 5.4.5 and 5.4.6.

5.4.1 Data Structure

The data structure used in this scheme is based on nontrivial adaptations of the interval-based approach by Agrawal et al. [6], which treats TC computation as solving reachability problems over a graph. Typically, for a TC scheme T, the rule managed within the data structure $\mathcal{R}_d^T$ is a single transitive closure rule (like (R6)) that axiomatises a relation R as transitive; while any other rules that derive the same predicate are included in $\mathcal{R}_n^T$.

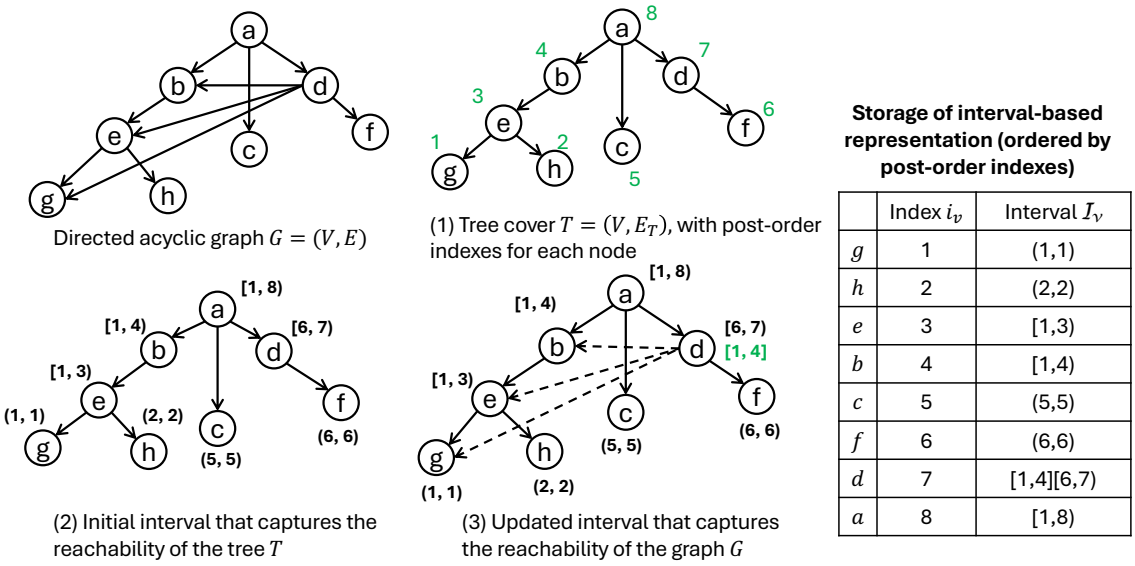
For a given set of input facts P represented as a directed graph G', the data structure constructed based on these facts can encapsulate the transitive closure of P, i.e., $(\mathcal{R}_d^T)^\infty[P]$. Following [6], a condensed directed acyclic graph (DAG) G is obtained by treating each strongly connected component (SCC) in G' as a node. Let each node $v \in G$ be associated with an index i_v and a set of intervals $\mathcal{I}_v$ so that the indexes of nodes that are reachable from v are included in $\mathcal{I}_v$. The index and intervals can be initialised as follows:

1. Compute a tree cover of the graph G. The index i_v for a node $v \in G$ is assigned as the post-order index of the tree.

   A *tree cover* of a directed graph $G = (V, E)$ is a spanning directed rooted tree $\mathsf{T} = (V, E_T)$ with $E_T \subseteq E$. On a chosen tree cover T, perform a post-order traversal and assign each node v a unique integer i_v by traversal order so that for every node v, all descendants of v in T receive smaller post-order numbers than i_v. This indexing will be helpful for step 2, in which we compute an initial interval representation that captures the transitive closure of the tree.

   Any valid tree cover can be used in this step. However, to improve storage efficiency, we adopt a tree cover that minimises the number of intervals required to represent the transitive closure compactly. This optimisation was studied by Agrawal et al. [6], and we follow their algorithmic approach in our implementation. Readers interested in the details of optimal tree cover selection are referred to [6].

2. For each node, initialise its interval $\mathcal{I}_v$ as an interval with its index i_v as the upper bound, with the smallest lower-end among its descendants' intervals as the lower bound. This interval can capture the reachability of the tree T. Please note that $\mathsf{T} = (V, E_T)$ with $E_T \subseteq E$ only contains a subset of the edges of graph G. Therefore, for a node v, the intervals $\mathcal{I}_v$ constructed in this step only include the indices of nodes that are reachable from v using edges in E_T.



Storage of interval-based representation (ordered by post-order indexes)

	Index i_v	Interval $\mathcal{I}_v$
g	1	(1,1)
h	2	(2,2)
e	3	[1,3)
b	4	[1,4)
c	5	(5,5)
f	6	(6,6)
d	7	[1,4][6,7)
a	8	[1,8)

Figure 5.3: Example of interval-based representation: computation and storage.

3. To fully capture the reachability relations in graph G, we traverse the graph in reverse topological order, so that when processing node v, all descendants of v have already been processed. For every edge $(u, v) \in E$, we update $\mathcal{I}_u := \mathcal{I}_u \cup \mathcal{I}_v \cup \{i_v\}$, meaning that u can reach whatever v can reach and v itself. After this process, for every node $v \in V$, the updated interval $\mathcal{I}_v$ contains the indices of all nodes reachable from v in graph G.

Figure 5.3 provides an example of the process to compute indices and intervals. Please note that the indices and intervals of each node are stored in an ordered table sorted by the post-order indices i_v, as shown on the right side of Figure 5.3. Additionally, in practice, we use floating-point numbers for indices, and some gaps in the intervals of leaf nodes can also be considered, which will be beneficial for incremental updates. For example, instead of $(1, 1)$ for node g in Figure 5.3, we can use $[0.5, 1)$, which allows additional indices to be introduced when edges such as (g, x) are inserted. This will be discussed in more detail in Section 5.4.2.

The serialisation of this data structure can be done by traversing each node $v \in G$, and obtaining nodes of which the index is included in $\mathcal{I}_v$. Formally,

$$I^T = \bigcup_{v \in G} \{\{C(v) \times C(u)\} \text{ if } i_u \in \mathcal{I}_v\}, \tag{5.4}$$

in which C maps each SCC to the set of elements (or nodes) that compose the current SCC. During reasoning, the serialisation of facts in different domains follows a similar process, but with different target intervals applied for each SCC. Therefore, we will

primarily focus on how the target intervals are computed for the following discussion. As shown in lines 2-12 of Algorithm 15, the data structure in the TC scheme is constructed when the *insert* function is called for the first time in the derive function shown in Algorithm 10, using scheduled facts stored in Δ_n^T. Please note that in line 8 and 12, intervals initialised as discussed above are first populated into the set $\mathcal{D}_v$ for each $v \in G$, representing facts in the domain 'Δ'. These intervals will be merged into $\mathcal{I}_v$ by the *merge* function to merge facts in Δ^T to I^T, which will be discussed in more detail below.

5.4.2 Addition

When the *insert* function is called for the second time or later, the data structure is incrementally maintained.

Schedule: The *schedule* function uses its internal *insertable* function to determine whether a fact t can be scheduled into Δ_n^T. This function returns true if the current fact is not present in the data structure, considering both I^T and Δ^T domains. The *insertable* function is the same as the *insertable* function of the default scheme presented in Algorithm 14, which we will not reiterate here.

Incremental Insertion: The *insert* function needs to maintain the data structure so that new facts (or new reachable pairs in the graph) are incorporated into the scheme and distinguished from existing ones. The incremental maintenance performed by the *insert* function is presented in lines 13-23 of Algorithm 15. To distinguish new reachable pairs, extra intervals $\mathcal{D}_v$ that are initially empty are introduced for each SCC $v \in G$. Each scheduled edge (i, j) is first inserted into the condensed graph G, followed by an update of its associated intervals. Please note that new SCCs will be introduced in G if i or j was not in the graph before, with its index assigned following the post order, all its intervals initialising as empty. We address the process of updating intervals based on two scenarios: whether existing SCCs remain distinct or merge into a new one. The former is presented in lines 17-19 of Algorithm 15, and one example is illustrated in Figure 5.4, in which (u, v) is the corresponding edge in the condensed graph of inserted fact (i, j). The left side shows that when v is not a new SCC just introduced, the insertion of (u, v) introduces a bridge between u's ancestors (node a in Figure 5.4 for example) and v's descendants (node d). The interval $\mathcal{I}_v + \mathcal{D}_v$ of v captures indexes of v's descendants, and should be added to the interval of u. A singleton interval that only contains the index of v, i.e., $[i_v]$ should be added as well. Let the interval $\mathcal{D}_u$ capture new reachable nodes from u, it can

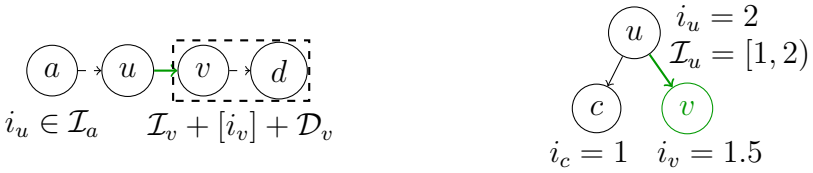


Figure 5.4: Addition Case Illustrations in which no SCCs merge.

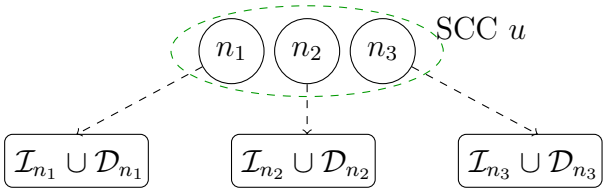


Figure 5.5: Addition Case Illustrations in which SCCs merge.

be updated by adding intervals that include indexes of v and v's descendants if they are not present in u's original interval $\mathcal{I}_u$. Formally, $\mathcal{D}_u = \mathcal{D}_u \cup (\mathcal{I}_v + [i_v] + \mathcal{D}_v) \setminus \mathcal{I}_u$, as shown in line 19 of Algorithm 15. The $\mathcal{D}_a$ interval of every ancestor a of u is updated similarly by the *propagate* function presented in Algorithm 16, as shown in line 23. The right side of Figure 5.4 shows a case in which v is a new SCC introduced because of the insertion of (u, v). Following post-order, a valid index assigned to v is 1.5 that is included in the original interval $\mathcal{I}_u$ of u and should be excluded when accessing original reachable pairs. We handle this case by recording such indexes in a global interval $\mathcal{N} = \mathcal{N} \cup [i_v]$, as shown in line 18 of Algorithm 15. One advantage of assigning index to v in this matter is that the original interval of u and u's ancestors can include the index i_v without the need to update the intervals of all ancestors. The target intervals computed for an SCC s that is not newly merged are shown in Table 5.9, in which $\mathcal{N}$ is removed from $\mathcal{I}_s$ when accessing I^T and included when accessing Δ^T.

Another addition case in which several existing SCCs merge into a new one is presented in lines 20-22 of Algorithm 15, and its illustration is presented in Figure 5.5. An example of this situation is the insertion of fact $R(a_1, a_3)$ as discussed in Section 5.2, which causes node a_1, a_2, and a_3 merge into a new SCC. In general, let a random SCC u among original SCCs $O(u) = \{n_1, n_2, n_3\}$ as a representative to represent the SCC after merging. After the merge, the SCC u can reach all nodes that are reachable from n_1, n_2, and n_3 (denoted as $\mathcal{I}_n \cup \mathcal{D}_n$ for each $n \in O(u)$), as well as n_1, n_2, and n_3 themselves. In this case, we use $\mathcal{D}_u$ to memorise nodes that are reachable from u after the merge, while target intervals for domain 'Δ' and 'I' are computed on the fly using $\mathcal{D}_u$ and intervals associated with original SCCs, as shown

in Table 5.9. Formally, $\mathcal{D}_u = \bigcup_{n \in O(u)} \{\mathcal{I}_n + [i_n] + \mathcal{D}_n\}$, as presented in line 22 of Algorithm 15. When accessing reachable nodes from one of the original SCCs $n \in O(u)$ in I^T, newly introduced SCCs in $\mathcal{N}$ are removed from n's original interval $\mathcal{I}_n$; while Δ^T is computed by removing original reachable nodes captured by $\mathcal{I}_n$ from the nodes that can be reached after the merge ($\mathcal{D}_u$), and extracting newly introduced SCCs in $\mathcal{N}$ from $\mathcal{I}_n$. Please note that intervals of original SCCs of u will not be deleted from the ordered list L at this point, which will be used when accessing facts in domain 'Δ' and 'I'.

$s \in G$	Δ_s^T	I_s^T
if s is not newly merged	$\mathcal{D}_s + (\mathcal{N} \cap \mathcal{I}_s)$	$\mathcal{I}_s \backslash \mathcal{N}$
if s is newly merged, for $n \in O(s)$	$(\mathcal{D}_s \backslash \mathcal{I}_n) + (\mathcal{N} \cap \mathcal{I}_n)$	$\mathcal{I}_n \backslash \mathcal{N}$

Table 5.9: Target intervals of SCCs in different domains.

For both cases, the changes of the interval $\mathcal{D}_u$ is passed to every ancestor a of u by the *propagate* function presented in Algorithm 16, as a should reach whatever u can reach. The interval $\mathcal{I}_u + [i_u] + \mathcal{D}_u$ covering the index of u, as well as u's descendants, is first added to $\mathcal{D}_a$ for every direct incoming edge (a, u) of u in line 4. Please note that the interval $\mathcal{D}_a$ is used differently depending on whether a is a newly merged SCC, as introduced above. If a is not a newly merged SCC, $\mathcal{D}_a$ should only include fresh reachable nodes from a. We handle this case by removing $\mathcal{I}_a$ from $\mathcal{D}_a$ in line 5. In the end, changes are passed to a's direct incoming edges by calling *propagate*(a) in line 6. The termination of the *propagate* function is guaranteed by: (1) the graph G is acyclic; (2) the stop condition presented in line 3, which avoids redundant operations by checking whether a can reach nodes in $\mathcal{D}_u$ already.

Merge Change: As presented in Algorithm 17, the *merge* function merges Δ^T to I^T by adding the interval $\mathcal{D}_s$ to $\mathcal{I}_s$ for each SCC $s \in G$, and emptying $\mathcal{D}_s$. For a newly merged SCC s, its original SCCs $O(s)$ except the representative s, can be deleted from the ordered list L as they are no longer needed. The elements contained in s are updated as the union of elements of SCCs in $O(s)$, i.e., $\bigcup_{n \in O(s)} \{C(n)\}$, in which C is the map from SCC to its contained elements. Finally, the interval $\mathcal{N}$ capturing indices of fresh SCCs that are just introduced during the previous *insert* call, is emptied.

5.4.3 Overdeletion

The *delete* function incrementally updates the data structure, allowing reachable pairs to be deleted from the transitive closure are serialised in Δ^T while preserving existing

Algorithm 15 *insert* function of TC scheme

Result: initialise the data structure using Δ_n^T or incrementally insert Δ_n^T.
if data structure is not initialised **then** ▷ Initialise the data structure
    represent facts in Δ_n^T as a directed graph G'
    compute a tree cover and a condensed directed acyclic graph G of G'
    initialise a list $L = \emptyset$, index $i = 1$, global interval capturing fresh SCCs $\mathcal{N} = \emptyset$
    for node $v \in G$ in post-order traversal of the tree cover **do**
        initialise node index $i_v = i$, interval $\mathcal{I}_v = \emptyset$
        $\mathcal{D}_v$ = [smallest lower bound among its descendants' intervals, i_v]
        push node v to the end of the list L
        increment current index i by the size of the strongly connected component v
    for node $u \in G$ in reversed topological order of G **do**
        for edge $(u, v) \in G$ **do update** the interval of u by $\mathcal{D}_u := \mathcal{D}_u + \mathcal{D}_v + \{i_v\}$
else ▷ Incrementally update the data structure
    for edge $(i, j) \in \Delta_n^T$ **do**
        find i/j's SCC u/v, initialise fresh SCC(s) when i and/or j is not present before
        update the condensed graph G by inserting edge (u, v)
        if existing SCCs remain distinct **then**
            if v is a fresh SCC **then** $\mathcal{N} := \mathcal{N} \cup [i_v]$, **then continue**
            else $\mathcal{D}_u := \mathcal{D}_u \cup (\mathcal{I}_v + [i_v] + \mathcal{D}_v)$, **do** $\mathcal{D}_u := \mathcal{D}_u \backslash \mathcal{I}_u$ if u is not newly merged
        else the insertion will cause existing SCCs merge **then**
            find SCCs that will merge, u = representative among original SCCs $O(u)$
            update the interval of u by $\mathcal{D}_u := \bigcup_{n \in O(u)} \{\mathcal{I}_n + [i_n] + \mathcal{D}_n\}$
        propagate(u) ▷ Pass changes to u's ancestors

facts in I^T.

Schedule: The *schedule* function utilises the scheme's internal *deletable* function to determine whether the input fact t can be scheduled for deletion, by verifying that t exists in the scheme and has not been deleted. Specifically, this function checks if $t \in I^T \backslash \Delta^T$, the same as the *deletable* function of the default scheme presented in Algorithm 14.

Incremental Deletion: The deletion uses interval $\mathcal{D}_s$ for each SCC $s \in G$ to record indexes of nodes that s can not reach after the deletion. As presented in Algorithm 18, the process starts by deleting edges scheduled in Δ_n^T from G. If the deletion causes an SCC s to split into multiple new SCCs, as shown in the left part of Figure 5.6, each newly formed SCC n will be assigned an index i_n, and its interval $\mathcal{I}_n$ is initialised the same as $\mathcal{I}_s$. Before the split, the SCC s can reach the nodes captured by interval $\mathcal{I}_s$, including itself. Therefore, we have $i_s \in \mathcal{I}_s$. To preserve reachability relations among nodes contained in s, we assign a number that is included in the interval $\mathcal{I}_s$ as the index for newly formed SCCs so that $\mathcal{I}_n$ captures indexes of the same set of nodes. After G

Algorithm 16 Internal *propagate* function of TC scheme

1: **Input:** an SCC $u \in G$; **Result:** pass $\mathcal{D}_u$ to the ancestors of u
2: **for** every $a \in G$ that a direct edge (a, u) exists in G **do**
3: **if** $\mathcal{D}_u \subseteq (\mathcal{I}_a + \mathcal{D}_a)$ **then** continue ▷ The break condition
4: $\mathcal{D}_a := \mathcal{D}_a \cup (\mathcal{I}_u + [i_u] + \mathcal{D}_u)$ ▷ Include u's descendants in $\mathcal{D}_a$
5: **if** a is not newly merged **then** $\mathcal{D}_a := \mathcal{D}_a \backslash \mathcal{I}_a$ ▷ Exclude intervals in domain 'I'
6: **propagate**(a)

Algorithm 17 *merge* function of TC scheme

1: **Result:** update the data structure so that encapsulated facts in Δ^T are merged into I^T, and Δ^T is emptied.
2: **for** node $s \in L$ **do**
3: **update** intervals of s, $\mathcal{I}_s := \mathcal{I}_s + \mathcal{D}_s$, $\mathcal{D}_s := \emptyset$
4: **if** s is newly merged **then**
5: $C(s) := \bigcup_{n \in O(s)} \{C(n)\}$, **delete** n from L for every $n \in O(s), n \neq s$.
6: $\mathcal{N} := \emptyset$

is updated, for each SCC $s \in G$, the interval $\mathcal{D}_s$ that captures deleted reachable nodes can be computed using its original interval $\mathcal{I}_s$ that includes reachable nodes from s before the deletion and remaining edges in G. Specifically, traverse SCCs in the graph by reversed topological order to ensure that when processing current SCC u, all its descendants have already been processed. As shown in the right part of Figure 5.6, two remaining edges from u are (u, c_1), (u, c_2), in which c_1 have been processed and $\mathcal{I}_{c_1} \backslash \mathcal{D}_{c_1}$ represents nodes that are reachable from c_1 after the deletion. The same holds for c_2 as well. The nodes that are reachable from u after the deletion can be represented as $\mathcal{D}'_u = \bigcup_{c \in \text{child}(u)} \{(\mathcal{I}_c \backslash \mathcal{D}_c) + [i_c]\}$, while the interval $\mathcal{D}_u$ representing deleted reachable nodes can be obtained by removing $\mathcal{D}'_u$ from $\mathcal{I}_u$.

Serialisation & Merge Change: For each SCC $s \in G$, the target interval that can be used to access deleted reachable nodes (Δ^T) and previous reachable nodes (I^T) is $\mathcal{D}_s$ and $\mathcal{I}_s$, respectively. The *remove* function presented in Algorithm 19 updates $\mathcal{I}_s$ for each SCC $s \in G$ as $\mathcal{I}_s \backslash \mathcal{D}_s$, adds $\mathcal{D}_s$ to $\mathcal{D}^m_s$, in which the interval $\mathcal{D}^m_s$ is used to record deleted facts that are revisited during rederivation, and finally empties D_s.

5.4.4 Rederivation

During rederivation, deleted facts are serialised by using the target interval $\mathcal{D}^m_s$ for every SCC $s \in G$. Rederivation verifies whether the deleted fact can be derived in a single application of the program using the remaining facts. Since our incremental deletion algorithm accurately maintains the transitive closure based on the specified

Algorithm 18 *delete* function of TC scheme

Result: incrementally delete Δ_n^T from the data structure Δ^T.
for edge $(i, j) \in \Delta_n^T$ **do**
    find corresponding SCC s and s' of i and j, respectively
    if $s = s'$ **then** the SCC s will break, **compute** newly formed SCCs after the deletion
        for every newly formed SCC n **do**
            assign an unoccupied index $i \in \mathcal{I}_s$ as the index i_n of n, $\mathcal{I}_n := \mathcal{I}_s$
update the condensed graph G, and the map C
for node $u \in G$ in reversed topological order of G **do**
    $\mathcal{D}'_u := \bigcup_{c \in \text{child}(u)} \{(\mathcal{I}_c \backslash \mathcal{D}_c) + [i_c]\}$
                ▷ $\mathcal{D}'_u$ covers nodes that u can reach after deletion
    if the size of the SCC $|u| > 1$ or the edge $(C(u), C(u))$ exists **then** $\mathcal{D}'_u := \mathcal{D}'_u + [i_u]$
    update the interval $\mathcal{D}_u$ to record deleted reachable nodes by $\mathcal{D}_u := \mathcal{I}_u \setminus \mathcal{D}'_u$

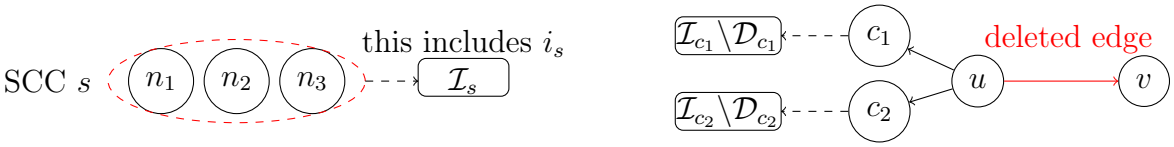


Figure 5.6: Deletion Case Illustrations.

deleted facts, no facts can be rederived through a single-step application of the transitive closure rule. Therefore, we only consider the one-step application of rules in $\mathcal{R}_n^T$ when calling *rederivable* function. Specifically, the rederivation can be performed by backward evaluation, i.e., matching a deleted fact t to the head of rules in $\mathcal{R}_n^T$, and subsequently verifying whether the body of the rule can be satisfied using remaining facts. The rederived fact t will be scheduled for addition, and the rest of the propagation will be completed during addition. The rederivation is only responsible for triggering the initial cascade reaction.

5.4.5 Complexity and Correctness

The *insert* and *delete* functions of the TC scheme maintain the data structure by preserving the transitive closure property of the underlying graph, which is equivalent to iteratively applying the transitive closure rule in $\mathcal{R}_d^T$ until the whole closure is derived. Formally, properties of the *insert* and *delete* functions are presented in Claim 1 and 2, respectively. Intuitively, $I^T = (\mathcal{R}_d^T)^\infty[N]$ represents all reachable pairs of the graph with edges in N, and captures all facts in the transitive closure of the graph G. When facts in Δ_n^T are added by the *insert* function, new reachable pairs, that exist in the updated graph with edges $N \cup \Delta_n^T$ but are not present before,

Algorithm 19 *remove* function of TC scheme

1: **Result:** update the data structure so that encapsulated facts in Δ^T are removed from I^T, and added to D^T.
2: **for** node $s \in L$ **do**
3: $\quad \mathcal{I}_s := \mathcal{I}_s \backslash \mathcal{D}_s$, $\mathcal{D}_s^m := \mathcal{D}_s^m + \mathcal{D}_s$, $\mathcal{D}_s := \emptyset$

are captured by Δ^T, as shown in Equation 5.6. Correspondingly, when facts in Δ_n^T are deleted by the *delete* function, remaining reachable pairs of the graph with edges $N \backslash \Delta_n^T$ can be represented as $(\mathcal{R}_d^T)^\infty[N \backslash \Delta_n^T]$. The deleted facts Δ^T can be accessed by removing the remaining facts from I^T, as shown in Equation 5.8. The proof of Claims 1 and 2 is provided in Appendix C.4.

Claim 1 *During the multi-scheme DRed algorithm, given facts Δ_n^T that are to be inserted, the* `insert` *function of the TC scheme updates the data structure such that the encapsulated facts in domain 'I' and 'Δ' are maintained as follows:*

$$I^T = (\mathcal{R}_d^T)^\infty[N], \tag{5.5}$$

$$\Delta^T = (\mathcal{R}_d^T)^\infty[N \cup \Delta_n^T] \setminus I^T, \tag{5.6}$$

in which N represents existing edges of the underlying graph G before the incremental insertion.

Claim 2 *During the multi-scheme DRed algorithm, given facts Δ_n^T that are to be deleted, the* `delete` *function of the TC scheme updates the data structure such that the encapsulated facts in domain 'I' and 'Δ' are maintained as follows:*

$$I^T = (\mathcal{R}_d^T)^\infty[N], \tag{5.7}$$

$$\Delta^T = I^T \setminus (\mathcal{R}_d^T)^\infty[N \backslash \Delta_n^T], \tag{5.8}$$

in which N represents facts that are present in the underlying graph G before the incremental deletion.

As discussed in Section 5.3, the correctness of the multi-scheme DRed algorithm directly follows from the requirement that every scheme satisfies the invariants in Definition 4. With the claims introduced above, it is straightforward to verify the following lemma. Details are included in C.4.

Lemma 5 *The TC scheme satisfies the invariants in Definition 4.*

In the example introduced in Section 5.2, we discussed the time and space complexities of initial materialisation using the TC scheme and seminaïve evaluation. Specifically, given a directed graph $G = (V, E)$ represented by facts, the seminaïve evaluation derives the transitive closure in $O(|V|^3)$ time, and storing the closure requires $O(|V|^2)$ space. In contrast, constructing the interval graph uses $O(|V| + |E|)$ time because computing the tree cover and initialising intervals only requires a traversal of the graph, as shown in lines 2-12 of Algorithm 15. This reduces the overall computational cost by avoiding the explicit computation of the transitive closure. The interval representation requires $O(c|V|)$ space in general, in which c is the maximum number of intervals stored. The worst-case space requirement is $O(|V|^2)$ space, which can be reduced using an optimal tree cover introduced by Agrawal et al. [6], and is evaluated to be efficient in our experiments in Section 5.7.

With regard to the incremental addition, for an edge $R(i, j)$ to be inserted, the seminaïve evaluation is expected to be completed in $O(|V|^2)$ steps. This is because the insertion of edge $R(i, j)$ introduces $O(|V|)$ direct fresh rule instances with the form $R(i, j), R(j, k) \rightarrow R(i, k)$ and $R(k, i), R(i, j) \rightarrow R(k, j)$, in which the choice of k is bounded by the number of vertices in G. Each of these rule instances is processed by the seminaïve method at most once, which derives $O(|V|)$ facts. Each of these facts will introduce $O(|V|)$ new rule instances, giving $O(|V|^2)$ time complexity in total. In contrast, the worst-case complexity of incremental addition performed by the TC scheme can be determined by the loop in line 23, which requires $O(l)$ time given that l is the maximum length of a path in G.

Deriving consequences of the deletion of a fact $R(i, j)$ using the standard seminaïve approach requires $O(|V|^2)$ running time, similarly to the case of incremental inserting a single edge. Thus, deleting n facts would require $O(n|V|^2)$ time. In contrast, the TC scheme only requires $O(|V| + |E|)$ time because the intervals can be updated by traversing the graph, as shown in line 8 of Algorithm 18. Please note that the complexity of the TC scheme's *delete* function is independent of the number of facts to be deleted, while the worst-case complexity of the seminaïve evaluation is proportional to the number of facts to be deleted. This improvement in complexity is particularly significant for the efficiency of incremental materialisation with large deletions using the multi-scheme DRed algorithm, which is also demonstrated by our experiments in Section 5.7.

The interval-based representations of facts optimises computation and storage efficiency, but incurs a trade-off with slower access times. Retrieving facts using the TC scheme requires $O(c \cdot |V| log(|V|) + n)$ in which n represents the number of retrieved

facts and c represents the maximum number of target intervals for a node. Please note that this complexity does not depend on the reasoning phase and the domain, as the same procedure is applied uniformly across all scenarios: for each node v in the graph, (a set of) target interval(s) $\mathcal{T}_v$ about the reasoning phase and the domain is computed in constant time; for each target interval $\mathcal{T}_v^i \in \mathcal{T}_v$, we retrieve nodes whose indexes are included in the interval $\mathcal{T}_v^i$ by first locating the lower bound of $\mathcal{T}_v^i$ in the ordered list L using $O(log(|V|))$ time, and then sequentially scanning L until its upper bound is met. In contrast, retrieving n facts in a plain table requires $O(n)$ time. Although the TC scheme incurs slower access times for retrieving facts, it compensates by significantly reducing the time required to compute and incrementally update the transitive closure. As a result, reasoning using the multi-scheme DRed algorithm remains highly efficient, particularly for datasets with large transitive closure structures. Moreover, our approach achieves substantial memory savings while maintaining acceptable query performance, striking a balance between computational efficiency and storage optimisation: our evaluation presented in Section 5.7.5 shows that, regarding query answering, the TC scheme performs as efficiently as the standard approach for queries with a cardinality smaller than 1 thousand; and only 1.7 to 5 times slower than the standard approach in general.

5.4.6 Data Access

In this section, we first describe how facts stored in the TC scheme are accessed, and then present a special case in which two relations may be managed together to improve storage and computational efficiency.

Accessing facts in a scheme depends on two parameters: the *domain*: either I or Δ; and the *access pattern*. The domain indicates whether we are accessing the full materialisation (I^T) or only the newly derived facts (Δ^T). The access pattern specifies which arguments of a predicate are *bound* to constants and which are *free* variables. For instance, in the atom $R(a, ?x)$, the access pattern is `(bound, free)`, meaning that lookup is guided by a single bound argument. Access to the facts in a scheme is required in both the reasoning and querying phases: (1) As discussed in Section 5.3.5, during reasoning, the rule plan specifies which subset of facts is needed from the scheme managing the R facts, along with the appropriate access pattern for retrieval. (2) During querying, the domain is fixed as I, and the access pattern is determined by the query. These two factors jointly determine which subset of the facts is serialised and how the underlying data structures are organised to support efficient access.

We then discuss how data is serialised under four distinct access patterns:

1. `(free, free)`: Fact retrieval in the TC scheme, when both variables are unbound, can be efficiently implemented by traversing all strongly connected components (SCCs) $s \in G$ and computing the target intervals of s based on the provided input domain. These target intervals capture the set of reachable nodes from s, allowing for the reconstruction of reachable pairs. Repeating this process for all SCCs enables the retrieval of all facts within the scheme for a given domain.

2. `(bound, free)`: When the first argument is bound, fact retrieval in the TC scheme remains efficient. Given a source node, we locate the strongly connected component (SCC) that contains it and compute its corresponding target interval based on the specified domain. The set of reachable nodes can then be directly obtained from this interval.

3. `(free, bound)`: However, answering queries where the second argument is bound, such as retrieving all nodes that can reach a given node, poses additional challenges. Unlike forward reachability, which is naturally supported by the interval encoding, backward reachability requires extra processing. We refer to the interval computed from the original graph as the *forward interval encoding*. To efficiently support backward reachability, we additionally compute and maintain both indexes and intervals over the reverse graph of G. The reverse graph encodes backward reachability and enables efficient access to all nodes that are reachable *from* a given node. For access patterns of the form `(free, bound)`, we use the interval associated with the bound constant, which is computed from the reverse graph, to retrieve all such nodes. For simplicity, we refer to the intervals computed from the reverse graph as *backward interval encoding*.

4. `(bound, bound)`: This access pattern can be efficiently supported using either the forward or backward interval encoding. Take $R(a, b)$ as an example: using the forward interval encoding, we first locate the SCC containing the bound argument a, then compute its target interval based on the specified domain. Finally, we check whether the index associated with b falls within this interval.

A Special Case: We now discuss a special case in which two relations can be managed within a single TC scheme. Suppose we have two relations R_1 and R_2, each defined by transitive rules, and furthermore, these relations are *mutually reversed*;

that is, the rules $R_1(x, y) \rightarrow R_2(y, x)$ and $R_2(x, y) \rightarrow R_1(y, x)$ are both present. In this setting, the forward interval encoding of R_1 coincides with the backward interval encoding of R_2, and vice versa. As a result, the transitive closures of R_1 and R_2 can be maintained within the same TC scheme: backward reachability in R_1 corresponds exactly to forward reachability in R_2, and vice versa. In particular, if a TC scheme manages R_1, and R_2 is a mutually reversed, transitive relation, then R_2 can be supported without additional indexing by reusing the stored reverse reachability information of R_1.

During incremental materialisation, when processing an inserted edge $R_1(u, v)$, we maintain the graph G and its reverse graph by inserting (u, v) and (v, u), respectively. Similarly, when fact $R_2(u, v)$ is inserted, we maintain the graph G and its reverse graph by inserting (v, u) and (u, v), respectively. The same stands for the deletion as well. In this way, the TC scheme supports all required access patterns. As discussed in Section 5.3.5, schemes that provide full access pattern and domain support are sufficient for rule evaluation within our multi-scheme framework.

5.5 Union Scheme

Another type of rule that can be optimised is union rules with the form: $A(x, y) \rightarrow U(x, y), B(x, y) \rightarrow U(x, y)$. These rules derive facts with the predicate U by using the instantiations from facts with the predicate A and B. A specialised union scheme T is initialised for predicate $P_T = U$ if there are two or more associated union rules.[3] These union rules are included in $\mathcal{R}_d^T$. Any other rules that derive U are included in $\mathcal{R}_n^T$. Define a set of predicates $\mathcal{U}_T$ as the predicates present in the body atoms of these union rules. For facts with predicate U that are derived by union rules, the existence of a fact $U(a, b)$ can be determined by checking its underlying facts with predicates in $\mathcal{U}_T$. Specifically, $U(a, b)$ holds if either $A(a, b)$ or $B(a, b)$ exists. Therefore, we don't have to apply union rules and store consequences, but calculate corresponding facts on demand. While for rules in $\mathcal{R}_n^T$ that cannot be handled specially, they still need to be applied and their consequences are stored explicitly. We use a plain table denoted as L_U to store derivations of $\mathcal{R}_n^T$. Facts in table L_U with the label 'Δ' and 'I' are denoted as L_U^{Δ} and L_U^I respectively.

A fact $U(x, y)$ exists if $U(x, y)$ is stored explicitly, or any of $R(x, y)$ exists for $R \in \mathcal{U}_T$. Define an operator $\Gamma(F)$ that returns a set of facts by replacing the predicate

[3]Here, we focus on explaining the rule shapes that can benefit from the optimisation of the union scheme. The detailed approach to scheme initialisation is provided in Section 5.7.1, with further discussions presented in Section 5.9.

of a fact $t \in F$ as U if t has a predicate in $\mathcal{U}_T$. Formally, $\Gamma(F) = \{U(x, y)$ for $R(x, y) \in F$ and $R \in \mathcal{U}_T\}$. The serialised U facts in the union scheme T are

$$I^T = L_U^I \cup \Gamma(\bigcup_{R \in \mathcal{U}_T} \{I^R\}), \tag{5.9}$$

in which the operator Γ can be considered a replacement for applying union rules in $\mathcal{R}_d^T$. Then we discuss how incremental maintenance is performed and how facts in different domains are defined.

5.5.1 Addition

The *insertable* function checks if the given fact is present in $\Delta^T \cup I^T$. In the *deriveForAddition* function, normal rules $\mathcal{R}_n^T$ are first applied and new derivations are inserted to table L_U with an associated label 'Δ' to modified facts, by calling the *insert* function. While derivations of union rules $\mathcal{R}_d^T$ are obtained by applying the operator Γ to tracked changes M_a^T, when needed. The fresh additions Δ^T is defined as the union of derivations by $\mathcal{R}_n^T$ and $\mathcal{R}_d^T$ with facts in I^T are removed:

$$\Delta^T = \left(L_T^\Delta \cup \Gamma(M_a^T)\right) \setminus I^T. \tag{5.10}$$

The serialisation of I^T during addition is defined as follows:

$$I^T = L_T^I \cup \Gamma(\bigcup_{R \in \mathcal{U}_T} \{I^R \setminus M_a^T\}), \tag{5.11}$$

in which L_T^I represents stored facts with the 'I' label in L_U, and facts from underlying schemes are rewritten by the operator Γ. The *merge* function updates the label of facts marked as 'Δ' to 'I' for facts stored in L_U, while the status of derivations from union rules is dependent on corresponding underlying schemes and does not require further updates.

5.5.2 Overdeletion

In the *deriveForDeletion* function, normal rules $\mathcal{R}_n^T$ are first applied and their derivations can be scheduled for deletion if it is still in $T^I \backslash T^\Delta$, which is determined by the *deletable* function. The *delete* function adds a label 'Δ' to facts that are scheduled for deletion. Serialised facts in Δ^T should include derivations from $\mathcal{R}_n^T$ and $\mathcal{R}_d^T$. The former includes facts in L_U with label 'Δ' (L_U^Δ), while the latter is can be obtained

by $\Gamma(M_d^T)$. For each fact $t \in \Gamma(M_d^T)$, it is a fresh deletion if $t \in I^T$. Finally, fresh deleted facts are defined as follows:

$$\Delta^T = L_T^{\Delta} \cup (\Gamma(M_d^T) \cap I^T). \tag{5.12}$$

The I^T during overdeletion is defined as:

$$I^T = L_U^I \cup \Gamma(\bigcup_{R \in \mathcal{U}_T} \{I^R\} \cup M_d^T), \tag{5.13}$$

in which the first part denotes the I facts stored explicitly in table L_T, while the second part rewrites facts in $I^R \cup M_d^T$ for each supporting predicate $R \in \mathcal{U}_T$. Deleted facts in M_d^T need to be included because underlying facts are deleted from their corresponding schemes, but the union consequences $\Gamma(M_d^T)$ are not. The update function *remove* deletes labels 'I' and 'Δ' for facts in L_T^{Δ}, and adds an extra label 'D' to mark deleted facts, just like plain table in default scheme.

5.5.3 Rederivation

The deleted facts can be obtained by collecting facts with the label 'D' in table L_U (defined as L_U^D) and performing the rewrite operator Γ to deleted facts of underlying schemes. However, this approach would be very time-consuming. Instead, we just record deleted facts D^T in the union scheme when calling the *remove* function. For each deleted fact $t \in D^T$, backward evaluation is performed to determine whether it can be rederived.

5.5.4 Correctness

The union scheme stores derivations of normal rules $\mathcal{R}_n^T$ in a plain table L_U, implemented similarly to the plain table in the default scheme, as presented in Algorithm 14. However, the union scheme uses different procedures to compute facts in various domains. Specifically, derivations of normal rules $\mathcal{R}_n^T$ are retrieved from the table L_U; while derivations of union rules $\mathcal{R}_d^T$ are computed and retrieved on the fly by rewriting facts with predicates in $\mathcal{U}_T$ using the operator Γ. For a set of facts F, applying Γ is equivalent to applying the union rules $\mathcal{R}_d^T$, i.e., $\Gamma(F) = \mathcal{R}_d^T[F]$. Combining with the definition of facts in various domains during different reasoning phases shown in Equations 5.10-5.13, the union scheme has the properties presented in Claims 3 and 4. Then Lemma 6 stating that the union scheme satisfies the invariants in Definition 4 can be easily verified. The proof is presented in C.5.

Claim 3 *Given facts Δ_n^T that are to be inserted and current tracked changes M_a^T, the* `insert` *function updates the data structure so that the encapsulated facts in domain 'Δ' are as follows:*

$$\Delta^T = (\Delta_n^T \cup \mathcal{R}_d^T[M_a^T]) \setminus I^T, \tag{5.14}$$

which is exactly the lower bound of Δ^T defined in Equation 5.2 in Definition 4.

Claim 4 *Given facts Δ_n^T that are to be deleted and current tracked changes M_d^T, the* `delete` *function updates the data structure so that the encapsulated facts in domain 'Δ' are as follows:*

$$\Delta^T = \Delta_n^T \cup (\mathcal{R}_d^T[M_d^T] \cap I^T), \tag{5.15}$$

which adheres to the bounds of Δ^T in Equation 5.3 in Definition 4.

Lemma 6 *The union scheme satisfies the invariants in Definition 4.*

5.5.5 Complexity

The union scheme optimises memory usage by avoiding the application of union rules $\mathcal{R}_d^T$ and refraining from storing the derivations of these rules, instead handling them dynamically during reasoning. Given tracked changes M with the size n, applying union rules by the seminaïve approach requires $O(n)$ time, and storing these consequences requires $O(n)$ space. In contrast, the union scheme skips the application of union rules, and requires $O(kn)$ space in which k represents the fraction of facts that are stored explicitly, and $0 \leq k \leq 1$. This design saves memory but comes at the cost of increased access time, as the dynamic application of union rules during reasoning introduces additional computational overhead compared to precomputing and storing derivations. Specifically, retrieving all the facts from a plain table storing n facts would require $O(n)$ time; while retrieving from the union scheme would require $O(cl)$ time in which c is the number of supporting predicates $\mathcal{U}_T$, and l is the maximum cardinality of facts with these predicates.

5.5.6 Data Access

As discussed in Section 5.3.5, schemes should support data access with all possible access patterns (i.e., indicating which arguments are bound and which are free) and domains (i.e., indicating which subset of facts is required). Data access across different domains is covered by Equations (5.10)–(5.13), which formally specify which subsets

of facts should be included in the I and Δ domains. Therefore, we mainly discuss how various access patterns are supported in the union scheme below.

In the standard approach, derivations of union rules (denoted as U-facts) are stored explicitly and can thus be accessed directly. In contrast, under the union scheme, union facts are not materialised and stored directly. Instead, accessing a union fact involves checking the facts with the predicate that derives U-facts, i.e., predicates in $\mathcal{U}_T$. Given a specific access pattern, it decomposes the query into corresponding subqueries over the relations in $\mathcal{U}_T$ to enable efficient retrieval. For example, a request to access $U(a, ?x)$ would be translated into queries over $A(a, ?x)$ and $B(a, ?x)$, assuming $A, B \in \mathcal{U}_T$. Here, we use the term *query* to refer broadly to both access operations invoked during rule evaluation and those issued in response to external queries.

5.6 Optimisations by Counting

In the discussion above, rederivation is achieved by backward evaluation. In this section, we show how rederivation can be optimised by tracking the number of recursive derivations and non-recursive derivations, following the idea proposed by Hu et al. [75]. The number of recursive and non-recursive derivations are maintained during addition for each derived fact, depending on whether it was derived by a recursive or a non-recursive rule. When performing overdeletion, a fact cannot be deleted if it still has non-recursive derivations. Thus, non-recursive counting can help to stop the propagation of overdeletion. During rederivation, a fact that is deleted during overdeletion can be recovered if it still has a positive number of recursive derivations. The counting technique avoids backward evaluation, which improves efficiency.

Counting in Default Scheme: The *insertable*, *deletable* and *rederivable* functions of default schemes can be further optimised to integrate counting. In the table, two numbers are associated with each stored fact representing the number of recursive and non-recursive derivations, respectively. For every fact derivation, *insertable* function increments corresponding counting by one depending on whether a recursive or non-recursive rule derives it. The *deletable* function decrements corresponding counting by one, and returns true if the current fact no longer has non-recursive counting. For every deleted fact, the *rederivable* function returns true if its recursive counting is positive.

Counting in TC Scheme: Only non-recursive counting is integrated into the TC scheme because attaching counting to every fact in compact storage is impractical.

We maintain a non-recursive backbone graph G_{nr} of the transitive closure, in which each edge in this graph is labelled with a non-negative integer indicating the number of non-recursive derivations. If the input fact is derived from a non-recursive rule, then the *insertable* function adds it to G_{nr}. If the edge already exists, the count associated with the edge will be incremented accordingly.

When *deletable* is called, the counting of input fact t decrements by one if it is derived by a non-recursive rule. This function returns true only when t doesn't have any non-recursive counting in G_{nr}. The rederivation of the TC scheme still uses backward evaluation since no recursive counting is maintained.

Counting in Union Scheme: The union scheme uses both recursive and non-recursive counting. In its internal table L_U, two numbers are associated with each stored fact representing the number of recursive and non-recursive derivations of rules $\mathcal{R}_n^T$. The *insertable* function maintains these counts for every fact derived by $\mathcal{R}_n^T$.

The number of derivations of union rules $\mathcal{R}_d^T$ is recovered by visiting its underlying schemes. For each predicate $R \in \mathcal{U}_T$, if its associated rule $R(x,y) \to U(x,y)$ in $\mathcal{R}_d^T$ is a recursive rule, then the existence of $R(x,y)$ counts as one recursive derivation of $U(x,y)$. Similarly, if the rule is a non-recursive rule, $R(x,y)$ counts as one non-recursive derivation. Define $\mathcal{U}_T^r$ and $\mathcal{U}_T^{nr}$ as the predicates that can recursively and non-recursively derive U respectively. For a fact $t = U(a,b)$, let $N_r^{L_t}$ and $N_{nr}^{L_t}$ be the recursive counting and non-recursive counting records in the table L_U, respectively. If $U(a,b)$ is not in L_U, $N_r^{L_t} = N_{nr}^{L_t} = 0$. The final recursive and non-recursive counting for the fact $U(a,b)$ can be obtained by combining the number of derivations of $\mathcal{R}_n^T$ and $\mathcal{R}_d^T$:

$$N_r^t = N_r^{L_t} + |\{R(a,b) \in I^R \text{ for } R \in \mathcal{U}_T^r\}|, \tag{5.16}$$

$$N_{nr}^t = N_{nr}^{L_t} + |\{R(a,b) \in I^R \text{ for } R \in \mathcal{U}_T^{nr}\}|, \tag{5.17}$$

in which $|S|$ denotes the cardinality of the set S. In this way, the full counting is recovered.

When the *deletable* function is called, the counting in L_U is first maintained. Then whether the input fact t can be deleted depends on whether t still has non-recursive derivations computed by Equation 5.17.

The Δ^T defined during overdeletion is updated as

$$\Delta^T = L_T^{\Delta} \cup \big((\Gamma(M_d^T) \cap I^T) \cap N_{nr}^0\big), \tag{5.18}$$

in which L_T^{Δ} denotes derivations of $\mathcal{R}_n^T$ that are in I^T and do not have non-recursive derivation; and the second part denotes the derivations of $\mathcal{R}_d^T$ that satisfy the same

condition, N^0_{nr} denotes a set of facts that have no non-recursive counting, i.e., $N^0_{nr} = \{t \in \Gamma(M^T_d) \mid N^t_{nr} = 0\}$. During rederivation, the *rederivable* function checks the recursive counting of t by Equation 5.16, and returns true if a recursive derivation can be found.

Correctness: The introduction of counting in various schemes will not affect the correctness of the multi-scheme DRed algorithm, as it simply records the (partial) number of derivations for each fact. The effect of counting can be summarised as "delete less, rederive more", since the additional information about non-recursive derivation counts allows the algorithm to identify and retain facts with remaining valid derivations during the overdeletion phase, while the information about recursive derivation counts can be used to recover facts during the rederivation phase. We prove that even in the extreme case where both recursive counting and non-recursive counting are enabled, the resulting overdeleted and rederived facts still conform to the bounds defined in Table 5.7. Then, it is straightforward to see that if only partial counting is enabled, like in the TC scheme, the overdeleted and rederived facts also adhere to the defined bounds. This ensures that the correctness of the reasoning process is maintained, as the counting mechanisms merely optimise the handling of deletions and rederivations without violating the modular framework's defined constraints. The proof of Lemma 7 is provided in C.6.

Lemma 7 *Enabling both recursive and non-recursive counting does not violate the correctness bounds. Specifically, the overdeleted facts and rederived facts adhere to the bounds defined in Table 5.7.*

5.7 Implementation and Evaluation

In this section, we first discuss an implementation detail related to scheme selection in Section 5.7.1. We then introduce various approaches in Section 5.7.2 that are included in the evaluation. In Section 5.7.3, we consider a purely transitive setting where only a transitive closure rule is included, examining how different approaches perform initial materialisation, incremental insertion and deletion for the transitive property. Section 5.7.4 evaluates initial materialisation and incremental maintenance across several benchmarks. Finally, Section 5.7.5 presents results on query answering. The discussion of hypotheses, dataset choices, and result analyses for each experiment is provided in the corresponding sections from Section 5.7.3 to Section 5.7.5.

5.7.1 Scheme Initialisation

We first present how scheme initialisation is currently achieved in the implementation; while potential issues and optimisations for this approach are presented in Section 5.9.

Current Approach: In the implementation, schemes are initialised when the Datalog program Π is imported into the system. Specialised schemes are first determined by examining the syntax of specific rules that support customised storage approaches. Currently, we look for transitive closure rules and union rules in Π and construct specialised storage schemes accordingly. To prevent multiple storage optimisations are enabled for the same relation, an order is imposed among the implemented optimisations. Currently, in our implementation, the transitive closure rule is given priority over the union rule. Predicates without an identified specialised storage scheme are then stored using the default scheme.

5.7.2 Experiment Setup

To allow for a direct and fair comparison, we implemented three approaches in the same codebase: the *standard* approach employs the DRed algorithm with the counting algorithm [75] for (incremental) materialisation, storing all facts in a single table structure identical to the plain table described in Section 5.3; the *TCModule* approach uses the optimised application method for transitive closure rules proposed by Hu et al. [77] and similarly relies on the plain table for storage; our proposed *multi-scheme* approach utilises customised storage schemes for transitive and union rules, while the plain table is used for remaining facts. Notably, if no module is activated in the TCModule approach, and no custom schemes are activated in the multi-scheme framework, all three methods behave identically. We stress that the goal of our experiments is to compare methods rather than systems, and for this reason we did not consider other Datalog systems in the experiments. All of our experiments are conducted on a Dell PowerEdge R730 server with 512GB RAM and 2 Intel Xeon E5-2640 2.60GHz processors, running Fedora 33, kernel version 5.10.8.

5.7.3 Pure Transitive Setting

We first compare the *standard*, *TCModule*, and *multi-scheme* approaches in a setting that includes only a transitive rule. This allows for a comprehensive evaluation of the proposed TC scheme algorithms in incremental insertion and deletion tasks.

Hypothesis: We hypothesise that our proposed TC scheme outperforms both the standard and TCModule approaches in three aspects: (1) the time required to construct the transitive closure, (2) the time required for its incremental maintenance, and (3) the storage required to represent the full closure.

Dataset: The experiment in this section is based on *broader* facts extracted from DBpedia [94] and a single transitive closure rule that defines the *broader* property as transitive. Details on dataset construction and experimental setup are provided in the following sections.

5.7.3.1 TC Insertion

We first examine how insertion tasks can be performed in a purely transitive setting by the three approaches, comparing their efficiency and scalability in handling initial materialisation and incremental insertions. We extracted two sets of *broader* facts from DBpedia [94] and created a program with a transitive rule for *broader*. For each dataset, we inserted the facts in four rounds: the first insertion added some facts for initial materialisation (shown in the first column of Table 5.10), while the next three insertions each added 1,000 new facts as E^+ to test incremental maintenance (the last three columns). For the smaller dataset (the upper rows), the TCModule approach optimised the running time to a large extent compared to the standard approach, but not on memory consumption. In contrast, our TC scheme approach is around 100-1000x faster than the standard approach, but only uses about $1/8 \sim 1/5$ memory, in all the tasks. For the larger dataset (the lower rows), the standard approach failed to finish the initial insertion. Our TC scheme approach finished all the tasks and only used around 1/35 of the memory used by the TCModule. Our TC scheme can also maintain the data structure quickly under addition (around 7-100x faster than the TCModule), which is beneficial for the recursive and incremental reasoning scenario.

5.7.3.2 TC Deletion

We now evaluate how incremental deletion tasks are handled in this purely transitive setting by three approaches, analysing their efficiency and robustness in updating the materialisation after fact deletions. We use extracted *broader* facts from *DBpedia* [94] and a single rule that declares the transitive property of this relation. In this case, only a TC scheme is used in the multi-scheme framework, and since our scheme can maintain the transitive closure accurately concerning deletions, the rederivation and addition phase will not be invoked. In contrast, the TCModule approach potentially triggers additional rederivation and insertion procedures, as the TC module cannot

Runs	initial materialisation			+1000 facts			+1000 facts			+1000 facts		
	0.2M ▷ 29.1M			**29.1M ▷ 29.8M**			**29.8M ▷ 30.7M**			**30.7M ▷ 53.9M**		
	time	peak	static	time	peak	static	time	peak	static	time	peak	static
Standard	2.8k	1.5k	1.3k	96.0	1.5k	1.3k	92.1	1.5k	1.3k	6.4k	2.8k	2.3k
TCModule	28.2	1.6k	1.3k	15.5	1.6k	1.4k	2.8	1.6k	1.4k	39.4	2.8k	2.4k
MS	**8.4**	**0.2k**	**0.2k**	**0.8**	**0.2k**	**0.2k**	**0.9**	**0.2k**	**0.2k**	**6.1**	**0.3k**	**0.3k**
	1.4M ▷ 1,949.3M			**1,949.3M ▷ 1,950.4M**			**1,950.4M ▷ 1,951.4M**			**1,951.3M ▷ 1,953.8M**		
	time	peak	static	time	peak	static	time	peak	static	time	peak	static
Standard	>38h	-	-	-	-	-	-	-	-	-	-	-
TCModule	3.3k	97.9k	82.3k	0.1k	97.9k	82.1k	30.0	97.9k	82.1k	1.5k	98.4k	82.5k
MS	**0.4k**	**2.4k**	**2.3k**	**2.2**	**2.4k**	**2.3k**	**3.0**	**2.4k**	**2.3k**	**14.8**	**2.4k**	**2.3k**

Table 5.10: Performance Evaluation of TC Scheme Insertion Algorithms on DBpedia's *broader* relation. The bold text in the title indicates changes in the fact count before and after materialisation. The *time* is in second, *peak* and *static* stand for the peak memory usage during the reasoning and the static memory used by the data structure, respectively. In all cases, the memory is reported in MB.

perform precise deletions. This experiment assesses the impact of varying deletion sizes on performance by comparing the efficiency of different approaches, including incremental maintenance and full re-initialisation, where materialisation is recomputed from scratch using the remaining facts. The experiment results are presented in Figure 5.7, in which we deleted various constant numbers of facts as well as a percentage of the total facts. The standard approach cannot finalise the initial materialisation, thus any subsequent deletions cannot be performed. Therefore only the TCModule (TCM) and Multi-Scheme (MS) approach is shown in Figure 5.7.

Regarding running time, the incremental materialisation performed by the multi-scheme approach ('MS-inc') is generally faster than the initialisation materialisation ('MS-init') and computing from scratch ('MS-reinit') for deletion sizes that are smaller than 0.5%. The initialisation and re-initialisation do not have significant differences for the MS method because the size of the graph is similar and the initialisation construction has the complexity of $O(|V| + |E|)$ in which $|V|$ and $|E|$ represent the number of nodes and edges, respectively. For the TCModule approach, the incremental deletion ('TCM-inc') only ran to completion when deleting 20, 50 and 250 facts, which uses a similar time compared to the initial materialisation ('TCM-init') and recomputing ('TCM-reinit'). Please note that the TCModule successfully deleted 250 facts, but failed to delete 100 facts. A potential explanation is that some facts have multiple derivation paths, and deleting only a subset of facts may partially invalidate certain derivations, requiring additional effort to search for alternative proofs. In contrast, deleting a larger set of 250 facts may eliminate entire derivation paths, thereby

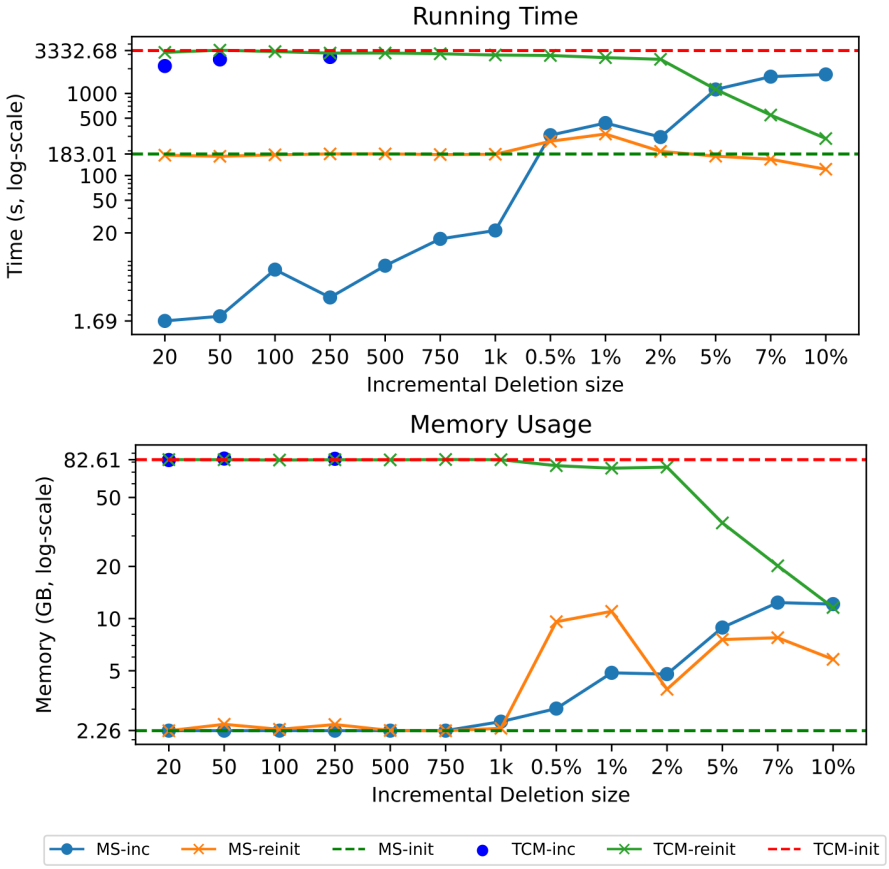


Figure 5.7: The Running Time and Memory Usage for different deletion sizes in TC schemes. The 'MS' and 'TCM' mean the Multi-Scheme and TCModule approach, respectively; while 'inc' stands for incremental deletion, 'init' represents the initial materialisation before performing incremental deletion, and 'reinit' stands for materialisation from scratch using remaining facts for each deletion.

reducing the re-derivation workload. The re-initialisation of TCModule approach uses a similar time as initial computation until a relatively large portion is deleted, i.e., after 2% deletion. Overall, our TC scheme can perform incremental materialisation in small deletions efficiently by a few seconds.

Regarding memory usage of the multi-scheme approach, the initial computation ('MS-init'), incremental materialisation ('MS-inc'), and re-initialisation ('MS-reinit') use a similar space for constant number deletions. This is because small number deletion will not cause a significant change regarding the graph structure; while for large percentage deletions, MS-reinit and MS-inc use more space compared with the initial materialisation, which is caused by potentially a larger number of SCCs included in the graph than the initial materialisation since deletion of edges can split existing

SCC. Another reason is the tree cover used for determining the indexes of SCCs is not optimal. The initialisation and re-initialisation will choose the optimal tree cover following [6], while the incremental method still uses the same indexes obtained in the initial tree cover, and only maintains the intervals so that the reachability relations are captured. The performance of the TC scheme is highly dependent on the underlying graph structure. On the other hand, the TC module generally consumes more memory until a large portion is deleted (after 2%), but still higher than the TC scheme approach.

5.7.3.3 Result Analysis

We now evaluate the results with respect to the hypothesis. Since only the TC scheme is used in the multi-scheme approach, we refer to it simply as the TC scheme in the discussion below. Overall, for incremental insertion tasks on the transitive closure, the TC scheme consistently outperforms both the standard approach and the TC-Module approach in terms of both running time and memory usage, which supports the hypothesis. For incremental deletion tasks, the TC scheme generally achieves better results than the TCModule approach for small deletions, further supporting the hypothesis. However, the incremental deletion process of the TC scheme may not always be optimal when compared to fully recomputing the transitive closure using the TC scheme, especially for large deletions. The reasons for this have been discussed in Section 5.7.3.2. Nevertheless, the TC scheme overall achieves strong performance for both incremental insertion and deletion tasks, confirming its advantages in this setting.

5.7.4 Initial and Incremental Materialisation

This section evaluates the performance of both initial and incremental materialisation. Section 5.7.4.1 introduces hypotheses. Section 5.7.4.2 presents the benchmarks used. Section 5.7.4.3 discusses the experimental results.

5.7.4.1 Hypotheses

We begin by introducing the hypotheses for this set of experiments:

1. **Storage optimisation:** The multi-scheme approach can optimise storage compared to the standard and TCModule approaches when datasets contain transitive and/or union rules.

2. **Running time optimisation:** In datasets where transitive closure rules are present, the multi-scheme approach can also optimise running time for both initial materialisation and incremental maintenance. This is because the TC scheme employs specialised storage such that, during the construction and maintenance of the data structure, the transitive closure is simultaneously computed and maintained. According to the complexity analysis in Section 5.4.5, the running time should also improve in the presence of transitive rules.

Please note that we do not expect the running time to improve when union rules are present. Although the union scheme avoids deriving and storing redundant derivations for union rules, accessing facts managed by the union scheme during reasoning can incur additional overhead. Specifically, when such facts are needed, the system must access all base relations involved in deriving the union predicate, leading to increased retrieval overhead during reasoning. Therefore, we only expect storage optimisation benefits from the union scheme, not improvements in running time. This section focuses on reasoning performance, while the impact of access time will be evaluated in Section 5.7.5.

5.7.4.2 Benchmarks

The benchmarks used are presented in Table 5.11, in which the *program* column presents the number of rules contained ($|\Pi|$), number of TC and union schemes; the *fact changes* column shows the number of facts before and after the initial materialisation. The *TC* column here presents the number of facts that are fed into the scheme (more specifically, scheduled facts in Δ_n^T) and the number of facts encapsulated in the transitive closure. This number shows that the transitive closure rule can expand the datasets extensively. The *union* column shows the number of facts with the union predicate. In all these benchmarks, almost 99% of union facts are supported by underlying tables, instead of stored explicitly, so we only show the number of facts after the materialisation. We use several general-purpose datasets (i.e., DBpedia, LUBM, and Wikidata) that contain transitive closure and/or union rules, as well as several benchmarks (i.e., TC, TC+, U, and TCU) designed to test the specialised schemes and the interactions between different types of schemes.

DBpedia: We use different sizes of *DBpedia* [94] to evaluate the scalability and performance of our approaches under varying data volumes. The DBpedia consists of structured information from Wikipedia, in which the *SKOS vocabulary*[4] is used to

[4]https://www.w3.org/TR/skos-reference/

present some background knowledge within Wikipedia categories. A Datalog subset of the *SKOS RDF* scheme is used as the program for DBpedia, in which two predicates (*broader* and *narrower*) that are mutually reversed, will be identified and included in one TC scheme. The *DB75%** uses the same fact set as *DB75%*, but with a modified program by removing the transitive rule of *narrower*, and two mutually reverse rules between *narrower* and *broader*. This dataset is useful for identifying bottlenecks when materialising.

Wikidata: *Wikidata* [147] is a collaboratively curated knowledge base that stores structured data to support Wikimedia projects and external applications. The hierarchical relationships in Wikidata are represented by the *subclassOf* (P279) and *instanceOf* (P31) properties. The *subclass* relation defines a taxonomy of classes, and thus is transitive. The *instanceOf* property links an entity to a class it belongs to. These properties form the backbone of Wikidata's ontological structure. We extracted facts with these two properties as the dataset, and translated the ontology meaning of these properties as two Datalog rules to enable logical reasoning.

LUBM: Finally, the *LUBM* [65] dataset models a university domain. We use the data generated for 500 universities and a LUBM L variant rules created by Zhou et al. [158]. The Wikidata and LUBM datasets are utilised to evaluate our methods on widely recognised benchmarks.

TC: The *TC* dataset is the DAG-R dataset used by Hu et al. [77], containing a randomly generated acyclic graph represented as a set of edges and a single rule that declares the edge predicate as transitive. This dataset is useful to test the performance of the specialised TC scheme.

TC+: The *TC+* dataset includes a program containing transitive closure rules and several rules that use the transitive predicate non-recursively and recursively, as well as synthetic facts. This dataset is useful to test the interaction between the TC scheme and default schemes.

U and TCU: The *U* and *TCU* datasets are generated to test the union scheme, and the union scheme with one of its underlying predicates is a TC predicate, respectively. These datasets capture most general cases encountered in practice associated with TC and union rules. The Datalog programs for TC, TC+, U and TCU are provided in Appendix D.

	Program			Fact Changes							
	|Π|	TC	U	Facts in I				TC			Union
DBpedia25%	41	2	2	23M	⇒	33M	(x1.4)	0.7k	⇒	4M	4M
DBpedia50%				46M	⇒	0.6B	(x13)	1M	⇒	0.3B	0.3B
DBpedia75%				69M	⇒	8B	(x124)	2M	⇒	4B	4B
DBpedia				92M	⇒	35B	(x380)	3M	⇒	17B	17B
DBpedia75%*	38	2	2	69M	⇒	4B	(x62)	2M	⇒	2B	2B
TC	2	1	0	0.1M	⇒	23M	(x228)	0.1M	⇒	23M	-
TC+	10	1	0	22k	⇒	12M	(x550)	37k	⇒	8M	-
U	14	0	1	7M	⇒	14M	(x2)	-	-	-	7M
TCU	11	2	1	13k	⇒	19M	(x1k)	6k	⇒	7M	4M
Wikidata	2	1	0	20M	⇒	0.5B	(x25)	3M	⇒	141M	-
LUBM	98	1	1	67M	⇒	91M	(x1.4)	160k	⇒	310k	2.3M

Table 5.11: Benchmarks used in Section 5.7.4.

5.7.4.3 Experiment Result

The experiments are conducted by first performing the initial materialisation, followed by deleting 100 facts to test incremental small deletions. These 100 facts are then added back for incremental small addition. Lastly, 5% facts are deleted for incremental large deletion. Please note that a large addition is omitted because the initial materialisation can be counted as a large addition. The running time of the standard, TCModule (TCM) and Multi-Scheme (MS) approaches for different tasks is presented in Table 5.12. The memory usage for different datasets is reported in Table 5.13.

For DB25%, neither the TCM nor MS approach shows significant improvement because of the overhead of customised methods. However, for other larger sizes of DBpedia datasets, the MS approach generally performs the best, even when the standard and TCM approaches cannot finalise the initial materialisation. In the DB50% dataset, the TCModule approach is faster than the standard approach, because of its optimisation for TC rules, but requires more memory. In the DB75%* dataset, the standard approach is stuck at the transitive closure rule, while the TCModule approach exhausts the memory and is subsequently terminated by the system. This experiment shows both the time and memory bottleneck associated with the transitive rule, which can be addressed by the multi-scheme approach. For the TC and TC+ datasets, the MS approach performs the best across different materialisation tasks. Specifically, our approach is about 40x-1000x faster than the standard approach, and 2x-180x faster than the TCM approach, which shows that the multi-scheme framework can handle transitive closure rules efficiently, and is suitable for use during the materialisation even when multiple rules are included. **This result partially sup-**

ports Hypothesis 2, as the benefit of the TC scheme becomes more pronounced when the transitive closure is large, in which case both the standard and TCModule approaches encounter bottlenecks.

At the same time, **we observe certain results that deviate from the expectations of Hypothesis 2**. For Wikidata and LUBM, the underlying graph of the transitive closure is very sparse, as evidenced by the expansion factor shown in Table 5.11. Consequently, the memory savings and materialisation running time improvements are not significant, as the overhead of fact retrieval and interval representation becomes pronounced in sparse graphs compared to simply storing facts plainly. Moreover, for computing and maintaining a sparse graph, the standard approach and the TCModule may outperform the TC scheme. For example, using the standard seminaïve algorithm, computing the transitive closure may require only a few recursive steps. In contrast, the TC scheme must still traverse the entire graph to construct and update the interval data structures, which can introduce additional overhead. However, our method demonstrates high efficiency in handling large deletions, which can be attributed to the effectiveness of the proposed deletion algorithm. This can be demonstrated by the complexity analysis in Section 5.4.5, as incrementally deleting n facts would require $O(n|V|^2)$ time using the seminaïve method, but only require $O(|V| + |E|)$ time using the TC scheme, in which $|V|$ and $|E|$ is the number of vertices and edges in the underlying graph, respectively.

For the U dataset, our approach demonstrates acceptable running time while using only half the memory. This is because the union scheme primarily conserves memory, but requires more time for serialisation as it involves accessing multiple underlying schemes during materialisation. For the TCU dataset, the result shows that our approach can handle the interaction between the TC scheme and union scheme well, which is significantly faster (140x-210,000x than the standard approach, and 1.3x-4,300x than the TCModule appraoch) and uses only about 40% memory. Overall, storage optimisation is observed across most datasets, **which supports Hypothesis 1**. This demonstrates the effectiveness of the optimisations in both the TC scheme and the Union scheme. However, there are exceptions in DBpedia25%, Wikidata, and LUBM. As discussed above, the benefits of the TC scheme become more pronounced as the graph size and density increase.

In summary, regarding the transitive closure rule, our proposed TC scheme can not only efficiently perform the (incremental) materialisation tasks, but also save memory, which is essential for large datasets. Additionally, our multi-scheme framework can handle the interaction between multiple rules, enabling efficient materialisation across

	Materialisation			Small Deletion (-100)		
Dataset	Standard	TCM	MS	Standard	TCM	MS
DB25%	**28.753**	35.427	39.342	**0.026**	0.065	0.580
DB50%	14,575.000	1,158.820	**989.835**	**0.100**	105.419	2.365
DB75%	>86h	>86h	**15,885.308**	-	-	**5.505**
DB75%*	>64h	>64h	**9,256.462**	-	-	**6.074**
DBpedia	-	-	**59,574.736**	-	-	**4.765**
TC	3,869.660	44.027	**22.639**	1,097.720	83.025	**6.516**
TC+	6,025.190	20.829	**5.753**	10,269.100	110.372	**19.908**
U	**9.099**	9.442	18.384	0.009	0.008	**0.006**
TCU	4,833.920	23.245	**17.577**	8,045.380	164.556	**0.038**
Wikidata	1,776.650	**840.300**	1,947.002	**0.140**	0.349	13.758
LUBM	80.185	**73.551**	179.797	**0.008**	0.009	0.145
	Large Deletion (5%)			**Small Addition (+100)**		
Dataset	Standard	TCM	MS	Standard	TCM	MS
DB25%	**3.113**	7.382	10.031	0.010	**0.009**	0.625
DB50%	13,378.200	1,699.600	**1,166.558**	0.103	**0.089**	1.474
DB75%	-	-	**6,648.236**	-	-	**2.662**
DB75%*	-	-	**6,728.426**	-	-	**3.337**
DBpedia	-	-	**27,442.412**	-	-	**3.096**
TC	5,566.400	162.786	**9.490**	18.333	9.451	**0.468**
TC+	9,869.970	168.840	**27.400**	5.454	17.017	**0.092**
U	0.582	**0.577**	2.760	0.009	0.009	**0.007**
TCU	7,709.500	341.499	**0.179**	8.498	30.551	**0.062**
Wikidata	728.774	429.41	**378.846**	**0.052**	0.070	2.805
LUBM	7.744	7.514	**6.748**	**0.013**	0.013	0.167

Table 5.12: Running time in seconds of initial and incremental materialisation.

different schemes. While our approach outperforms the standard method in many cases, an important aspect is that even when it is not the fastest, its performance remains competitive. In contrast, as our results show, the standard approach can be orders of magnitude slower in certain cases. This robustness ensures that our method provides consistently good performance across a wide range of scenarios, without suffering from severe worst-case slowdowns.

5.7.5 Query Answering

Compared to storing all derived facts explicitly in memory, storing facts in the TC scheme and Union scheme using specialised data structures introduces additional data access overhead. This represents a trade-off between storage efficiency and access time. Accessing data from these structures is expected to take longer than accessing explicitly stored facts. In this section, we evaluate a set of queries to measure the access time for both the Union and TC schemes.

Dataset	Standard	TCM	MS	DT	Other T	**Dataset**	Standard	TCM	MS	DT	Other T
DB25%	**2.1G**	2.7G	3.2G	0.9G	2.3G	DB50%	25.4G	26.3G	**7.8G**	1.9G	5.9G
DB75%	≈500G	-	**15G**	2.8G	12.2G	DB75%*	≈250G	-	**13G**	2.8G	10.2G
DBpedia	≈2000G	-	**15G**	3.8G	11.2G	TC	903M	1.0G	**893M**	4M	889M
TC+	0.54G	0.56G	**0.19G**	0.16G	0.03G	U	0.66G	0.66G	**0.38G**	0.33G	0.05G
TCU	0.78G	0.80G	**0.34G**	0.32G	0.02G	Wikidata	**23G**	24G	25G	15G	10G
LUBM	**5.5G**	5.6G	5.8G	3.5G	2.3G						

Table 5.13: Memory used to store materialisation of different datasets. The 'DT' and 'Other T' columns represent estimated memory usage for the default table and customised schemes in the multi-scheme framework, respectively.

Dataset	DB50%											TC+	TCU
TC	0.1B	0.1B	0.6k	0.4k	0.3k	5	0.1B	0.8M	1.1M	12	7	4M	3.6M
S	1.99	17.2	0.001	0	0	0	11.6	0.054	0.083	0	0	0.053	0.049
MS	10.023	15.14	0	0	0	0	39.03	0.189	0.285	0	0	0.09	0.084
Dataset	DB50%											U	TCU
Union	0.3B	0.3B	0.8M	1.1M	0.3k	12	0.3B	0.8M	1.1M	0.3k	12	7M	4M
S	3.52	32.7	0.057	0.087	0	0	34.2	0.053	0.082	0	0	0.089	0.049
MS	323.869	343.526	0.406	0.607	0.001	0	433.169	1.978	2.96	0	0	6.967	4.424

Table 5.14: Query answering time in seconds, the cardinality of each query is shown in the 2nd and the 6th row. The 'S' stands for the standard approach, while the 'MS' means the multi-scheme approach.

We tested 13 conjunctive queries that use the TC predicate and 13 queries that use the union predicate on DB50%, TC+, Union, and TCU datasets. Please refer to Appendix E for a complete list of queries used in this test. The results are presented in Table 5.14, in which corresponding datasets are illustrated as shown in the *Dataset* row. The first column presents the time to answer a query that retrieves all facts with the TC predicate and the union predicate in DB50%. Please note that we only compare the standard approach and our multi-scheme approach, because TCM uses the same storage as the standard approach, which will not affect the query answering. We can only compare query answering efficiency on datasets that all methods can finalise the materialisation, otherwise the QA is inapplicable. For queries with a cardinality smaller than 1,000, the multi-scheme framework and the standard approach exhibit similarly low query times, with no measurable difference in our experiments. This suggests that any performance difference, if present, is negligible in practical scenarios where queries typically return small results. For the TC queries, our approach is 1.7-5 times slower than the standard approach. For most union queries, our approach is about 7-40x times slower. While for the first query which retrieves all facts with the union predicate, is 90x times slower because multiple

underlying schemes are accessed, and when retrieving, schemes that have already been processed need to be checked to avoid repetitive facts. One might doubt the usefulness of the union scheme. However, in datasets such as DBpedia, most union facts are actually supported by the data structure underlying TC schemes: if we deactivate the union scheme, then we will not be able to obtain massive space improvement as depicted in Table 5.13.

5.8 Related Work on Transitive Closure

The problem of computing and representing transitive closure has been extensively studied across multiple domains, notably in databases and graph theory. Prior work has addressed efficient computation [15, 79], compressed or label-based representations [6, 41, 43, 82, 84], and indexing techniques to accelerate reachability queries [40, 42, 68, 155]. In contrast to these general approaches, our focus is on supporting *incremental maintenance* of transitive closure, and enabling efficient *domain-sensitive access*, where queries may target either the full materialisation or recent delta changes. These aspects are crucial for efficient rule evaluation and integration into a multi-scheme framework.

In this section, we review existing work on transitive closure in the literature and identify key limitations that restrict their applicability in the context of Datalog-based reasoning, particularly in incremental maintenance scenarios.

Efficient Computation and Maintenance: Some prior studies have examined efficient full materialisation of transitive closure [15, 79], but they do not sufficiently address how to store and incrementally update the materialisation. In the graph theory literature, transitive closure maintenance has been explored extensively [46, 69, 78, 91, 93, 120] as a general graph reachability problem. These approaches typically classify algorithms into *incremental*, *decremental*, or *fully dynamic* categories, depending on whether they support edge insertions, deletions, or both. Despite these advancements, these studies primarily focus on designing data structures for fast updates and reachability queries, often optimised for small-scale retrieval tasks rather than large-scale incremental reasoning, which requires efficiently tracking newly derived and deleted facts (analogous to newly formed or removed reachability pairs), maintaining derivation dependencies, and ensuring consistency across multiple updates. This is necessary to avoid redundant derivations and ensure correctness, a requirement that existing transitive closure maintenance techniques do not fully support. Moreover, they typically address a *single relation* with transitive closure

properties, whereas Datalog reasoning involves multiple interdependent rules and relations. This complexity necessitates a more generalised approach that can efficiently handle diverse rule structures, incremental updates, and dependency management within a broader reasoning framework.

The incremental maintenance of recursive Datalog programs has also been studied through *change structures* and *program derivatives* [9], providing a theoretical framework applicable to transitive closure updates. However, these approaches do not specifically address the storage efficiency and query performance requirements of large-scale transitive closure applications. Additionally, these general approaches do not exploit the specific structural properties of graphs that can be leveraged for more efficient transitive closure maintenance.

In the context of efficient computation and maintenance of transitive closure, it is worth noting the theoretical foundations established in descriptive complexity theory. Patnaik and Immerman [114] introduced the class of dynamic first-order logic problems to characterise properties of relations that can be maintained under single-tuple updates using first-order queries. Remarkably, reachability - whilst not expressible in first-order logic itself - possesses this dynamic property, meaning that transitive closure can be incrementally maintained with significantly less computational effort than reconstructing it from scratch. This theoretical insight provides important justification for the practical value of incremental transitive closure maintenance algorithms, as the maintenance cost is inherently lower than the initial construction cost.

Compact Storage: There have been various studies on compact representations of relations that can be viewed as derivations of transitive closure Datalog rules. Several works have explored efficient representations of transitively closed relations [6, 41, 43, 82, 84, 145], often modelling them as pairs of reachable nodes in a directed graph. These reachability relations can be compactly stored using auxiliary data structures designed to reduce redundancy. However, most of these studies do not thoroughly address the incremental maintenance of compact data structures, limiting their applicability in dynamic reasoning settings. Additionally, as discussed above, they lack mechanisms for efficiently identifying newly derived or deleted facts, which is crucial for avoiding redundant derivations and ensuring correctness in incremental reasoning. Moreover, it remains unclear how such compact representations can be effectively integrated into general Datalog reasoning, where rules exhibit diverse structures and complex dependencies beyond simple reachability computations. Achieving this integration requires a more generalised approach that can efficiently handle rule inter-

actions, dependency tracking, and incremental updates within the broader reasoning framework.

To address these shortcomings discussed above, we propose a Transitive Closure Scheme that satisfies the properties of our multi-scheme framework, enabling efficient storage, retrieval, and incremental updates. This scheme is designed to seamlessly integrate into the framework, ensuring compatibility with the broader materialisation and reasoning workflow while improving scalability and maintenance efficiency.

5.9 Conclusion

In this chapter, we proposed a multi-scheme framework that can perform initial and incremental materialisation for Datalog programs. The proposed framework allows for the integration of optimised storage schemes for derivations of certain types of Datalog rules with standard reasoning approaches. Specifically, we introduced general scheme interfaces that provide flexibility, as well as customised scheme implementations targeted at transitive closure rules and union rules. Our evaluation results show that the proposed approach often significantly outperforms standard approaches in terms of both space and time. Our framework thus provides a flexible and extensible alternative to existing Datalog reasoning techniques for large-scale knowledge bases. In future, we will consider integrating novel schemes for challenging rules like chain rules, symmetric transitive closure rules, general transitive closure rules, etc, enhancing the applicability of our work. Moreover, it would be interesting to develop algorithms that are able to estimate the size of the materialisation, so that one can determine in advance whether to enable space-efficient storage schemes.

We now highlight two promising lines of discussion regarding future storage scheme design: one targeting structural patterns introduced by Datalog rules derived from RDF Schema and ontology mappings, and the other concerning strategies for efficient scheme initialisation.

Future Storage Scheme Design: In addition to the transitive closure and union storage schemes explored in this thesis, it would be valuable to investigate further storage schemes that are motivated by the structure of graph-structured knowledge bases and common reasoning patterns. In particular, RDF Schema (RDFS) and lightweight Description Logics such as OWL 2 EL, QL, and RL have been studied for their mappings to Datalog rule sets [25, 53, 116, 129], often exhibiting characteristic patterns such as hierarchy propagation, typing constraints, and bounded existential reasoning. Identifying such patterns could enable the design of specialised storage

schemes optimised for these classes of rules, analogous to how the TC scheme targets transitive relations. Exploring these possibilities offers an interesting direction for further optimising the framework, particularly for reasoning over RDF-based knowledge graphs and OWL profiles.

Enhancing Scheme Initialisation Strategies: Regarding scheme initialisation approaches, while the current approach introduced in Section 5.7.1 effectively assigns storage schemes based on the syntax of rules, there are potential issues and areas for optimisation. One limitation is the rigid prioritisation order, which may not be optimal for all use cases. For example, in certain scenarios, a union rule might be more performance-efficient than a transitive closure, depending on the structure of the data and the query workload. Therefore, a more dynamic strategy that considers data distribution and access patterns could improve performance.

Additionally, the current implementation only supports one storage scheme per predicate. This constraint simplifies the design but may limit optimisation opportunities in complex Datalog programs where different rules for the same predicate could benefit from distinct storage strategies. Allowing multiple schemes for the same predicate, combined with a strategy to select the most efficient scheme during query execution, could provide further improvements. Another potential issue is the reliance on syntactic pattern matching for rule identification. This approach may fail to detect optimisation opportunities when the rules are expressed in logically equivalent but syntactically different forms. A more robust solution would involve semantic analysis to capture the underlying logical properties of rules, enabling more flexible and accurate scheme selection.

Future work could explore adaptive scheme selection based on runtime statistics, integrating cost-based optimisers, and supporting hybrid schemes for complex predicates. These enhancements would make the framework more flexible and better suited for dynamic and large-scale Datalog workloads.

Chapter 6

Conclusion and Outlook

6.1 Summary

This thesis explores optimisations for Datalog materialisation, focusing on efficient computation, incremental maintenance, and customised storage strategies. These contributions serve as a significant contribution to Datalog materialisation techniques, facilitating the scalability and applicability of large-scale knowledge graphs and materialisation-based reasoning systems.

6.1.1 Computational Efficiency of Cyclic Rules

Focusing on the computational efficiency issue of cyclic rules, this thesis explores the integration of hypertree decomposition into Datalog materialisation and incremental reasoning. While standard seminaïve evaluation often leads to redundant computations and inefficiencies in complex recursive rules, hypertree decomposition provides a structured way to optimise joins and reduce redundancy. However, its direct application to materialisation introduces additional overhead in both runtime and memory consumption. To address this, we developed a modular framework that selectively applies hypertree-based evaluation for complex rules while retaining standard Datalog reasoning for simpler cases. Our empirical evaluation demonstrates that this combined approach significantly improves performance, especially for programs involving deeply recursive rules. These findings highlight the potential of rule-specific evaluation strategies in enhancing the scalability of Datalog reasoning, paving the way for further optimisations in large-scale and dynamic knowledge systems.

Adaptability to broader cases of Datalog: These hypertree decomposition-based optimisations, as well as the hybrid approach combining HD techniques with standard

seminaïve evaluation, are applicable to broader cases of Datalog beyond the RDF subset. This generalisability stems from the fact that hypertree decomposition is defined for general multi-arity relations, without any constraints specific to unary or binary predicates. Moreover, the underlying algorithms are formulated independently of the RDF data model, allowing them to be seamlessly applied to more complex Datalog programs with higher-arity relations and diverse rule structures. Therefore, while this thesis focuses on the RDF subset, the proposed approaches have the potential to advance the state of Datalog reasoning in a wide range of knowledge-based systems, contributing to improved efficiency and scalability in real-world applications.

6.1.2 Storage Efficiency of Materialisations

Focusing on the storage efficiency of materialisations, this thesis investigates the integration of specialised storage schemes into Datalog reasoning to address the scalability challenges posed by large materialised fact bases. Many knowledge-based systems use Datalog rules to encode domain knowledge, and materialisation-based reasoning allows queries to be directly answered over precomputed consequences. However, when complex rules derive an excessive number of facts, traditional materialisation approaches may become impractical due to excessive space requirements, leading to system failure.

While certain data structures can compactly represent results derived from specific types of Datalog rules, their application in general Datalog reasoning remains unclear. To bridge this gap, we propose a general-purpose framework that enables the seamless integration of customised storage schemes into standard materialisation and incremental maintenance processes. Additionally, we develop specialised storage schemes for transitive and union rules - two rule patterns frequently used in knowledge-based systems. Our experimental results demonstrate that the proposed framework successfully computes materialisations even when the standard approach fails due to space constraints. Furthermore, it significantly enhances both storage and computational efficiency, underscoring the potential of rule-aware storage solutions in large-scale reasoning systems.

Adaptability to broader cases of Datalog: The multi-scheme framework proposed is applicable to broader cases of Datalog as it does not impose any restrictions on the arity of predicates or the complexity of rule structures. The framework is designed to be general and flexible, allowing integration with various storage schemes without being limited to unary or binary predicates. However, to support standard

rule evaluations through multiple schemes, as discussed in Section 5.3.5, if the same evaluation strategy is applied, supporting all possible access patterns in higher-arity schemes poses significant challenges. In particular, efficient join processing and index management become increasingly complex as the arity of predicates grows, necessitating more advanced indexing and retrieval mechanisms.

The union scheme can be extended to handle higher arities because it relies on decomposing access queries into sub-queries over its underlying relations, which can be generalised beyond binary predicates. However, the TC scheme utilises the graph structure of RDF triples, effectively reducing the problem to a reachability problem for transitive closure rules. This design inherently assumes two-arity relations, leveraging graph traversal techniques that are not easily generalised to higher-arity predicates. Extending the TC scheme to support higher arities would require fundamentally different approaches to encode and traverse complex hypergraphs or multi-dimensional data structures.

In summary, this thesis presents a series of optimisations for Datalog materialisation, focusing on both computation and storage efficiency. By integrating hypertree decomposition for efficient reasoning over cyclic rules and introducing a multi-scheme framework for customised storage, we address key scalability challenges in large-scale knowledge-based systems. Our contributions demonstrate that tailored reasoning and storage strategies can significantly improve the performance of materialisation-based approaches, making them more practical for real-world applications.

6.2 Future Work

While this thesis presents significant advancements in optimising Datalog materialisation, several challenges remain open for future research. In particular, further improvements can be made in optimising evaluation plans, refining storage schemes for more complex rule patterns, and exploring additional strategies for integrating materialisation with dynamic querying. We outline these future directions in the following discussion.

Extending to Datalog Variants: A natural extension of this work is to generalise the proposed techniques to more expressive Datalog variants and extensions, such as Datalog with existential quantifiers [11], Datalog$^{\pm}$ [32, 34], DatalogMTL [29, 30, 148], stratified [44, 115], well-founded [143], or stable [58] negation-as-failure, disjunction [50, 59], and aggregation [51, 89, 121]. These extensions introduce additional

reasoning complexity, such as existential quantification, temporal constraints, and non-monotonic inference, which require further adaptations to both storage and incremental reasoning frameworks. Investigating how the multi-scheme framework and hypertree-based optimisations can be adapted to support these extensions would be a valuable future direction.

Additionally, as discussed in Section 6.1, evaluating whether our proposed approach can be extended to **higher-arity** data structures represents an interesting and valuable future direction. Research has shown that, to support rule-based reasoning on Knowledge Graphs, it is sometimes necessary to use tuples of arity higher than three [92]. Extending our framework to accommodate higher-arity relations would not only broaden its applicability but also uncover unique challenges and opportunities for optimisation. Future work could explore the development of tailored storage and evaluation techniques that leverage the structural properties of higher-arity tuples, thereby enhancing both scalability and performance in advanced rule-based reasoning systems.

From another perspective, while exploring the applicability of our framework to higher-arity Datalog programs is an interesting direction, it is equally valuable to consider **optimisations specific to the arity $\leq$ 2 case**. Many practical knowledge graphs and ontologies are inherently graph-structured, where binary predicates naturally represent edges. This observation aligns with the success of graph databases such as Neo4j[1], which leverage specialised data structures and optimised query languages like Cypher [56] to efficiently support graph traversal and pattern matching. Similarly, in our framework, exploiting structural properties of labelled property graphs - the natural representation of facts with binary predicates (arity ≤ 2) — such as sparsity patterns, transitive relations, and efficient path indexing, could further improve storage and query performance. Labelled property graphs, as adopted by graph database systems such as Neo4j, model data as nodes and edges with associated key-value properties, aligning closely with binary Datalog facts. Designing such targeted optimisations for the graph case complements the broader goal of supporting general-arity Datalog programs, while enhancing performance in common real-world applications.

Beyond DRed: More Complex Incremental Algorithms: The incremental maintenance techniques explored in this thesis are generally based on the DRed algorithm, which provides a modular and efficient approach to handling fact insertions and deletions. However, more sophisticated incremental maintenance algorithms,

[1] `https://neo4j.com`

such as the *Forward/Backward Filtering (FBF)* algorithm [108], have been proposed to further optimise materialisation updates by reducing unnecessary recomputations. Future work could explore the integration of such advanced algorithms into the multi-scheme framework, potentially improving the efficiency of incremental reasoning for complex rule dependencies.

Optimising Execution Order and Program Structure Analysis: This thesis explores how to optimise the computation and storage of facts resulting from individual rule structures. However, it does not extensively investigate the execution order or overall program structure analysis for optimising rule evaluation. Recent research has shown that scheduling strategies can significantly enhance the efficiency of Datalog evaluation by optimising the order of rule execution and minimising redundant computations. For instance, Singh et al. [127] modelled recursive rule materialisation as a scheduling problem on a directed acyclic graph, effectively handling dependencies among rules to improve performance.

The approaches proposed in this thesis could also benefit from such advanced scheduling techniques. By incorporating a dependency-aware scheduler that dynamically determines the optimal execution order, the framework could further reduce computation time and enhance memory efficiency. Additionally, exploring a hybrid approach that integrates the multi-scheme framework with a scheduler could improve incremental maintenance by minimising unnecessary updates and recomputations. Future work should investigate how these scheduling strategies can be combined with the proposed optimisations to enhance not only individual rule evaluation but also the overall program execution, particularly for complex Datalog programs with cyclic dependencies and intricate inter-rule interactions.

Heuristic and Quantitative-Based Approaches: While the current work focuses on leveraging structural properties of rules for customised optimisations, another promising direction is to incorporate heuristic and quantitative information from datasets to further guide reasoning processes. Estimating the cardinality of Datalog materialisations remains a challenging problem, particularly given that even conjunctive query cardinality estimation is non-trivial [74, 130]. However, exploring effective estimation techniques could significantly enhance the efficiency of incremental reasoning, enabling more informed optimisation strategies for rule application and storage management. For example, heuristic-based approaches could be used to guide forward and backward reasoning procedures, dynamically adapting evaluation strategies based on estimated derivation sizes. Additionally, incorporating statistics from

various storage schemes could enable more advanced join evaluation plans, allowing the framework to optimise query execution based on data distribution. Such adaptive techniques could be modularly integrated into the multi-scheme framework, further enhancing its flexibility and efficiency.

Advanced Query Optimisation: Although this thesis primarily focuses on materialisation, many of the proposed techniques could be extended to optimise query answering over materialised knowledge bases. In particular, leveraging structural properties and dataset heuristics could enable the development of adaptive query execution strategies, dynamically selecting the most efficient evaluation plans based on runtime statistics. Further exploration in this direction could bridge the gap between materialisation-based reasoning and on-demand query processing, offering a more comprehensive solution for large-scale knowledge-based systems.

Hardware Acceleration and Distributed Datalog Reasoning: This thesis primarily focuses on algorithm-level exploration of efficient Datalog materialisation and incremental maintenance. The proposed techniques are CPU-oriented and single-threaded. However, significant opportunities remain to enhance performance and scalability by exploring modern hardware architectures and decentralised, distributed reasoning systems. Modern hardware advancements, such as GPUs, TPUs, and heterogeneous computing environments, provide substantial computational power and memory bandwidth that can be leveraged for Datalog processing. For example, Sun et al. [134] explored GPU-oriented storage layouts optimised for high memory throughput and massive parallelism. Building on this, future work could investigate GPU-accelerated implementations of the proposed multi-scheme framework, focusing on optimising data transfers and parallel joins for transitive closure computations. Additionally, TPUs and FPGAs could be considered for custom data flow designs, tailored to efficiently handle cyclic rule evaluations and complex join patterns.

Decentralised and distributed reasoning also presents a promising avenue for scaling Datalog materialisation to large knowledge bases. Distributed Datalog engines [5], have demonstrated the feasibility of partitioning rules and facts across multiple nodes to achieve scalability and fault tolerance. Extending the multi-scheme framework to support distributed environments would involve developing adaptive partitioning strategies that consider data locality and communication overhead. Furthermore, integrating decentralised storage solutions, such as distributed hash tables and peer-to-peer networks, could provide scalable and resilient storage for dynamic and geographically distributed knowledge graphs. Additionally, as data volume and complexity

grow, it would be beneficial to explore hybrid storage architectures that dynamically adapt to varying workloads. For example, hybrid CPU-GPU storage systems could be designed to leverage the strengths of both architectures, enabling efficient processing of highly interconnected data.

Parallelisation presents an important future direction for the broader materialisation framework proposed in this thesis. Specifically, for transitive closure approaches, whilst this work primarily focuses on sequential algorithms for transitive closure computation, the parallelisation of interval-based approaches presents an intriguing avenue for future research. The unique characteristics of interval-based representations may introduce novel challenges distinct from those encountered in parallelising standard materialisation techniques. For instance, the dependency on post-order traversal indices and the maintenance of ordered interval structures could complicate parallel decomposition strategies. Additionally, the incremental update procedures that leverage interval gaps may require careful synchronisation mechanisms to preserve correctness across concurrent modifications. Understanding whether existing parallel materialisation frameworks can be directly adapted or whether fundamentally new parallel algorithms are needed remains an open question. Further investigation into these parallelisation challenges could potentially unlock significant performance improvements for large-scale transitive closure applications, particularly in distributed computing environments where the benefits of interval-based storage efficiency would be most pronounced.

In conclusion, the optimisations presented in this thesis provide a significant step forward in improving the efficiency and scalability of Datalog reasoning. However, as knowledge bases continue to grow in size and complexity, further advancements will be needed to ensure robust and adaptive reasoning capabilities. By extending the current approaches to more expressive Datalog variants, incorporating advanced incremental maintenance techniques, and integrating heuristic-driven optimisation strategies, future research can continue to push the boundaries of scalable and efficient reasoning in knowledge-based systems.

Appendix A

Proofs of Lemmas in Chapter 4

A.1 Proof of Claims

Before providing the proofs for the three lemmas, we first prove several useful claims, which will facilitate our discussion later. In particular, we will show that during the execution of functions Add^r, Del^r, and Red^r made in the DRed algorithm, the intermediate result inst_p^I of each node $p \in N^r$ for every r is correctly maintained.

To better capture the computation of inst_p^I, we define $\Pi_p[I]$ as:

$$\Pi_p[I] = \{\chi^r(p)\sigma \mid \lambda^r(p)\sigma \subseteq I\}, \tag{A.1}$$

in which σ is a substitution. Recall that $\Pi_p[I, \Delta]$ is already defined in section 4 of the main submission, as follows:

$$\Pi_p[I, \Delta] = \{\chi^r(p)\sigma \mid \lambda^r(p)\sigma \subseteq I \text{ and } \lambda^r(p)\sigma \cap \Delta \nsubseteq \emptyset\}, \tag{A.2}$$

in which I and Δ are sets of facts with $\Delta \subseteq I$.

Also, since $\mathsf{inst}_p^{I \dot{:} \Delta^+}$ is computed by $\Pi_p[I, \Delta]$ in line 3 of Algorithm 4, we could safely infer that $\mathsf{inst}_p^{I \dot{:} \Delta^+}$ should contain the tuples as below:

$$\mathsf{inst}_p^{I \dot{:} \Delta^+} = \{\chi^r(p)\sigma \mid \lambda^r(p)\sigma \nsubseteq I \backslash \Delta^+ \text{ and } \lambda^r(p)\sigma \subseteq I \text{ and } \chi^r(p)\sigma \notin \mathsf{inst}_p^I\}, \tag{A.3}$$

in which Δ^+ is newly added tuples and $\Delta^+ \subseteq I$, σ is every possible substitution defined on $\mathsf{var}(\lambda^r(p))$ (similarly below).

Similarly, for $\mathsf{inst}_p^{I \dot{:} \Delta^-}$ that is also computed by $\Pi_p[I, \Delta]$ in line 3 of Algorithm 6, we have:

$$\mathsf{inst}_p^{I \dot{:} \Delta^-} = \{\chi^r(p)\sigma \mid \lambda^r(p)\sigma \nsubseteq I \backslash \Delta^- \text{ and } \lambda^r(p)\sigma \subseteq I \text{ and } \chi^r(p)\sigma \in \mathsf{inst}_p^I\}, \tag{A.4}$$

in which Δ^- is newly deleted tuples and $\Delta^- \subseteq I$.

By expressions (A.3) and (A.4) above, we could notice that the values of $\mathsf{inst}_p^{I \dot{:} \Delta^+}$ and $\mathsf{inst}_p^{I \dot{:} \Delta^-}$ also depend on the previous status of inst_p^I. Moreover, inst_p^I will be updated by $\mathsf{inst}_p^{I \dot{:} \Delta^+}$ in line 6 of Add^r, line 6 of Red^r and by $\mathsf{inst}_p^{I \dot{:} \Delta^-}$ in line 8 of Del^r. Therefore, whether inst_p^I is correctly maintained is related to whether inst_p^I was correctly maintained before.

Observe that during the execution of DRed, function calls to Add^r, Del^r, and Red^r are not made in an arbitrary order, but follow a certain pattern. As such, following Hu et al. [77], we define a call history H^r of the form (A.5) for each rule r as a finite and nonempty sequence of *runs* with length m; each Q_i^r with $0 \leq i \leq m$ is a finite and nonempty sequence of *calls* of the form (A.6); each $C_{i,j}^r$ is a call of function Add^r, Del^r, or Red^r.

$$H^r = Q_0^r, \ldots, Q_m^r, \tag{A.5}$$

$$Q_i^r = C_{i,1}^r, \ldots, C_{i,h_i}^r. \tag{A.6}$$

Intuitively, each run represents a sequence of calls for r during one execution of the DRed algorithm. For example, Q_0^r represents the initial materialisation, and thus it would be one Red^r call, followed by a series of Add^r calls (note that when $E = \emptyset$ function Del^r is not called due to line 10). Moreover, for each $0 \leq i \leq m$ and $1 \leq j \leq h_i$, let $(I_{i,j-1}^r, \Delta_{i,j-1}^r)$ be the arguments passed to call $C_{i,j}^r$; additionally, for each $1 \leq i \leq m$, let $I_{i-1,h_{i-1}}^r = I_{i,0}^r$.

We first show for each r, inst_p^I is correctly maintained in the call sequence of Q_0^r. Since for the first time that Red^r is called, both its parameters are empty, so we start our proof by discussing Add^r, i.e., $C_{0,j}^r$ for $1 < j \leq h_0$.

Claim 5 *For every node p, the set inst_p^I will be updated corrected by the call $\mathsf{Add}^r[I, \Delta^+]$ ($C_{0,j}^r(I_{0,j-1}^r, \Delta_{0,j-1}^r)$ with $1 < j \leq h_0$), from $\Pi_p[I_{0,j-1}^r \backslash \Delta_{0,j-1}^r]$ to $\Pi_p[I_{0,j-1}^r]$ during the initial materialisation (Q_0^r).*

Since Add^r is called recursively in Q_0^r and the correctness of Claim 5 also depends on whether inst_p^I is updated correctly when Add^r is called last time, therefore, we use induction to prove Claim 5.

1. **Induction Hypothesis:** Assume that after the execution of $C_{0,j-1}^r(I_{0,j-2}^r, \Delta_{0,j-2}^r)$, we have $\mathsf{inst}_p^I = \Pi_p[I_{0,j-2}^r]$ for every node p.

2. *Induction Base*: If the current call is $C^r_{0,2}(I^r_{0,1}, \Delta^r_{0,1})$, then before $C^r_{0,2}(I^r_{0,1}, \Delta^r_{0,1})$ is executed, inst^I_p is (initialised as) an empty set, which satisfies $\Pi_p[I^r_{0,1} \backslash \Delta^r_{0,1}] = \Pi_p[\emptyset] = \emptyset$.

3. *Induction Steps*: The current call of Add^r ($C^r_{0,j}(I^r_{0,j-1}, \Delta^r_{0,j-1})$ with $1 < j \leq h_0$) can correctly update inst^I_p if: (1) the values of inst^I_p before updating is $\Pi_p[I^r_{0,j-1} \backslash \Delta^r_{0,j-1}]$; (2) in line 3 of Algorithm 4, $\mathsf{inst}_p^{I \dot{:} \Delta^+}$ can be correctly updated to $\Pi_p[I^r_{0,j-1}] \backslash \Pi_p[I^r_{0,j-1} \backslash \Delta^r_{0,j-1}]$, then inst^I_p can be correctly updated to $\Pi_p[I^r_{0,j-1}]$ in line 6 by adding $\mathsf{inst}_p^{I \dot{:} \Delta^+}$ to inst^I_p.

   For (1), if Add^r is called for the first time (i.e. $C^r_{0,2}$) or the previous call of Add^r ($C^r_{0,j-1}(I^r_{0,j-2}, \Delta^r_{0,j-2})$ when $2 < j \leq h_0$) updates inst^I_p correctly (i.e., to $\Pi_p[I^r_{0,j-2}] = \Pi_p[I^r_{0,j-1} \backslash \Delta^r_{0,j-1}]$), then (1) is satisfied. For (2), we will prove it by showing its soundness and completeness.

   (2a) **Soundness**: for every $f = \chi^r(p)\sigma$ that $f \notin \Pi_p[I^r_{0,j-1}] \backslash \Pi_p[I^r_{0,j-1} \backslash \Delta^r_{0,j-1}]$ in which σ is the corresponding substitution, when line 3 of Add^r is executed $f \notin \Pi_p[I^r_{0,j-1}, \Delta^r_{0,j-1}] \backslash \mathsf{inst}^I_p$ (or equivalently, $f \notin \mathsf{inst}_p^{I \dot{:} \Delta^+}$).

   (i) If $f \in \Pi_p[I^r_{0,j-1}]$ and $f \in \Pi_p[I^r_{0,j-1} \backslash \Delta^r_{0,j-1}]$, then $f \in \mathsf{inst}^I_p$, therefore, $f \notin \Pi_p[I^r_{0,j-1}, \Delta^r_{0,j-1}] \backslash \mathsf{inst}^I_p$; (ii) If $f \notin \Pi_p[I^r_{0,j-1}]$, then $\lambda^r(p)\sigma \not\subseteq I^r_{0,j-1}$. According to the expression (4.1) of $\Pi_p[I, \Delta]$, $f \notin \Pi_p[I^r_{0,j-1}, \Delta^r_{0,j-1}]$. Therefore, $f \notin \Pi_p[I^r_{0,j-1}, \Delta^r_{0,j-1}] \backslash \mathsf{inst}^I_p$.

   (2b) **Completeness**: for every $f = \chi^r(p)\sigma$ that $f \in \Pi_p[I^r_{0,j-1}]$ but $f \notin \Pi_p[I^r_{0,j-1} \backslash \Delta^r_{0,j-1}]$, f can be derived by line 3 and inserted to $\mathsf{inst}_p^{I \dot{:} \Delta^+}$.

   Since $f \in \Pi_p[I^r_{0,j-1}]$ but $f \notin \Pi_p[I^r_{0,j-1} \backslash \Delta^r_{0,j-1}]$, we can have $\lambda^r(p)\sigma \subseteq I^r_{0,j-1}$ but $\lambda^r(p)\sigma \not\subseteq I^r_{0,j-1} \backslash \Delta^r_{0,j-1}$. Equivalently, $\lambda^r(p)\sigma \cap \Delta^r_{0,j-1} \neq \emptyset$. According to the expression (4.1) of $\Pi_p[I, \Delta]$, $f \in \Pi_p[^r_{0,j-1}, \Delta^r_{0,j-1}]$. Because of the condition (1) above, when line 3 is executed, inst^I_p equals to $\Pi_p[I^r_{0,j-1} \backslash \Delta^r_{0,j-1}]$. Therefore, $f \in \Pi_p[I^r_{0,j-1}, \Delta^r_{0,j-1}] \backslash \mathsf{inst}^I_p$ such that $f \in \mathsf{inst}_p^{I \dot{:} \Delta^+}$. $\square$

Without loss of generality, we can assume for each run Q^r_i when $i \geq 1$, it consists of a series of calls of Del^r ($C^r_{i,1}$ to C^r_{i,d_i}), one call of Red^r (C^r_{i,d_i+1}), and a series of calls of Add^r (C^r_{i,d_i+2} to C^r_{i,h_i}) where d_i is the number of deletion calls. We will first show this series Del^r calls in Q^r_1 will maintain the inst^I_p correctly.

Claim 6 *During the first incremental update (i.e., run Q^r_1), for every node p, inst^I_p will be updated correctly by the first call of Del^r (i.e., $C^r_{1,1}[I^r_{1,0}, \Delta^r_{1,0}]$), from $\Pi_p[I^r_{1,0}]$ to $\Pi_p[I^r_{1,0} \backslash \Delta^r_{1,0}] \backslash \Pi_p[I^r_{1,0}, \Delta^r_{1,0}]$.*

We first show after Q_0^r and before any calls of Q_1^r, the values in inst_p^I are exactly $\Pi_p[I_{1,0}^r]$. As we defined above, $I_{1,0}^r = I_{0,h_0}^r$. Following Claim 5, we can have after C_{0,h_0}^r, inst_p^I can be updated to $\Pi_p[I_{0,h_0}^r] = \Pi_p[I_{1,0}^r]$.

Then, we prove that $\Pi_p[I_{1,0}^r] \backslash (\Pi_p[I_{1,0}^r \backslash \Delta_{1,0}^r] \backslash \Pi_p[I_{1,0}^r, \Delta_{1,0}^r])$ ($F_{1,0}^r$ for simplicity) will be added to $\mathsf{inst}_p^{I \dot{:} \Delta^-}$ in line 3 of Algorithm 6 and then removed from inst_p^I in line 8 as below:

1. **Soundness**: For every $f = \chi^r(p)\sigma \notin F_{1,0}^r$, f will not be inserted to $\mathsf{inst}_p^{I \dot{:} \Delta^-}$.

   There are two cases:

   (1a) $f \notin \Pi_p[I_{1,0}^r]$ and $f \notin \Pi_p[I_{1,0}^r, \Delta_{1,0}^r]$.

   (1b) $f \in \Pi_p[I_{1,0}^r]$, $f \in \Pi_p[I_{1,0}^r \backslash \Delta_{1,0}^r]$, but $f \notin \Pi_p[I_{1,0}^r, \Delta_{1,0}^r]$.

   Both cases consist of $f \notin \Pi_p[I_{1,0}^r, \Delta_{1,0}^r]$, therefore, there is no way f will be included in $\mathsf{inst}_p^{I \dot{:} \Delta^-}$ in line 3 of Algorithm 6.

2. **Completeness**: For every $f = \chi^r(p)\sigma \in F_{1,0}^r$, f will be inserted to $\mathsf{inst}_p^{I \dot{:} \Delta^-}$.

   Since $f \in F_{1,0}^r$, then there are several cases: $f \in \Pi_p[I_{1,0}^r]$, and (1) $f \notin \Pi_p[I_{1,0}^r \backslash \Delta_{1,0}^r]$; (2) $f \in \Pi_p[I_{1,0}^r \backslash \Delta_{1,0}^r]$, but $f \in \Pi_p[I_{1,0}^r, \Delta_{1,0}^r]$. For (1), we have $\lambda^r(p)\sigma \subseteq I_{1,0}^r$, and $\lambda^r(p)\sigma \nsubseteq I_{1,0}^r \backslash \Delta_{1,0}^r$. According to the expression (4.1), $f \in \Pi_p[I_{1,0}^r, \Delta_{1,0}^r]$, then f will be added to $\mathsf{inst}_p^{I \dot{:} \Delta^-}$ in line 3. For (2), we have $\lambda^r(p)\sigma \subseteq I_{1,0}^r$, $\lambda^r(p)\sigma \subseteq I_{1,0}^r \backslash \Delta_{1,0}^r$, but $f \in \Pi_p[I_{1,0}^r, \Delta_{1,0}^r]$. In this case, since $f \in \Pi_p[I_{1,0}^r, \Delta_{1,0}^r]$, there exists another σ' s.t. $\chi^r(p)\sigma = \chi^r(p)\sigma'$, $\lambda^r(p)\sigma' \subseteq I_{1,0}^r$, and $\lambda^r(p)\sigma' \nsubseteq I_{1,0}^r \backslash \Delta_{1,0}^r$. Therefore, f will be overdeleted in line 3 of Algorithm 6. □

For simplicity, we first define a symbol that will be used below:

$$L_{i,j} := \Pi_p[I_{i,j+1}^r] \backslash \Pi_p[I_{i,j}^r, \Delta_{i,j}^r].$$

Claim 7 *For the first incremental updates (i.e., run Q_1^r), inst_p^I will be updated correctly, by the calls of Del^r after the first call (i.e., $C_{1,j+1}^r[I_{1,j}^r, \Delta_{1,j}^r]$ for $1 \leq j \leq d_1$), from $\Pi_p[I_{1,j}^r] \backslash \Pi_p[I_{1,j-1}^r, \Delta_{1,j-1}^r]$ ($L_{1,j-1}$) to $\Pi_p[I_{1,j}^r \backslash \Delta_{1,j}^r] \backslash \Pi_p[I_{1,j}^r, \Delta_{1,j}^r]$ (since $I_{1,j}^r \backslash \Delta_{1,j}^r = I_{1,j+1}^r$, so $\Pi_p[I_{1,j}^r \backslash \Delta_{1,j}^r] \backslash \Pi_p[I_{1,j}^r, \Delta_{1,j}^r]$ can be represented by $L_{1,j}$).*

1. *Induction Base*: Before $C_{1,2}^r$, the values of inst_p^I are exactly $L_{1,j-1}$ (i.e., $L_{1,0}$). Following Claim 6, inst_p^I is $\Pi_p[I_{1,0}^r \backslash \Delta_{1,0}^r] \backslash \Pi_p[I_{1,0}^r, \Delta_{1,0}^r]$, and $I_{1,1}^r = I_{1,0}^r \backslash \Delta_{1,0}^r$, so inst_p^I satisfy $\Pi_p[I_{1,j}^r] \backslash \Pi_p[I_{1,j-1}^r, \Delta_{1,j-1}^r]$ when $j = 1$.

2. *Induction Step*: The current call of Del^r ($C^r_{1,j+1}(I^r_{1,j}, \Delta^r_{1,j})$ with $2 \leq j \leq d_1$) can correctly update inst^I_p if: (1) the values of inst^I_p before updating is $L_{1,j-1}$; (2) in line 3 of Algorithm 6, $\mathsf{inst}_p^{I \dot{:} \Delta^-}$ can be correctly updated to $L_{1,j-1} \backslash L_{1,j}$, then inst^I_p can be correctly updated to $L_{1,j}$ in line 8 by removing $\mathsf{inst}_p^{I \dot{:} \Delta^-}$ from inst^I_p.

   For (1), if it is the call $C^r_{1,2}$ or the previous Del^r calls $C^r_{1,j}$ with $2 < j \leq d_1$ update inst^I_p correctly, then (1) is satisfied. For (2), we will prove it by showing its soundness and completeness.

   (2a) **Soundness**: for every $f \notin L_{1,j-1} \backslash L_{1,j}$, f will not be inserted to $\mathsf{inst}_p^{I \dot{:} \Delta^-}$.

   There are two cases:

   (i) $f \notin L_{1,j-1}$ and $f \notin \Pi_p[I^r_{1,j}, \Delta^r_{1,j}]$.

   (ii) $f \in L_{1,j-1}$ and $f \in L_{1,j}$, therefore $f \in \Pi_p[I^r_{1,j} \backslash \Delta^r_{1,j}]$ but $f \notin \Pi_p[I^r_{1,j}, \Delta^r_{1,j}]$.

   Both cases consist of $f \notin \Pi_p[I^r_{1,j}, \Delta^r_{1,j}]$, therefore, there is no way f will be inserted to $\mathsf{inst}_p^{I \dot{:} \Delta^-}$ in line 3 of Algorithm 6.

   (2b) **Completeness**: for every $f = \chi^r(p)\sigma$ that $f \in L_{1,j-1}$ but $f \notin L_{1,j}$, f can be derived by line 3 and inserted to $\mathsf{inst}_p^{I \dot{:} \Delta^-}$.

   Since $f \in L_{1,j-1} \backslash L_{1,j}$, then f must be in $L_{1,j-1}$ (i.e., $f \in \mathsf{inst}^I_p$) and

   (i) $f \notin \Pi_p[I^r_{1,j} \backslash \Delta^r_{1,j}]$, or

   (ii) $f \in \Pi_p[I^r_{1,j} \backslash \Delta^r_{1,j}]$, $f \in \Pi_p[I^r_{1,j}, \Delta^r_{1,j}]$.

   In both cases, f will be inserted to $\mathsf{inst}_p^{I \dot{:} \Delta^-}$. The proof will be similar to the completeness part of Claim 6: (i) represents the case in which $I^r_{1,j} \backslash \Delta^r_{1,j}$ cannot derive f; (ii) represents the case in which $I^r_{1,j} \backslash \Delta^r_{1,j}$ can derive f, but f is a direct consequence by deleting $\Delta^r_{1,j}$, because of $f \in \Pi_p[I^r_{1,j}, \Delta^r_{1,j}]$. □

Then, we prove the Red^r call next (i.e., C^r_{1,d_1+1}) can update inst^r_p correctly.

Claim 8 *For every node p,* inst^I_p *will be updated correctly by* $C^r_{1,d_1+1}(I^r_{1,d_1}, \Delta^r_{1,d_1})$*, from* $\Pi_p[I^r_{1,d_1}] \backslash \Pi_p[I^r_{1,d_1-1}, \Delta^r_{1,d_1-1}]$ *to* $\Pi_p[I^r_{1,d_1}]$.

First, following Claim 7, we show before C^r_{1,d_1+1}, the values of inst^I_p satisfy the condition described in Claim 8. After C^r_{1,d_1}, $\mathsf{inst}^I_p = \Pi_p[I^r_{1,d_1-1} \backslash \Delta^r_{1,d_1-1}] \backslash \Pi_p[I^r_{1,d_1-1}, \Delta^r_{1,d_1-1}]$, and $I^r_{1,d_1-1} \backslash \Delta^r_{1,d_1-1} = I^r_{1,d_1}$.

Oracle can accurately maintain the count [75] of instantiations (i.e., inst^I_p), therefore for $\mathsf{O}(f)$ will return true if and only if $f \in \Pi_p[I^r_{1,d_1}]$. Then, we only need to show

for every $f \in \Pi_p[I^r_{1,d_1-1}, \Delta^r_{1,d_1-1}]$ that needs to be checked, f also belong to inst^{re}_p. Since inst^{re}_p consists of all the overdeleted instantiations that are deleted during $C^r_{1,1}$ to C^r_{1,d_i}. For every $f = \chi^r(p)\sigma \in \Pi_p[I^r_{1,d_1-1}, \Delta^r_{1,d_1-1}]$, f will be deleted in C^r_{1,d_i} and inserted to inst^{re}_p. $\square$

Claim 9 *In the first incremental run Q^r_1, a series of Add^r calls $C^r_{1,d_1+k+1}(I^r_{1,d_1+k}, \Delta^r_{1,d_1+k})$ (for $1 \leq k < h_1 - d_1$) will update inst^I_p from $\Pi_p[I^r_{1,d_1+k} \backslash \Delta^r_{1,d_1+k}]$ to $\Pi_p[I^r_{1,d_1+k+1}]$.*

Before C^r_{1,d_1+2}, $\mathsf{inst}^I_p = \Pi_p[I^r_{1,d_1}]$ and $I^r_{1,d_1} = I^r_{1,d_1+1} \backslash \Delta^r_{1,d_1+1}$, which satisfies the condition described in Claim 9. According to Claim 5, inst^I_p will be updated corrected by each Add^r call $C^r_{1,d_1+k+1}(I^r_{1,d_1+k}, \Delta^r_{1,d_1+k})$, for $1 \leq k < h_1 - d_1$.

This is because Claim 5 establishes the correctness of a series of Add^r calls used recursively in the initial materialisation. For a series of Add^r calls $C^r_{1,d_1+k+1}(I^r_{1,d_1+k}, \Delta^r_{1,d_1+k})$ (for $1 \leq k < h_1 - d_1$) used recursively in incremental materialisation, the only difference bwteen this series of Add^r calls and the Add^r calls in the initial materialisation, is the beginning condition, i.e., the value of inst^I_p before any Add^r call in the series is called. Claim 5 proved that starting with $\mathsf{inst}^I_p = \emptyset$, inst^I_p will be updated corrected by the series of Add^r calls in the initial materialisation; Following Claim 8, before C^r_{1,d_1+2}, $\mathsf{inst}^I_p = \Pi_p[I^r_{1,d_1}]$ and $I^r_{1,d_1} = I^r_{1,d_1+1} \backslash \Delta^r_{1,d_1+1}$, which satisfies the condition described in Claim 9. Therefore, following the same principle of Claim 5, starting from a "correct" set of inst^I_p, the Add^r calls $C^r_{1,d_1+k+1}(I^r_{1,d_1+k}, \Delta^r_{1,d_1+k})$ (for $1 \leq k < h_1 - d_1$) will update inst^I_p as described in Claim 9. $\square$

Combining Claim 6, 7, 8, and 9, we can have the following claim:

Claim 10 *inst^I_p will be updated correctly by every call in run Q^r_1 as defined in Claim 6, 7, 8, and 9.*

Also, just like Claim 10, we can have the following claim:

Claim 11 *inst^I_p will be updated correctly by every call in run Q^r_i for $i \geq 1$. Specifically:*

- *the first Del^r call $C^r_{i,1}(I^r_{i,0}, \Delta^r_{i,0})$ will update inst^I_p from $\Pi_p[I^r_{i,0}]$ to $\Pi_p[I^r_{i,0} \backslash \Delta^r_{i,0}] \backslash \Pi_p[I^r_{i,0}, \Delta^r_{i,0}]$;*
- *the following Del^r calls $C^r_{i,j+1}(I^r_{i,j}, \Delta^r_{i,j})$ for $1 \leq j < d_i$ can update inst^I_p from $\Pi_p[I^r_{i,j}] \backslash \Pi_p[I^r_{i,j-1}, \Delta^r_{i,j-1}]$ to $\Pi_p[I^r_{i,j} \backslash \Delta^r_{i,j}] \backslash \Pi_p[I^r_{i,j}, \Delta^r_{i,j}]$;*
- *Then the next Red^r call C^r_{i,d_i+1} will update inst^I_p from $\Pi_p[I^r_{i,d_i}] \backslash \Pi_p[I^r_{i,d_i-1}, \Delta^r_{i,d_i-1}]$ to $\Pi_p[I^r_{i,d_i}]$;*

- *Finally, the* Add^r *calls* $C^r_{i,d_i+k+1}(I^r_{i,d_i+k}, \Delta^r_{i,d_i+k})$ *for* $1 \leq k < h_i - d_i$ *will update* inst^I_p *from* $\Pi_p[I^r_{i,d_i+k} \backslash \Delta^r_{i,d_i+k}]$ *to* $\Pi_p[I^r_{i,d_i+k}]$.

1. *Induction Base*: every step in Q^r_0 and Q^r_1 can correctly update inst^I_p just as described above (Claim 5 and 10).

2. *Induction Step*: for every call $C^r_{i,j}$ for $1 \leq j \leq h_i$, we can prove it updates inst^I_p correctly as described above. The proof will be just like Claim 6, 7, 8, and 9. □

Besides, we also prove that after the domain of each node p is fixed, the improved Yannakakis algorithm (Line 7 to 9 of Algorithm 5) can correctly join the data that is currently active in each node.

Claim 12 *Line 7 to 9 of Algorithm 5 will correctly join* inst^{ac}_p *for each* $p \in N^r$ *and project to* $\mathsf{var}(\mathsf{h}(r))$*, i.e., computing* $\pi_{\mathsf{var}(\mathsf{h}(r))}(\bowtie_{p \in N^r} \mathsf{inst}^{ac}_p)$.

Line 8 to 9 are steps of standard Yannakakis algorithms [153], therefore, line 8 and 9 correctly compute the data described in Claim 12. Then, if we could prove the values of inst^{ac}_p after using line 7 and 8 to reduce data are exactly the same as inst^{ac}_p when using only line 8 to reduce data. In other words, the left semi-join sequence we add in line 7 won't influence the results of the full reducer.

We prove this claim by following Theorem 1 of Bernstein and Chiu [21]. They show that for a tree query with n nodes, there exists an efficient semi-join sequence of length $2 \cdot (n-1)$ that reduces all nodes to retain only "useful" data. The sequence they construct first performs semi-joins in a breadth-first leaf-to-root order, followed by semi-joins from the root to each leaf node - this corresponds exactly to line 8 in Algorithm 5. Therefore, line 8 correctly performs full reduction of the data for all nodes $p \in N^r$. They further show that if a subsequence of left semi-joins LSJ_p, sufficient to fully reduce the data for p, is embedded within a larger left semi-join sequence LSJ, then LSJ will also fully reduce the data for p. This argument can be generalised as follows. Let the left semi-join sequence in line 8 be denoted by L_8, which fully reduces the data for all nodes. Now consider a larger semi-join sequence $\mathsf{L} = \mathsf{L}_7 \cup \mathsf{L}_8$, where L_7 is an additional sequence prepended before L_8 (as in line 7). Since L_8 alone suffices to fully reduce the data for all nodes, the combined sequence L will also perform full reduction. By adding this additional left semi-join sequence (line 7) before line 8, we ensure that a valid LSJ_p for every $p \in N^r$ remains embedded within the final left semi-join sequence. As a result, the outcome of applying both line 7 and line 8 is equivalent to applying only the necessary (line 8). □

A.2 Proof of Lemma 1

Lemma 1: The $\mathsf{Add}^r[I, \Delta^+]$ function shown in Algorithnm 4 computes $r[I, \Delta^+] \backslash I$.

Proof of Lemma 1. In this thesis, we use the complete hypertree decomposition T^r for every r, i.e., every B_i in $\mathsf{b}(r)$ is strongly covered. In other words, for every body atom $B_i \in \mathsf{b}(r)$, there exist at least one node p s.t. $B_i \in \lambda^r(p)$ and $\mathsf{var}(B_i) \subseteq \chi^r(p)$. We prove Lemma 1 by showing its soundness and completeness.

1. **Soundness**: For every $f \notin r[I, \Delta^+] \backslash I$, f will not be returned by Add^r.

   (1a) if $f \in r[I, \Delta^+]$ and $f \in I$, then f will be removed in line 8.

   (1b) if $f \notin r[I, \Delta^+]$, then $\mathsf{b}(r\sigma) \nsubseteq I$ or $\mathsf{b}(r\sigma) \subseteq I$ but $\mathsf{b}(r\sigma) \cap \Delta^+ = \emptyset$. For the former case, such a substitution σ is not a valid substitution for operator $\Pi_p[I, \Delta^+]$, so instantiations that use this σ will not be included in inst_p^I or $\mathsf{inst}_p^{I \vdots \Delta^+}$. Therefore, no data using this σ will be derived. For the latter case, if $\mathsf{b}(r\sigma) \cap \Delta^+ = \emptyset$, then for all $B_i \in \mathsf{b}(r)$, $B_i\sigma \cap \Delta^+ = \emptyset$. Therefore, for every node $p \in N^r$, $\lambda^r(p)\sigma \cap \Delta^+ = \emptyset$. So the instantiations that use σ are included in the inst_p^I. In line 3 of Algorithm 5, our algorithms force one node using the newly added instantiations by labeling this node as Δ^+. So in this case, f will not be derived.

2. **Completeness**: for every $f = \mathsf{h}(r\sigma)$ that $f \in r[I, \Delta^+] \backslash I$, f will be returned by Add^r.

   Since $f \in r[I, \Delta^+] \backslash I$, so we can have $\mathsf{h}(r\sigma) \notin I$, $\mathsf{b}(r\sigma) \subseteq I$, and $\mathsf{b}(r\sigma) \cap \Delta^+ \neq \emptyset$ for this σ. So there exists at least one B_i s.t. $B_i\sigma \subseteq I$, and $B_i\sigma \cap \Delta^+ \neq \emptyset$. We define a set $P^r_{B_i,\sigma}$ as a subset of N^r (the nodes in T^r) s.t. B_i is assigned to these nodes and $\mathsf{inst}_p^{I \vdots \Delta^+}$ consists of new instantiations using σ. Formally:

   $$P^r_{B_i,\sigma} = \{p \in N^r \mid B_i \in \lambda^r(p) \text{ and } \chi^r(p)\sigma \notin \mathsf{inst}_p^I\}. \tag{A.7}$$

   For this substitution σ that $f = \mathsf{h}(r\sigma)$, we define a set of nodes P^r_σ whose instantiations $\mathsf{inst}_p^{I \vdots \Delta^+}$ is updated by this substitution σ. Formally:

   $$P^r_\sigma = \{P^r_{B_i,\sigma} \text{ for every } B_i \in \mathsf{b}(r) \text{ if } B_i\sigma \cap \Delta^+ \neq \emptyset\} \tag{A.8}$$

   Since T^r is complete, there is at least one node in P^r_σ for every σ s.t. $f = \mathsf{h}(r\sigma) \in r[I, \Delta^+] \backslash I$. For node $p \in P^r_\sigma$, $\lambda^r(p)\sigma \cap \Delta^+ \neq \emptyset$ and $\chi^r(p)\sigma \notin \mathsf{inst}_p^I$, so the instantiation that uses σ will be derived in line 3 by operator $\Pi_p[I, \Delta^+]$ and will be included in $\mathsf{inst}_p^{I \vdots \Delta^+}$.

For other nodes that $p' \notin P^r_\sigma$, the instantiation that uses σ is included in the current inst^I_p. This is because: assume there is a $p' \notin P^r_\sigma$ and $\chi^r(p')\sigma \notin \mathsf{inst}^I_p$. Then (1) if there exists B_i s.t. $B_i \in \lambda^r(p')$ and $B_i\sigma \cap \Delta^+ \neq \emptyset$, then according to expression (A.7), $p' \in P^r_{B_i,\sigma} \subseteq P^r_\sigma$, which contradicts our assumption; (2) if no such B_i exists, then $\lambda^r(p')\sigma \cap \Delta^+ = \emptyset$, i.e., $\lambda^r(p')\sigma \subseteq I \backslash \Delta^+$, then according to Claim 5 and 11, $\chi^r(p')\sigma \in \mathsf{inst}^I_p = \Pi_p[I \backslash \Delta^+]$.

Let $\max(P^r_\sigma)$ be the nodes in P^r_σ whose index is the maximum in the prefixed order of all the nodes in N^r. When $\max(P^r_\sigma)$ is labelled as Δ^+, therefore the set of data $\mathsf{inst}_p^{I \dot{:} \Delta^+}$ is used; while any other nodes in P^r_σ are labelled as $I \cup \Delta^+$, so the set of data $\mathsf{inst}^I_p \cup \mathsf{inst}_p^{I \dot{:} \Delta^+}$ is used. In both $\mathsf{inst}_p^{I \dot{:} \Delta^+}$ and $\mathsf{inst}^I_p \cup \mathsf{inst}_p^{I \dot{:} \Delta^+}$, the instantiations that use σ are included. For $p' \notin P^r_\sigma$, its label will be I or $I \cup \Delta^+$, either way, the instantiations that use σ are included. Because line 7 to 9 of Algorithm 5 can correctly join data between nodes (Claim 12), f can be derived in line 4 of Algorithm 4 and returned by Add^r. □

A.3 Proof of Lemma 2

Lemma 2: The $\mathsf{Del}^r[I, \Delta^-]$ function shown in Algorithm 6 computes $r[I, \Delta^-] \cap (I \backslash \Delta^-)$.

Proof of Lemma 2. Just like the proof of Lemma 1, we can prove Lemma 2 by showing its soundness and completeness.

1. **Soundness**: for every $f \notin r[I, \Delta^-] \cap (I \backslash \Delta^-)$, f will not be returned by Del^r.

   (1a) $f \in r[I, \Delta^-]$ but $f \notin I \backslash \Delta^-$. Such a f will be removed in line 10 of Algorithm 6.

   (1b) $f \notin r[I, \Delta^-]$. This case is similar to the (1b) of the proof of Lemma 1.

2. **Completeness**: For every $f \in r[I, \Delta^-] \cap (I \backslash \Delta^-)$, f will be returned by Del^r.

   The proof of completeness is similar to Lemma 1. □

A.4 Proof of Lemma 3

Lemma 3: The $\mathsf{Red}^r[I, \Delta]$ function shown in Algorithm 7 computes $r[I] \cap \Delta$.

Proof of Lemma 3. We first define inst_p^σ as a single instantiation that is instantiated by the substitution σ, in which σ is defined on $\mathsf{var}(\mathsf{b}(r))$. Formally:

$$\mathsf{inst}_p^\sigma = \chi^r(p)\sigma. \tag{A.9}$$

For a tuple $f = \mathsf{h}(r\sigma)$ in the database, the Oracle will maintain the count of different valid substitutions to derive f. Two substitutions σ, σ' that can derive the same f are different if there exists at least a node $p \in N^r$ s.t. $\mathsf{inst}_p^\sigma \neq \mathsf{inst}_p^{\sigma'}$. For a tuple $f = \mathsf{h}(r\sigma)$ in the database, $\mathsf{Oracle}(f)$ will return true if there exists at least one remaining σ defined on $\mathsf{var}(\mathsf{b}(r))$ s.t. f can be derived by joining inst_p^σ for each $p \in N^r$ and $\mathsf{inst}_p^\sigma \in \mathsf{inst}_p^I$ (we refer the values of inst_p^I before line 6 of Algorithm 7, same below).

For an instantiation $f_p \in \mathsf{inst}_p^I$, the Oracle will maintain the count of different valid substitutions to derive f_p. Two different substitutions σ_p, σ'_p that can derive the same f_p are different if $\lambda^r(p)\sigma_p \neq \lambda^r(p)\sigma'_p$. Each node $p \in N^r$ can also be regarded as a subrule of the original rule r. Therefore, in some cases, the Oracle only maintains the count of "half" of the step of rule r. So after rederiving the instantiations in p, we need to evaluate the tree to rederive the data introduced by the original rule r.

Formally, for every $f = \mathsf{h}(r\sigma) \in r[I] \cap \Delta$, it will have two cases: (1) there exists a substitution σ s.t. $\mathsf{inst}_p^\sigma \in \mathsf{inst}_p^I$ (using the values of inst_p^I before line 6). In this case, f can be safely rederived by using Oracle in line 8 of Algorithm 7. (2) in the other case, there is at least one node $p' \in N^r$ that $\mathsf{inst}_{p'}^\sigma \notin \mathsf{inst}_{p'}^I$, so f can't be derived by (1) using Oracle directly. However, as we proved in Claim 11, in-node rederivation (line 3) can accurately update $\mathsf{inst}_{p'}^{I \dot{\cup} \Delta^+}$ to $\Pi_{p'}[I] \backslash \mathsf{inst}_{p'}^I$ and update $\mathsf{inst}_{p'}^I$ to $\Pi_{p'}[I]$ in line 6. Since $f \in r[I]$, we have $\mathsf{b}(r\sigma) \subseteq I$, and thus $\lambda^r(p')\sigma \subseteq \Pi_{p'}[I]$. Therefore, after in-node rederivation, $\mathsf{inst}_{p'}^\sigma \in \mathsf{inst}_{p'}^{I \dot{\cup} \Delta^+} \subseteq \Pi_{p'}[I]$. So we can also derive f in this case. The discussion above establishes the completeness of the approach.

For the soundness of Algorithm 7, note that we only examine tuples in Δ that are candidates for over-deletion. This strictly constrains the domain of data that can be rederived. For any fact $f \in \Delta$ but $f \notin r[I]$: (1) the call $\mathsf{Oracle}(f)$ will return `false` (indicating that f cannot be directly proved); (2) since $f \notin r[I]$, we have $\mathsf{b}(r\sigma) \nsubseteq I$, and thus $\lambda^r(p')\sigma \nsubseteq \Pi_{p'}[I]$ stands for every node p' in the decomposition tree of rule r. Hence, f cannot be derived through the call `crossNodeEvaluation` in line 4. This holds because Oracle is designed to maintain the correct derivation count for each node instantiation and fact. Therefore, Algorithm 7 is correct.

Please note that the proof above assumes the Oracle can maintain the count correctly. In this thesis, we implement Oracle using a counting-based approach following [75]. A more detailed proof of how the derivation count is maintained during materialisation, and how this count correctly reflects the remaining derivations, can be found in [75]. Moreover, the correctness of Oracle itself has been formally proven in their work (Theorem 1 of [75]). □

Appendix B

Cyclic Rules Used in Section 4.6

This appendix presents the cyclic rules used in our experiments presented in Section 4.6, including those from LUBM L+C, YAGO and Exp.

B.1 Cyclic Rules in LUBM L+C

B.1.1 Rules 1–8

```
# 1
a1:haveSameAdvisor[?p1, ?p2] :-
    a1:hasAlumnus[?o, ?p1],
    a1:advisor[?p1, ?advisor],
    a1:hasAlumnus[?o, ?p2],
    a1:advisor[?p2, ?advisor] .

# 2
a1:fromSameCourse[?student, ?teacher] :-
    a1:hasAlumnus[?uni, ?student],
    a1:takesCourse[?student, ?course],
    a1:teacherOf[?teacher, ?course],
    a1:hasAlumnus[?uni, ?teacher] .

# 3
a1:studentHaveAdvisor[?student] :-
    a1:member[?dep2, ?student],
    a1:member[?dep1, ?advisor],
    a1:subOrganizationOf[?dep2, ?uni],
```

```
    a1:subOrganizationOf[?dep1, ?uni],
    a1:advisor[?student, ?advisor] .

# 4
a1:studentInSameDepOfAdv[?student] :-
    a1:member[?dep, ?student],
    a1:advisor[?student, ?advisor],
    a1:member[?dep, ?advisor] .

# 5
a1:similarResearchers[?p1, ?p2] :-
    a1:member[?org1, ?p1],
    a1:degreeFrom[?p1, ?org2],
    a1:member[?org1, ?p2],
    a1:degreeFrom[?p2, ?org2] .

# 6
a1:possibleCollaborators[?s1, ?s2] :-
    a1:researchInterest[?s1, ?int],
    a1:member[?uni, ?s1],
    a1:researchInterest[?s2, ?int],
    a1:member[?uni, ?s2] .

# 7
a1:studentInDepWhereAdvGotDegree[?student] :-
    a1:member[?dep, ?student],
    a1:subOrganizationOf[?dep, ?org],
    a1:degreeFrom[?advisor, ?org],
    a1:advisor[?student, ?advisor] .

# 8
a1:studentGradFromAdvWorks[?student] :-
    a1:advisor[?student, ?advisor],
    a1:degreeFrom[?student, ?uni],
    a1:member[?org, ?advisor],
    a1:subOrganizationOf[?org, ?uni] .
```

B.1.2 Rules 9–16, adapted from Stefanoni et al. [130]

```
# 9, g1 in sumRDF
:q12xc[?X, ?C],
:q12au[?A, ?U],
:q12ay[?A, ?Y] :-
    a1:degreeFrom[?X, ?U],
    rdf:type[?X, a1:GraduateStudent],
    a1:advisor[?X, ?A],
    rdf:type[?A, a1:FullProfessor],
    a1:teacherOf[?A, ?C],
    rdf:type[?C, a1:GraduateCourse],
    a1:takesCourse[?Y, ?C],
    rdf:type[?Y, a1:GraduateStudent],
    a1:hasAlumnus[?U, ?Y] .

# 10, g2 in sumRDF
:q13xc[?X, ?C],
:q13au[?A, ?U] :-
    a1:advisor[?X, ?A],
    a1:teacherOf[?A, ?C],
    a1:takesCourse[?X, ?C],
    a1:headOf[?A, ?D],
    a1:subOrganizationOf[?D, ?U] .

# 11, g3 in sumRDF
:q14GraduateAndTAIn[?X, ?U] :-
    a1:teachingAssistantOf[?X, ?C],
    a1:teacherOf[?H, ?C],
    a1:headOf[?H, ?D],
    a1:subOrganizationOf[?D, ?U],
    rdf:type[?U, a1:University],
    a1:degreeFrom[?X, ?U] .

# 12, g6 in sumRDF
:q15GraduateAndTAWhereHeWorks[?X, ?P] :-
    a1:subOrganizationOf[?D, ?U],
```

```
    a1:worksFor[?P, ?D],
    a1:teacherOf[?P, ?C],
    a1:teachingAssistantOf[?X, ?C],
    a1:degreeFrom[?X, ?U],
    rdf:type[?C, a1:Course] .

# 13, g7 in sumRDF
:q16GraduateAndTakeCourseWhereHeWorks[?X, ?P] :-
    a1:subOrganizationOf[?D, ?U],
    a1:worksFor[?P, ?D],
    a1:teacherOf[?P, ?C],
    a1:takesCourse[?X, ?C],
    a1:degreeFrom[?X, ?U],
    rdf:type[?C, a1:GraduateCourse] .

# 14, g12 in sumRDF
:q18WorksForWhereAdvisorGraduate[?X, ?Y] :-
    a1:worksFor[?X, ?Z],
    a1:subOrganizationOf[?Z, ?U],
    a1:undergraduateDegreeFrom[?Y, ?U],
    a1:advisor[?Y, ?X] .

# 15, g13 / q2 in sumRDF
:q19BSFromAndMemberOf[?X, ?Y] :-
    a1:undergraduateDegreeFrom[?X, ?Y],
    a1:memberOf[?X, ?Z],
    a1:subOrganizationOf[?Z, ?Y],
    rdf:type[?X, a1:GraduateStudent],
    rdf:type[?Z, a1:Department],
    rdf:type[?Y, a1:University] .

# 16, g15 in sumRDF
:q20BSFromAndMemberOf[?X, ?Y] :-
    a1:undergraduateDegreeFrom[?X, ?Y],
    a1:memberOf[?X, ?Z],
    a1:subOrganizationOf[?Z, ?Y],
```

```
    rdf:type[?X, a1:UndergraduateStudent],
    rdf:type[?Z, a1:Department],
    rdf:type[?Y, a1:University] .
```

B.2 Cyclic Rules in YAGO

The following cyclic rules are adapted from queries used in [113]. For the 19 non-recursive rules, we use the original query structure as the rule body and add a new predicate to form a non-recursive rule. For the 4 recursive rules, we reuse one of the predicates from the body as the rule head to introduce recursion. These rules are useful for testing recursive derivation behaviour in our evaluation.

```
# 19 non recursive rules
# 1: Clique_6, f_Q_0_11.txt.sparql
<http://Q1x0x1>[?x0, ?x1],
<http://Q1x2x3>[?x2, ?x3] :-
    <http://edge46>[?x0, ?x2] ,
    <http://edge17>[?x0, ?x3] ,
    <http://edge46>[?x1, ?x0] ,
    <http://edge46>[?x1, ?x3] ,
    <http://edge17>[?x2, ?x1] ,
    <http://edge17>[?x2, ?x3] .

# 2: Cycle_6, f_Q_0_12.txt.sparql
<http://Q2x0x1>[?x0, ?x1],
<http://Q2x2x3>[?x2, ?x3],
<http://Q2x4x5>[?x4, ?x5] :-
    <http://edge0>[?x0, ?x5] ,
    <http://edge2>[?x1, ?x0] ,
    <http://edge2>[?x1, ?x2] ,
    <http://edge25>[?x2, ?x3] ,
    <http://edge11>[?x4, ?x3] ,
    <http://edge0>[?x4, ?x5] .

# 3: Cycle_6, f_Q_5_14.txt.sparql
<http://Q3x0x1>[?x0, ?x1],
<http://Q3x2x3>[?x2, ?x3],
```

```
<http://Q3x4x5>[?x4, ?x5] :-
    <http://edge8>[?x0, ?x1] ,
    <http://edge11>[?x2, ?x1] ,
    <http://edge11>[?x2, ?x3] ,
    <http://edge8>[?x4, ?x3] ,
    <http://edge2>[?x5, ?x0] ,
    <http://edge2>[?x5, ?x4] .

# 4: Cycle_6, f_Q_3_7.txt.sparql
<http://Q4x0x1>[?x0, ?x1],
<http://Q4x2x3>[?x2, ?x3],
<http://Q4x4x5>[?x4, ?x5] :-
    <http://edge22>[?x0, ?x1] ,
    <http://edge22>[?x0, ?x5] ,
    <http://edge17>[?x1, ?x2] ,
    <http://edge57>[?x2, ?x3] ,
    <http://edge57>[?x4, ?x3] ,
    <http://edge17>[?x4, ?x5] .

# 5: Cycle_6, f_Q_1_12.txt.sparql
<http://Q5x0x1>[?x0, ?x1],
<http://Q5x2x3>[?x2, ?x3],
<http://Q5x4x5>[?x4, ?x5] :-
    <http://edge8>[?x0, ?x1] ,
    <http://edge11>[?x2, ?x1] ,
    <http://edge0>[?x2, ?x3] ,
    <http://edge0>[?x4, ?x3] ,
    <http://edge2>[?x5, ?x0] ,
    <http://edge2>[?x5, ?x4] .

# 6: Cycle_6, f_Q_0_10.txt.sparql
<http://Q6x0x1>[?x0, ?x1],
<http://Q6x2x3>[?x2, ?x3],
<http://Q6x4x5>[?x4, ?x5] :-
    <http://edge5>[?x1, ?x0] ,
    <http://edge54>[?x1, ?x2] ,
```

```
    <http://edge5>[?x2, ?x3] ,
    <http://edge17>[?x3, ?x4] ,
    <http://edge5>[?x5, ?x0] ,
    <http://edge21>[?x5, ?x4] .

# 7: Cycle_6, f_Q_3_3.txt.sparql
<http://Q7x0x1>[?x0, ?x1],
<http://Q7x2x3>[?x2, ?x3],
<http://Q7x4x5>[?x4, ?x5] :-
    <http://edge47>[?x0, ?x1] ,
    <http://edge36>[?x2, ?x1] ,
    <http://edge50>[?x2, ?x3] ,
    <http://edge55>[?x4, ?x3] ,
    <http://edge55>[?x4, ?x5] ,
    <http://edge29>[?x5, ?x0] .

# 8: Flower_6, f_Q_3_3.txt.sparql
<http://Q8x0x1>[?x0, ?x1],
<http://Q8x2x3>[?x2, ?x3],
<http://Q8x4x5>[?x4, ?x5] :-
    <http://edge25>[?x0, ?x1] ,
    <http://edge48>[?x0, ?x2] ,
    <http://edge8>[?x0, ?x3] ,
    <http://edge48>[?x0, ?x4] ,
    <http://edge25>[?x5, ?x3] ,
    <http://edge48>[?x5, ?x4] .

# 9: Flower_6, f_Q_1_1.txt.sparql
<http://Q9x0x1>[?x0, ?x1],
<http://Q9x2x3>[?x2, ?x3],
<http://Q9x4x5>[?x4, ?x5] :-
    <http://edge17>[?x0, ?x1] ,
    <http://edge21>[?x2, ?x0] ,
    <http://edge17>[?x3, ?x4] ,
    <http://edge17>[?x4, ?x0] ,
    <http://edge17>[?x5, ?x0] ,
```

```
    <http://edge17>[?x5, ?x3] .

# 10: Flower_6, f_Q_3_1.txt.sparql
<http://Q10x0x1>[?x0, ?x1],
<http://Q10x2x3>[?x2, ?x3],
<http://Q10x4x5>[?x4, ?x5] :-
    <http://edge46>[?x0, ?x1] ,
    <http://edge17>[?x2, ?x0] ,
    <http://edge5>[?x3, ?x4] ,
    <http://edge5>[?x3, ?x5] ,
    <http://edge17>[?x4, ?x0] ,
    <http://edge17>[?x5, ?x0] .

# 11: Flower_6, f_Q_2_1.txt.sparql
<http://Q11x0x1>[?x0, ?x1],
<http://Q11x2x3>[?x2, ?x3],
<http://Q11x4x5>[?x4, ?x5] :-
    <http://edge17>[?x0, ?x1] ,
    <http://edge36>[?x0, ?x2] ,
    <http://edge46>[?x0, ?x3] ,
    <http://edge46>[?x0, ?x4] ,
    <http://edge58>[?x5, ?x3] ,
    <http://edge58>[?x5, ?x4] .

# 12: Flower_6, f_Q_5_3.txt.sparql
<http://Q12x0x1>[?x0, ?x1],
<http://Q12x2x3>[?x2, ?x3],
<http://Q12x4x5>[?x4, ?x5] :-
    <http://edge25>[?x0, ?x1] ,
    <http://edge2>[?x2, ?x0] ,
    <http://edge2>[?x3, ?x0] ,
    <http://edge2>[?x3, ?x4] ,
    <http://edge2>[?x5, ?x0] ,
    <http://edge2>[?x5, ?x4] .

# 13: Flower_6, f_Q_3_5.txt.sparql
```

```
<http://Q13x0x1>[?x0, ?x1],
<http://Q13x2x3>[?x2, ?x3],
<http://Q13x4x5>[?x4, ?x5] :-
    <http://edge17>[?x0, ?x3] ,
    <http://edge21>[?x1, ?x0] ,
    <http://edge21>[?x2, ?x0] ,
    <http://edge17>[?x3, ?x4] ,
    <http://edge21>[?x5, ?x0] ,
    <http://edge5>[?x5, ?x4] .

# 14: Petal_6, f_Q_0_12.txt.sparql
<http://Q14x0x1>[?x0, ?x1],
<http://Q14x2x3>[?x2, ?x3],
<http://Q14x4x5>[?x4, ?x5] :-
    <http://edge31>[?x0, ?x2] ,
    <http://edge0>[?x0, ?x4] ,
    <http://edge5>[?x3, ?x1] ,
    <http://edge36>[?x3, ?x2] ,
    <http://edge5>[?x5, ?x1] ,
    <http://edge0>[?x5, ?x4] .

# 15: Petal_6, f_Q_3_7.txt.sparql
<http://Q15x0x1>[?x0, ?x1],
<http://Q15x2x3>[?x2, ?x3],
<http://Q15x4x5>[?x4, ?x5] :-
    <http://edge17>[?x0, ?x4] ,
    <http://edge39>[?x1, ?x3] ,
    <http://edge5>[?x1, ?x5] ,
    <http://edge21>[?x2, ?x0] ,
    <http://edge39>[?x2, ?x3] ,
    <http://edge17>[?x5, ?x4] .

# 16: Petal_6, f_Q_1_12.txt.sparql
<http://Q16x0x1>[?x0, ?x1],
<http://Q16x2x3>[?x2, ?x3],
<http://Q16x4x5>[?x4, ?x5] :-
```

```
    <http://edge31>[?x0, ?x2] ,
    <http://edge31>[?x0, ?x4] ,
    <http://edge5>[?x3, ?x1] ,
    <http://edge35>[?x3, ?x2] ,
    <http://edge22>[?x5, ?x1] ,
    <http://edge35>[?x5, ?x4] .

# 17: Petal_6, f_Q_0_10.txt.sparql
<http://Q17x0x1>[?x0, ?x1],
<http://Q17x2x3>[?x2, ?x3],
<http://Q17x4x5>[?x4, ?x5] :-
    <http://edge17>[?x0, ?x2] ,
    <http://edge17>[?x0, ?x4] ,
    <http://edge46>[?x1, ?x3] ,
    <http://edge58>[?x2, ?x3] ,
    <http://edge17>[?x5, ?x1] ,
    <http://edge46>[?x5, ?x4] .

# 18: Petal_6, f_Q_3_3.txt.sparql
<http://Q18x0x1>[?x0, ?x1],
<http://Q18x2x3>[?x2, ?x3],
<http://Q18x4x5>[?x4, ?x5] :-
    <http://edge17>[?x0, ?x4] ,
    <http://edge17>[?x1, ?x3] ,
    <http://edge21>[?x2, ?x0] ,
    <http://edge21>[?x2, ?x3] ,
    <http://edge17>[?x4, ?x5] ,
    <http://edge17>[?x5, ?x1] .

# 19: Petal_6, f_Q_1_9.txt.sparql
<http://Q19x0x1>[?x0, ?x1],
<http://Q19x2x3>[?x2, ?x3],
<http://Q19x4x5>[?x4, ?x5] :-
    <http://edge17>[?x0, ?x2] ,
    <http://edge17>[?x0, ?x4] ,
    <http://edge5>[?x3, ?x1] ,
```

```
    <http://edge21>[?x3, ?x2] ,
    <http://edge22>[?x5, ?x1] ,
    <http://edge21>[?x5, ?x4] .

# 4 recursive rules
# 1: Clique_6, f_Q_0_11.txt.sparql
<http://edge46>[?x0, ?x3],
<http://Q1x1x2>[?x1, ?x2] :-
    <http://edge46>[?x0, ?x2] ,
    <http://edge17>[?x0, ?x3] ,
    <http://edge46>[?x1, ?x0] ,
    <http://edge46>[?x1, ?x3] ,
    <http://edge17>[?x2, ?x1] ,
    <http://edge17>[?x2, ?x3] .

# 2: Cycle_6, f_Q_0_12.txt.sparql
<http://edge0>[?x0, ?x1],
<http://Q2x2x3>[?x2, ?x3],
<http://Q2x4x5>[?x4, ?x5] :-
    <http://edge0>[?x0, ?x5] ,
    <http://edge2>[?x1, ?x0] ,
    <http://edge2>[?x1, ?x2] ,
    <http://edge25>[?x2, ?x3] ,
    <http://edge11>[?x4, ?x3] ,
    <http://edge0>[?x4, ?x5] .

# 3: Flower_6, f_Q_3_3.txt.sparql
<http://edge25>[?x0, ?x2],
<http://Q3x1x3>[?x1, ?x3],
<http://Q3x4x5>[?x4, ?x5] :-
    <http://edge25>[?x0, ?x1] ,
    <http://edge48>[?x0, ?x2] ,
    <http://edge8>[?x0, ?x3] ,
    <http://edge48>[?x0, ?x4] ,
    <http://edge25>[?x5, ?x3] ,
    <http://edge48>[?x5, ?x4] .
```

```
# 4: Petal_6, f_Q_0_12.txt.sparql
<http://edge31>[?x0, ?x1],
<http://Q4x2x3>[?x2, ?x3],
<http://Q4x4x5>[?x4, ?x5] :-
    <http://edge31>[?x0, ?x2] ,
    <http://edge0>[?x0, ?x4] ,
    <http://edge5>[?x3, ?x1] ,
    <http://edge36>[?x3, ?x2] ,
    <http://edge5>[?x5, ?x1] ,
    <http://edge0>[?x5, ?x4] .
```

B.3 Cyclic Rules in Exp

The following presents three rules from the Exp dataset, representing the evaluation of subtraction, addition, and multiplication expressions.

```
# subtraction expression
:eval[?s , ?e],
:instance[?e , ?i],
:value[?e , ?v] :-
    :hasType[?s , :Minus],
    :hasLhs[?s , ?lhs],
    :hasRhs[?s , ?rhs],
    # RHS
    :eval[?rhs , ?erhs],
    :value[?erhs , ?vrhs],
    :instance[?erhs , ?i],
    # LHS
    :eval[?lhs , ?elhs],
    :value[?elhs , ?vlhs],
    :instance[?elhs , ?i],
    # Value
    BIND(?vlhs - ?vrhs as ?v),
    BIND(SKOLEM("Eval", ?s, ?i) AS ?e) .
```

```
# addition expression
:eval[?s , ?e],
:instance[?e , ?i],
:value[?e , ?v] :-
    :hasType[?s , :Plus],
    :hasLhs[?s , ?lhs],
    :hasRhs[?s , ?rhs],
    # RHS
    :eval[?rhs , ?erhs],
    :value[?erhs , ?vrhs],
    :instance[?erhs , ?i],
    # LHS
    :eval[?lhs , ?elhs],
    :value[?elhs , ?vlhs],
    :instance[?elhs , ?i],
    # Value
    BIND(?vlhs + ?vrhs as ?v),
    BIND(SKOLEM("Eval", ?s, ?i) AS ?e) .

# multiplication expression
:eval[?s , ?e],
:instance[?e , ?i],
:value[?e , ?v] :-
    :hasType[?s , :Times],
    :hasLhs[?s , ?lhs],
    :hasRhs[?s , ?rhs],
    # RHS
    :eval[?rhs , ?erhs],
    :value[?erhs , ?vrhs],
    :instance[?erhs , ?i],
    # LHS
    :eval[?lhs , ?elhs],
    :value[?elhs , ?vlhs],
    :instance[?elhs , ?i],
    # Value
    BIND(?vlhs * ?vrhs as ?v),
```

```
BIND(SKOLEM("Eval", ?s, ?i) AS ?e) .
```

Appendix C

Proofs of Theorems and Lemmas in Chapter 5

C.1 Adapted Modular DRed Algorithm

Before presenting the formal proofs, we introduce an adapted version of the modular DRed algorithm, based on the variant proposed by Hu et al. [77]. The original modular DRed algorithm [77] enables different subsets of a Datalog program to be evaluated using distinct algorithms for improved performance, but it maintains a unified storage for the resulting facts. In contrast, our multi-scheme variant, shown in Algorithm 12, can be seen as an improvement based on their modular framework by introducing tailored storage schemes for facts derived by different subsets of Datalog programs. Despite these differences, their framework serves as a valid theoretical foundation for our correctness proof. In particular, we adopt the same semantic definitions of upper and lower bounds on derivations used in the addition, deletion, and rederivation phases, as introduced by Hu et al. [77]. They have proved that, if the evaluation functions assigned to different subsets of the Datalog program satisfy these lower and upper bounds, their modular DRed algorithm can maintain the materialisation correctly (Theorem 15 in [77]). In this section, we first present the modular version of the DRed algorithm in Algorithm 20, which follows the same fundamental principles as in [77], but adopts modified notation to better align with our multi-scheme framework, making it more suitable for discussion in the proof.

The modular version of the DRed algorithm, as shown in Algorithm 20, enables different parts of a program to be evaluated using customised algorithms. For a program Π, rules are partitioned into disjoint rule sets, and each module T either implements tailored evaluation algorithms for rules Π_T or defaults to the standard

seminaïve evaluation. Taking the *add* procedure presented in lines 17-24 as an example, line 23 tracks the modified part, just like the seminaïve algorithm; while line 20 applies rules within each module using the modular operator Π_T^+. Hu et al. [77] define lower and upper bounds for the resulting derivations of the operator Π_T^+ in the context of rules Π_T, current facts I, and fresh derivations M, as presented in Table C.1. Specifically, the lower bound corresponds to the seminaïve evaluation of rules in Π_T, while any fact included in the final updated materialisation $\Pi^\infty[E']$ where $E' = (E \backslash E^-) \cup E^+$ can be derived by the operator Π_T^+ without influencing the correctness of reasoning. Moreover, to ensure the termination of the algorithm, Π_T^+ is restricted from returning facts that are already included in I.

Similarly, operator Π_T^- computes consequences that are potentially affected by deleting M, fresh deletions tracked during the *overdeletion* procedure, as shown in line 9. The bounds of Π_T^- are defined in the context of rules Π_T, currently existing facts I, and tracked changes M. The operator Π_T^- would delete any fact that is derivable from one-step seminaïve evaluation given Π_T, I, and M and does not hold in the final updated materialization $\Pi^\infty[E']$, as shown in the lower bound. Meanwhile, to accommodate various implementations, Π_T^- is allowed to overdelete any currently existing fact in I, as presented in the upper bound. Finally, operator Π_T^r in line 16 is used to examine whether overdeleted facts D collected in line 13 have alternative proofs using remaining facts I. It can recover any fact in D that is one-step rederivable by applying Π_T to I, as shown in the lower bound. Moreover, it can also rederive any fact that holds in the final updated materialisation $\Pi^\infty[E']$, as presented in the upper bound.

The bounds defined in Table C.1 accommodate various types of evaluation algorithms, ensuring flexibility; as long as a module implements evaluation functions that adhere to these bounds, the correctness of the overall reasoning process is preserved. Our multi-scheme algorithm builds upon this modular version as a non-trivial improvement. While the modular algorithm allows the use of different evaluation algorithms, it retains a unified storage structure for all derivations, thereby failing to address the storage challenges discussed in this work. Nonetheless, the bounds defined for each modular operation remain critical when partitioning programs into separate sets of rules - an unavoidable step when integrating diverse storage schemes for the consequences of different rules.

Algorithm 20 The Modular Version of the DRed Algorithm

1: **Requires:** program Π, materialisations I, initial dataset E, fact additions E^+ and deletions E^-
2: **Result:** update I from $\Pi^\infty[E]$ to $\Pi^\infty[(E \backslash E^-) \cup E^+]$
3: $E^- := (E^- \cap E) \backslash E^+$, $E^+ := E^+ \backslash E$
4: OVERDELETE, REDERIVE, ADD
5: $E := (E \backslash E^-) \cup E^+$
6: **procedure** OVERDELETE
7: $M := E^-$, $I := I \backslash M$, $D := \emptyset$
8: **loop**
9: **for** each module T **do** $\Delta^T := \Pi_T^-[I, M]$
10: **if** derived nothing **then break**, **else** $M := \emptyset$
11: **for** each module T **do**
12: $M := M \cup \Delta^T$
13: $I := I \backslash \Delta^T$, $D := D \cup \Delta^T$, $\Delta^T := \emptyset$
14: **procedure** REDERIVE
15: $A := \emptyset$
16: **for** each module T **do** $A := A \cup \Pi_T^r[I, D]$
17: **procedure** ADD
18: $M := (A \cup (E \backslash E^-) \cup E^+) \backslash I$, $I := I \cup M$
19: **loop**
20: **for** each module T **do** $\Delta^T := \Pi_T^+[I, M]$
21: **if** derived nothing **then break**, **else** $M := \emptyset$
22: **for** each module T **do**
23: $M := M \cup \Delta^T$
24: $I := I \cup \Delta^T, \Delta^T := \emptyset$

C.2 Proof of Theorem 2

Theorem 2 establishes the correspondence between the resulting set of facts obtained after invoking the derivation procedures and the bounds defined in Table 5.7. It is straightforward to observe that the bounds in Table 5.7 are consistent with those in the DRed algorithm, as defined in Table C.1. Therefore, to prove Theorem 2, we directly demonstrate the correspondence between the resulting set of facts and the modular bounds. The proof of Theorem 2 is divided as proving claims 13, 14, and 15 stated below.

Claim 13 *For a scheme T that satisfies the invariants in Definition 4, derivations encapsulated in Δ^T after invoking the* `deriveForAddition` *function conform to the lower and upper bounds of the modular operate Π_T^+ defined in Table C.1.*

Modular Operations	lower bound	upper bound
$\Pi_T^+[I, M],\ M \subseteq I$	$\Pi_T[I, M] \setminus I$	$\Pi^\infty[E'] \setminus I$
$\Pi_T^r[I, D], D \nsubseteq I$	$\Pi_T[I] \cap D$	$\Pi^\infty[E'] \cap D$
$\Pi_T^-[I, M],\ M \nsubseteq I$	$(I \cap \Pi_T[I \cup M, M]) \setminus \Pi^\infty[E']$	I

Table C.1: The bounds of modular operations, in which $E' = (E \backslash E^-) \cup E^+$, and $\Pi^\infty[E']$ represents the materialisations after incrementally inserting facts E^+ and deleting E^-.

Proof. As stated in the theorem, the scheme T satisfies the invariants in Definition 4. As such, facts in domain 'Δ' satisfy

$$(\Delta_n^T \cup \mathcal{R}_d^T[I \cup \Delta_n^T]) \setminus I^T \subseteq \Delta^T \subseteq (\mathcal{R}_d^T)^\infty[I \cup \Delta_n^T] \setminus I^T, \tag{C.1}$$

in which Δ_n^T denotes derivations of seminaïve evaluation of $\mathcal{R}_n^T$ using current available facts I and tracked changes for the scheme M_a^T. Therefore, Δ^T is defined with regard to $\Pi_T = \mathcal{R}_d^T \cup \mathcal{R}_n^T$, $I = \bigcup_{T \in \mathbb{T}} \{I^T\}$, and M_a^T. The bounds of the modular operator Π_T^+ are defined in the context of Π_T, I, and M. Please note that the difference between M for a modular operator and M_a^T for a scheme is that M collects global fresh additions; while M_a^T only includes additions that are relevant to the application of rules in Π_T. This is because M_a^T only considers facts with the predicate appearing in the body of rules in Π_T (i.e., B_T). Therefore, we have $M_a^T \subseteq M$, and $\Pi_T[I, M] = \Pi_T[I, M_a^T]$. The same applies to M_d^T during overdeletion, so we will not reiterate below.

We first prove that facts in Δ^T conform to the lower bound of Π_T^+, by proving any fact $t \in \Pi_T[I, M] \backslash I$ that is included in the lower bound is contained in Δ^T. The consequences of $\Pi_T[I, M]$ can be computed by applying rules $\mathcal{R}_d^T$ and $\mathcal{R}_n^T$ separately using I and M. Therefore, $\Pi_T[I, M] = \mathcal{R}_d^T[I, M] \cup \mathcal{R}_n^T[I, M]$. For a fact $t \in \mathcal{R}_n^T[I, M] \backslash I$ that is included in the lower bound and derived by $\mathcal{R}_n^T$, t is included in Δ^T because

$$\mathcal{R}_n^T[I, M] \backslash I = \mathcal{R}_n^T[I, M_a^T] \backslash I^T = \Delta_n^T \backslash I^T \subseteq \Delta^T. \tag{C.2}$$

Similarly, for a fact $t \in \mathcal{R}_d^T[I, M] \setminus I$ that is included in the lower bound and derived by $\mathcal{R}_d^T$, t is included in Δ^T because

$$\mathcal{R}_d^T[I, M] \backslash I = \mathcal{R}_d^T[I, M_a^T] \backslash I^T \subseteq \mathcal{R}_d^T[I] \backslash I^T \subseteq \mathcal{R}_d^T[I \cup \Delta_n^T] \backslash I^T \subseteq \Delta^T. \tag{C.3}$$

Therefore, any fact $t \in \Pi_T[I, M] \backslash I$ that is included in the lower bound of the operator Π_T^+ is contained in Δ^T, so we have: facts in Δ^T conform to the lower bound of Π_T^+. Similarly, we prove Δ^T conforms to the upper bound of Π_T^+ as follows:

$$\Delta^T \subseteq (\mathcal{R}_d^T)^\infty[I \cup \Delta_n^T] \setminus I^T \subseteq \Pi^\infty[E'] \backslash I. \tag{C.4}$$

This is because current available facts in I are valid consequences of rules in Π, and Δ_n^T denotes derivations of $\mathcal{R}_n^T \subseteq \Pi$. Applying $\mathcal{R}_d^T \subseteq \Pi$ to derivations of rules that are included in Π yields facts contained in the final materialisation $\Pi^\infty[E']$. Combining the statements above, we have: Δ^T conforms to the lower and upper bounds of Π_T^+. $\square$

Claim 14 *For a scheme T that satisfies the invariants in Definition 4, derivations encapsulated in Δ^T after invoking the* `deriveForDeletion` *function conform to the lower and upper bounds of the modular operate Π_T^- defined in Table C.1.*

Proof. According to the Definition 4, during overdeletion, facts in domain Δ satisfy:

$$\left[\Delta_n^T \cup (I \cap X)\right] \setminus \Pi^\infty[E'] \quad \subseteq \quad \Delta^T \quad \subseteq \quad I^T, \tag{C.5}$$

in which $X = \mathcal{R}_d^T[I \cup M_d^T, M_d^T \cup \Delta_n^T]$, and $\Delta_n^T = \mathcal{R}_n^T[I \cup M_d^T, M_d^T] \cap I$. We first prove Δ^T conform to the lower bound of Π_T^-. For any fact $t \in (I \cap \Pi_T[I \cup M, M]) \backslash \Pi^\infty[E']$ that is included in the lower bound, it is either derived by $(I \cap \mathcal{R}_n^T[I \cup M, M]) \backslash \Pi^\infty[E']$ or $(I \cap \mathcal{R}_d^T[I \cup M, M]) \backslash \Pi^\infty[E']$. We can prove each of them is included in Δ^T as follows:

$$I \cap \mathcal{R}_n^T[I \cup M, M] = I \cap \mathcal{R}_n^T[I \cup M_d^T, M_d^T] = \Delta_n^T, \tag{C.6}$$

$$\Delta_n^T \backslash \Pi^\infty[E'] \subseteq \Delta^T, \tag{C.7}$$

$$\mathcal{R}_d^T[I \cup M, M] = \mathcal{R}_d^T[I \cup M_d^T, M_d^T] \subseteq X, \tag{C.8}$$

$$(I \cap X) \backslash \Pi^\infty[E'] \subseteq \Delta^T. \tag{C.9}$$

The facts in Δ^T conform to the upper bound of Π_T^-, which can be proved simply by $I^T \subseteq I$. The scheme can delete as many as I^T while the modular operator Π_T^- is permitted to delete all available facts I, therefore, any fact that can be overdeleted by the scheme is included in the upper bound of Π_T^-. $\square$

Claim 15 *For a scheme T that satisfies the invariants in Definition 4, facts recovered by the* `rederive` *function conform to the lower and upper bounds of the modular operate Π_T^r defined in Table C.1.*

Proof. The `rederive` function of the scheme T examines every overdeleted facts $t \in D^T$. If t can be derived by applying Π_T to current materialisation I, i.e., $t \in \Pi_T[I]$, then t has an alternative proof and can be rederived. Moreover, if t is contained in the final updated materialisation $\Pi^\infty[E']$, it is also safe to derive t at this stage. The

`rederivable` function for each scheme T is used to determine whether an overdeleted fact t satisfies the two conditions stated above. According to the invariants defined in Definition 4, `rederivable`(t) returns `true` if $t \in \Pi_T[I]$ or $t \in \Pi^\infty[E']$. Consequently, the `rederive` function will return a set of facts contained somewhere between $\Pi_T[I] \cap D$ and $\Pi^\infty[E'] \cap D$, as defined in Table C.1. □

Theorem 4 *For a scheme that satisfies the invariants in Definition 4, the derivations encapsulated in Δ^T after invoking the `deriveForDeletion`, `rederive`, and `deriveForAddition` functions conform to the lower and upper bounds of the modular operations Π_T^+, Π_T^-, and Π_T^r as specified in Table C.1, respectively.*

Proof. Combining claims 13, 14, and 15, Theorem 4 stands. It is straightforward to see the correspondence between Theorem 4 and Theorem 2, therefore Theorem 2 is also proved. □

C.3 Proof of Lemma 4

Lemma 4. *The default scheme satisfies the invariants in Definition 4.*

Proof. The default scheme T handles all the rules using seminaïve evaluation, therefore, $\mathcal{R}_n^T = \Pi_T, \mathcal{R}_d^T = \emptyset$. So the bounds defined for Δ^T after calling the *insert* function in Definition 4 can be updated as:

$$\Delta_n^T \backslash I^T = \Delta_n^T \subseteq \Delta^T \subseteq (I \cup \Delta_n^T) \backslash I^T. \tag{C.10}$$

Please note that $\Delta_n^T \backslash I^T = \Delta_n^T$ holds because Δ_n^T is a set of facts derived by $\mathcal{R}_n^T$ (line 3 in Algorithm 10) that satisfying the *insertable* condition, as shown in the *schedule* function in Algorithm 11, ensuring they are not already present in I^T. The *insert* function defined in Algorithm 14 inserts facts in Δ_n^T to the scheme's internal table and adds a label 'Δ'. Then, these facts with the label 'Δ' are returned when accessing Δ^T. Therefore, after calling the *insert* function, Δ^T serialised from the data structure is $\Delta^T = \Delta_n^T$, which is exactly the lower bound of Δ^T considering $\mathcal{R}_d^T = \emptyset$. Additionally, the *insert* function will not change the facts with the label 'I' or 'D', so facts in domain 'I' and 'D' remain unchanged before and after the *insert* call. The *insert* is invoked after the *rederive* function being called, in which D^T is emptied. So, D^T are not affected when calling *insert* and remains empty. Finally, the Condition 1 in Definition 4 holds for a default scheme.

For Condition 2, the *merge* function changes the label of facts with the label 'Δ' to 'I'. The facts in domain 'I' are defined as facts with the label 'I'. Therefore, the

merge function pushes every fact that is originally in Δ^T to I^T by changing its label; while Δ^T is empty after the call since there is no fact with the label 'Δ'. Additionally, facts in domain 'D' remain unchanged because the *merge* function did not change or add any fact with the label 'D'. Therefore, the condition 2 in Definition 4 holds.

With $\mathcal{R}_d^T = \emptyset$, the bounds defined for Δ^T after calling the *delete* function can be updated as:

$$\Delta_n^T \backslash \Pi^\infty[E'] \subseteq \Delta^T \subseteq I^T. \tag{C.11}$$

Similar to the *insert* function, after calling the *delete* function, facts in domain 'Δ' serialised from the data structure is $\Delta^T = \Delta_n^T$, which satisfies the bounds in Equation C.11. The upper bound is satisfied because Δ_n^T is processed by the *deletable* function, which ensures that $\Delta_n^T \subseteq I^T$. Therefore, Condition 3 holds for a default scheme.

For Condition 4, the *remove* function adds label 'D' and 'I' to facts currently with the label 'Δ', which is equivalent to adding Δ^T to I^T and D^T. The label 'Δ' is then deleted for these facts, therefore, Δ^T is emptied. Therefore, Condition 4 holds.

Finally, the *insertable* and *deletable* functions satisfy the required conditions, provided that I^T and Δ^T are correctly implemented. Therefore, we omit the proof for these functions, as the same reasoning applies to the specialised schemes introduced below. The *rederivable* function of the default scheme employs backward evaluation for each overdeleted fact $t \in D^T$, which is equivalent to verifying whether $t \in \Pi_T[I]$. Since this satisfies the stated condition, Lemma 4 is proved. □

C.4 Correctness of the TC Scheme: Lemma 5

We first prove the correctness of each function in the TC scheme independently under specific preconditions from C.4.1 to C.4.3. Next, we demonstrate that these preconditions are satisfied within the multi-scheme DRed algorithm, thereby establishing the validity of Claim 1 and Claim 2 in C.4.4. Finally, we prove that the TC scheme satisfies the invariants in Definition 4 in C.4.5. Specifically, the TC scheme employs the *insert* and *merge* functions to update its internal data structure, ensuring that encapsulated facts in the domains I' and Δ' are consistently maintained. The *insert* function guarantees that the facts in domain 'Δ' represent fresh derivations in the transitive closure, while the *merge* function integrates Δ^T into I^T. The correctness of the *insert* and *merge* functions for the initial construction is established in Claims 16–17 of C.4.1, while their correctness for incremental maintenance is demonstrated in Claims 18–20 in C.4.2.

Additionally, the TC scheme uses the *delete* and *merge* functions to maintain the data structure during the overdeletion procedure, ensuring that facts encapsulated in the domains Δ' and I' are updated appropriately. The *delete* function derives the consequences of fact deletions and ensures that these consequences are encapsulated in Δ^T, while the *remove* function removes Δ^T from I^T. The correctness of the *delete* function is discussed in Claim 21, and the *remove* function is analysed in Claim 22, both are provided in C.4.3.

C.4.1 Correctness of Initial Construction

Claim 16 *Given facts Δ_n^T that are to be inserted, the first* `insert` *call ensures that encapsulated facts in domain 'I' and 'Δ' are:*

$$I^T = \emptyset, \tag{C.12}$$

$$\Delta^T = (\mathcal{R}_d^T)^\infty[\Delta_n^T]. \tag{C.13}$$

Proof. The initial construction of the data structure, presented in lines 2–12 of Algorithm 15, is invoked when the *insert* function is called for the first time. This construction algorithm follows the approach of Agrawal et al. [6], who proved that the intervals and indexes computed in this manner correctly capture the full transitive closure of the input graph. Lines 2–12 of Algorithm 15 adopt the same structure as in Agrawal et al. [6], with modifications to align with our framework and notation. In particular, we store the assigned intervals for each node s in $\mathcal{D}_s$, as every reachable pair is first recorded in Δ^T, and subsequently added to I^T via the *merge* function. Provided that the intervals in $\mathcal{D}_s$ for each node s correctly capture the reachability relation from s (as ensured by the correctness of the interval computation), the set of reachable pairs Δ^T, decoded from the intervals $\mathcal{D}_s$, represents all reachable pairs in the graph constructed from the input facts Δ_n^T. Therefore, Δ^T contains exactly the facts derived by recursively applying the transitive closure rule $\mathcal{R}_d^T$ to Δ_n^T until a fixpoint is reached, as formalised in Equation C.13. Meanwhile, according to the target intervals defined in Table 5.9, I^T is empty because $\mathcal{I}_s = \emptyset$ for every node s. Therefore, Claim 16 is proved. □

Claim 17 *The* `merge` *function, which is invoked after the initial* `insert` *call, ensures the data structure is maintained such that:*

$$I^T = I^T \cup \Delta^T = (\mathcal{R}_d^T)^\infty[\Delta_n^T] = (\mathcal{R}_d^T)^\infty[N], \tag{C.14}$$

$$\Delta^T = \emptyset, \tag{C.15}$$

in which $N = \Delta_n^T$ represents edges that are added to the underlying graph G by the initial `insert` *function.*

Proof. The *merge* function updates the $\mathcal{I}_s$ interval by adding $\mathcal{D}_s$ for every node s. The interval $\mathcal{N}$ remains empty during the initial *insert* and *merge* function call. According to Table 5.9, the target interval $\mathcal{T}_s^\Delta$ of a node s for domain 'Δ' is:

$$\mathcal{D}_s + (\mathcal{N} \cap \mathcal{I}_s) = \mathcal{D}_s, \tag{C.16}$$

$$(\mathcal{D}_s \backslash \mathcal{I}_n) + (\mathcal{N} \cap \mathcal{I}_n) = (\mathcal{D}_s \backslash \mathcal{I}_n), \tag{C.17}$$

in which Equation C.16 represents the case in which s is not an SCC that is newly merged during the initial *insert* call; while Equation C.17 represents the case in which s is a newly merged SCC and n is one of its original SCCs. Equations C.16 and C.17 captures the indexes of nodes that are reachable from s in domain 'Δ'. The target interval of a node s for domain 'I' is $\mathcal{T}_s^I = \mathcal{I}_s \backslash \mathcal{N} = \mathcal{I}_s$, which includes $\mathcal{T}_s^\Delta$ in both cases:

$$\mathcal{D}_s \subseteq I_s, \tag{C.18}$$

$$\mathcal{D}_s \backslash \mathcal{I}_n \subseteq I_s. \tag{C.19}$$

Therefore, by adding $\mathcal{D}_s$ to $\mathcal{I}_s$, the target interval in domain 'Δ' is included in the target interval in domain 'I', $\mathcal{T}_s^\Delta \subseteq \mathcal{T}_s^I$, thus $\Delta^T \subseteq I^T$. Considering I^T is initially empty, $I^T = I^T \cup \Delta^T = (\mathcal{R}_d^T)^\infty[\Delta_n^T]$. It is straightforward to see that $(\mathcal{R}_d^T)^\infty[\Delta_n^T] = (\mathcal{R}_d^T)^\infty[N]$ because edges in Δ_n^T are added to the underlying graph G in the initial *insert* call. Then, the *merge* function empties $\mathcal{D}_s$, which makes the target interval of domain 'Δ' empty: $\mathcal{T}_s^\Delta = \emptyset$. Thus, $\Delta^T = \emptyset$. Claim 17 is proved. □

C.4.2 Correctness of Incremental Addition

We first establish several properties of intervals that are consistently maintained during the execution of the *insert* function for incremental addition. The correctness of inserting a single edge using the *insert* function is then formally proven in Claim 18. Subsequently, Claim 19 demonstrates that the *insert* function performs incremental addition correctly, given the precondition that the data structure is in a correct state before execution. Finally, the correctness of the subsequent *merge* function is established in Claim 20.

Property 1 *For an SCC u that is not newly merged, its intervals satisfy:*

$$\mathcal{D}_u \cap \mathcal{I}_u = \emptyset. \tag{C.20}$$

Proof. This property is guaranteed by how $\mathcal{D}_u$ is updated. As presented in line 19 of Algorithm 15, when u is not newly merged, $\mathcal{D}_u := [\mathcal{D}_u \cup (\mathcal{I}_v + \mathcal{D}_v + [i_v])] \setminus \mathcal{I}_u$. When changes of intervals are passed to u during the *propagate* function, similar procedures are invoked in lines 4-5 of Algorithm 3, which ensures that the interval $\mathcal{D}_u$ only contains new intervals that are not included in $\mathcal{I}_u$, as presented in Equation C.20. □

Property 2 *For an SCC s that is newly merged, its intervals satisfy:*

$$\mathcal{I}_s \subseteq \mathcal{D}_s. \tag{C.21}$$

Proof. This property is guaranteed by how $\mathcal{D}_s$ is updated, as presented in line 22 of Algorithm 15. When s is newly merged, the interval $\mathcal{D}_s$ of s is used to record nodes that this new SCC can reach after merging. Specifically, s can reach all the nodes that its original SCCs $O(s)$ can reach, as well as $O(s)$ themselves. So, $\mathcal{D}_s$ collects $\mathcal{I}_n + \mathcal{D}_n + [i_n]$ for every original SCC $n \in O(s)$. Therefore, we have $\mathcal{I}_s \subseteq \mathcal{D}_s$, as shown in Equation C.21. □

Property 3 *For an SCC s, regardless of whether it is newly merged or not, the target interval covering all the nodes that s can reach in domain 'I' or 'Δ' is as follows:*

$$\mathcal{T}_s^I \cup \mathcal{T}_s^\Delta = \mathcal{T}_s^{I\cup\Delta} = \mathcal{I}_s + \mathcal{D}_s. \tag{C.22}$$

Proof. This property can be derived directly by the target intervals in different domains defined in Table 5.9. For an SCC s that is not newly merged, we have:

$$\mathcal{T}_s^{I\cup\Delta} = \mathcal{T}_s^I \cup \mathcal{T}_s^\Delta = \mathcal{D}_s + (\mathcal{N} \cap \mathcal{I}_s) + \mathcal{I}_s \backslash \mathcal{N} = \mathcal{I}_s + \mathcal{D}_s, \tag{C.23}$$

which aligns with Property 3. Similarly, we can prove Property 3 for an SCC u that is newly merged as follows:

$$\begin{aligned}
\mathcal{T}_s^{I\cup\Delta} = \mathcal{T}_s^I \cup \mathcal{T}_s^\Delta &= \bigcup_{n\in O(s)} \{\mathcal{T}_n^I + \mathcal{T}_n^\Delta + [i_n]\} && \text{(C.24)}\\
&= \bigcup_{n\in O(s)} \{\mathcal{I}_n + \mathcal{D}_n + [i_n]\} && \text{(C.25)}\\
&= \mathcal{D}_s = \mathcal{I}_s + \mathcal{D}_s, && \text{(C.26)}
\end{aligned}$$

in which Equation C.24 represents the SCC s after merging can reach n and nodes that n can reach in domain 'I' or 'Δ', for every original SCC $n \in O(s)$. The C.24 = C.25 assumes every original SCC $n \in O(s)$ is the first time being merged, and $\mathcal{T}_n^I \cup \mathcal{T}_n^\Delta = \mathcal{I}_n + \mathcal{D}_n$ stands for an SCC n that is not newly merged, as just proved in

Equation C.23. The Equation C.26 can be derived because of $\mathcal{I}_s \subseteq \mathcal{D}_s$, as shown in Property 2. The proof can be easily extended for an original SCC $n \in O(s)$ that is newly merged already, as $\mathcal{T}_n^I \cup \mathcal{T}_n^\Delta = \mathcal{I}_n + \mathcal{D}_n$ is also proved from Equation C.24-C.26. Therefore, Property 3 is proved. □

Definition 5 *For a map C that associates each SCC s in the ordered list L with its contained elements, we extend the definition of C to operate over an interval $\mathcal{T}$ as follows:*

$$C(\mathcal{T}) = \bigcup_{s \in L} \{C(s) \mid i_s \in \mathcal{T}\}, \tag{C.27}$$

in which i_s is the index of the SCC s.

Then, we show the *insert* function can update the data structure incrementally while preserving the transitive closure property and distinguishing between old and new reachable pairs. Without loss of generality, we present the correctness of incrementally inserting a single edge, as the set of edge Δ_n^T is processed by the *insert* function edge by edge. Specifically, let $e \in \Delta_n^T$ be the single edge to be inserted next. Let G_1, Δ_1 be the graph and the edges that have been processed so far (before e), and G_2, Δ_2 be the graph that includes e and $\Delta_2 = \Delta_1 + \{e\}$. The encapsulated facts in domain 'I' before and after inserting e is notated as I^{T_1} and I^{T_2}, respectively, in which T_1 and T_2 represents the scheme before and after the insertiong of e. The same for facts in domain 'Δ'. We have the following claim:

Claim 18 *Assume $I^{T_1} = (\mathcal{R}_d^T)^\infty(N) = I$, $\Delta^{T_1} = (\mathcal{R}_d^T)^\infty(N \cup \Delta_1) \setminus I$. The execution of the insert function for a single edge $e = (i, j)$ can update the data structure correctly so that I^{T_2} stay unchanged, Δ^{T_2} returns $(\mathcal{R}_d^T)^\infty(N \cup \Delta_2) \setminus I$.*

Proof. Let u, v be the corresponding SCCs of i and j, respectively. We prove this lemma by various insertion cases below. Please note that Case 1 and Case 2 are proved regardless whether u is a fresh SCC. In the following cases, we will prove that $I^{T_1} = I^{T_2}$, and then show that Δ^{T_2} additionally returns $[(\mathcal{R}_d^T)^\infty(N \cup \Delta_2) \setminus I] \setminus \Delta^{T_1}$ comparing to Δ^{T_1}.

Case 1. SCCs u, v remain distinct, v is new: In this case, v is a fresh node and its index i_v is initialised as an index $i \in \mathcal{I}_u$ that has not been occupied yet. Then, this index i_v is added to $\mathcal{N}$ in line 18 of Algorithm 15 to record fresh nodes. Therefore, we have $i_v \in (\mathcal{N} \cap \mathcal{I}_u)$. The insertion of (u, v) makes u and u's ancestors can reach v

additionally in Δ^{T_2}, formally $\Delta^{T_2}= \Delta^{T_1}+ \{X \times C(v)\}$, in which X is a set containing elements of u and all ancestors of u, $C(v)$ maps the SCC v to its contained elements, and '$\times$' represents the cross product between two sets. Let $\mathcal{T}_u^{\Delta_1}$ and $\mathcal{T}_u^{\Delta_2}$ be the target intervals of u in domain 'Δ' before and after inserting the edge e, respectively. According to Table 5.9, we have:

$$\mathcal{T}_u^{\Delta_1} = \mathcal{D}_u^1 + (\mathcal{N}^1 \cap \mathcal{I}_u^1) \tag{C.28}$$

$$\mathcal{T}_u^{\Delta_2} = \mathcal{D}_u^2 + (\mathcal{N}^2 \cap \mathcal{I}_u^2), \tag{C.29}$$

in which the superscript 1 and 2 represents the intervals of u in T_1 and T_2, respectively. According to Algorithm 15, in this case, only the $\mathcal{N}$ interval is updated by including the index i_v of v, so $\mathcal{N}^2 = \mathcal{N}^1 \cup [i_v]$, and $\mathcal{D}_u^1 = \mathcal{D}_u^2$. Therefore, $\mathcal{T}_u^{\Delta_2} = \mathcal{T}_u^{\Delta_1} \cup [i_v]$, and for the SCC u, the nodes u can reach additionally in G_2 comparing to G_1 is the node v, which is as expected. The same applies for all ancestors a of u because $\mathcal{I}_u \subseteq (\mathcal{I}_a + \mathcal{D}_a)$ since a can reach whatever u can reach, then $i_v \in [\mathcal{N} \cap (\mathcal{I}_a + \mathcal{D}_a)]$, which is included in the target interval of a in domain 'Δ' regardless whether a is newly merged, according to Table 5.9. Therefore, u and all ancestors of u, can additionally reach v in domain 'Δ', and Δ^{T_2} is correct.

For I^{T_2}, the target intervals of u in domain 'I' are as follows, in which $\mathcal{I}_u^1 = \mathcal{I}_u^2$, and $\mathcal{N}_u^2 = \mathcal{N}_u^1 \cup [i_v]$:

$$\mathcal{T}_u^{I_1} = \mathcal{I}_u^1 \backslash \mathcal{N}^1, \tag{C.30}$$

$$\mathcal{T}_u^{I_2} = \mathcal{I}_u^2 \backslash \mathcal{N}^2. \tag{C.31}$$

Please note that although $\mathcal{I}_u^1 = \mathcal{I}_u^2$, applying $\mathcal{I}_u^1$ to the ordered table in T_1 and T_2 would retrieve different nodes. This is because a fresh node v is inserted into the ordered table when inserting e, and applying the same interval to the ordered table in T_2 will retrieve v additionally than applying to T_1. By including i_v in $\mathcal{N}^2$, the node v will be discarded from the target interval $\mathcal{T}_u^{I_2}$. Therefore, applying $\mathcal{T}_u^{I_1}$ to T_1 and applying $\mathcal{T}_u^{I_2}$ to T_2 retrieve the same set of nodes. The map C that maps each SCC to its contained elements is not updated for existing SCCs when calling the *insert* function. Therefore, encapsulated facts for node u in T_1 (denoted as $I_u^{T_1}$) are the same as those in T_2 (as $I_u^{T_2}$), formally, $I_u^{T_1} = \{C(u) \times C(\mathcal{T}_u^{I_1})\} = \{C(u) \times C(\mathcal{T}_u^{I_2})\} = I_u^{T_2}$ in which $C(\mathcal{T}_u^{I_1})$ represents the union of elements of SCCs retrieved by applying $\mathcal{T}_u^{I_1}$ to T_1; while $C(\mathcal{T}_u^{I_2})$ represents the union of elements of SCCs retrieved by applying $\mathcal{T}_u^{I_2}$ to T_2. The same applies to all ancestors of u. Therefore, we have $I^{T_2} = \bigcup_{u \in G}\{I_u^{T_2}\} = \bigcup_{u \in G}\{I_u^{T_1}\} = I^{T_1}$.

Case 2. SCCs u, v remains distinct, v is not new: In this case, the $\mathcal{D}_u$ interval of u is updated by line 19 of Algorithm 15, and the $\mathcal{D}_a$ interval of every ancestor a

of u is updated by Algorithm 16. The $\mathcal{I}_s$ interval for every node s and the global $\mathcal{N}$ interval remain unchanged. According to Table 5.9, target interval $\mathcal{T}_s^I$ of domain 'I' is unchanged when incrementally inserting e. So, $I^{T_2} = I^{T_1}$.

The insertion of (u, v) builds a connection between every ancestor a of u and every descendant d of v, as presented in Figure 5.4. Therefore, u and ancestors of u, should reach v and v's descendants in Δ^{T_2} after the insertion, if it cannot reach in I^{T_1}. Considering the precondition that I^{T_1} and Δ^{T_1} are correct, for node u, target intervals of u covering indexes that u can reach in domain 'I' and 'Δ' in T_1 are as follows:

$$\mathcal{T}_u^{I_1} = \mathcal{I}_u \backslash \mathcal{N}, \tag{C.32}$$

$$\mathcal{T}_u^{\Delta_1} = \mathcal{D}_u^1 + (\mathcal{N} \cap \mathcal{I}_u), \tag{C.33}$$

in which the superscript 1 represents intervals of u in T_1 before inserting the edge e. Intervals without an superscript are not changed when inserting e, and thus are not distinguished for brevity. Nodes that v can reach in G_1 are:

$$\mathcal{T}_v^{I_1} + \mathcal{T}_v^{\Delta_1} = \mathcal{I}_v + \mathcal{D}_v, \tag{C.34}$$

which captures nodes that v can reach regardless the domains. Please note that nodes that v can reach are not changed in G_1 and G_2, since the intervals of v are not updated by Algorithm 15 in this case. Then, we present an interval representing nodes that u can reach in Δ^{T_2} but not in I^{T_1}, using target intervals of u and v in T_1; and show how this interval is exactly aligned to the target interval of u for domain 'Δ' in T_2:

$$(\mathcal{T}_u^{\Delta_1} + \mathcal{T}_v^{I_1} + \mathcal{T}_v^{\Delta_1} + [i_v]) \setminus \mathcal{T}_u^{I_1} = \tag{C.35}$$

$$(\mathcal{D}_u^1 + (\mathcal{N} \cap \mathcal{I}_u) + \mathcal{I}_v + \mathcal{D}_v + [i_v]) \setminus (\mathcal{I}_u \backslash \mathcal{N}) = \tag{C.36}$$

$$[(\mathcal{D}_u^1 + \mathcal{I}_v + \mathcal{D}_v + [i_v]) \setminus (\mathcal{I}_u \backslash \mathcal{N})] + [(\mathcal{N} \cap \mathcal{I}_u) \setminus (\mathcal{I}_u \backslash \mathcal{N})] = \tag{C.37}$$

$$[\mathcal{D}_u^2 \setminus (\mathcal{I}_u \backslash \mathcal{N})] + [(\mathcal{N} \cap \mathcal{I}_u) \setminus (\mathcal{I}_u \backslash \mathcal{N})] = \tag{C.38}$$

$$\mathcal{D}_u^2 + (\mathcal{N} \cap \mathcal{I}_u) = \tag{C.39}$$

$$\mathcal{T}_u^{\Delta_2}, \tag{C.40}$$

in which C.37 = C.38 matches directly with the operation in line 19 of Algorithm 15, and C.38 = C.39 is because of the property when u is not a newly merged SCC: $\mathcal{D}_u \cap \mathcal{I}_u = \emptyset$, as discussed in Property 1. Therefore, by applying $\mathcal{T}_u^{\Delta_2}$ to T_2, the node u can additionally reach v and v's descendants comparing to nodes that u can reach in T_1, if u cannot reach it before (as '$\backslash \mathcal{T}_u^{I_1}$). The same applies for all ancestors of u. When a, an ancestor of u, is not newly merged, $\mathcal{T}_a^{\Delta_2}$ capturing fresh reachable

nodes among v and v's descendants can be proved similarly to Equations C.35-C.40. Otherwise, when a is newly merged, we show that every original SCC $n \in O(a)$ satisfies this condition as following. Specifically, the target intervals of n in T_1 are presented as:

$$\mathcal{T}_n^{I_1} = \mathcal{I}_n \backslash \mathcal{N}, \tag{C.41}$$

$$\mathcal{T}_n^{\Delta_1} = (\mathcal{D}_s^1 \backslash \mathcal{I}_n) + (\mathcal{N} \cap \mathcal{I}_n). \tag{C.42}$$

Then, we show $\mathcal{T}_n^{\Delta_2}$ can capture v and v's descendants if n cannot reach it before by:

$$(\mathcal{T}_n^{\Delta_1} + \mathcal{T}_v^{I_1} + \mathcal{T}_v^{\Delta_1} + [i_v]) \setminus \mathcal{T}_n^{I_1} = \tag{C.43}$$

$$[(\mathcal{D}_s^1 \backslash \mathcal{I}_n) + (\mathcal{N} \cap \mathcal{I}_n)] \backslash (\mathcal{I}_n \backslash \mathcal{N}) + (\mathcal{T}_v^{I_1} + \mathcal{T}_v^{\Delta_1} + [i_v]) \backslash (\mathcal{I}_n \backslash \mathcal{N}) = \tag{C.44}$$

$$[(\mathcal{D}_s^1 \backslash \mathcal{I}_n) + \mathcal{T}_v^{I_1} + \mathcal{T}_v^{\Delta_1} + [i_v]] \backslash (\mathcal{I}_n \backslash \mathcal{N}) \ + \ (\mathcal{N} \cap \mathcal{I}_n) \backslash (\mathcal{I}_n \backslash \mathcal{N}) = \tag{C.45}$$

$$(\mathcal{D}_s^2 \backslash \mathcal{I}_n) + (\mathcal{N} \cap \mathcal{I}_n) = \tag{C.46}$$

$$\mathcal{T}_n^{\Delta^2}, \tag{C.47}$$

in which C.45 = C.46 matches directly with how interval changes are passed into ancestors of u in line 4 of Algorithm 16. Therefore, for u and all the ancestors of u, their target intervals in domain 'Δ' covers v and v's descendants, if they cannot reach it before. Finally, the correctness Δ^{T_2} is proved.

Case 3. SCCs merge: This section presents a case in which the insertion of (u, v) causes the merging of SCCs $\{n_1, \ldots, n_m\}$. Let $s \in \{n_1, \ldots, n_m\}$ be a representative node. We call s a newly merged SCC, and denote original SCCs as $O(s) = \{n_1, \ldots, n_m\}$. This case is processed by lines 20-22. Before merging, encapsulated facts of original SCCs in $O(s)$ in domain 'I' can be retrieved by $I_{O(s)}^{T_1} = \bigcup_{n \in O(s)} \{C(n) \times C(\mathcal{T}_n^{I_1})\}$. While facts of SCCs in $O(s)$ after merging can be represented as $I_{O(s)}^{T_2} = \bigcup_{n \in O(s)} \{C(n) \times C(\mathcal{T}_n^{I_2})\}$. We now prove $I^{T_1} = I^{T_2}$ by proving $\mathcal{T}_n^{I_1} = \mathcal{T}_n^{I_2}$ as follows:

$$\mathcal{T}_n^{I_1} = \mathcal{I}_n \backslash \mathcal{N} = \mathcal{T}_n^{I_2}, \tag{C.48}$$

in which intervals $\mathcal{I}_n$ and $\mathcal{N}$ stay unchanged during the *insert* function call. The same stands for all s's ancestors. Therefore, $I^{T_1} = I^{T_2}$ is proved. Similarly, we can prove $\Delta^{T_1} = \Delta^{T_2}$ by showing that for every original SCC $n \in O(s)$, the target interval of n in domain 'Δ' includes new reachable nodes that are caused by the merge, it n cannot reach before. The merge of $\{n_1, \ldots, n_m\}$ makes each of them can reach the descendants of each other, as well as themselves, as presented in Figure 5.5. Formally,

we prove $\mathcal{T}_n^{\Delta_2} = (\mathcal{T}_n^{\Delta_1} + \bigcup_{m \in O(s)} \{\mathcal{T}_m^{\Delta_1} + \mathcal{T}_m^{I_1} + [i_m]\}) \setminus \mathcal{T}_n^{I_1}$ stands for every original SCC $n \in O(s)$:

$$(\mathcal{T}_n^{\Delta_1} + \bigcup_{m \in O(s)} \{\mathcal{T}_m^{\Delta_1} + \mathcal{T}_m^{I_1} + [i_m]\}) \setminus \mathcal{T}_n^{I_1} = \tag{C.49}$$

$$(\mathcal{D}_n^1 + (\mathcal{N} \cap \mathcal{I}_n) + \bigcup_{m \in O(s)} \{\mathcal{I}_m + \mathcal{D}_m^1 + [i_m]\}) \setminus (\mathcal{I}_n \backslash \mathcal{N}) = \tag{C.50}$$

$$(\mathcal{D}_s^2 \backslash \mathcal{I}_n) + (\mathcal{N} \cap \mathcal{I}_n) = \tag{C.51}$$

$$\mathcal{T}_n^{\Delta_2}, \tag{C.52}$$

in which C.49 = C.50 can be proved by using the expressions of $\mathcal{T}_n^{\Delta_1}$ and $\mathcal{T}_n^{I_1}$ defined in Table 5.9, and $\mathcal{T}_m^{\Delta_1} + \mathcal{T}_m^{I_1} = \mathcal{I}_m + \mathcal{D}_m^1$ is true for every original SCC $m \in O(s)$ according to Property 3. The C.50 = C.51 can be matched directly with how the interval $\mathcal{D}_s$ of the representative s is updated in line 22 of Algorithm 15. Therefore, facts retrieved for every original SCC $n \in O(s)$ in Δ^{T_2} is correct. The same can be easily proved for every ancestor a of s. So Δ^{T_2} is correct when SCCs merge. □

Claim 18 demonstrates that the *insert* function correctly update the data structure during incremental insertion of a single edge, ensuring that encapsulated facts in various domains are maintained consistently. Then we show the correctness of the *insert* function when inserting a set of facts Δ_n^T in Claim 19.

Claim 19 *Assume that the encapsulated facts in the data structure before the* `insert` *call are as follows:*

$$I^T = (\mathcal{R}_d^T)^\infty[N], \tag{C.53}$$

$$\Delta^T = \emptyset. \tag{C.54}$$

Then, after the `insert` *call, the encapsulated facts are updated as follows:*

$$I^T = (\mathcal{R}_d^T)^\infty[N], \tag{C.55}$$

$$\Delta^T = (\mathcal{R}_d^T)^\infty[N \cup \Delta_n^T] \setminus I^T, \tag{C.56}$$

in which N represents edges of the underlying graph before the call, and Δ_n^T represents facts that are incrementally added to the data structure during the `insert` *call.*

Proof. We prove this claim by induction.

- *Induction base:* Let the edges that have been processed be Δ_1. Before any edge $e \in \Delta_n^T$ is inserted, i.e., $\Delta_1 = \emptyset$, facts encapsulated in domain 'I' and 'Δ' are $I^{T_1} = (\mathcal{R}_d^T)^\infty[N]$, $\Delta^T = \emptyset = (\mathcal{R}_d^T)^\infty[N] \setminus I^T = (\mathcal{R}_d^T)^\infty[N \cup \Delta_1] \setminus I^T$, just as defined in Claim 19. Therefore, before any edge is inserted, I^T and Δ^T can access facts as expected.

- *Induction step:* The pre-condition stated in Claim 18 is satisfied if previous insertions of edges in Δ_1 is correct. The correctness of inserting a single edge $e \in \Delta_n^T$ is proved by Claim 18. Therefore, the insertion of $\Delta_1 = \Delta_1 \cup \{e\}$ is correct, and encapsulated facts are as defined in Equations C.55-C.56.

As discussed above, when the pre-condition stated in Claim 19 is satisfied, the *insert* function can maintain the data structure incrementally as defined in the claim. □

Claim 20 *The* `insert` *function inserts Δ_n^T incrementally and update the data structure, ensuring that encapsulated facts are shown as Equations C.55-C.56. The subsequent* `merge` *function ensures encapsulated facts are:*

$$I^T = (\mathcal{R}_d^T)^\infty[N], \tag{C.57}$$

$$\Delta^T = \emptyset, \tag{C.58}$$

in which N represents facts in the underlying graph G.

Proof. For clarity, we use I^{T_1} and I^{T_2} to denote facts in domain 'I' before and after the *merge* call, respectively. The same for Δ^{T_1} and Δ^{T_2}. Then, we prove that, the *merge* function updates I^T as follows:

$$I^{T_2} = I^{T_1} \cup \Delta^{T_1} = (\mathcal{R}_d^T)^\infty[N^1 \cup \Delta_n^T] = (\mathcal{R}_d^T)^\infty[N^2], \tag{C.59}$$

$$\Delta^{T_2} = \emptyset, \tag{C.60}$$

in which N^1 represents the facts in the underlying graph G before the *insert* call, and $N^2 = N^1 \cup \Delta_n^T$ represents the facts in the underlying graph during the *merge* call. Please note that the *merge* call does not change the graph, therefore N^2 matches with the N stated in Claim 20. Recall that the *merge* function shown in Algorithm 17 updates the interval $\mathcal{I}_s$ by adding $\mathcal{D}_s$ to it, and D_s is emptied. For an SCC s that is newly merged during the previous *insert* call, its contained elements are updated by updating the map C. We use C^1 and C^2 to denote the map before and after the update, respectively. Finally, the interval N covering fresh introduced SCCs is emptied. According to Table 5.9, we can prove $\Delta^{T_2}= \emptyset$ by showing that the target interval in domain 'Δ' after the *merge* call is empty for every SCC s:

$$\mathcal{T}_s^{\Delta_2} = \mathcal{D}_s^2 + (\mathcal{N}^2 \cap \mathcal{I}_s^2) = \emptyset. \tag{C.61}$$

Then, we prove that I^{T_2} is exactly as defined in Equation C.59. For an SCC s that is not newly merged during the previous *insert* call, its target interval in domain 'I' satisfies the following:

$$\mathcal{T}_s^{I_2} = \mathcal{I}_s^2 \backslash \mathcal{N}^2 = \mathcal{I}_s^1 + \mathcal{D}_s^1 = \mathcal{T}_s^{I_1} + \mathcal{T}_s^{\Delta_1} = \mathcal{T}_s^{I_1 \cup \Delta_1}, \tag{C.62}$$

in which the Property 3 is applied. The facts recovered from the SCC s in domain 'I' after the *merge* call can be represented as:

$$I_s^{T_2} = \{C^2(s) \times C^2(\mathcal{T}_s^{I_2})\}, \tag{C.63}$$

which is equal to facts recovered from the same SCC s in domain 'I' and 'Δ' before the *merge* call defined as follows:

$$I_s^{T_1} \cup \Delta_s^{T_1} = \{C^1(s) \times C^1(\mathcal{T}_s^{I_1 \cup \Delta_1})\}. \tag{C.64}$$

The C.64 = C.63 can be derived because $C^1(s) = C^2(s)$, since s is not a newly merged SCC, and its contained elements are not updated. For $C^1(\mathcal{T}_s^{I_1 \cup \Delta_1}) = C^1(\mathcal{T}_s^{I_2}) = C^2(\mathcal{T}_s^{I_2})$, we prove this by two cases: (1) consider an SCC u that $i_u \in \mathcal{T}_s^{I_2}$ and u is not a newly merged SCC, so $C^1(u) = C^2(u)$; (2) u is a newly merged SCC and $i_u \in \mathcal{T}_s^{I_2}$. Let $O(u) = \{m_1, \ldots, m_n\}$ be u's original SCCs. The interval $\mathcal{T}_s^{I_1 \cup \Delta_1}$ covers all the nodes that s can reach, including the indices of all the original SCCs in $O(u)$. The *merge* function removes all the original SCCs in $O(u)$ and only keeps the representative u. The map $C^2(u)$ is updated to contain all elements that are used to belong to one of u's original SCCs. Therefore, we have $\bigcup_{m \in O(u)} \{C^1(m)\} = C^2(u)$. Additionally, $\mathcal{T}_s^{I_1 \cup \Delta_1}$ captures all SCCs in $O(u)$, while $\mathcal{T}_s^{I_2}$ only captures u. Finally, we have $C^1(\mathcal{T}_s^{I_1 \cup \Delta_1}) = C^2(\mathcal{T}_s^{I_2})$. As discussed above, for an SCC s that is not newly merged during the last *insert* call, the facts recovered from the SCC s in domain 'I' is exactly what is stated in Claim 20.

For an SCC s that is newly merged, the facts recovered in T_1 can be retrieved by accessing each original SCC $n \in O(s)$; while for T_2, the same set of facts can be retrieved by accessing s solely. Formally:

$$I_s^{I_1} \cup \Delta_s^{I_1} = \bigcup_{n \in O(s)} \{C^1(n) \times C^1(\mathcal{T}_n^{I_1 \cup \Delta_1})\}, \tag{C.65}$$

$$I_s^{I_2} = \{C^2(s) \times C^2(\mathcal{T}_s^{I_2})\}, \tag{C.66}$$

in which the target intervals $\mathcal{T}_n^{I_1 \cup \Delta_1}$ and $\mathcal{T}_s^{I_2}$ are defined as follows:

$$\mathcal{T}_n^{I_1 \cup \Delta_1} = \mathcal{I}_s^1 + \mathcal{D}_s^1 = \mathcal{D}_s^1, \tag{C.67}$$

$$\mathcal{T}_s^{I_2} = \mathcal{I}_s^2 \setminus \mathcal{N}^2 = \mathcal{I}_s^1 + \mathcal{D}_s^1 = \mathcal{D}_s^1. \tag{C.68}$$

As discussed above, we have $\bigcup_{n \in O(s)} \{C^1(n)\} = C^2(s)$, and $C^1(\mathcal{T}_n^{I_1 \cup \Delta_1}) = C^2(\mathcal{T}_s^{I_2})$ because $\mathcal{T}_s^{I_2} = \mathcal{T}_n^{I_1 \cup \Delta_1}$. Therefore, C.65 = C.66. So for all SCCs, after the *merge* function, the I^{T_2} is exactly what is claimed in Claim 20. □

C.4.3 Correctness of Incremental Deletion

We first present some properties that hold during the *delete* call, and then prove the correctness of incremental deletion.

Property 4 *Let s be an SCC that breaks during the* `delete` *call, and $O(s) = \{n_1, \ldots, n_m\}$ represent the newly formed SCCs after the deletion. The contained elements of s and $O(s)$ have the following property:*

$$C^1(s) = \bigcup_{n \in O(s)} \{C^2(n)\}, \tag{C.69}$$

in which $C^1(s)$ represents the elements originally contained in s, and $C^2(n)$ denotes the elements associated with each newly formed SCC $n \in O(s)$.

Proof. This property can be derived directly from how the map C is updated in line 7 of Algorithm 18. Specifically, during the *delete* call, when an SCC s breaks into smaller SCCs $O(s) = \{n_1, \ldots, n_m\}$, the map C is updated to reflect the redistribution of elements. □

Property 5 *Let s be an SCC that breaks during the* `delete` *call, and $O(s) = \{n_1, \ldots, n_m\}$ represent the newly formed SCCs after the deletion. For every original ancestor a of s, the following property holds:*

$$i_n \in \mathcal{I}_a, \forall n \in O(s). \tag{C.70}$$

Proof. The index of each newly formed SCC $n \in O(s)$ is assigned such that $i_n \in \mathcal{I}_s$, as shown in line 6 of Algorithm 18. Because a is one of s's original ancestors before the *delete* call, then a can reach s, a can reach whatever s can reach, formally, $\mathcal{I}_s \subseteq \mathcal{I}_a$. Therefore, we have $i_n \in \mathcal{I}_s \subseteq \mathcal{I}_a$, for each newly formed SCC $n \in O(s)$. □

Claim 21 *Assume that the encapsulated facts in the data structure before the* `delete` *call are as follows:*

$$I^T = (\mathcal{R}_d^T)^\infty[N], \tag{C.71}$$

$$\Delta^T = \emptyset. \tag{C.72}$$

Then, after the `delete` *call, the encapsulated facts are updated as follows:*

$$I^T = (\mathcal{R}_d^T)^\infty[N], \tag{C.73}$$

$$\Delta^T = I^T \setminus (\mathcal{R}_d^T)^\infty[N \setminus \Delta_n^T], \tag{C.74}$$

in which N represents edges of the underlying graph before the call, and Δ_n^T represents facts that are incrementally deleted from the data structure during the `delete` *call.*

Proof. The *delete* function recomputes reachable pairs after the deletion in line 9 of Algorithm 18, and then target intervals for the domain 'Δ' are computed in line 12. We will first show I^T after the deletion satisfy Equation C.73, and reachable pairs that are still valid after the deletion $R^T = (\mathcal{R}_d^T)^\infty[N \backslash \Delta_n^T]$, finally we show $\Delta^T = I^T \setminus R^T$, as defined in Equation C.74. For clarity, we use the superscript 1 and 2 to represent the associated information before and after the update.

For an SCC s that didn't break during the *delete* call, the facts recovered from s in domain 'I' before and after the deletion can be represented as:

$$I_s^{T_1} = \{C^1(s) \times C^2(\mathcal{I}_s^1)\}, \tag{C.75}$$

$$I_s^{T_s} = \{C^1(s) \times C^2(\mathcal{I}_s^2)\}, \tag{C.76}$$

in which $\mathcal{I}_s^1 = \mathcal{I}_s^2 = \mathcal{I}_s$ since the *delete* function didn't change the $\mathcal{I}_s$ interval of s. The contained elements in s remain unchanged: $C^1(s) = C^2(s)$. For every node n that is included in the interval $\mathcal{I}_s$, such that $i_n \in \mathcal{I}_s$, we have $C^1(n) = C^2(n)$ provided that n is not a break SCC during the *delete* function. When n is an SCC that breaks during the deletion, $i_n \in \mathcal{I}_s$ implies that s is an ancestor of n. Following Property 5, for any other newly formed SCC $n' \in O(n)$, n' is also included in the interval $\mathcal{I}_s$: $i_{n'} \in \mathcal{I}_s$. So we have $C^1(n) = \bigcup_{n' \in O(s)}\{C^2(n')\}$, as discussed in Property 4. Therefore, $C^1(\mathcal{I}_s^1) = C^2(\mathcal{I}_s^2)$, and $I_s^{T_1} = I_s^{T_2}$ stands for an SCC s that didn't break. For an SCC s that breaks, facts in domain 'I' before the deletion are the same as Equation C.75, and that after deletion can be retrieved by accessing facts for each newly formed SCC $n \in O(s)$: $\bigcup_{n \in O(s)}\{C^2(n) \times C^2(\mathcal{I}_n^2)\}$. The interval $\mathcal{I}_n^2$ of newly formed SCC n is initialised as $\mathcal{I}_s$, as shown in line 6 of Algorithm 18. So $\mathcal{I}_s^1 = \mathcal{I}_n^2$. As discussed above, we have $C^1(\mathcal{I}_s^1) = C^2(\mathcal{I}_n^2)$. According to Property 4, $C^1(s) = \bigcup_{n \in O(s)}\{C^2(n)\}$. Then, $I_s^{T_1} = \bigcup_{n \in O(s)}\{C^2(n) \times C^2(\mathcal{I}_n^2)\}$ is proved for an SCC s that breaks. We have $I^{T_1} = I^{T_2}$.

Then we prove reachable pairs after the deletion $K^T = (\mathcal{R}_d^T)^\infty[N \backslash \Delta_n^T]$ is correct using the interval $\mathcal{D}'_u$ computed in line 9 of Algorithm 18. As shown in line 8, this interval is computed by traversing the graph in reversed topological order, and using

intervals of the current node's children, which have already been processed. Therefore, we prove this by induction:

1. *Induction base:* For each leaf node u that has no children, the interval $\mathcal{D}'_u = \emptyset$ in line 9. Whether u can reach itself is determined in line 11: if the SCC u contains more than one resource, then the SCC u can reach all elements in u; Or if the fact $(C(u), C(u))$ exists, u can also reach itself. In both cases, i_u is added to $\mathcal{D}'_u$ to correctly represent the nodes that u can reach after the deletion. So, $\mathcal{D}'_u$ reflects nodes that are reachable from u. And the $\mathcal{D}_u$ interval computed in line 12 represents nodes that u cannot reach after the deletion, and thus can be used to retrieve Δ^T.

2. *Induction step:* For the current node u, the interval $\mathcal{D}'_u$ computed in line 9 covers every child c of u, as well as nodes that c can reach after the deletion. This step represent a path $u \to c \dashrightarrow d$ from u to one of c's descendants d, provided that c has been processed, and $\mathcal{I}_c \backslash \mathcal{D}_c$ represents nodes that c can reach. Whether u can reach itself is determined in line 11, as discussed in the induction base. Therefore, the interval $\mathcal{D}'_u$ can represent nodes that u can reach after the deletion correctly; while the interval $\mathcal{D}_u$ covers nodes that can no longer be reached after the deletion.

As discussed above, we have Δ^{T_2} is correct. □

Claim 22 *The* `delete` *function deletes* Δ_n^T *incrementally from the data structure, ensuring that encapsulated facts are shown as Equations C.71-C.72. The subsequent* `remove` *function ensures encapsulated facts are:*

$$I^T = (\mathcal{R}_d^T)^\infty[N], \tag{C.77}$$
$$D^T = D^T \cup \Delta^T, \tag{C.78}$$
$$\Delta^T = \emptyset, \tag{C.79}$$

in which N *represents edges in the underlying graph.*

Proof. The map C is not changed during the *remove* function, and the target intervals in domain 'I', 'Δ', and 'D' are defined as $\mathcal{I}_s$, $\mathcal{D}_s$, $\mathcal{D}_s^m$, respectively. Let I^T should be updated as follows:

$$I^{T_2} = I^{T_1} \backslash \Delta^{T_1} = (\mathcal{R}_d^T)^\infty[N^1] \setminus (I^{T_1} \backslash ((\mathcal{R}_d^T)^\infty[N^1 \backslash \Delta_n^T])) \tag{C.80}$$
$$= (\mathcal{R}_d^T)^\infty[N^1 \backslash \Delta_n^T] \tag{C.81}$$
$$= (\mathcal{R}_d^T)^\infty[N^2], \tag{C.82}$$

in which $N^2 = N^1 \backslash \Delta_n^T = N$ represents the edges in the underlying graph after the *delete* call, and N^1 represents the edges before the previous *delete* call. The *remove* function removes the interval $\mathcal{D}_s$ from $\mathcal{I}_s$, which is equivalent to removing Δ^T from I^T. Similarly, $\mathcal{D}_s$ is added to $\mathcal{D}_s^m$, which is equivalent to adding Δ^T to D^T. Finally, $\mathcal{D}_s$ is emptied, so Δ^T is emptied. □

C.4.4 Proof of Claims 1-2

Claim 19 defines encapsulated facts in the TC scheme after the *insert* function with a pre-condition that states the status of the TC scheme before the *insert* call. Then, we show this pre-condition is preserved every time the *insert* is called during the reasoning process presented in Algorithm 12. This directly leads to the correctness of Claim 1 showing that the *insert* function can update the data structure correctly and encapsulated facts satisfy Definition 4.

Claim 1. *During the multi-scheme DRed algorithm, given facts Δ_n^T that are to be inserted, the* `insert` *function of the TC scheme updates the data structure such that the encapsulated facts in domain 'I' and 'Δ' are maintained as follows:*

$$I^T = (\mathcal{R}_d^T)^\infty[N], \tag{C.83}$$

$$\Delta^T = (\mathcal{R}_d^T)^\infty[N \cup \Delta_n^T] \setminus I^T, \tag{C.84}$$

in which N represents existing edges of the underlying graph G before the incremental insertion. Proof. We prove this lemma by showing that the pre-condition stated in Equation C.53-C.54 of Claim 19 is preserved during the multi-scheme DRed algorithm. The multi-scheme DRed algorithm calls the function following a certain pattern, as defined below.

Definition 6 *Following Hu et al [77], we define a call history of H^T of the form C.85 for each scheme T as a finite and nonempty sequence of runs with length m; each Q_i^T with $0 \leq i \leq m$ is a finite and nonempty sequence of calls of the form C.86; each $C_{i,j}^T$ is a call of functions* `insert`, `merge`, `delete`, `remove`, *or* `rederive`, *in which $0 \leq j \leq h_i$ and h_i is the length of the ith call history.*

$$H^T = Q_0^T, \ldots, Q_m^T, \tag{C.85}$$

$$Q_i^T = C_{i,1}^T, \ldots, C_{i,h_i}^T. \tag{C.86}$$

Specifically, for a scheme T, Q_0^T represents the call history of internal functions in T during the initial materialisation, and thus Q_0^T would be a sequence of *insert* and

merge calls, as the *overdelete* and *rederive* procedures are not invoked during the initial materialisation. Let $h_0 = 2n$, each call $C^T_{0,2j-1}$ with $1 \leq j \leq n$ in the call history is a *insert* call maintaining the data structure to populate the facts in domain 'Δ', while each all $C^T_{0,2j}$ with $1 \leq j \leq n$ in the call history is a *merge* call to merge facts in domain 'Δ' to domain 'I'. While for any Q^T_i call history with $0 < i \leq m$ represents the incremental materialisation. Let $1 \leq d = 2k < h_i$ be the length of calls used in the *overdelete* procedure, then each call $C^T_{i,2j-1}$ with $1 \leq j \leq k$ in the call history is a *delete* call to compute facts in domain 'Δ', while each all $C^T_{i,2j}$ with $1 \leq j \leq k$ in the call history is a *remove* call to remove facts in 'Δ' from the domain 'I'. Then the call $C^T_{i,d+1}$ is a *rederive* call used in the rederivation procedure. The following calls $C^T_{i,j}$ with $d + 1 < j \leq h_i$ are *insert* and *merge* calls, in which $C^T_{i,j}$ is a *insert* call when j is even, and is a *merge* call when j is odd.

Following the definition above, it is strightforward to see that the *insert* function will be used in the following 3 cases. We prove that the pre-condition is satisfied in these 3 cases:

1. During the initial materialisation with a call history Q^T_0, the first *insert* call $C^T_{0,0}$ is used to initially construct the data structure. Therefore, the facts N in the underlying graph is empty: $N = \emptyset$. So encapsulated facts in domain 'I' and 'Δ' are empty:

$$I^T = (\mathcal{R}^T_d)^\infty[N] = \emptyset, \tag{C.87}$$

$$\Delta^T = \emptyset, \tag{C.88}$$

   which aligns with the pre-condition in Equations C.53-C.54.

2. During the incremental materialisation with a call history Q^T_i in which $i > 1$, the first *insert* call $C^T_{i,d+2}$ is invoked after the overdeletion and rederivation procedure, in which d is the length of calls used during the overdeletion procedure. The calls $C^T_{i,j}$ with $j \leq d$ in this call history are a sequence of *delete* and *remove* calls, which update the graph and maintain encapsulated facts consistently. Following Claims 21 and 22, after the last *remove* call $C^T_{i,d}$, the encapsulated facts are updated as shown in Equations C.77-C.79, which aligns with the pre-conditions presented in Equations C.53-C.54. The *rederive* call $C^T_{i,d+1}$ does not update the data structure, nor encapsulated facts in domain 'I' and 'Δ'. Therefore, the pre-condition is satisfied before the first *insert* call $C^T_{i,d+2}$.

3. The *insert* function is also used during the loop in lines 16-23 of Algorithm 12, representing the recursive application happened during reasoning. This case covers the *insert* calls $C^T_{0,2j-1}$ with $2 \leq j \leq n$ and $h_0 = 2n$ during the initial materialisation with a call history Q^T_0, and the insert calls $C^T_{i,2j}$ with $i \geq 1$, and $2j$ is an even number that $d{+}2 < 2j \leq h_i$ during the incremental materialisation with a call history Q^T_i, and d represents the length of calls invoked during the overdeletion procedure. Then, we discuss these two cases separately: (1) For every call $C^T_{0,2j-1}$ in the initial materialisation, the calls before $C^T_{0,2j-1}$ are a sequence of *insert* and *merge* calls. When $j = 2$, the previous calls $C^T_{0,0}$ and $C^T_{0,1}$ are the initial *insert* and *remove* functions, respectively. The property of the initial *insert* and *remove* calls is discussed in Claims 16-17, ensuring that the encapsulated facts in domain 'I' and 'Δ' before the call $C^T_{0,3}$ satisfy the pre-condition defined in Equations C.53-C.54; When $j > 2$, the previous *insert* and *merge* calls perform the incremental addition. As discussed in Claims 19-20, these previous calls can maintain the data structure such that the pre-condition is satisfied; (2) For the calls $C^T_{i,2j}$ during the incremental maintenance, the previous calls include the overdeletion and rederivation procedure. In point 2, we discussed the pre-condition is satisfied before the first *insert* call $C^T_{i,d+2}$. The *insert* call $C^T_{i,d+2}$ and the *merge* call $C^T_{i,d+3}$ perform the incremental addition, whose correctness is discussed in Claims 19-20. Therefore, the pre-condition is satisfied before the *insert* call $C^T_{i,d+4}$. The same applies for all the calls $C^T_{i,2j}$ with $i \geq 1$, and $d + 2 < 2j \leq h_i$.

The discussion above covers all cases in which the *insert* function is invoked, and we have shown that the pre-condition is always satisfied before each of the *insert* calls. Therefore, following Claim 19, the *insert* function maintains the data structure as described in Claim 1, during the execution of the multi-scheme DRed algorithm. □

Similarly, we can prove the correctness of the *delete* function during the execution of the multi-scheme DRed algorithm.

Claim 2. *During the multi-scheme DRed algorithm, given facts Δ^T_n that are to be deleted, the* `delete` *function updates the data structure so that the encapsulated facts in domain 'I' and 'Δ' are maintained as follows:*

$$I^T = (\mathcal{R}^T_d)^\infty[N], \tag{C.89}$$

$$\Delta^T = I^T \setminus (\mathcal{R}^T_d)^\infty[N \backslash \Delta^T_n], \tag{C.90}$$

in which N represents facts that are present in the underlying graph G before the incremental deletion.

Proof. The *delete* function is invoked in the following cases:

1. During the incremental materialisation with a call history Q_1^T, the first call $C_{1,1}^T$ is a *delete* call. In this case, the previous run Q_0^T represents the initial materialisation. The call history of Q_0^T is a sequence of *insert* and *merge* calls. As discussed in Claim 1 and Claim 20, the initial materialisation updates the data structure correctly so that the pre-condition stated in Equations C.71-C.72 is satisfied.

2. The *delete* is called recursively during the loop in lines 8-12 of Algorithm 12. The calls $C_{1,2j+1}^T$ with $1 \leq j \leq (d/2-1)$ and d is the length of calls used in the overdeletion procedure in the run Q_1^T. As discussed in point 1, the pre-condition is satisfied before the call $C_{1,1}^T$. Following Claims 21-22, the initial *delete* call $C_{1,1}^T$ and *remove* call $C_{1,2}^T$ can maintain the data structure so that the encapsulated facts are as described in Equations C.77-C.79, which aligns with the pre-condition presented in Equations C.71-C.72. Therefore, before the call $C_{1,3}^T$, the pre-condition is satisfied. The same stands for all the calls $C_{1,2j+1}^T$ with $1 \leq j \leq (d/2-1)$.

3. During the incremental materialisation with a call history Q_i^T in which $i > 1$, the first call $C_{i,1}^T$ is a *delete* call. Take the run Q_2^T as an example, we need to prove that the previous run Q_1^T maintains the data structure correctly so that the pre-condition is satisfied before the call $C_{2,1}^T$. The point 2 has already proved the correctness of the overdeletion procedure, i.e., the pre-condition is satisfied before the *rederive* call $C_{1,d+1}^T$ in which d is the number of calls invoked during the overdeletion procedure. The *rederive* call does not change the data structure, and the correctness of the following calls $C_{1,j}^T$ with $(d+1) < j \leq h_1$ invoked during the addition procedure is discussed in the point 3 of the proof of Claim 1, therefore we have the run Q_1^T is correct. Finally, the pre-condition is satisfied before the first *delete* call $C_{2,1}^T$. The same applies to the other initial *delete* calls $C_{i,1}^T$ with $i > 1$ of runs Q_i^T.

4. The *delete* calls $C_{i,2j+1}^T$ with $i > 1$, and $1 \leq j \leq (d/2-1)$ are invoked recursively in runs Q_i^T, before which the pre-condition is also satisfied. This can be proved similarly to the point 2 discussed above.

In summary, during the multi-scheme DRed algorithm, the pre-condition is satisfied for the *delete* calls of all cases. □

C.4.5 Correctness of the TC scheme

Finally, we prove that the TC scheme satisfies the invariants in Definition 4. The properties of the *insert* and *delete* functions are established in Claims 1–2. Specifically, these functions process the facts to be inserted or deleted while maintaining the data structure and preserving the transitive closure. Moreover, changes to facts are encapsulated in the domain 'Δ', which can be accessed through the serialisation of the data structure. In the proof below, we demonstrate that the encapsulated facts in Δ^T adhere to the bounds specified in Definition 4. Additionally, we verify that the *merge*, *remove*, and *rederive* functions also conform to the requirements of Definition 4.

Lemma 5. *The TC scheme satisfies the invariants in Definition 4.*

Proof. We demonstrate that the TC scheme adheres to Definition 4 by verifying the correctness of each function individually.

1. As described in Claim 1, the *insert* function updates the data structure so that encapsulated facts in domain 'I' are not changed, and $\Delta^T = (\mathcal{R}_d^T)^\infty[N \cup \Delta_n^T] \setminus I^T$, in which N is the edges in the underlying graph before the *insert* call, and Δ_n^T represents a set of edges to be inserted. Provided that $N \subseteq I = \bigcup_{T \in \mathbb{T}} \{I^T\}$ and $\Delta_n^T \subseteq (\mathcal{R}_d^T)^\infty[N \cup \Delta_n^T]$, we have:

$$(\Delta_n^T \cup \mathcal{R}_d^T[I \cup \Delta_n^T]) \setminus I^T \quad \subseteq \quad \Delta^T = (\mathcal{R}_d^T)^\infty[N \cup \Delta_n^T] \setminus I^T \quad \subseteq \quad (\mathcal{R}_d^T)^\infty[I \cup \Delta_n^T] \setminus I^T, \tag{C.91}$$

   which aligns with Definition 4. The *rederive* function empties the D^T, after which the *insert* and *merge* functions are invoked. Therefore, D^T remains empty during the addition procedure, i.e., when *insert* and *merge* functions are used. This conforms to the requirements described in Definition 4.

2. The *merge* function is discussed in Claim 20, stating that Δ^T is added to I^T and emptied afterwards.

3. The Claim 2 discusses the *delete* function of the TC scheme, which keeps I^T unchanged and updates encapsulated facts in domain 'Δ' as $\Delta^T = I^T \setminus$

$(\mathcal{R}_d^T)^\infty[N \backslash \Delta_n^T]$. We need to prove that Δ_n^T adhere to the lower and upper bounds as below:

$$\left[\Delta_n^T \cup (I \cap X)\right] \setminus \Pi^\infty[E'] \;\subseteq\; \Delta^T = I^T \setminus (\mathcal{R}_d^T)^\infty[N \backslash \Delta_n^T] \;\subseteq\; I^T, \qquad \text{(C.92)}$$

in which $X = \mathcal{R}_d^T[I \cup M_d^T, M_d^T \cup \Delta_n^T]$, and it is straightforward to verify the upper bound. Then, for the lower bound, we first prove the following equation:

$$\Delta_n^T \setminus \Pi^\infty[E'] \;\subseteq\; I^T \setminus (\mathcal{R}_d^T)^\infty[N \backslash \Delta_n^T], \qquad \text{(C.93)}$$

since for any fact $t \in \Delta_n^T \backslash \Pi^\infty[E']$ that is to be deleted and is not included in the final materialisation $\Pi^\infty[E']$, t will not be included in the updated transitive closure $(\mathcal{R}_d^T)^\infty[N \backslash \Delta_n^T]$. Provided that $t \in \Delta_n^T \subseteq I^T$, and $t \notin (\mathcal{R}_d^T)^\infty[N \backslash \Delta_n^T]$, we have $t \in I^T \backslash (\mathcal{R}_d^T)^\infty[N \backslash \Delta_n^T]$. Therefore, Equation C.93 stands. Then, we prove that

$$(I \cap X) \backslash \Pi^\infty[E'] \;\subseteq\; I^T \setminus (\mathcal{R}_d^T)^\infty[N \backslash \Delta_n^T]. \qquad \text{(C.94)}$$

The right-hand side can be rewritten as $I^T \setminus (\mathcal{R}_d^T)^\infty[N \backslash \Delta_n^T] = (\mathcal{R}_d^T)^\infty[N] \setminus (\mathcal{R}_d^T)^\infty[N \backslash \Delta_n^T]$, which includes direct and indirect consequences of applying $\mathcal{R}_d^T$ using facts Δ_n^T. For the left-hand side, $X = \mathcal{R}_d^T[I \cup M_d^T, M_d^T \cup \Delta_n^T] = \mathcal{R}_d^T[I \cup M_d^T, M_d^T] \cup \mathcal{R}_d^T[I \cup M_d^T, \Delta_n^T]$ in which $I \cap \mathcal{R}_d^T[I \cup M_d^T, M_d^T] = \emptyset$: the rule $\mathcal{R}_d^T$ only uses the transitive predicate in its body and is only applicable to facts in M_d^T with this predicate. These facts are the facts deleted from the transitive closure in the previous round. Thus, applying $\mathcal{R}_d^T$ to M_d^T will not derive any fact that currently exists in the transitive closure of the remaining graph, i.e., in I^T. For any fact $t \in (I \cap X) \backslash \Pi^\infty[E']$ that is a direct consequence of deleting Δ_n^T with regard to $\mathcal{R}_d^T$ and is not included in the updated transitive closure $\Pi^\infty[E']$, t will also be included in $I^T \setminus (\mathcal{R}_d^T)^\infty[N \backslash \Delta_n^T]$. Therefore, Equation C.94 is proved. Finally, the *delete* function meets the requirement described in Definition 4, as shown in Equation C.92.

4. As described in Claim 22, the *remove* function removes Δ^T from I^T, adds Δ^T to D^T, and then empties Δ^T.

5. The *rederivable* function uses the backward evaluation of rules in $\mathcal{R}_n^T$ for overdeleted facts $D^T \nsubseteq I^T$, more specifically, it recovers facts $\mathcal{R}_n^T[I] \cap D^T$, which equals to $\Pi_T[I] \cap D^T = (\mathcal{R}_n^T[I] \cup \mathcal{R}_d^T[I]) \cap D^T$. This is because the transitive closure is maintained so that $I^T = \mathcal{R}_d^T[I^T]$, then we have $\mathcal{R}_d^T[I] = \mathcal{R}_d^T[I^T] = I^T$ and $\mathcal{R}_d^T[I] \cap D^T = \emptyset$. Therefore, the *rederivable* function conforms to the requirement in Definition 4.

As discussed above, the TC scheme satisfies the invariants in Definition 4. □

C.5 Correctness of the Union Scheme: Lemma 6

In this section, we prove the correctness of the union scheme. This scheme computes consequences of the union rules in $\mathcal{R}_d^T$ on demand to save memory, rather than storing these facts explicitly. In this scheme, only the consequences of the normal rules $\mathcal{R}_n^T$ are stored in a default table L_T. We define Equations 5.10-5.11 and 5.12-5.13 to represent facts in various domains during the addition and the overdeletion procedures, respectively. We use an operator Γ in these equations to rewrite facts with the underlying predicates $\mathcal{U}_T$ into the union predicate U. These equations effectively define how facts in different domains are retrieved: for example, L_T^{Δ} represents retrieving corresponding facts from the default table L_T, while Γ denotes the rewriting process applied to the given facts.

Claims 3-4 establish the correctness of the *insert* and *delete* operations in the union scheme. Specifically, we prove that the equations governing the addition procedure ensure the correct encapsulation of newly derived facts, and the equations for the overdeletion procedure guarantee that derived consequences are correctly updated and removed. These claims collectively demonstrate that the union scheme satisfies the requirements for correctness while achieving its goal of memory efficiency by avoiding explicit storage of union rule derivations. The equivalence of Equation 5.14 in Claim 3 and Equation 5.10 can be easily verified by changing L_T^{Δ} to Δ_n^T, representing the fresh derivations of normal rules $\mathcal{R}_n^T$ stored in the default table L_T; and changing $\Gamma[M_a^T]$ to $\mathcal{R}_d^T[M_a^T]$, representing derivations of union rules $\mathcal{R}_d^T$. The same applies to Equation 5.15 in Claim 4 and Equation 5.12.

Next, we prove Lemma 6 by analysing each function of the union scheme individually, demonstrating how they collectively ensure correctness and adherence to the requirements of the scheme.

Lemma 6. *The union scheme satisfies the invariants in Definition 4.*

Proof. The requirements of the Definition 4 is discussed as follows:

1. We first prove that the encapsulated facts in domain 'Δ' after the *insert* function is exactly the lower bound of Δ^T stated in Definition 4, formally:

$$(\Delta_n^T \cup \mathcal{R}_d^T[M_a^T]) \setminus I^T = (\Delta_n^T \cup \mathcal{R}_d^T[I \cup \Delta_n^T]) \setminus I^T, \tag{C.95}$$

which can be proved by showing that $\mathcal{R}_d^T[M_a^T] \setminus I^T = \mathcal{R}_d^T[I \cup \Delta_n^T] \setminus I^T$. The M_a^T represents fact additions that are added in the last round, and $M_a^T \subseteq I$. So for every fact $t \in \mathcal{R}_d^T[M_a^T] \setminus I^T$, the fact t is also included in $\mathcal{R}_d^T[I \cup \Delta_n^T] \setminus I^T$, thus:

$$\mathcal{R}_d^T[M_a^T] \setminus I^T \subseteq \mathcal{R}_d^T[I \cup \Delta_n^T] \setminus I^T. \tag{C.96}$$

On the other hand, $\mathcal{R}_d^T$ represents union rules and is not applicable to Δ_n^T containing facts with the union predicate: $\mathcal{R}_d^T[\Delta_n^T] = \emptyset$. For every fact $t \in I \setminus M_a^T$ that has been added to I in at least two rounds ago, then union rules $\mathcal{R}_d^T$ have been applied to t if the fact t has a predicate in $\mathcal{U}_T$, formally, $\mathcal{R}_d^T[t] \in I^T$. Therefore, we have:

$$\mathcal{R}_d^T[M_a^T] \setminus I^T \supseteq \mathcal{R}_d^T[I \cup \Delta_n^T] \setminus I^T. \tag{C.97}$$

Finally, Equation C.95 is proved. For the facts in domain 'I' represented by Equation 5.11, the encapsulated facts are not changed because the facts L_T^I stored explicitly in domain 'I' are not updated, and the only change is the facts with the supporting predicates in M_a^T are added into I, but their corresponding consequences with the predicate U are not. Therefore, by removing M_a^T from I^R when applying the rewriting to facts with the supporting predicate $R \in \mathcal{U}_T$, fresh additions derived by the union rules are excluded from I^T, which is as expected.

2. The *merge* function of the union scheme updates the label of facts with a label 'Δ' to 'I' in the default table, which is equivalent to adding L_T^Δ to L_T^I. The facts in domain 'I' can be expressed by Equation 5.9, in which the L_T^I captures fresh additions L_T^Δ, and $\Gamma(I^R)$ for every $R \in \mathcal{U}_T$ captures new derivations $\Gamma(M_a^T)$ of the union rules because M_a^T is already added to domain 'I' by corresponding schemes in the last round. Therefore, the *merge* function can updates encapsulated facts in domain 'I' correctly, just as stated in the Definition 4. For the facts in domain 'Δ' retrieved by Equation 5.10, $L_T^\Delta = \emptyset$ because there is no fact with the label 'Δ' in the default table, and $\Gamma(M_a^T) \setminus I^T = \emptyset$ since I^T includes new derivations $\Gamma(M_a^T)$ of the union rules, as discussed above. Therefore, the *merge* function adheres to the bounds specified in Definition 4.

3. Then, we prove the encapsulated facts in domain 'Δ' after the *delete* function adhere to the bounds specified in Definition 4. Formally, we need to prove that:

$$\left[\Delta_n^T \cup (X \cap I)\right] \setminus \Pi^\infty[E'] \;\subseteq\; \Delta_n^T \cup (\mathcal{R}_d^T[M_d^T] \cap I^T) \;\subseteq\; I^T, \tag{C.98}$$

in which $X = \mathcal{R}_d^T[I \cup M_d^T, M_d^T \cup \Delta_n^T]$ represents the one-step derivation of the union rules with fact changes M_d^T and Δ_n^T. The upper bound can be easily verified by the fact that $\Delta_n^T \subseteq I^T$ and $(\mathcal{R}_d^T[M_d^T] \cap I^T) \subseteq I^T$. For the lower bound, we first prove that:

$$(X \cap I) \quad \subseteq \quad (\mathcal{R}_d^T[M_d^T] \cap I^T), \tag{C.99}$$

which can be proved by showing that $\mathcal{R}_d^T[\Delta_n^T]$ is empty, and $\mathcal{R}_d^T[M_d^T] \cap I^T = \mathcal{R}_d^T[I \cup M_d^T, M_d^T] \cap I^T$. Therefore, $(X \cap I) \setminus \Pi^\infty[E'] \subseteq (\mathcal{R}_d^T[M_d^T] \cap I^T)$, and the lower bound of Equation C.98 is proved. For the facts in domain 'I', the encapsulated facts are not changed by using Equation 5.13, because the facts L_T^I stored explicitly in domain 'I' are not updated, and the only change is the facts with the supporting predicates in M_d^T are removed from I, but their corresponding consequences with the union predicate U are not. Therefore, by including tracked fact changes M_d^T when applying the rewriting to facts with the supporting predicates $\mathcal{U}_T$, fresh deletions derived by the union rules are included, which is as expected.

4. The *remove* function of the union scheme removes facts in domain 'Δ' from domain 'I'. Specifically, for facts stored explicitly in the default table L_T with a label 'Δ', the label 'Δ' is removed, and the label 'I' is added. For the consequences of the union rules that are not stored explicitly, the tracked fact changes M_d^T have already been deleted from domain 'I' by their corresponding schemes in the last round. The encapsulated facts in domain 'I' are retrieved by Equation 5.9, in which L_T^I captures the facts in the default table with the label 'I', in which the facts with the label 'Δ' are removed, and $\Gamma(I^R)$ for every $R \in \mathcal{U}_T$ captures the consequences of the union rules that are not stored explicitly. As discussed above, M_d^T is already removed from 'I^R' by the corresponding schemes in the last round, so $\Gamma(I^R)$ effectively eliminates the fresh deletions derived by the union rules. Therefore, the *remove* function updates the encapsulated facts in domain 'I' correctly.

5. The *rederivable* function of the union scheme uses the backward evaluation of rules in $\Pi_T = \mathcal{R}_n^T \cup \mathcal{R}_d^T$ for overdeleted facts $D^T \nsubseteq I^T$, more specifically, it recovers facts $\Pi_T[I] \cap D^T$, which matches with the requirement of Definition 4.

The proof for facts in domain 'D' is omitted because it is strightforward to verify that $D^T = \emptyset$ during the addition, and the *remove* function accumulate deleted facts in domain 'D'. Therefore, the union scheme satisfies the invariants in Definition 4. □

C.6 Proof of Lemma 7

In this section, we analyse the impact of including counting mechanisms in the schemes. The effect of counting can be summarised as "delete less, rederive more", leveraging auxiliary information about the number of derivations for each fact. Specifically, the number of derivations is updated within the *insertable* and *deletable* functions. In the *deletable* function, it checks whether the number of non-recursive derivations has dropped to zero. If so, the fact can be safely deleted; otherwise, it is preserved from overdeletion, potentially improving efficiency by reducing unnecessary rederivations. Similarly, the *rederivable* function verifies whether a given fact still has any recursive or non-recursive derivations. If so, the fact is rederived, ensuring that the correctness of the reasoning process is maintained while benefiting from the reduced computational overhead enabled by counting.

Take the *deriveForDeletion* function shown in Algorithm 10 as an example, the counting mechanisms does not change the rule applications in line 8, including the maintenance of the data structure in line 10, but changes which facts are schedulable into the set Δ_n^T in line 9. Therefore, rather than proving the correctness of the counting mechanisms within individual schemes, we demonstrate that the inclusion of counting mechanisms is permissible within the multi-scheme DRed algorithm and that the algorithm's correctness is preserved when these mechanisms are applied. Specifically, we prove Lemma 7 by demonstrating that the bounds of the corresponding modular operators permit the use of counting mechanisms, ensuring that the algorithm's correctness is preserved.

Lemma 7. *Enabling both recursive and non-recursive counting does not violate the correctness bounds. Specifically, the overdeleted facts and the rederived facrs adhere to the bounds defined in Table 5.7.*

Proof. Let A_c and A represent the set of facts that are derived after invoking the *deriveForDeletion* function with and without the counting mechanisms, respectively, we prove A_c adhere to the bounds defined for the operators Π_T^-, provided that A already adheres to the bounds of Π_T^-. Since the counting mechanisms can only make the set of facts that are overdeleted smaller, we only prove A still adhere to the lower bound of Π_T^-, formally:

$$(I \cap \Pi_T[I \cup M]) \setminus \Pi_T^\infty[E'] \quad \subseteq \quad A_c \quad \subseteq \quad A. \tag{C.100}$$

To prove Equation C.100, we prove that for every fact $t \in (A \backslash A_c)$, $t \notin \big(I \cap \Pi_T[I \cup M]\big) \setminus \Pi_T^\infty[E']$. Intuitively, the lower bound represents derivations within one step, while A

includes derivations from one or more steps, and A_c can be derived from A by removing facts with at least one non-recursive counting. Therefore, for every $t \in (A \backslash A_c)$, t has at least one non-recursive derivation. We discuss the following two cases: (1) t is not included in the one-step application of Π_T, then $t \notin \big(I \cap \Pi_T[I \cup M]\big) \setminus \Pi_T^\infty[E']$ stands. (2) t is included in the one-step application of Π_T, then $t \in \big(I \cap \Pi_T[I \cup M]\big)$. However, t has at least one non-recursive derivation, so $t \in \Pi_T^\infty[E']$. Therefore, $t \notin \big(I \cap \Pi_T[I \cup M]\big) \setminus \Pi_T^\infty[E']$ stands.

Then, we prove that the rederived facts adhere to the bounds defined for the operator Π_T^r. Let B_c and B represent the set of facts that are rederived after invoking the *rederive* function with and without the counting mechanisms, respectively, we prove B_c adhere to the bounds defined for the operator Π_T^r, provided that B already adheres to the bounds of Π_T^r. Since the counting mechanisms can only make the set of facts that are rederived larger, we only prove B still adhere to the upper bound of Π_T^r, formally:

$$B \subseteq B_c \subseteq \Pi^\infty[E'] \cap D. \tag{C.101}$$

To prove Equation C.101, we prove that for every fact $t \in (B_c \backslash B)$, $t \in \Pi^\infty[E'] \cap D$. Following the counting mechanisms, $B_c \backslash B$ includes facts that are in D and have at least one non-recursive or recursive derivation. Please note that the rederive procedure is invoked after the overdeletion procedure, so facts with at least one counting can be represented as $\Pi^\infty[I]$ in which I is the current materialisations when the rederive procedure is invoked. Therefore, $\Pi^\infty[I] \subseteq \Pi^\infty[E']$. Finally, for every fact $t \in (B_c \backslash B)$, $t \in \Pi^\infty[E'] \cap D$ stands. □

C.7 Proof of Theorem 3

Theorem 3. *The DRed algorithm with multiple schemes presented in Algorithm 12 is correct if all schemes satisfy Definition 4.*

Proof. In this section, we prove the correctness of the multi-scheme DRed algorithm by showing that Algorithm 12 is equivalent to the modular DRed algorithm shown in Algorithm 20. Both algorithms follow a general DRed procedure, including the addition, deletion, and rederivation of facts. The multi-scheme DRed algorithm uses the schemes to store consequences of different types of rules, while the modular DRed algorithm uses a default table to store all derivations. During the overdeletion procedure, the modular algorithm first initialises the fact changes M as facts to be deleted

E^-, and removes M from I in in line 7. Then, the changes M and current materialisation I is used in line 9 to derive facts to be deleted, whose bounds are defined by the operator Π_T^- in Table C.1. Then, the derivation is applied recursively with the fact changes updated in line 12 and the materialisation updated in line 13.

The multi-scheme algorithm uses the schemes to store the consequences of different types of rules, and the fact changes are stored in the schemes. In line 6, the facts to be deleted E^- are scheduled into the schemes. Please note that here E^- is not yet deleted from the materialisation I. These facts are processed and encapsulated in the domain 'Δ' in line 8, and removed from the materialisation I in line 12. This will not influence the correctness of the algorithm, just potentially change the order of the fact processing. The multi-scheme algorithm uses an internal *flagChanges* function to update fact changes in line 11. Instead of using a global M, the multi-scheme algorithm uses the M_d^T to represent the fact changes during the deletion for each scheme T. Each scheme identifies fact changes with the predicate B_T to collect facts that are useful when applying the rules Π_T managed by the current scheme T. This will only influence the effectiveness of the reasoning process, but not the correctness of the algorithm, as only "relavant" facts are identified for each scheme.

Provided that the *deriveForDeletion* function of each scheme T applies rules and encapsulates the derivations Δ^T in scheme's internal data structure, and Δ^T adhere to the bounds of the modular operator Π_T^-, as proved in Theorem 2, the scheme can be considered as a valid module when computing the derivations during the overdeletion procedure. Moreover, as shown in Definition 4, the encapsulated facts in domain 'I'remain unchanged, just like the I in line 9 of the modular algorithm. The derivations Δ^T for each scheme T is flagged as fact changes $M_d^{T'}$ for any relevant schemes T' in line 11, which is equivalent to updating the global fact changes M in the modular algorithm in line 12. Finally, the scheme uses a *remove* function to remove the facts in domain 'Δ' from the materialisation I in line 12, which is equivalent to the removal of the fact changes M from the materialisation I in line 13 of the modular algorithm, as shown in Definition 4.

For the rederivation procedure, the multi-scheme algorithm uses the *rederive* function of a scheme T to rederive facts D^T that are overdeleted from the data structure. The overdeleted facts D^T are collected in line 12 using the *remove* function, just like the D in line 13 of the modular algorithm. Following Theorem 2, the rederived facts in the multi-scheme algorithm adhere to the bounds of the modular operator Π_T^r. Therefore, the scheme in the multi-scheme algorithm can be considered as a valid module when computing the rederivations during the rederivation procedure.

Finally, for the addition procedure, the multi-scheme algorithm uses the *deriveForAddition* function of each scheme T to derive facts to be added and encapsulate them in the data structure of the domain 'Δ'. The Theorem 2 establishes that the derivations Δ^T for each scheme T adhere to the bounds of the modular operator Π^+_T. Additionally, the encapsulated facts in domain 'I' remain unchanged, just like the I in line 20 of the modular algorithm. The derivations Δ^T for each scheme T are flagged as fact changes $M_a^{T'}$ for any relevant schemes T' in line 22, which is equivalent to updating the global fact changes M in the modular algorithm in line 23. Finally, the scheme uses a *merge* function to merge the facts in domain 'Δ' into the materialisation I in line 23, which is equivalent to the merging of Δ^T into the materialisation I in line 24 of the modular algorithm, as shown in Definition 4.

Therefore, the multi-scheme DRed algorithm is correct if all schemes satisfy Definition 4. □

Appendix D

Datalog Programs in Benchmarks

This section presents Datalog programs for the benchmarks that are designed to test specialised schemes and the interaction between different types of schemes.

D.1 TC

The Datalog program used in the TC dataset is shown below. It defines a simple transitive closure over a directed graph. The predicate `p:path` represents whether there is a path from node `?X` to node `?Y`. The first rule captures direct edges in the graph, while the second rule recursively derives reachable pairs by chaining existing path facts. The second rule declares the relation `p:path` as transitive.

```
p:path[?X,?Y] :- p:edge[?X,?Y] .
p:path[?X,?Z] :- p:path[?X,?Y], p:path[?Y,?Z] .
```

D.2 TC+

The Datalog program used in the TC+ dataset is shown below. It defines two transitive relations, `R` and `R2`, which are mutually reverse: `R2` captures the reverse of `R`, and vice versa. As such, both relations can be efficiently managed within a single TC scheme, as discussed in Section 5.4.6. The predicate `R` can be derived from various combinations of base relations (`A`, `B`, `C`) non-recursively, and from `D` and itself recursively; while `R2` is non-recursively derived from `F`, and recursively derived from `E`, `R`, and `X`. Additionally, the relation `X` depends on `R2`, introducing further interaction between transitive relations and normal relations. This program is designed to test the performance of the TC scheme under more complex recursive and mutually dependent transitive relations.

```
<http://R>[?X,?Z] :- <http://R>[?X,?Y], <http://R>[?Y,?Z] .
<http://R>[?X,?Y] :- <http://A>[?X,?Y] .
<http://R>[?X,?Y] :- <http://B>[?X,?Y], <http://C>[?X,?Y] .
<http://R>[?X,?Z] :- <http://R>[?X,?Y], <http://D>[?Y,?Z] .
<http://R2>[?X,?Z] :- <http://R2>[?X,?Y], <http://R2>[?Y,?Z] .
<http://R2>[?X,?Y] :- <http://R>[?Y,?X] .
<http://R>[?X,?Y] :- <http://R2>[?Y,?X] .
<http://R2>[?X,?Y] :- <http://F>[?X,?Y] .
<http://X>[?X,?Y] :- <http://R2>[?X,?Y] .
<http://R2>[?Y,?X] :- <http://X>[?X,?Y], <http://E>[?X,?Y] .
```

D.3 U

The Datalog program of the U dataset is shown as below, in which `U` is the union of other 14 relations.

```
<http://U>[?X,?Y] :- <http://A>[?X,?Y] .
<http://U>[?X,?Y] :- <http://B>[?X,?Y] .
<http://U>[?X,?Y] :- <http://C>[?X,?Y] .
<http://U>[?X,?Y] :- <http://D>[?Y,?X] .
<http://U>[?X,?Y] :- <http://E>[?X,?Y] .
<http://U>[?X,?Y] :- <http://F>[?X,?Y] .
<http://U>[?X,?Y] :- <http://G>[?X,?Y] .
<http://U>[?X,?Y] :- <http://H>[?X,?Y] .
<http://U>[?X,?Y] :- <http://I>[?X,?Y] .
<http://U>[?X,?Y] :- <http://J>[?X,?Y] .
<http://U>[?X,?Y] :- <http://K>[?X,?Y] .
<http://U>[?X,?Y] :- <http://L>[?X,?Y] .
<http://U>[?X,?Y] :- <http://M>[?X,?Y] .
<http://U>[?X,?Y] :- <http://N>[?X,?Y] .
```

D.4 TCU

The Datalog program for the TCU dataset is shown below. In this program, the relation `U` is defined as the union of the relations `A` and `B`. The relation `A` is transitive and mutually reverse with another transitive relation, `A2`. Moreover, there are recursive dependencies between `A`, `A2`, and `U`, making this program a representative

case for evaluating interactions between transitive and union relations within the multi-scheme framework.

```
<http://U>[?X,?Y] :- <http://A>[?X,?Y] .
<http://U>[?X,?Y] :- <http://B>[?X,?Y] .
<http://C>[?X,?Y] :- <http://U>[?X,?Y] .
<http://B>[?X,?Y] :- <http://C>[?X,?Y] .
<http://U>[?X,?Y] :- <http://D>[?X,?Y], <http://E>[?X,?Y] .
<http://A>[?X,?Y] :- <http://A>[?X,?Z], <http://A>[?Z,?Y] .
<http://A2>[?X,?Y] :- <http://A2>[?X,?Z], <http://A2>[?Z,?Y] .
<http://A>[?X,?Y] :- <http://A2>[?Y,?X] .
<http://A2>[?X,?Y] :- <http://A>[?Y,?X] .
<http://U>[?X,?Y] :- <http://A2>[?X,?Y] .
<http://F>[?X,?Y] :- <http://U>[?X,?Y], <http://D>[?X,?Y] .
```

Appendix E

Queries used in Query Answer Experiments

This section presents test queries using the TC predicate and the union predicate, used in Section 5.7.5.

```
# 1. Queries using TC predicate

# Queries evaluated over DBpedia50# TC Q_0
Select (count(*) as ?cnt) where {
    ?x skos:narrowerTransitive ?y .
}

# TC Q_1
Select (count(*) as ?cnt) where {
    ?x   a    skos:Concept .
    ?x   skos:narrowerTransitive  ?y .
}

# TC Q_2
Select (count(*) as ?cnt) where {
    ?x   term:subject  db:Reproducibility .
    ?x   skos:narrowerTransitive  ?y .
}

# TC Q_3
Select (count(*) as ?cnt) where {
```

```
    ?x   term:subject   db:Uncertainty .
    ?x   skos:narrowerTransitive ?y .
}

# TC Q_4
Select (count(*) as ?cnt) where {
    ?x   term:subject db:Supercritical_angle_fluorescence_microscopy .
    ?x   skos:narrowerTransitive ?y .
}

# TC Q_5
Select (count(*) as ?cnt) where {
    ?x   term:subject   db:Micrograph .
    ?x   skos:narrowerTransitive ?y .
}

# TC Q_6
Select (count(*) as ?cnt) where {
    ?y   a   skos:Concept .
    ?x   skos:narrowerTransitive ?y .
}

# TC Q_7
Select (count(*) as ?cnt) where {
    ?y   term:subject   db:Reproducibility .
    ?x   skos:narrowerTransitive ?y .
}

# TC Q_8
Select (count(*) as ?cnt) where {
    ?y   term:subject   db:Uncertainty .
    ?x   skos:narrowerTransitive ?y .
}

# TC Q_9
Select (count(*) as ?cnt) where {
```

```
    ?y   term:subject      db:Supercritical_angle_fluorescence_microscopy .
    ?x   skos:narrowerTransitive ?y .
}

# TC Q_10
Select (count(*) as ?cnt) where {
    ?y   term:subject   db:Micrograph .
    ?x   skos:narrowerTransitive ?y .
}

# Queries evaluated over TC+
Select ?x ?y where {  ?x <http://R> ?y }

# Queries evaluated over TCU
Select ?x ?y where { ?x <http://A> ?y }

# 2. Queries using Union predicate

# Queries evaluated over DBpedia50# Union Q_0
Select ?x ?y where {
    ?x    <http://www.w3.org/2004/02/skos/core#semanticRelation> ?y .
}

# Union Q_1
Select ?x ?y where {
    ?x    a <http://www.w3.org/2004/02/skos/core#Concept>.
    ?x    <http://www.w3.org/2004/02/skos/core#semanticRelation> ?y .
}

# Union Q_2
Select ?x ?y where {
    ?x term:subject <http://dbpedia.org/resource/Reproducibility>.
    ?x <http://www.w3.org/2004/02/skos/core#semanticRelation> ?y .
}

# Union Q_3
```

```
Select ?x ?y where {
    ?x    term:subject <http://dbpedia.org/resource/Uncertainty>.
    ?x    <http://www.w3.org/2004/02/skos/core#semanticRelation> ?y .
}

# Union Q_4
Select ?x ?y where {
    ?x term:subject db:Supercritical_angle_fluorescence_microscopy.
    ?x <http://www.w3.org/2004/02/skos/core#semanticRelation> ?y .
}

# Union Q_5
Select ?x ?y where {
    ?x    term:subject <http://dbpedia.org/resource/Micrograph>.
    ?x    <http://www.w3.org/2004/02/skos/core#semanticRelation> ?y .
}

# Union Q_6
Select ?x ?y where {
    ?y    a <http://www.w3.org/2004/02/skos/core#Concept>.
    ?x    <http://www.w3.org/2004/02/skos/core#semanticRelation> ?y .
}

# Union Q_7
Select ?x ?y where {
    ?y    term:subject <http://dbpedia.org/resource/Reproducibility>.
    ?x    <http://www.w3.org/2004/02/skos/core#semanticRelation> ?y .
}

# Union Q_8
Select ?x ?y where {
    ?y    term:subject <http://dbpedia.org/resource/Uncertainty>.
    ?x    <http://www.w3.org/2004/02/skos/core#semanticRelation> ?y .
}

# Union Q_9
```

```
Select ?x ?y where {
    ?y    term:subject db:Supercritical_angle_fluorescence_microscopy .
    ?x    <http://www.w3.org/2004/02/skos/core#semanticRelation> ?y .
}

# Union Q_10
Select ?x ?y where {
    ?y     term:subject <http://dbpedia.org/resource/Micrograph>.
    ?x     <http://www.w3.org/2004/02/skos/core#semanticRelation> ?y .
}

# Queries evaluated over U
Select ?x ?y where { ?x <http://U> ?y }

# Queries evaluated over TCU
Select ?x ?y where { ?x <http://U> ?y }
```

Bibliography

[1] Daniel J Abadi, Adam Marcus, Samuel R Madden, and Kate Hollenbach. Scalable Semantic Web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 411–422, 2007. (Cited on pages 32 and 33.)

[2] Christopher R Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. Old techniques for new join algorithms: A case study in RDF processing. In *2016 IEEE 32nd International Conference on Data Engineering Workshops (ICDEW)*, pages 97–102. IEEE, 2016. (Cited on pages 7, 40, and 63.)

[3] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)*, 42(4):1–44, 2017. (Cited on page 63.)

[4] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995. (Cited on pages 1, 3, 4, 14, 15, 16, 17, and 18.)

[5] Serge Abiteboul, Meghyn Bienvenu, Alban Galland, and Marie-Christine Rousset. Distributed Datalog revisited. In *International Datalog 2.0 Workshop*, pages 252–261. Springer, 2010. (Cited on page 127.)

[6] Rakesh Agrawal, Alexander Borgida, and Hosagrahar Visvesvaraya Jagadish. Efficient management of transitive relationships in large data and knowledge bases. *ACM SIGMOD Record*, 18(2):253–262, 1989. (Cited on pages 70, 72, 88, 97, 111, 118, 119, and 161.)

[7] Afnan Alhazmi, Tom Blount, and George Konstantinidis. ForBackBench: a benchmark for chasing vs. query-rewriting. *Proc. VLDB Endow.*, 15(8): 1519–1532, April 2022. ISSN 2150-8097. doi: 10.14778/3529337.3529338. URL `https://doi.org/10.14778/3529337.3529338`. (Cited on page 4.)

[8] Waqas Ali, Muhammad Saleem, Bin Yao, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. A survey of RDF stores & SPARQL engines for querying knowledge graphs. *The VLDB Journal*, pages 1–26, 2022. (Cited on page 31.)

[9] Mario Alvarez-Picallo, Alex Eyers-Taylor, Michael Peyton Jones, and C.-H. Luke Ong. Fixing incremental computation: Derivatives of fixpoints, and the recursive semantics of Datalog. *CoRR*, abs/1811.06069, 2018. URL `http://arxiv.org/abs/1811.06069`. (Cited on page 119.)

[10] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M Hellerstein, and Russell Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European Conference on Computer Systems*, pages 223–236, 2010. (Cited on pages 1 and 14.)

[11] Mario Alviano, Wolfgang Faber, Nicola Leone, and Marco Manna. Disjunctive Datalog with existential quantifiers: Semantics, decidability, and complexity issues. *CoRR*, abs/1210.2316, 2012. URL `http://arxiv.org/abs/1210.2316`. (Cited on page 124.)

[12] Samuel Arch, Xiaowen Hu, David Zhao, Pavle Subotić, and Bernhard Scholz. Building a join optimizer for Soufflé. In *International Symposium on Logic-Based Program Synthesis and Transformation*, pages 83–102. Springer, 2022. (Cited on page 36.)

[13] Samuel Isaac Arch. *Automatic Index Selection for Inequalities*. PhD thesis, The University of Sydney Australia 22, 2020. (Cited on page 36.)

[14] Molham Aref, Balder ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. Design and implementation of the LogicBlox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1371–1382, 2015. (Cited on pages 3, 4, 28, and 37.)

[15] Francois Bancilhon. Naive evaluation of recursively defined relations. In *On Knowledge Base Management Systems*, pages 165–178. Springer, 1986. (Cited on pages 27 and 118.)

[16] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. In *Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 269–284, 1987. (Cited on page 3.)

[17] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM (JACM)*, 30(3): 479–513, 1983. (Cited on page 42.)

[18] Luigi Bellomarini, Georg Gottlob, and Emanuel Sallinger. The Vadalog system: Datalog-based reasoning for knowledge graphs. *arXiv preprint arXiv:1807.08709*, 2018. (Cited on pages 3 and 4.)

[19] Luigi Bellomarini, Georg Gottlob, and Emanuel Sallinger. Datalog-based reasoning for knowledge graphs. In *AMW*, 2019. (Cited on page 2.)

[20] Bilal Ben Mahria, Ilham Chaker, and Azeddine Zahi. An empirical study on the evaluation of the RDF storage systems. *Journal of Big Data*, 8:1–20, 2021. (Cited on page 31.)

[21] Philip A Bernstein and Dah-Ming W Chiu. Using semi-joins to solve relational queries. *Journal of the ACM (JACM)*, 28(1):25–40, 1981. (Cited on pages 43 and 135.)

[22] Philip A Bernstein and Nathan Goodman. Power of natural semijoins. *SIAM Journal on Computing*, 10(4):751–771, 1981. (Cited on page 42.)

[23] Manuel Bichler, Michael Morak, and Stefan Woltran. lpopt: A rule optimization tool for answer set programming. *Fundamenta Informaticae*, 177(3-4):275–296, 2020. (Cited on page 64.)

[24] Meghyn Bienvenu, Balder Ten Cate, Carsten Lutz, and Frank Wolter. Ontology-based data access: A study through disjunctive Datalog, CSP, and MMSNP. *ACM Transactions on Database Systems (TODS)*, 39(4):1–44, 2014. (Cited on page 2.)

[25] Meghyn Bienvenu, Stanislav Kikot, Roman Kontchakov, Vladimir V Podolskii, and Michael Zakharyaschev. Theoretically optimal Datalog rewritings for OWL 2 QL ontology-mediated queries. *arXiv preprint arXiv:1604.05258*, 2016. (Cited on page 120.)

[26] Robert Binna, Wolfgang Gassler, Eva Zangerle, Dominic Pacher, and Günther Specht. SpiderStore: A native main memory approach for graph storage. *Grundlagen Von Datenbanken*, 733, 2011. (Cited on page 33.)

[27] Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. OWLIM: A family of scalable semantic repositories. *Semantic Web*, 2(1):33–42, 2011. (Cited on page 4.)

[28] Angela Bonifati, Wim Martens, and Thomas Timm. An analytical study of large SPARQL query logs. *arXiv preprint arXiv:1708.00363*, 2017. (Cited on page 59.)

[29] Sebastian Brandt, Elem Güzel Kalaycı, Roman Kontchakov, Vladislav Ryzhikov, Guohui Xiao, and Michael Zakharyaschev. Ontology-based data access with a horn fragment of metric temporal logic. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017. (Cited on page 124.)

[30] Sebastian Brandt, Elem Guüzel Kalaycı, Vladislav Ryzhikov, Guohui Xiao, and Michael Zakharyaschev. Querying log data with metric temporal logic. *Journal of Artificial Intelligence Research*, 62:829–877, 2018. (Cited on page 124.)

[31] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *International Semantic Web Conference*, pages 54–68. Springer, 2002. (Cited on page 31.)

[32] Andrea Calì, Georg Gottlob, Thomas Lukasiewicz, and Andreas Pieris. Datalog+/-: A family of languages for ontology querying. In *Datalog Reloaded: First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, pages 351–368. Springer, 2011. (Cited on pages 65 and 124.)

[33] Francesco Calimeri, Simona Perri, and Jessica Zangari. Optimizing answer set computation via heuristic-based decomposition. *Theory and Practice of Logic Programming*, 19(4):603–628, 2019. (Cited on page 64.)

[34] Andrea Calì, Georg Gottlob, Thomas Lukasiewicz, Bruno Marnette, and Andreas Pieris. Datalog+/-: A family of logical knowledge representation and query languages for new applications. In *2010 25th Annual IEEE Symposium on Logic in Computer Science*, pages 228–242, 2010. doi: 10.1109/LICS.2010.27. (Cited on pages 14 and 124.)

[35] David Carral, Irina Dragoste, Larry González, Ceriel Jacobs, Markus Krötzsch, and Jacopo Urbani. Vlog: A rule engine for knowledge graphs. In *International Semantic Web Conference*, pages 19–35. Springer, 2019. (Cited on pages 3, 4, 8, and 35.)

[36] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In *VLDB*, volume 91, pages 577–589, 1991. (Cited on pages 20 and 29.)

[37] Stefano Ceri, Georg Gottlob, Letizia Tanca, et al. What you always wanted to know about Datalog (and never dared to ask). *IEEE transactions on knowledge and data engineering*, 1(1):146–166, 1989. (Cited on page 1.)

[38] Tanvi Chawla, Girdhari Singh, Emmanuel S. Pilli, and M.C. Govil. Storage, partitioning, indexing and retrieval in big RDF frameworks: A survey. *Computer Science Review*, 38:100309, 2020. ISSN 1574-0137. doi: https://doi.org/10.1016/j.cosrev.2020.100309. URL `https://www.sciencedirect.com/science/article/pii/S1574013720304093`. (Cited on pages 31 and 32.)

[39] Xiaojun Chen, Shengbin Jia, and Yang Xiang. A review: Knowledge reasoning over knowledge graph. *Expert Systems with Applications*, 141: 112948, 2020. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2019.112948. URL `https://www.sciencedirect.com/science/article/pii/S0957417419306669`. (Cited on page 2.)

[40] Yangjun Chen. General spanning trees and reachability query evaluation. In *Proceedings of the 2nd Canadian Conference on Computer Science and Software Engineering*, pages 243–252, 2009. (Cited on page 118.)

[41] Yangjun Chen and Yibin Chen. An efficient algorithm for answering graph reachability queries. In *2008 IEEE 24th International Conference on Data Engineering*, pages 893–902. IEEE, 2008. (Cited on pages 118 and 119.)

[42] Yangjun Chen and Gagandeep Singh. Graph indexing for efficient evaluation of label-constrained reachability queries. *ACM Transactions on Database Systems (TODS)*, 46(2):1–50, 2021. (Cited on page 118.)

[43] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003. (Cited on pages 118 and 119.)

[44] David J Tena Cucala, Przemysław A Wałęga, Bernardo Cuenca Grau, and Egor Kostylev. Stratified negation in Datalog with metric temporal operators. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 6488–6495, 2021. (Cited on page 124.)

[45] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys (CSUR)*, 33(3):374–425, 2001. (Cited on page 14.)

[46] Camil Demetrescu and Giuseppe F Italiano. Fully dynamic transitive closure: breaking through the o (n/sup 2/) barrier. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 381–389. IEEE, 2000. (Cited on page 118.)

[47] Hasanat M Dewan, David Ohsie, Salvatore J Stolfo, Ouri Wolfson, and Sushil Da Silva. Incremental database rule processing in paradiser. *Journal of Intelligent Information Systems*, 1(2):177–209, 1992. (Cited on page 28.)

[48] Guozhu Dong, Jianwen Su, and Rodney Topor. Nonrecursive incremental evaluation of Datalog queries. *Annals of Mathematics and Artificial Intelligence*, 14(2):187–223, 1995. (Cited on page 30.)

[49] Jason Eisner and Nathaniel W. Filardo. Dyna: Extending Datalog for modern ai. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, pages 181–220, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-24206-9. (Cited on page 1.)

[50] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems (TODS)*, 22(3):364–418, 1997. (Cited on page 124.)

[51] Wolfgang Faber, Gerald Pfeifer, and Nicola Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, 175(1):278–298, 2011. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2010.04.002. URL `https://www.sciencedirect.com/science/article/pii/S000437021000038X`. John McCarthy's Legacy. (Cited on page 124.)

[52] Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh M. Patel. Scaling-up in-memory Datalog processing: Observations

and techniques. *CoRR*, abs/1812.03975, 2018. URL `http://arxiv.org/abs/1812.03975`. (Cited on page 3.)

[53] Alessio Fiorentino, Jessica Zangari, and Marco Manna. DaRLing: A Datalog rewriter for OWL 2 RL ontological reasoning under SPARQL queries. *Theory and Practice of Logic Programming*, 20(6):958–973, 2020. (Cited on page 120.)

[54] Jörg Flum, Markus Frick, and Martin Grohe. Query evaluation via tree-decompositions. *Journal of the ACM (JACM)*, 49(6):716–752, 2002. (Cited on page 43.)

[55] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(82)90020-0. URL `https://www.sciencedirect.com/science/article/pii/0004370282900200`. (Cited on page 35.)

[56] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 international conference on management of data*, pages 1433–1445, 2018. (Cited on page 125.)

[57] Hector Garcia-Molina. *Database systems: the complete book*. Pearson Education India, 2008. (Cited on pages 14 and 57.)

[58] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski, Bowen, and Kenneth, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080. MIT Press, 1988. URL `http://www.cs.utexas.edu/users/ai-lab?gel88`. (Cited on page 124.)

[59] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991. (Cited on page 124.)

[60] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences*, 64(3): 579–627, 2002. (Cited on page 42.)

[61] Georg Gottlob, Giorgio Orsi, Andreas Pieris, and Mantas Šimkus. *Datalog and Its Extensions for Semantic Web Databases*, pages 54–77. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-33158-9. doi: 10.1007/978-3-642-33158-9_2. URL https://doi.org/10.1007/978-3-642-33158-9_2. (Cited on page 2.)

[62] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. Hypertree decompositions: Questions and answers. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 57–74, 2016. (Cited on pages 7, 39, 40, 43, and 63.)

[63] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. Datalog and recursive query processing. *Foundations and Trends® in Databases*, 5 (2):105–195, 2013. ISSN 1931-7883. doi: 10.1561/1900000017. URL http://dx.doi.org/10.1561/1900000017. (Cited on page 3.)

[64] Todd J Green, Shan Shan Huang, Boon Thau Loo, Wenchao Zhou, et al. Datalog and recursive query processing. *Foundations and Trends® in Databases*, 5(2):105–195, 2013. (Cited on page 1.)

[65] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2-3):158–182, 2005. (Cited on pages 59 and 113.)

[66] Ashish Gupta, Dinesh Katiyar, and Inderpal Singh Mumick. Counting solutions to the view maintenance problem. In *Workshop on deductive databases, JICSLP*, pages 185–194. Citeseer, 1992. (Cited on page 28.)

[67] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. Maintaining views incrementally. *ACM SIGMOD Record*, 22(2):157–166, 1993. (Cited on pages 6, 20, 28, and 29.)

[68] Jiawei Han, Ghassen Qadah, and Chinying Chaou. The processing and evaluation of transitive closure queries. In *International Conference on Extending Database Technology*, pages 49–75. Springer, 1988. (Cited on page 118.)

[69] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Faster fully dynamic transitive closure in practice. *arXiv preprint arXiv:2002.00813*, 2020. (Cited on page 118.)

[70] Stephen Harris and Nicholas Gibbins. 3store: Efficient bulk RDF storage. 2003. (Cited on page 31.)

[71] Steve Harris, Nick Lamb, Nigel Shadbolt, et al. 4store: The design and implementation of a clustered RDF store. In *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, volume 94, page 10, 2009. (Cited on page 25.)

[72] Andreas Harth and Stefan Decker. Optimized index structures for querying RDF from the web. In *Third Latin American Web Congress (LA-WEB'2005)*, pages 10–pp. IEEE, 2005. (Cited on page 33.)

[73] Ian Horrocks, Peter F Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, Mike Dean, et al. SWRL: A Semantic Web rule language combining OWL and RuleML. *W3C Member submission*, 21(79):1–31, 2004. (Cited on pages 2 and 23.)

[74] Pan Hu and Boris Motik. Accurate sampling-based cardinality estimation for complex graph queries. *ACM Transactions on Database Systems*, 49(3):1–46, 2024. (Cited on page 126.)

[75] Pan Hu, Boris Motik, and Ian Horrocks. Optimised maintenance of Datalog materialisations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018. (Cited on pages 30, 53, 104, 107, 133, and 139.)

[76] Pan Hu, Jacopo Urbani, Boris Motik, and Ian Horrocks. Datalog reasoning over compressed RDF knowledge bases. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 2065–2068, 2019. (Cited on page 34.)

[77] Pan Hu, Boris Motik, and Ian Horrocks. Modular materialisation of Datalog programs. *Artificial Intelligence*, 308:103726, 2022. (Cited on pages 30, 54, 55, 58, 78, 107, 113, 130, 154, 155, and 174.)

[78] Toshihide Ibaraki and Naoki Katoh. On-line computation of transitive closures of graphs. *Information Processing Letters*, 16(2):95–97, 1983. (Cited on page 118.)

[79] Yannis E Ioannidis. On the computation of the transitive closure of relational operators. In *VLDB*, volume 86, pages 25–28, 1986. (Cited on page 118.)

[80] Alex Ivliev, Stefan Ellmauthaler, Lukas Gerlach, Maximilian Marx, Matthias Meißner, Simon Meusel, and Markus Krötzsch. Nemo: First glimpse of a new rule engine. *Electronic Proceedings in Theoretical Computer Science*, 385: 333–335, September 2023. ISSN 2075-2180. doi: 10.4204/eptcs.385.35. URL `http://dx.doi.org/10.4204/EPTCS.385.35`. (Cited on pages 3 and 35.)

[81] Alex Ivliev, Lukas Gerlach, Simon Meusel, Jakob Steinberg, and Markus Krötzsch. Nemo: Your Friendly and Versatile Rule Reasoning Toolkit. In *Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning*, pages 743–754, 8 2024. doi: 10.24963/kr.2024/70. URL `https://doi.org/10.24963/kr.2024/70`. (Cited on page 3.)

[82] HV1093244 Jagadish. A compression technique to materialize transitive closure. *ACM Transactions on Database Systems (TODS)*, 15(4):558–598, 1990. (Cited on pages 118 and 119.)

[83] Michael Jakl, Reinhard Pichler, and Stefan Woltran. Answer-set programming with bounded treewidth. In *IJCAI*, volume 9, pages 816–822, 2009. (Cited on page 63.)

[84] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. Efficiently answering reachability queries on very large directed graphs. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 595–608, 2008. (Cited on pages 118 and 119.)

[85] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*, pages 422–430. Springer, 2016. (Cited on pages 3, 34, 35, and 36.)

[86] Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. A specialized B-tree for concurrent Datalog evaluation. In *Proceedings of the 24th symposium on principles and practice of parallel programming*, pages 327–339, 2019. (Cited on page 36.)

[87] Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. Specializing parallel data structures for Datalog. *Concurrency and Computation: Practice and Experience*, 34(2):e5643, 2022. (Cited on page 36.)

[88] Philippe Jégou and Samba Ndojh Ndiaye. On the notion of cycles in hypergraphs. *Discrete Mathematics*, 309(23):6535–6543, 2009. ISSN 0012-365X. doi: https://doi.org/10.1016/j.disc.2009.06.035. URL `https://www.sciencedirect.com/science/article/pii/S0012365X09003446`. (Cited on page 42.)

[89] David B. Kemp and Peter J. Stuckey. Semantics of logic programs with aggregates. pages 387–401, December 1991. International Logic Programming Symposium 1991, ILPS 1991 ; Conference date: 28-10-1991 Through 01-11-1991. (Cited on page 124.)

[90] Bas Ketsman, Paraschos Koutris, et al. Modern Datalog engines. *Foundations and Trends® in Databases*, 12(1):1–68, 2022. (Cited on pages 3, 31, and 35.)

[91] Valerie King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*, pages 81–89. IEEE, 1999. (Cited on page 118.)

[92] Markus Krötzsch. Efficient rule-based inferencing for OWL EL. In *IJCAI*, volume 11, pages 2668–2673, 2011. (Cited on page 125.)

[93] Johannes A La Poutré and Jan van Leeuwen. Maintenance of transitive closures and transitive reductions of graphs. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 106–120. Springer, 1987. (Cited on page 118.)

[94] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web*, 6(2):167–195, 2015. doi: 10.3233/SW-140134. URL `https://doi.org/10.3233/SW-140134`. (Cited on pages 66, 108, and 112.)

[95] Bjørnar Luteberget, Christian Johansen, and Martin Steffen. Rule-based consistency checking of railway infrastructure designs. In *International Conference on Integrated Formal Methods*, pages 491–507. Springer, 2016. (Cited on pages 1 and 14.)

[96] Xuedong Lyu, Xin Wang, Yuan-Fang Li, Zhiyong Feng, and Junhu Wang. GraSS: An efficient method for RDF subgraph matching. In *Web Information Systems Engineering–WISE 2015: 16th International Conference, Miami, FL, USA, November 1-3, 2015, Proceedings, Part I 16*, pages 108–122. Springer, 2015. (Cited on page 33.)

[97] David Maier. *The theory of relational databases*, volume 11. Computer science press Rockville, 1983. (Cited on page 42.)

[98] David Maier, K. Tuncay Tekle, Michael Kifer, and David S. Warren. *Datalog: concepts, history, and outlook*, page 3–100. Association for Computing Machinery and Morgan & Claypool, 2018. ISBN 9781970001990. URL `https://doi.org/10.1145/3191315.3191317`. (Cited on page 3.)

[99] Brian McBride. Jena: Implementing the RDF model and syntax specification. In *Semantic Web Workshop, WWW*, volume 2001. Citeseer, 2001. (Cited on pages 25 and 32.)

[100] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR*, 2013. (Cited on page 36.)

[101] Eric Miller. An introduction to the resource description framework. *D-lib Magazine*, 1998. (Cited on pages 2 and 22.)

[102] Michael Morak and Stefan Woltran. Preprocessing of Complex Non-Ground Rules in Answer Set Programming. In Agostino Dovier and Vítor Santos Costa, editors, *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12)*, volume 17 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 247–258, Dagstuhl, Germany, 2012. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-939897-43-9. doi: 10.4230/LIPIcs.ICLP.2012.247. URL `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ICLP.2012.247`. (Cited on page 64.)

[103] Boris Motik, Peter F Patel-Schneider, Bijan Parsia, Conrad Bock, Achille Fokoue, Peter Haase, Rinke Hoekstra, Ian Horrocks, Alan Ruttenberg, Uli Sattler, et al. OWL 2 Web Ontology Language: Structural specification and functional-style syntax. *W3C recommendation*, 27(65):159, 2009. (Cited on pages 2 and 23.)

[104] Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks, and Dan Olteanu. Parallel materialisation of Datalog programs in centralised, main-memory RDF systems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, 2014. (Cited on pages 25, 27, 34, 37, and 85.)

[105] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. Combining rewriting and incremental materialisation maintenance for Datalog programs with equality. *arXiv preprint arXiv:1505.00212*, 2015. (Cited on page 30.)

[106] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. Handling OWL: sameAs via rewriting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015. (Cited on pages 8, 30, and 37.)

[107] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. Incremental update of Datalog materialisation: the backward/forward algorithm. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015. (Cited on page 29.)

[108] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. Maintenance of Datalog materialisations revisited. *Artificial Intelligence*, 269:76–136, 2019. (Cited on pages 6, 19, 20, 28, 29, 67, and 126.)

[109] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. RDFox: A highly-scalable RDF store. In *International Semantic Web Conference*, pages 3–20. Springer, 2015. (Cited on pages 3, 4, 34, 36, and 37.)

[110] Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19:91–113, 2010. (Cited on page 33.)

[111] Hung Q Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: New developments in the theory of join algorithms. *ACM SIGMOD Record*, 42(4):5–16, 2014. (Cited on pages 6 and 39.)

[112] Jean-Marie Nicolas and Kioumars Yazdanian. An outline of BDGEN: A deductive DBMS. In *IFIP Congress*, pages 711–717, 1983. (Cited on page 28.)

[113] Yeonsu Park, Seongyun Ko, Sourav S Bhowmick, Kyoungmin Kim, Kijae Hong, and Wook-Shin Han. G-CARE: A framework for performance benchmarking of cardinality estimation techniques for subgraph matching. In *Proceedings of the*

2020 ACM SIGMOD International Conference on Management of Data, pages 1099–1114, 2020. (Cited on pages 59 and 144.)

[114] Sushant Patnaik and Neil Immerman. Dyn-FO: A parallel, dynamic complexity class. *Journal of Computer and System Sciences*, 55(2):199–209, 1997. ISSN 0022-0000. doi: https://doi.org/10.1006/jcss.1997.1520. URL `https://www.sciencedirect.com/science/article/pii/S0022000097915208`. (Cited on page 119.)

[115] Teodor C. Przymusinski. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5(2):167–205, 1989. doi: 10.1007/BF00243002. URL `https://doi.org/10.1007/BF00243002`. (Cited on page 124.)

[116] Haya Majid Qureshi and Wolfgang Faber. Meta-reasoning over OWL 2 QL using Datalog. In *Datalog*, pages 181–187, 2022. (Cited on page 120.)

[117] Raghu Ramakrishnan, Divesh Srivastava, and S Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1991. (Cited on page 27.)

[118] Raghu Ramakrishnan, Divesh Srivastava, S Sudarshan, and Praveen Seshadri. The CORAL deductive system. *The VLDB Journal*, 3:161–210, 1994. (Cited on page 1.)

[119] SQL RDF. Efficient RDF storage and retrieval in Jena2. 2003. (Cited on page 32.)

[120] Liam Roditty and Uri Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM Journal on Computing*, 37(5):1455–1471, 2008. (Cited on page 118.)

[121] Kenneth A Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. *Journal of Computer and System Sciences*, 54(1):79–97, 1997. ISSN 0022-0000. doi: https://doi.org/10.1006/jcss.1997.1453. URL `https://www.sciencedirect.com/science/article/pii/S0022000097914537`. (Cited on page 124.)

[122] Leonid Ryzhyk and Mihai Budiu. Differential Datalog. *Datalog*, 2:4–5, 2019. (Cited on pages 35 and 37.)

[123] Francesco Scarcello, Gianluigi Greco, and Nicola Leone. Weighted hypertree decompositions and optimal query plans. *Journal of Computer and System Sciences*, 73(3):475–506, 2007. (Cited on pages 42 and 57.)

[124] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On fast large-scale program analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 196–206, 2016. (Cited on page 3.)

[125] Jiwon Seo, Stephen Guo, and Monica S Lam. SociaLite: Datalog extensions for efficient social network analysis. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 278–289. IEEE, 2013. (Cited on pages 3, 14, and 36.)

[126] Ahmedur Rahman Shovon, Thomas Gilray, Kristopher Micinski, and Sidharth Kumar. Towards iterative relational algebra on the GPU. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 1009–1016, 2023. (Cited on page 35.)

[127] Shikha Singh, Sergey Madaminov, Michael A. Bender, Michael Ferdman, Ryan Johnson, Benjamin Moseley, Hung Ngo, Dung Nguyen, Soeren Olesen, Kurt Stirewalt, and Geoffrey Washburn. A scheduling approach to incremental maintenance of Datalog programs. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 864–873, 2020. doi: 10.1109/IPDPS47924.2020.00093. (Cited on pages 27, 28, 37, and 126.)

[128] Martin Staudt and Matthias Jarke. Incremental maintenance of externally materialized views. In *VLDB*, volume 96, pages 3–6. Citeseer, 1996. (Cited on pages 6, 20, and 29.)

[129] Giorgio Stefanoni, Boris Motik, and Ian Horrocks. Small Datalog query rewritings for EL. In *Proc. 25th Int'l Workshop on Description Logics*, 2012. (Cited on page 120.)

[130] Giorgio Stefanoni, Boris Motik, and Egor V Kostylev. Estimating the cardinality of conjunctive queries over RDF data using graph summarisation. In *Proceedings of the 2018 World Wide Web Conference*, pages 1043–1052, 2018. (Cited on pages iv, 59, 126, and 142.)

[131] Julien Subercaze, Christophe Gravier, Jules Chevalier, and Frederique Laforest. Inferray: fast in-memory RDF inference. In *VLDB*, volume 9, 2016. (Cited on page 30.)

[132] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: A large ontology from Wikipedia and WordNet. *Journal of Web Semantics*, 6(3):203–217, 2008. (Cited on page 59.)

[133] Yihao Sun, Ahmedur Rahman Shovon, Thomas Gilray, Kristopher K Micinski, and Sidharth Kumar. GDlog: A GPU-accelerated deductive engine. *CoRR*, 2023. (Cited on pages 35 and 36.)

[134] Yihao Sun, Sidharth Kumar, Thomas Gilray, and Kristopher Micinski. Column-oriented Datalog on the GPU. *arXiv preprint arXiv:2501.13051*, 2025. (Cited on pages 3, 28, 35, and 127.)

[135] Efthymia Tsamoura, David Carral, Enrico Malizia, and Jacopo Urbani. Materializing knowledge bases via trigger graphs. *Proc. VLDB Endow.*, 14(6): 943–956, February 2021. ISSN 2150-8097. doi: 10.14778/3447689.3447699. URL `https://doi.org/10.14778/3447689.3447699`. (Cited on pages 27 and 28.)

[136] Susan Tu and Christopher Ré. Duncecap: Query plans using generalized hypertree decompositions. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 2077–2078, 2015. (Cited on pages 7 and 40.)

[137] Jeffrey D Ullman. Implementation of logical query languages for databases. *ACM Transactions on Database Systems (TODS)*, 10(3):289–321, 1985. (Cited on page 3.)

[138] Jeffrey D Ullman. Bottom-up beats top-down for Datalog. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 140–149, 1989. (Cited on pages 3 and 4.)

[139] Jeffrey D Ullman. Principles of database and knowledge-base systems. *Computer Science Press*, 1989. (Cited on page 3.)

[140] Jeffry D Ullman. The database approach to knowledge representation. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, pages 1346–1349, 1996. (Cited on page 14.)

[141] Jacopo Urbani, Spyros Kotoulas, Jason Maassen, Frank Van Harmelen, and Henri Bal. WebPIE: A web-scale parallel inference engine using MapReduce. *Journal of Web Semantics*, 10:59–75, 2012. (Cited on page 4.)

[142] Jacopo Urbani, Ceriel Jacobs, and Markus Krötzsch. Column-oriented Datalog materialization for large knowledge graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016. (Cited on page 34.)

[143] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):619–649, July 1991. ISSN 0004-5411. doi: 10.1145/116825.116838. URL `https://doi.org/10.1145/116825.116838`. (Cited on page 124.)

[144] Frank Van Harmelen, Vladimir Lifschitz, and Bruce Porter. *Handbook of knowledge representation*. Elsevier, 2008. (Cited on page 14.)

[145] Sebastiaan J. van Schaik and Oege de Moor. A memory efficient reachability data structure through bit vector compression. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 913–924, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306614. doi: 10.1145/1989323.1989419. URL `https://doi.org/10.1145/1989323.1989419`. (Cited on page 119.)

[146] Todd L Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm. *arXiv preprint arXiv:1210.0481*, 2012. (Cited on page 38.)

[147] Denny Vrandečić and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014. (Cited on page 113.)

[148] Przemyslaw Andrzej Walega, B Cuenca Grau, Mark Kaminski, and Egor V Kostylev. DatalogMTL: Computational complexity and expressive power. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, 1886–92*. International Joint Conferences on Artificial Intelligence, 2019. (Cited on pages 65 and 124.)

[149] Qichen Wang, Bingnan Chen, Binyang Dai, Ke Yi, Feifei Li, and Liang Lin. Yannakakis+: Practical acyclic query evaluation with theoretical guarantees, 2025. URL `https://arxiv.org/abs/2504.03279`. (Cited on page 43.)

[150] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for Semantic Web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008. (Cited on page 33.)

[151] Kevin Wilkinson and Kevin Wilkinson. Jena property table implementation, 2006. (Cited on page 32.)

[152] Zhe Wu, George Eadon, Souripriya Das, Eugene Inseok Chong, Vladimir Kolovski, Melliyal Annamalai, and Jagannathan Srinivasan. Implementing an inference engine for RDFS/OWL constructs and user-defined rules in Oracle. In *2008 IEEE 24th International Conference on Data Engineering*, pages 1239–1248. IEEE, 2008. (Cited on pages 3 and 4.)

[153] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, volume 81, pages 82–94, 1981. (Cited on pages 40, 43, 64, and 135.)

[154] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale RDF data. *Proceedings of the VLDB Endowment*, 6(4):265–276, 2013. (Cited on page 33.)

[155] Chao Zhang, Angela Bonifati, Hugo Kapp, Vlad Ioan Haprian, and Jean-Pierre Lozi. A reachability index for recursive label-concatenated graph queries. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 67–81. IEEE, 2023. (Cited on page 118.)

[156] Xinyue Zhang, Pan Hu, Yavor Nenov, and Ian Horrocks. Enhancing Datalog reasoning with hypertree decompositions. In Edith Elkind, editor, *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI-23*, pages 3383–3393. International Joint Conferences on Artificial Intelligence Organization, 8 2023. doi: 10.24963/ijcai.2023/377. URL `https://doi.org/10.24963/ijcai.2023/377`. Main Track. (Cited on pages 13 and 39.)

[157] Xinyue Zhang, Pan Hu, Yavor Nenov, and Ian Horrocks. Optimised storage for Datalog reasoning. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'24/IAAI'24/EAAI'24. AAAI Press, 2024. ISBN 978-1-57735-887-9. doi: 10.1609/aaai.v38i9.28947. URL `https://doi.org/10.1609/aaai.v38i9.28947`. (Cited on pages 13 and 66.)

[158] Yujiao Zhou, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, and Jay Banerjee. Making the most of your triple store: query answering in OWL 2 using an RL reasoner. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1569–1580, 2013. (Cited on pages 59 and 113.)

[159] Lei Zou, Jinghui Mo, Lei Chen, M Tamer Özsu, and Dongyan Zhao. gStore: answering SPARQL queries via subgraph matching. *Proceedings of the VLDB Endowment*, 4(8):482–493, 2011. (Cited on page 33.)