21st International Conference on Database Theory

ICDT 2018, March 26–29, 2018, Vienna, Austria

Edited by

Benny Kimelfeld

Yael Amsterdamer

Editors

Benny Kimelfeld
Computer Science Department
Technion, Israel
bennyk@cs.technion.ac.il

Yael Amsterdamer
Department of Computer Science
Bar-Ilan University, Israel
first.last@biu.ac.il

ACM Classification 2012
Information systems → Data management systems, Information systems → Database design and models, Information systems → Database query processing, Information systems → Query languages, Information systems → Relational database model, Information systems → Parallel and distributed DBMSs, Theory of computation → Complexity classes, Theory of computation → Interactive computation, Theory of computation → Complexity theory and logic, Mathematics of computing → Graph theory, Computing methodologies → Knowledge representation and reasoning

ISBN 978-3-95977-063-7

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-95977-063-7.

Publication date
March, 2018

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/LIPIcs.ICDT.2018.0

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

http://www.dagstuhl.de/lipics

Contents

Invited Talks

Regular Papers

21st International Conference on Database Theory (ICDT 2018).
Editors: Benny Kimelfeld and Yael Amsterdamer

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

The 21th International Conference on Database Theory (ICDT 2018) was held in Vienna, Austria, on March 26-29, 2018. Originally biennial, the ICDT conference has been held annually and jointly with the conference on Extending Database Technology (EDBT) since 2009. The proceedings of ICDT 2018 includes an overview of the keynote talks by Virginia Vassilevska Williams (MIT EECS, MA, USA), by Ke Yi (Hong Kong University of Science and Technology, Hong Kong, China), and by Thomas Zeume (TU Dortmund University, Germany), as well as 17 research papers that were selected by the Program Committee.

Out of the 17 accepted papers, the Program Committee selected the paper *"Evaluation and Enumeration Problems for Regular Path Queries" by Wim Martens and Tina Trautner (University of Bayreuth, Germany)* for the ICDT 2018 Best Paper Award.

We wish to thank the many contributors to ICDT 2018: the authors of the submitted papers, the external reviewers, the keynote speakers, the EDBT/ICDT 2018 local organization officers, and the conference General Chair Reinhard Pichler. We wish to especially thank the members of the ICDT 2018 Program Committee, who invested considerable time and effort providing thorough paper reviews and conducting careful discussions. We also wish to thank ICDT Council Chair Wim Martens and ICDT 2017 Program Committee Chair Michael Benedikt for their guidance and support, as well as ICDT 2017 Proceedings Chair Giorgio Orsi for useful advice on the publicity and production of the proceedings. Last, we wish to acknowledge Michael Wagner and Marc Herbstritt from Dagstuhl Publishing for their support with the publication of the proceedings in the LIPIcs (Leibniz International Proceedings in Informatics) Series.

Benny Kimelfeld and Yael Amsterdamer
March 2018

21st International Conference on Database Theory (ICDT 2018).
Editors: Benny Kimelfeld and Yael Amsterdamer
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Organization

Conference Chair

Vim Martens (University of Bayreuth, Germany)

Program Chair

Benny Kimelfeld (Technion - Israel Institute of Technology, Israel)

Publication and Proceedings Chair

Yael Amsterdamer (Bar-Ilan University, Israel)

Program Committee Members

Antoine Amarilli (Institut Télécom; Télécom ParisTech; CNRS LTCI, France)

Yael Amsterdamer (Bar-Ilan University, Israel)

Pablo Barcelo (DCC, Universidad de Chile, Chile)

Keren Censor-Hillel (Technion - Israel Institute of Technology, Israel)

Claire David (Universite Paris-Est Marne-la-Vallee, France)

Martin Grohe (RWTH Aachen, Germany)

Aidan Hogan (DCC, Universidad de Chile, Chile)

Benny Kimelfeld (Technion - Israel Institute of Technology, Israel)

Roman Kontchakov (Birkbeck, University of London, UK)

Ilya Mironov (Google, US)

Filip Murlak (University of Warsaw, Poland)

Jelani Nelson (Harvard University, US)

Matthias Niewerth (University of Bayreuth, Germany)

Rasmus Pagh (IT University of Copenhagen, Denmark)

Sudeepa Roy (Duke University, US)

Dan Suciu (University of Washington, US)

Yufei Tao (Chinese University of Hong Kong, Hong Kong)

Jan Van den Bussche (Hasselt University; Transnational University of Limburg, Belgium)

Stijn Vansummeren (Université Libre de Bruxelles, Belgium)

Victor Vianu (UC San Diego, US)

David P. Woodruff (IBM Almaden, US)

21st International Conference on Database Theory (ICDT 2018).
Editors: Benny Kimelfeld and Yael Amsterdamer

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

External Reviewers

Foto Afrati

Arvind Arasu

Daniel Deutch

Alin Deutsch

Nadime Francis

Dominik D. Freydenberger

Herman Haverkort

Dylan Hutchison

Neil Immerman

Silvio Lattanzi

Yi Li

Leonid Libkin

Andrew McGregor

Hung Q. Ngo

Thomas Pellissier Tanon

Sylvain Schmitz

Luc Segoufin

Francesco Silvestri

Xiaorui Sun

Martin Ugarte

21st International Conference on Database Theory (ICDT 2018).
Editors: Benny Kimelfeld and Yael Amsterdamer
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Contributing Authors

Shqiponja Ahmetaj
Aws Albarghouthi
Antoine Amarilli
Pablo Barcelo
Christoph Berkholz
Elena Botoeva
Pierre Bourhis
Robert Brijder
Diego Calvanese
Nofar Carmeli
David Carral
Benjamin Cogrel
Graham Cormode
Jacques Dark
Johannes Doleschal
Floris Geerts
Jean Christoph Jung
Ahmet Kara
Jens Keppeler
Bas Ketsman
Paraschos Koutris
Markus Kröll
Markus Krötzsch
Carsten Lutz
Mauricio Martel
Wim Martens
Maximilian Marx
Stefan Mengel
Mikaël Monet
Frank Neven
Dan Olteanu
Magdalena Ortiz
Ana Ozaki
Miguel Romero
Sebastian Rudolph
Thomas Schneider
Nicole Schweikardt
Pierre Senellart
Mantas Simkus
Yufei Tao
Tina Trautner
Jan Van den Bussche
Brecht Vandevoort
Virginia Vassilevska Williams
Timmy Weerwag
Adam Witkowski
Guohui Xiao
Ke Yi
Thomas Zeume

21st International Conference on Database Theory (ICDT 2018).
Editors: Benny Kimelfeld and Yael Amsterdamer
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

ICDT 2018 Test of Time Award

In 2013, the International Conference on Database Theory (ICDT) began awarding the ICDT Test of Time (ToT) award, with the goal of recognizing one paper, or a small number of papers, presented at ICDT at least a decade earlier, that have best met the "test of time". The ICDT ToT award for 2018 was presented during the EDBT/ICDT 2018 Joint Conference, March 26–29, 2018 in Vienna, Austria. The ICDT 2018 ToT Award Committee was charged with selecting the paper(s) from the ICDT 1999 and 2001 proceedings that have had the most impact in terms of research, methodology, conceptual contribution, or transfer to practice over the past decade. After careful consideration, the committee selected the following papers as joint award winners for 2018:

Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, Uri Shaft:
When Is "Nearest Neighbor" Meaningful?

This much cited, thought-provoking paper challenged the conventional wisdom in high-dimensional similarity search by highlighting a surprising phenomenon: the distance to a point's nearest neighbor is almost the same as the distance to its farthest neighbor, in high dimensions. This opened up new fundamental questions about the utility of indexes for high-dimensional similarity search and pointed out a major weakness in the methodology of empirical evaluations of prior works.

Peter Buneman, Sanjeev Khanna, Wang Chiew Tan:
Why and Where: A Characterization of Data Provenance

A seminal contribution to the foundations of data provenance, this article considered for the first time the "where" and "why" flavors of provenance in a unified, application independent framework, using a general-purpose semi-structured data model. "Why" provenance refers to the portion of the database that justifies the presence of an item in the answer to a query, while "where" provenance identifies actual locations in the database from which the answer values were extracted. This influential and highly cited paper opened the way to an important research area focusing on various semantic and computational aspects of data provenance, still very active today.

Pablo Barcelo	Richard B. Hull	Victor Vianu
Univ. of Chile	IBM T.J. Watson Research Center	UC San Diego

The ICDT Test-of-Time Award Committee for 2018

21st International Conference on Database Theory (ICDT 2018).
Editors: Benny Kimelfeld and Yael Amsterdamer
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Fine-grained Algorithms and Complexity

Virginia Vassilevska Williams
MIT EECS, CSAIL, Cambridge, MA, USA
virgi@mit.edu

Abstract

A central goal of algorithmic research is to determine how fast computational problems can be solved in the worst case. Theorems from complexity theory state that there are problems that, on inputs of size n, can be solved in $t(n)$ time but not in $O(t(n)^{1-\varepsilon})$ time for $\varepsilon > 0$. The main challenge is to determine where in this hierarchy various natural and important problems lie. Throughout the years, many ingenious algorithmic techniques have been developed and applied to obtain blazingly fast algorithms for many problems. Nevertheless, for many other central problems, the best known running times are essentially those of their classical algorithms from the 1950s and 1960s.

Unconditional lower bounds seem very difficult to obtain, and so practically all known time lower bounds are conditional. For years, the main tool for proving hardness of computational problems have been NP-hardness reductions, basing hardness on P neq NP. However, when we care about the exact running time (as opposed to merely polynomial vs non-polynomial), NP-hardness is not applicable, especially if the problem is already solvable in polynomial time. In recent years, a new theory has been developed, based on "fine-grained reductions" that focus on exact running times. Mimicking NP-hardness, the approach is to (1) select a key problem X that is conjectured to require essentially $t(n)$ time for some t, and (2) reduce X in a fine-grained way to many important problems. This approach has led to the discovery of many meaningful relationships between problems, and even sometimes to equivalence classes.

The main key problems used to base hardness on have been: the 3SUM problem, the CNF-SAT problem (based on the Strong Exponential Time Hypothesis (SETH)) and the All Pairs Shortest Paths Problem. Research on SETH-based lower bounds has flourished in particular in recent years showing that the classical algorithms are optimal for problems such as Approximate Diameter, Edit Distance, Frechet Distance, Longest Common Subsequence etc.

In this talk I will give an overview of the current progress in this area of study, and will highlight some exciting new developments and their relationship to database theory.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness, Theory of computation → Design and analysis of algorithms

Keywords and phrases algorithms, complexity, fine-grained

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.1

Category Invited Talk

Join Algorithms: From External Memory to the BSP

Ke Yi
Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong, China
yike@ust.hk

Abstract

Database systems have been traditionally disk-based, which had motivated the extensive study on external memory (EM) algorithms. However, as RAMs continue to get larger and cheaper, modern distributed data systems are increasingly adopting a main memory based, shared-nothing architecture, exemplified by systems like Spark and Flink. These systems can be abstracted by the BSP model (with variants like the MPC model and the MapReduce model), and there has been a strong revived interest in designing BSP algorithms for handling large amounts of data.

With hard disks starting to fade away from the picture, EM algorithms may now seem less relevant. However, we observe that many of the recently developed join algorithms under the BSP model have a high degree of resemblance with their counterparts in the EM model. In this talk, I will present some recent results on join algorithms in the EM and BSP model, examine their relationships, and discuss a general theoretical framework for converting EM algorithms to the BSP.

2012 ACM Subject Classification Theory of computation → Data structures and algorithms for data management

Keywords and phrases External memory model, BSP, join algorithms

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.2

Category Invited Talk

21st International Conference on Database Theory (ICDT 2018).
Editors: Benny Kimelfeld and Yael Amsterdamer; Article No. 2; pp. 2:1–2:1

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

An Update on Dynamic Complexity Theory

Thomas Zeume
TU Dortmund University, Germany
thomas.zeume@cs.tu-dortmund.de

Abstract

In many modern data management scenarios, data is subject to frequent changes. In order to avoid costly re-computing query answers from scratch after each small update, one can try to use auxiliary relations that have been computed before. Of course, the auxiliary relations need to be updated dynamically whenever the data changes.

Dynamic complexity theory studies which queries and auxiliary relations can be updated in a highly parallel fashion, that is, by constant-depth circuits or, equivalently, by first-order formulas or the relational algebra. After gently introducing dynamic complexity theory, I will discuss recent results of the area with a focus on the dynamic complexity of the reachability query.

2012 ACM Subject Classification Theory of computation → Models of computation, Theory of computation → Finite Model Theory

Keywords and phrases Dynamic descriptive complexity, SQL updates, Reachability

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.3

Category Invited Talk

Funding Part of the work reported on in this talk was supported by DFG grant SCHW 678/6-2.

21st International Conference on Database Theory (ICDT 2018).
Editors: Benny Kimelfeld and Yael Amsterdamer; Article No. 3; pp. 3:1–3:1

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Rewriting Guarded Existential Rules into Small Datalog Programs

Shqiponja Ahmetaj
TU Wien, Vienna, Austria

Magdalena Ortiz
TU Wien, Vienna, Austria

Mantas Šimkus
TU Wien, Vienna, Austria

Abstract

The goal of this paper is to understand the relative expressiveness of the query language in which queries are specified by a set of guarded (disjunctive) tuple-generating dependencies (TGDs) and an output (or 'answer') predicate. Our main result is to show that every such query can be translated into a (disjunctive) Datalog program, which has polynomial size if the maximal number of variables in the (disjunctive) TGDs is bounded by a constant. To overcome the challenge that Datalog has no direct means to express the existential quantification present in TGDs, we define a two-player game that characterizes the satisfaction of the dependencies, and design a Datalog query that can decide the existence of a winning strategy for the game. For guarded disjunctive TGDs, we can obtain Datalog rules with disjunction in the heads. However, the use of disjunction is limited, and the resulting rules fall into a fragment that can be evaluated in deterministic single exponential time. We proceed quite differently for the case when the TGDs are not disjunctive and we show that we can obtain a plain Datalog query. Notably, unlike previous translations for related fragments, our translation requires only *polynomial time* under the reasonable assumption that the maximal number of variables in the (disjunctive) TGDs is bounded by a constant.

2012 ACM Subject Classification Theory of computation → Database query languages (principles), Theory of computation → Logic and databases, Theory of computation → Incomplete, inconsistent, and uncertain databases

Keywords and phrases Existential rules, Expressiveness, Query Rewriting

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.4

Acknowledgements This work was supported by the Austrian Science Fund (FWF) projects P30360, P30873, and W1255.

1 Introduction

This paper contributes to the understanding of the relative expressiveness and succinctness of *tuple-generating dependencies (TGDs)* [6] and their extension with disjunction *(DTGDs)* [11] as query languages for possibly incomplete data. TGDs and DTGDs are families of existential rules that play a crucial role as constraint languages for databases, especially in areas like data exchange and data integration. In recent years, they have also gained popularity in knowledge representation, where they are used as languages for writing *ontologies*, which can enrich possibly incomplete extensional data with intensional domain knowledge that can be leveraged at query time to obtain more complete and accurate answers.

When (D)TGDs are viewed as a query language for incomplete data, a *query* takes the form $Q = (\Sigma, q)$, where Σ is a set of DTGDs and q is a predicate. Q can be seen as a close

relative of the ontology-mediated queries (OMQs) considered in the literature on *Description Logics (DLs)* [25, 26, 1]. Given a database instance D, such a query asks to retrieve all tuples of values that are present in the relation q, over all databases that contain D and satisfy the dependencies in Σ. The corresponding query answering problem is only decidable if suitable restrictions are imposed on Σ. *Guardedness* is one of the best known such restrictions. Inspired by modal and first order logic [2], it yields a computationally robust family of TGDs whose complexity is quite well understood [9].

The problem that we study is the *rewritability* into Datalog. That is, given such a query $Q = (\Sigma, q)$, we want to build from it a Datalog query (P_Q, q_Q) that has the same certain answers over every input database instance over the signature of Σ. From the theoretical perspective, this problem is important: the existence of such a rewriting in the general case, and its size, allow us to understand the relative expressiveness and succinctness of guarded (D)TGDs as a query language. It also has practical relevance. Rewriting into standard query languages is considered one the most promising approaches to achieve scalability of expressive query languages, by reusing existing optimized database technologies. In the field of *ontology mediated querying*, for instance, rewritings have been a major research line for most of the last decade. Even 'impracticable' rewritings developed only for theoretical purposes can lay the groundwork for practical rewriting techniques, and shed light on their limits.

Our central result is a novel translation of such queries into 'small' Datalog programs. Our translation is non-trivial because guarded DTGDs allow for existential quantification in rule heads, while disjunctive Datalog (Datalog$^\vee$) does not support it. To overcome this challenge, the paper employs a game-theoretic characterization of answers to guarded DTGD queries. In particular, we tie the inclusion of a tuple in a query answer to the existence of a winning strategy in a two-player game. The Datalog$^\vee$ query that results from the translation aims to compute the answer to the input query by actually checking the existence of such winning strategies.

The translation takes polynomial time in the size of Q, if we assume that the guarded DTGDs only use a bounded number of variables. In particular, since our dependencies are guarded, this applies to theories where the arities of predicates are bounded by a constant, and no arbitrary conjunctions of atoms occur in the consequent of the dependencies. We stress that this case, although apparently restrictive, is very relevant, since in practice the arities of predicates are often not high, especially in settings that are tailored to deal with incomplete information (e.g., standard DLs use at most binary predicate symbols). If the number of variables per rule are not bounded, the translation is exponential in the size of Q, which is unavoidable under common assumptions in complexity theory (see Section 6). From the translation and the combined complexity of Datalog$^\vee$ queries, we can easily infer a coNExpTime upper bound for answering guarded DTGD queries under the assumption of bounded number of variables. However, we can do better: by analyzing the shape of the Datalog$^\vee$ queries resulting from our translation, we conclude that the program can be evaluated in deterministic exponential time in its size, which yields a new worst-case optimal algorithm for answering guarded DTGD queries with bounded number of variables.

Finally, we show that when the dependencies in the query are expressed using non-disjunctive guarded TGDs, we can provide a data-independent translation into plain Datalog queries. Again, the translation takes only polynomial time, assuming bounded number of variables. We note that translations into plain (resp., disjunctive) Datalog from guarded TGDs (resp., guarded DTGDs) do exist in the literature (see Related Work), but their techniques are quite different from ours, and more importantly, they are all exponential even under bounded number of variables.

Related Work. In the database community, there is a considerable amount of works on query rewritings for variants of guarded TGDs. For instance, it is known that when q is a (union of) conjunctive queries (rather than a predicate), (Σ, q) can be rewritten into a plain Datalog program, for Σ a set of guarded TGDs or more general classes of dependencies [3, 18]. For guarded *disjunctive* TGDs and q a union of conjunctive queries, a disjunctive Datalog rewriting can be found in [7]. The rewritings in [3, 18, 7] take *exponential time*, even if the number of variables in each TGD is bounded by a constant. In [16], the authors provide a translation into non-recursive Datalog for guarded TGDs, which is polynomial if the size of the schema of the input theory is bounded, which is a significantly stronger restriction that bounding the arity only. Moreover, this rewriting adopts a so-called *combined approach* that modifies the data incorporating inferences from the TGDs. The same authors proposed in [17] a rewriting into non-recursive Datalog queries of polynomial size for linear existential rules, without bounding the schema, or the arity. We remark that, for the case of bounded arity, this result follows from [19, 14]. The polynomial rewritings into non-recursive Datalog in [19, 17, 14] assume that the data contains some fixed number of constants. Our rewritings also use a few constants, which are supported by the Datalog variants we employ as target query languages.

In the context of DLs, rewritability into traditional query languages of queries enriched with an ontology is considered one of the most central questions. For instance, the *DL-Lite* family of DLs is popular because they often support rewritability into *first-order (FO) queries* [10]. In [23], the authors introduced the *combined approach* as a means to obtain rewritings into FO queries for more expressive languages like $\mathcal{EL}$. The authors of [19] proposed a rewriting into non-recursive Datalog queries of polynomial size for the prominent *DL-Lite*$_{\mathcal{R}}$. The succinctness problem of this and related FO rewritings has been thoroughly investigated in [14]. For more expressive DLs and *(unions of) conjunctive queries (UCQs)*, which are often not FO-rewritable, there is a considerable amount of work on rewritings into Datalog queries. The authors in [21] first showed that instance queries in an expressive DL with disjunction can be rewritten into in a disjunctive Datalog a program of exponential size. For variants of expressive DLs with disjunction and (U)CQs the existence of exponential rewritings into disjunctive Datalog is known [7, 22]. The authors of [27] propose a polynomial time Datalog translation of instance queries for an expressive *Horn-DL* without disjunction. Some of the above rewritings lie at the core of implemented systems, e.g., [28, 13, 29].

This paper is inspired by the technique in [1], where it was shown that instance queries mediated by ontologies in the expressive Description Logic $\mathcal{ALCHIO}$ can be rewritten in polynomial time into a Datalog$^{\vee}$ program. Guarded DTGDs can be seen as an ontology language that is orthogonal to $\mathcal{ALCHIO}$, which only supports unary and binary predicates and has a rather restricted syntax, yet it features *nominals* (essentially, constants). The guarded DTGDs considered here allow for predicates of arbitrary arities, but we disallow constants. In general, the higher arities and the rather relaxed syntax of guarded DTGDs makes the adaptation of the results in [1] highly non-trivial.

2 Preliminaries

General Technical Definitions. Let Δ_c, Δ_n and Δ_v be infinite mutually disjoint sets of *constants*, *labeled nulls*, and *variables*, respectively. Elements in $\Delta_c \cup \Delta_n \cup \Delta_v$ are *terms*. If no confusion arises, we may abuse notation and write a tuple of terms in the place of the set of its elements. An *atom* α is an expression of the form $R(t_1, \ldots, t_n)$, where R is a

predicate name with *arity* n, and $t_1, \ldots, t_n$ are terms. We let $\mathsf{terms}(\alpha) = \{t_1, \ldots, t_n\}$ and $\mathsf{vars}(\alpha) = \mathsf{terms}(\alpha) \cap \Delta_v$. If $\mathsf{terms}(\alpha) \subseteq \Delta_c$, then α is *ground*. For a set Γ of atoms, we let $\mathsf{terms}(\Gamma) = \bigcup_{\alpha \in \Gamma} \mathsf{terms}(\alpha)$ and $\mathsf{vars}(\Gamma) = \mathsf{terms}(\Gamma) \cap \Delta_v$. An instance I is a (possibly infinite) set of atoms with terms from $\Delta_n \cup \Delta_c$. A *database* D is a finite instance with only ground atoms. We denote with $\mathsf{dom}(I)$ the set of terms that occur in I. A *substitution* is a partial function h from a set of symbols S to a set of symbols S'. We let $h(\vec{t}) = (h(t_1), \ldots, h(t_n))$ for a tuple $\vec{t}$, $h(R(\vec{t})) = R(h(\vec{t}))$ for an atom $R(\vec{t})$, and $h(\Gamma) = \{h(R_1(\vec{t}), \ldots, h(R_n(\vec{t}))\}$ for a set of atoms $\Gamma = \{R_1(\vec{t}), \ldots, R_n(\vec{t})\}$.

Disjunctive Tuple-Generating Dependencies. A *disjunctive tuple-generating dependency (DTGD)* (a.k.a. *disjunctive existential rule*) σ is an expression of the form

$$\forall \vec{x}(\varphi(\vec{x}) \rightarrow \bigvee_{i=1}^{n} \exists \vec{y}_i.\psi_i(\vec{x}, \vec{y}_i)), \tag{1}$$

where $n \geq 1$, $\vec{x}, \vec{y}_1, \ldots, \vec{y}_n \subseteq \Delta_v$, and $\varphi, \psi_1 \ldots \psi_n$, are conjunctions of atoms with terms from Δ_v only. If $n = 1$, then σ is simply called a *tuple-generating dependency (TGD)* (a.k.a. *existential rule*). For simplicity, we use a comma for conjoining atoms and we omit the universal quantifiers in front of DTGDs. An instance I *satisfies* σ, denoted $I \models \sigma$, if for every substitution h from the variables in $\vec{x}$ to $\mathsf{dom}(I)$ such that $h(\varphi) \subseteq I$ there exists an $i \in [n]$ and a substitution h' from the variables in $\mathsf{vars}(\psi_i)$ to $\mathsf{dom}(I)$ such that $h'(\psi_i) \in I$ and $h'(x) = h(x)$ for all $x \in \vec{x}$. We say σ is *guarded* if there exists an atom α in φ, called a *guard*, such that $\vec{x} \subseteq \mathsf{vars}(\alpha)$. We refer the reader to [9] and [8] for more details on guarded (D)TGDs.

A set Σ of (D)TGDs is called a *theory*. The *schema* of Σ, denoted by $\mathsf{sch}(\Sigma)$, is the set of predicate names that appear in Σ, and for a set of atoms Γ, $\mathsf{sch}(\Gamma)$, is the set of all predicates that occur in Γ. A theory is *guarded* if all its (D)TGDs are guarded. An instance I *satisfies* a theory Σ, written $I \models \Sigma$, if $I \models \sigma$ for each $\sigma \in \Sigma$. For theories Σ, Σ', we say Σ entails Σ', written $\Sigma \models \Sigma'$, if for all instances I, $I \models \Sigma$ implies $I \models \Sigma'$.

For a database D, we write $I \models D$ if $D \subseteq I$. Given a database D and a theory Σ, a *model* of (Σ, D) is an instance I, denoted $I \models (\Sigma, D)$, such that $I \models D$ and $I \models \Sigma$. Given a ground atom α, we say α is *entailed* by Σ and D, written $(\Sigma, D) \models \alpha$, if $\alpha \in I$ for every model I of (Σ, D).

Queries. A *query* is a pair (Σ, q), where Σ is a theory and q is a predicate symbol. Given a query (Σ, q) and a database D, we let

$$\mathsf{ans}(\Sigma, q, D) = \{\vec{c} \in (\Delta_c)^n \mid (\Sigma, D) \models q(\vec{c})\},$$

where n is the arity of q. We call $\mathsf{ans}(\Sigma, q, D)$ the *(certain) answer* to (Σ, q) over D.

Normal Form. To simplify some technical constructions, we focus on guarded DTGDs with a restricted syntactic structure:

▶ **Definition 1.** (Normalized DTGDs) A theory Σ of DTGDs is in *normal form* if each $\sigma \in \Sigma$ is of one of the following forms:

$$B \rightarrow \exists \vec{y}.H \qquad \text{where } B, H \text{ are atoms with terms from } \Delta_v \tag{2}$$

$$\varphi \rightarrow \bigvee_{i=1}^{n} H_i \qquad \text{where each } H_i \text{ is an atom and } \varphi \text{ is a set of atoms over terms in } \Delta_v \tag{3}$$

In other words, a normalized theory Σ of DTGDs consists of a set of TGDs with one atom in the body and one atom in the head, and a set of guarded DTGDs without existentially quantified variables. By means of fresh predicate names, a theory of guarded DTGDs can be converted into the above normal form while preserving atom entailment (see, e.g., [18]). We state this more precisely next.

▶ **Proposition 2.** *Every query (Σ, q) can be transformed in polynomial time into a query (Σ', q) such that*

(a) $\mathsf{ans}(\Sigma, q, D) = \mathsf{ans}(\Sigma', q, D)$ *for every database D over* $\mathsf{sch}(\Sigma)$*;*

(b) Σ' *is in normal form;*

(c) *if Σ is disjunctive guarded, then Σ' is disjunctive guarded;*

(d) *if Σ has only guarded (non-disjunctive) TGDs, then Σ' also has only TGDs.*

Proof sketch. Assume a query (Σ, q). The idea is to replace every DTGD of the form (1) that appears in Σ by

- the DTGD $\forall\vec{x}(\varphi(\vec{x}) \rightarrow \bigvee_{i=1}^{n} F_i(\vec{x}_i))$, were $\vec{x}_i$ is a list of variables from $\vec{x}$ that appear in $\psi_i(\vec{x}, \vec{y}_i)$, and each F_i is a fresh predicate symbol with arity $|\vec{x}_i|$,
- for all $1 \leq i \leq n$, the TGD $\forall\vec{x}_i(F_i(\vec{x}_i) \rightarrow \exists\vec{y}_i.F_i'(\vec{x}_i, \vec{y}_i))$, where F_i' is a fresh predicate name of arity $|\vec{x}_i| + |\vec{y}_i|$, and
- for all $1 \leq i \leq n$ and each atom H in $\psi_i(\vec{x}, \vec{y}_i)$, the TGD $\forall\vec{x}_i\forall\vec{y}_i(F_i'(\vec{x}_i, \vec{y}_i) \rightarrow H)$.

The points *(b)*, *(c)*, and *(d)* follow easily. The point *(a)* holds because (Σ', D) is a *conservative extension* of (Σ, D) for any database D over $\mathsf{sch}(\Sigma)$. In particular, (i) any model of (Σ', D) is a model of (Σ, D), and (ii) every model of (Σ, D) that is over $\mathsf{sch}(\Sigma)$ can be extended to a model of (Σ', D) by properly populating the predicates F_i, F_i' introduced during the normalization process. ◀

In the rest of the paper, we consider only normalized guarded theories Σ. We let $\Sigma_\exists$ and $\Sigma_\forall$ denote the guarded DTGDs of the form (2) and of the form (3) that appear in Σ, respectively. The *width* of Σ, written *width*(Σ), is the maximal arity over the predicate in $\mathsf{sch}(\Sigma)$.

In the discussion below, and in particular in the complexity upper bounds, we often refer to (normalized) theories with *bounded width* (or *bounded predicate arity*), in which *width*(Σ) is bounded by a constant. We remark that the normalization process increases the predicate arities (the F_i and F_i' may exceed the width of the original theory). The width of a normalized theory, however, is bounded whenever there is only a bounded number of variables in each DTGD. It is also bounded if the original theory (before normalization) has bounded width, and in the consequent of DTGDs, $\psi_i(\vec{x}, \vec{y}_i)$ is a single atom rather than a conjunction of atoms. Therefore, the results below that refer to normalized theories of bounded width apply to these relevant cases.

Datalog with Disjunction (Datalog$^\vee$). A *rule* ρ is an expression of the form $H_1 \vee \ldots \vee H_n \leftarrow B_1, \ldots, B_k$, where $H_1, \ldots, H_n, B_1, \ldots, B_k$ are atoms with terms from $\Delta_v \cup \Delta_c$. The atoms $H_1, \ldots, H_n$ are called *head* atoms, and $B_1, \ldots, B_k$ are called *body* atoms. We require that each variable that appears in ρ also occurs in a body atom. A rule with no body atoms of the form $H \leftarrow$ is called a *fact*. A rule ρ with no head atoms of the form $\leftarrow B_1, \ldots, B_k$ is a *constraint*. A finite set P of rules is called a *program*. If every rule in a program P has at most one head atom, then P is called a (plain) Datalog program. The *grounding* $\mathsf{ground}(P)$ of a program P is the ground (i.e., variable-free) program that is obtained from P by replacing each rule ρ of P by its ground instances, i.e., rules that can be obtained from ρ by substituting its variables with constants of P.

A database D is a *model* of a program P, if $\{B_1, \ldots, B_k\} \subseteq D$ implies $D \cap \{H_1, \ldots, H_n\} \neq \emptyset$ for all rules $H_1 \vee \ldots \vee H_n \leftarrow B_1, \ldots, B_k$ in $\mathsf{ground}(P)$. A DATALOG$^\vee$ *query* is a pair (P, q), where P is a program, and q is a predicate symbol from P. We let $\mathsf{ans}(P, q, D)$ denote the set of all n-tuples $\vec{c}$ of values from Δ_c, where n is the arity of q, such that $q(\vec{c}) \in D'$ for all models D' of $P \cup \{\alpha \leftarrow \mid \alpha \in D\}$. If P is a plain DATALOG program, then (P, q) is a plain DATALOG query.

3 Counter Models

Assume Σ is a guarded theory of DTGDs. We want to decide whether $(\Sigma, D) \not\models \alpha$ for a given database D and a ground atom α. That is, we want to decide the existence of a model I of Σ and D such that $\alpha \notin I$. Rather than aiming at constructing such a (possibly infinite) I, we proceed as follows:

(1) We search for a 'small' part of such a possible I, which we call a *core instance* D_c for (Σ, D). Intuitively, core instances are databases that extend D to ensure the satisfaction of $\Sigma_\forall$ (but no nulls are added, and the satisfaction of $\Sigma_\exists$ is not guaranteed). They fix how the constants of D participate in all predicates, while ensuring that α is false.

(2) For each candidate core instance D_c, we verify if it can be extended to also satisfy $\Sigma_\exists$, while preserving satisfaction of $\Sigma_\forall$. When extending D_c, entailment of ground atoms is preserved, and hence so are the satisfaction of D and the non-entailment of α.

Core instances and their extensions are defined next:

▶ **Definition 3.** (Core instances) A *core instance* for (Σ, D) is a database D_c with predicates from $\mathsf{sch}(\Sigma)$ such that:

(c1) $\mathsf{dom}(D) = \mathsf{dom}(D_c)$, and

(c2) $D_c \models (\Sigma_\forall, D)$

An instance I is called an *extension* of D_c, if D_c is the result of restricting I to $\mathsf{dom}(D)$.

A core and its extensions coincide on the ground atoms they entail over $\mathsf{dom}(D)$. Hence, for a given query (Σ, q) and a tuple $\vec{c}$, deciding that $\vec{c} \notin \mathsf{ans}(\Sigma, q, D)$ amounts to deciding whether there is a core instance that does not entail $q(\vec{c})$, and that can be extended into a model of (Σ, D). Defining a disjunctive DATALOG program whose models are the core instances described above is not hard. The second part, that is, verifying if a core can be extended to a full model, is more challenging. In fact, it corresponds to testing satisfiability of a database (in this case, a candidate D_c) w.r.t. a theory Σ of guarded DTGDs, which is known to be EXPTIME-complete when predicate arities are bounded, and 2EXPTIME-complete otherwise [20, 15].

In order to obtain a set of rules that solves this problem (and that has polynomial size if *width*(Σ) is bounded), we characterize it as a game, revealing a simple algorithm that admits an elegant implementation in DATALOG$^\vee$. The game relies on *types* which we define next.

▶ **Definition 4.** (Types) For a theory Σ, we assume an order over some special variables $\mathbf{x}_1, \ldots, \mathbf{x}_w$, such that $w = \mathit{width}(\Sigma)$. A *type* τ over a theory Σ is a set of atoms over $\mathsf{sch}(\Sigma)$ such that $\mathsf{terms}(\tau) \subseteq \{\mathbf{x}_1, \ldots, \mathbf{x}_w\}$. We denote by *types*$(\Sigma)$ the set of all types over Σ.

Given an instance I and a tuple $\vec{c} = (c_1, \ldots, c_k)$ with $k \leq w$ and $\vec{c} \subseteq \mathsf{dom}(I)$, we let the type $\mathit{type}(\vec{c}, I) = \{R(\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_j}) \mid R(c_{i_1}, \ldots, c_{i_j}) \in I, \{c_{i_1}, \ldots, c_{i_j}\} \subseteq \vec{c}\}$. A type τ is *realized* in I if there is some tuple $\vec{c} \subseteq \mathsf{dom}(I)$ such that $\mathit{type}(\vec{c}, I) = \tau$.

For a given Σ, the number of possible types that one can construct is bounded by $2^{n(w^w)}$, where n is the number of predicates occurring in $\mathsf{sch}(\Sigma)$, and w is the *width*(Σ); if w is

bounded, there are only exponentially many types. In the rest of the paper, we write $\mathbf{X}$ to denote the set of special variables $\{\mathbf{x}_1, \ldots, \mathbf{x}_w\}$ and $\vec{\mathbf{x}}$ to denote the tuple $(\mathbf{x}_1, \ldots, \mathbf{x}_w)$.

▶ **Definition 5.** Let $\tau \in \mathit{types}(\Sigma)$ be a type over the special variables $\mathbf{X}$ and let $\mathbf{Y} \subseteq \mathbf{X}$. We denote by $\tau|_{\mathbf{Y}} \subseteq \tau$ the type that contains each atom $R(\vec{t}) \in \tau$ with $\vec{t} \subseteq \mathbf{Y}$.

We now describe a game to decide whether a given core D_c can be extended into a model of (Σ, D). The game is played by Bob (the builder), who wants to extend D_c into a model, and Sam (the spoiler), who wants to spoil all of Bob's attempts. Sam starts by picking a tuple $\vec{c}$ such that $R(\vec{c}) \in D_c$. They look at its type $\mathit{type}(\vec{c}, D_c)$ and if it does not satisfy the DTGDs in $\Sigma_\forall$, Sam is declared the winner. Otherwise, in each turn Sam chooses a TGD $\sigma \in \Sigma_\exists$ of the form $B \to \exists \vec{y}.H$ and a substitution h such that $h(B)$ is satisfied by the current type, forcing Bob to pick a type that satisfies H and that coincides with the current type on the shared variables of B and H. The game continues for as long as Bob can respond to the challenges of Sam.

Formally, for a theory Σ with $\Sigma = \Sigma_\exists \cup \Sigma_\forall$ and an instance I, we define the set $\mathsf{LC}(\Sigma, I)$ of *locally consistent types* as the set that contains each type $\tau \in \mathit{types}(\Sigma)$ such that the following condition holds.

(LC$_\forall$) For all DTGDs $\sigma \in \Sigma_\forall$ of the form $\varphi \to \bigvee_{i=1}^{n} H_i$, and for all substitutions h from $\mathsf{vars}(\varphi)$ to $\mathbf{X}$, $h(\varphi) \subseteq \tau$ implies that there exists $i \in [n]$ such that $h(H_i) \in \tau$.

The game on an instance I and a theory Σ starts by Sam choosing a tuple $\vec{c}$ such that $R(\vec{c}) \in I$ and $\tau = \mathit{type}(\vec{c}, I)$ is set to be the *current type*. Then:

(♦) If $\tau \notin \mathsf{LC}(\Sigma, I)$ then Sam is declared winner.
Otherwise, Sam chooses a TGD $\sigma \in \Sigma_\exists$ of the form $B \to \exists \vec{y}.H$ and a substitution $h : \mathsf{vars}(B) \to \mathbf{X}$ such that $h(B) \in \tau$; if there is no such TGD in Σ, Bob is declared the winner. Otherwise, let $\mathbf{Z} = \mathsf{vars}(B) \cap \mathsf{vars}(H)$. Bob has to choose a type τ' such that the following conditions hold:

(C1) $\tau|_{h(\mathbf{Z})} = \tau'|_{h(\mathbf{Z})}$,
(C2) there exists a substitution $h' : \mathsf{vars}(H) \to \mathbf{X}$ with $h(\mathbf{Z}) = h'(\mathbf{Z})$ such that $h'(H) \in \tau'$.

The type τ' is set to be the current type, and the game continues with a new round, i.e. we go back to ♦.

A *run* of the game on an instance I is a (possibly infinite) sequence $\vec{c}\sigma_1 h_1 \tau_1 \sigma_2 h_2 \tau_2 \ldots$ where $\vec{c}$ is a tuple initially picked by Sam such that some atom $R(\vec{c}) \in I$, and each σ_i, h_i and τ_i are the TGD and the substitution picked by Sam and the type picked by Bob in round i, respectively. A *strategy for Bob*, to play on I and Σ, is a partial function *str* that maps a type τ, a TGD $\sigma \in \Sigma_\exists$ of the form $B \to \exists \vec{y}.H$, and a substitution $h : \mathsf{vars}(B) \to \mathbf{X}$ with $h(B) \in \tau$ to a type τ' that satisfies (C1) and (C2); intuitively, the strategy gives a move for Bob in response to the moves of Sam. A run $\vec{c}\sigma_1 h_1 \tau_1 \sigma_2 h_2 \tau_2 \ldots$ with $\mathit{type}(\vec{c}, I) = \tau_0$ *follows a strategy str* if $\tau_i = \mathit{str}(\tau_{i-1}, \sigma_i, h_i)$ for every $i \geq 1$.

For a finite run r, we let $\mathit{tail}(r) = \mathit{type}(\vec{c}, I)$ if $r = \vec{c}$, and $\mathit{tail}(r) = \tau_\ell$ if $r = \vec{c} \ldots \sigma_\ell h_\ell \tau_\ell$ with $\ell \geq 1$. The strategy *str* is called *non-losing* on I if for every finite run r that follows *str*:

(i) $\mathit{tail}(r) \in \mathsf{LC}(\Sigma, I)$, and
(ii) $\mathit{str}(\mathit{tail}(r), \sigma, h_\sigma)$ is defined for every $\sigma \in \Sigma_\exists$ of the form $B \to \exists \vec{y}.H$ and every substitution $h_\sigma : \mathsf{vars}(B) \to \mathbf{X}$ with $h_\sigma(B) \in \mathit{tail}(r)$.

The correctness of the game is shown in the following theorem.

▶ **Theorem 6.** *Let Σ be a theory of guarded DTGDs, D a database, and α a ground atom. Then $(\Sigma, D) \not\models \alpha$ iff there is a core instance D_c for (Σ, D) such that:*
(1) *$\alpha \notin D_c$, and*
(2) *there is a non-losing strategy for Bob on D_c.*

Proof sketch. We focus on showing that for any given core D_c, there is a non-losing strategy *str* for Bob on D_c if and only if D_c can be extended into an instance I that satisfies (Σ, D).

For "$\Leftarrow$", assume an arbitrary model I of (Σ, D) that is an extension of D_c. We extract from I a non-losing strategy *str* for Bob as follows. First, let $rlz(I)$ be the set of all types realized in I. Observe that $rlz(I) \subseteq \mathsf{LC}(\Sigma, D_c)$ holds since the core D_c must satisfy all the rules in $\Sigma_\forall$. It suffices to set, for each type $\tau \in rlz(I)$ and each $\sigma \in \Sigma_\exists$ of the form $B \to \exists \vec{y}.H$ with $h(B) \in \tau$, $str(\tau, \sigma, h) = \tau'$ for an arbitrarily chosen type $\tau' \in rlz(I)$ that satisfies (C1) and (C2) which exists because I is a model, thus it satisfies all the DTGDs in Σ.

For the "$\Rightarrow$" direction, from an arbitrary non-losing *str* for D_c, we build I as follows. We write $\vec{c}_\tau$ to denote a tuple of constants realizing τ; $rn(D_c, str)$ to denote the set of all finite runs $\vec{c}_\tau \sigma_1 h_1 \tau_1 \sigma_2 h_2 \tau_2 \dots$ that follow *str*; and $fv(\sigma)$ to denote the set $\mathsf{vars}(B) \cap \mathsf{vars}(H)$ of variables where $\sigma \in \Sigma$ is a TGD of the form $B \to \exists \vec{y}.H$. We let I be the set of atoms $R(t_1, \dots, t_\ell)$ such that one of the following holds:
(a) there exists a run $r = \vec{c}_\tau$ in $rn(D_c, str)$ with $R(\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_\ell}) \in \tau$ and $t_j = c_{i_j} \in \vec{c_\tau}$ for each $j \in [1, \ell]$ or
(b) there exists a run $r = \vec{c}_\tau \dots \sigma_i h_i \tau_i$ in $rn(D_c, str)$, where $i \geq 1$ and σ_i is of the form $B \to \exists \vec{y}.H$ such that $R(\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_\ell}) \in \tau_i$ and for each t_j with $j \in [1, \ell]$ the following holds:

(1) t_j is the labelled null $r'\mathbf{x}_{i_j}$, if
  (i) $r' = \vec{c}_\tau \dots \sigma_k h_k \tau_k$ and $r = r' \dots \sigma_i h_i \tau_i$ with $1 \leq k \leq i$,
  (ii) $\mathbf{x}_{i_j} \notin h_k(fv(\sigma_k))$, and
  (iii) $\mathbf{x}_{i_j} \in h_l(fv(\sigma_l))$ for each $k < l \leq i$ if $i \neq k$.
(2) t_j is the constant c_{i_j} if $\mathbf{x}_{i_j} \in h_l(fv(\sigma_l))$ for each $l \in [1, i]$.

This I satisfies (Σ, D) and it is an extension of D_c. For more details, see Appendix A. ◀

To decide whether Bob has a non-losing strategy on a given core, we use the type elimination procedure **Mark** presented in Algorithm 1. Intuitively, the algorithm *marks* all types from which Bob does not have a non-losing strategy. It takes as input a theory Σ, and an instance I which intuitively is the core being checked. The algorithm builds the set of all types, and then it starts marking the types that are not a winning choice for Bob. First, in step ($\mathrm{M}_\forall$), the algorithm marks the types that are not in $\mathsf{LC}(\Sigma, I)$. Then it iterates using ($\mathrm{M}_\exists$) to exhaustively mark types τ that allow Sam to pick a TGD for which Bob cannot reply with any type τ'.

The formal relationship between the game and the marking algorithm is established next.

▶ **Theorem 7.** *Let D_c be a core instance for (Σ, D). Then Bob has a non-losing strategy on D_c iff none of the types realized in D_c is marked by* **Mark**(Σ).

Proof sketch. For the "$\Rightarrow$" direction, we can show that if a type τ is marked, then it cannot occur in a non-losing *str* for Bob. The proof is by induction in the number of iterations that the algorithm **Mark**(Σ) requires to mark τ. For the "$\Leftarrow$" direction, a non-losing *str* for D_c is obtained by first taking all unmarked types $\tau \in types(\Sigma)$ (which are all contained in $\mathsf{LC}(\Sigma, I)$,

Algorithm 1: Mark.

input : theory Σ of guarded DTGDs

output : Set of (possibly) marked types

$N \leftarrow \{\tau \mid \tau \in \mathit{types}(\Sigma)\}$

($\mathrm{M}_\forall$) Mark each $\tau \in N$ such that there exists:
- $\sigma \in \Sigma_\forall$ of the form $\varphi \rightarrow \bigvee_{i=1}^{n} H_i$, and
- h from $\mathsf{vars}(\varphi)$ to $\mathbf{X}$, s.t. $h(\varphi) \subseteq \tau$, and $h(H_i) \notin \tau$ for each $i \in [n]$.

repeat

($\mathrm{M}_\exists$) Mark $\tau \in N$ if there exists a TGD $B \rightarrow \exists \vec{y}.H \in \Sigma_\exists$ and a substitution $h : \mathsf{vars}(B) \rightarrow \mathbf{X}$ with $h(B) \in \tau$ s.t. for each $\tau' \in N$, at least one of the following holds:

($\mathrm{C0}$) τ' is marked,
($\mathrm{C1}'$) $\tau|_{h(\mathbf{Z})} \neq \tau'|_{h(\mathbf{Z})}$, or
($\mathrm{C2}'$) $h'(H) \notin \tau'$ for each $h' : \mathsf{vars}(H) \rightarrow \mathbf{X}$ with $h(\mathbf{Z}) = h'(\mathbf{Z})$

where $\mathbf{Z} = \mathsf{vars}(B) \cap \mathsf{vars}(H)$.

until *no new type is marked*

return N

as otherwise they would be marked by ($\mathrm{M}_\forall$). Then, for each unmarked type τ, each TGD $\sigma \in \Sigma_\exists$ of the form $B \rightarrow \exists \vec{y}.H$ and each substitution $h : \mathsf{vars}(B) \rightarrow \mathbf{X}$ with $h(B) \in \tau$, we set $\mathit{str}(\tau, \sigma, h) = \tau'$ for an arbitrarily chosen unmarked type τ' that satisfies (C1) and (C2). ◀

4 Translation into Datalog$^\vee$

The goal of this section is to build, from a given query (Σ, q), a DATALOG$^\vee$ program P such that (Σ, q) and (P, q) have the same answers for all databases D over $\mathsf{sch}(\Sigma)$. Moreover, the size of P is polynomial in Σ whenever $\mathit{width}(\Sigma)$ is bounded. P has three major components:

(a) rules that non-deterministically generate a core instance D_c for (Σ, D), where D is an input database;

(b) rules that implement the marking algorithm presented in the previous section;

(c) rules that 'glue' (a) and (b), by ensuring that all types that occur in D_c are not marked.

We emphasize that the construction of P depends exclusively on (Σ, q), and is independent from a particular database D. In what follows, we let $w = \mathit{width}(\Sigma)$, and let n be the number of distinct predicate symbols occurring in $\mathsf{sch}(\Sigma)$.

(I) Collecting the constants. We first add rules to collect in the unary predicate const all the constants that occur in the input database. For each $R \in \mathsf{sch}(\Sigma)$ with arity ℓ, and for each $1 \leq i \leq \ell$ we add the rule:

$$\mathsf{const}(u_i) \leftarrow R(u_1, \ldots, u_\ell)$$

For each $1 < i \leq w$ we have an i-ary version of const that stores i-ary tuples of constants:

$$\mathsf{const}^i(u_1, \ldots, u_i) \leftarrow \mathsf{const}(u_1), \ldots, \mathsf{const}(u_i)$$

(II) Generating core instances. We use a fresh predicate $\overline{R}$ for each $R \in \mathsf{sch}(\Sigma)$, and add the following rules to P for each ℓ-ary predicate $R \in \mathsf{sch}(\Sigma)$, where $\vec{u}$ is an ℓ-ary tuple of variables:

$$R(\vec{u}) \vee \overline{R}(\vec{u}) \leftarrow \mathsf{const}^{\ell}(\vec{u}) \qquad\qquad \leftarrow R(\vec{u}), \overline{R}(\vec{u})$$

To ensure (c2) in Definition 3, for each $\sigma \in \Sigma_{\forall}$ of the form $\varphi \rightarrow \bigvee_{i=1}^{n} H_i$ we add the rule $\bigvee_{i=1}^{n} H_i \leftarrow \varphi$. For any input D, the models of the above rules (when restricted to the predicates in $\mathsf{sch}(\Sigma)$) are the core instances D_c of (Σ, D).

Towards checking whether D_c can be extended to satisfy all rules in Σ, we implement the algorithm **Mark** from Section 3. We assume a fixed, arbitrary enumeration $\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_m$ of all the atoms that can be constructed over the predicate symbols in $\mathsf{sch}(\Sigma)$ and the special variables in $\mathbf{X}$. The set of all these atoms $\{\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_m\}$ is denoted by $\mathbf{A}$. Note that $1 \leq m \leq n(w^w)$, hence if w is bounded by a constant, then $|\mathbf{A}|$ is polynomially bounded.

We assume also a pair $0, 1$ of special constants. Intuitively, we use an m-ary predicate symbol $\mathsf{Type} = \{0,1\}^m$ to store all types in $\mathit{types}(\Sigma)$. For instance, a tuple $(r_1, \ldots, r_m) \in \mathsf{Type}$, encodes the type $\tau = \{\boldsymbol{\alpha}_i \mid r_i = 1, 1 \leq i \leq m\}$. This relation is the basis to compute another m-ary relation $\mathsf{Marked} \subseteq \{0,1\}^m$ that contains precisely the types marked by the **Mark** algorithm. We next define the rules to compute Type and Marked, and other relevant relations.

(III) A linear order over types. The first ingredient is a linear order over all possible types. Of course, we could hardcode the list of exponentially many types in the Datalog program, but then the program would not be of polynomial size, and, therefore, we need to compute it instead. To this end, for every $1 \leq i \leq m$, we inductively define i-ary relations first^i and last^i, and a $2i$-ary relation next^i, which provide the first, the last, and the successor elements from a linear order on $\{0,1\}^i$. In particular, given $\vec{u}, \vec{v} \in \{0,1\}^i$, the fact $\mathsf{next}^i(\vec{u}, \vec{v})$ will be true if $\vec{v}$ follows $\vec{u}$ in the ordering of $\{0,1\}^i$. The rules to populate next^i are quite standard (see, e.g., Theorem 4.5 in [5]). For the case $i = 1$, we simply add the following facts:

$$\mathsf{first}^1(0) \leftarrow \qquad\qquad \mathsf{last}^1(1) \leftarrow \qquad\qquad \mathsf{next}^1(0,1) \leftarrow$$

Then, for all $1 < i \leq m - 1$ we add the following rules:

$$\begin{aligned}
\mathsf{next}^{i+1}(0, \vec{u}, 0, \vec{v}) &\leftarrow \mathsf{next}^i(\vec{u}, \vec{v}) \qquad & \mathsf{first}^{i+1}(0, \vec{u}) &\leftarrow \mathsf{first}^i(\vec{u}) \\
\mathsf{next}^{i+1}(1, \vec{u}, 1, \vec{v}) &\leftarrow \mathsf{next}^i(\vec{u}, \vec{v}) & \mathsf{last}^{i+1}(1, \vec{u}) &\leftarrow \mathsf{last}^i(\vec{u}) \\
\mathsf{next}^{i+1}(0, \vec{u}, 1, \vec{v}) &\leftarrow \mathsf{last}^i(\vec{u}), \mathsf{first}^i(\vec{v}) &&
\end{aligned}$$

We can now collect in the relation Type all types in $\mathit{types}(\Sigma)$:

$$\mathsf{Type}(\vec{u}) \leftarrow \mathsf{first}^m(\vec{u}) \qquad\qquad \mathsf{Type}(\vec{v}) \leftarrow \mathsf{next}^m(\vec{u}, \vec{v})$$

(IV) Implementing Step ($\mathrm{M}_{\forall}$). First, we add the auxiliary facts $\mathsf{F}(0) \leftarrow$ and $\mathsf{T}(1) \leftarrow$ to P. For an m-tuple of variables $\vec{u}$ and an atom $\boldsymbol{\alpha}_j$ in the enumeration $\boldsymbol{\alpha}_1, \ldots \boldsymbol{\alpha}_m$, we use $\boldsymbol{\alpha}_j \in \vec{u}$ to denote the atom $\mathsf{T}(u_j)$, and $\boldsymbol{\alpha}_j \notin \vec{u}$ to denote the atom $\mathsf{F}(u_j)$. Then the step ($\mathrm{M}_{\forall}$), which marks types violating some rule of $\Sigma_{\forall}$, is implemented using the following rule for every DTGD $\varphi \rightarrow \bigvee_{i=1}^{n} H_i \in \Sigma_{\forall}$. Assume φ is a set of atoms $B_1, \ldots B_l$. Then, for all substitutions $h : \mathsf{vars}(\varphi) \rightarrow \mathbf{X}$, add the rule:

$$\mathsf{Marked}(\vec{u}) \leftarrow \mathsf{Type}(\vec{u}), h(B_1) \in \vec{u}, \ldots, h(B_l) \in \vec{u}, h(H_1) \notin \vec{u}, \ldots, h(H_n) \notin \vec{u}$$

There are at most w^k possible substitutions, where k is the arity of the guard atom in φ.

(V) Implementing Step ($\mathrm{M}_\exists$). The following rules are added for all TGDs $\sigma = B \to \exists \vec{y}.H \in \Sigma_\exists$ and for all substitutions $h : \mathsf{vars}(B) \to \mathbf{X}$. Assume $\mathbf{Z} = \mathsf{vars}(B) \cap \mathsf{vars}(H)$. Recall that we need to mark a type τ if there exists a substitution $h : \mathsf{vars}(B) \to \mathbf{X}$ such that $h(B) \in \tau$, and *for each* type τ', either τ' is marked or at least one of (C1′) or (C2′) holds. First, for each TGD σ and substitution h, we use an auxiliary $2m$-ary relation $\mathsf{MarkedOne}_{\sigma,h}$ to collect all such types τ'.

For collecting each τ' that satisfies condition (C0), we add:

$$\mathsf{MarkedOne}_{\sigma,h}(\vec{u},\vec{v}) \leftarrow \mathsf{Type}(\vec{u}), \mathsf{Marked}(\vec{v})$$

For (C1′) we add a rule for each $R(\vec{x})$, such that $\vec{x} \subseteq h(\mathbf{Z})$:

$$\begin{aligned} \mathsf{MarkedOne}_{\sigma,h}(\vec{u},\vec{v}) &\leftarrow \mathsf{Type}(\vec{u}), \mathsf{Type}(\vec{v}), R(\vec{x}) \in \vec{u}, R(\vec{x}) \notin \vec{v} \\ \mathsf{MarkedOne}_{\sigma,h}(\vec{u},\vec{v}) &\leftarrow \mathsf{Type}(\vec{u}), \mathsf{Type}(\vec{v}), R(\vec{x}) \notin \vec{u}, R(\vec{x}) \in \vec{v} \end{aligned}$$

To implement condition (C2′), we collect each τ' such that $h'_i(H) \notin \tau$ for each substitution $h'_i : \mathsf{vars}(H) \to \mathbf{X}$ with $h(\mathbf{Z}) = h'_i(\mathbf{Z})$. Notice that for k variables in $\vec{y}$, there are w^k possible substitutions, that is $1 \leq i \leq w^k$. We add:

$$\mathsf{MarkedOne}_{\sigma,h}(\vec{u},\vec{v}) \leftarrow \mathsf{Type}(\vec{u}), \mathsf{Type}(\vec{v}), h'_1(H) \notin \vec{v}, \ldots, h'_{w^k}(H) \notin \vec{v}$$

Intuitively, we want to infer $\mathsf{Marked}(\vec{u})$ if $h(\mathsf{B})$ is set to true in $\vec{u}$ and $\mathsf{MarkedOne}_{\sigma,h}(\vec{u},\vec{v})$ is true for all types (bit vectors) $\vec{v}$. To this aim, we need another $2k$-ary relation $\mathsf{MarkedUntil}_{\sigma,h}$. We add:

$$\begin{aligned} \mathsf{MarkedUntil}_{\sigma,h}(\vec{u},\vec{t}) &\leftarrow \mathsf{MarkedOne}_{\sigma,h}(\vec{u},\vec{t}), \mathsf{first}^m(\vec{t}) \\ \mathsf{MarkedUntil}_{\sigma,h}(\vec{u},\vec{v}) &\leftarrow \mathsf{MarkedUntil}_{\sigma,h}(\vec{u},\vec{t}), \mathsf{next}^m(\vec{t},\vec{v}), \mathsf{MarkedOne}_{\sigma,h}(\vec{u},\vec{v}) \end{aligned}$$

Intuitively, with the above rules we traverse all types checking the conditions (C0), (C1′) and (C2′) described in ($\mathrm{M}_\exists$). If we manage to reach the last type, and if $h(\mathsf{B}) \in \vec{u}$, we know the condition is satisfied. We add the following rule:

$$\mathsf{Marked}(\vec{u}) \leftarrow \mathsf{MarkedUntil}_{\sigma,h}(\vec{u},\vec{t}), h(\mathsf{B}) \in \vec{u}, \mathsf{last}^m(\vec{t})$$

(VI) Forbidding marked types in the core. Finally, we need to forbid each tuple of constants in the generated core database from having a type from Marked. For all $0 \leq j \leq m$ we take a fresh $(j+w)$-ary relation symbol Proj^j. We first add:

$$\mathsf{Proj}^m(\vec{\mathbf{x}},\vec{u}) \leftarrow \mathsf{const}^w(\vec{\mathbf{x}}), \mathsf{Marked}(\vec{u})$$

Intuitively, we collect in $\mathsf{const}^w(\vec{\mathbf{x}})$ all the possible w-ary tuples of constants from the core database. We now project away bits from the Proj^j relations by looking at the actual types of tuples of constants. We add the following rules for all $1 \leq j \leq m$:

$$\mathsf{Proj}^{j-1}(\vec{\mathbf{x}},\vec{u}) \leftarrow \mathsf{Proj}^j(\vec{\mathbf{x}},\vec{u},1), \boldsymbol{\alpha}_j \qquad \mathsf{Proj}^{j-1}(\vec{\mathbf{x}},\vec{u}) \leftarrow \mathsf{Proj}^j(\vec{\mathbf{x}},\vec{u},0), \overline{\boldsymbol{\alpha}_j}$$

Finally, we need to rule out situations when the extracted type of a tuple of constants over the core database is marked by adding the constraint $\leftarrow \mathsf{Proj}^0(\vec{\mathbf{x}})$.

This completes the polynomial translation of a query (Σ, q) where Σ is a set of guarded DTGDs to a Datalog$^\vee$ query (P, q). Next, we provide a theorem that captures the correctness of the above translation.

▶ **Theorem 8.** *For a query (Σ, q), where Σ is a normalized theory of guarded DTGDs, we can build a query (P, q), where P is a set of disjunctive* Datalog *rules, such that $\mathsf{ans}(\Sigma, q, D) = \mathsf{ans}(P, q, D)$ for any given database D over $\mathsf{sch}(\Sigma)$. The construction requires only polynomial time whenever width(Σ) is bounded.*

From the complexity of Datalog$^\vee$ [12], we infer a coNExpTime upper bound in combined complexity for guarded DTGDs whose normalized form uses only predicates whose arities are bounded by a constant. However, we use disjunction in a limited way: only in the rules in (II), whose heads have only two atoms over predicates of bounded arity. The resulting program falls into a class of programs that can be evaluated in (deterministic) exponential time, see Appendix A for more details. This matches the ExpTime-completeness of satisfiability of guarded DTGDs with bounded predicate arities [20].

▶ **Proposition 9.** *Deciding $\vec{c} \in \mathsf{ans}(P, q, D)$, where P is the disjunctive* Datalog *program obtained from the above translation of a set Σ of guarded DTGDs, can be done in* ExpTime.

5 Non-Disjunctive TGDs

Now we consider the case of plain guarded TGDs, that have no disjunction. Recall that in normalized theories of TGDs, each σ takes the form $B \to \exists \vec{y}.H$ or $\varphi \to H$, where B, H are atoms and φ is a set of atoms, all with terms from Δ_v.

In this section, we provide a rewriting of queries given by a theory of TGDs into plain Datalog. That is, given a query of the form (Σ, q), where Σ is a theory of TGDs, we obtain (P, q), where P is a (non-disjunctive) plain Datalog program. The resulting program is of polynomial size whenever *width*(Σ) is bounded, unlike previous rewritings that exist in the literature for related query languages.

5.1 From Marked Types to Horn Rules

To obtain a non-disjunctive rewriting for TGDs, we leverage to a great extent the results of the previous section, but proceed somewhat differently. We start by observing that if in the input (Σ, q), the TGDs in Σ contain no disjunction, almost all rules of the program described above are in plain Datalog. However, the rules in step **(II)** are disjunctive, and this is a major issue. Indeed, the basic strategy that we described in Section 3, namely, to obtain a core and then check if it can be suitably extended, is intrinsically based on the ability to use disjunction to generate different core instances. Therefore, to obtain a program without disjunction, we must proceed differently. Our strategy is to use the marked types to obtain a new set of non-disjunctive TGDs Σ_{Horn}, which correctly capture the consequences of Σ. This TGDs are *full*, that is, they have no existentially quantified variables, and they can be easily realized by Datalog rules.

Towards obtaining Σ_{Horn}, let $\mathsf{Marked}(\Sigma)$ denote the set of all the types that are marked by Algorithm 1 executed on Σ. We show that (independently of whether Σ contains or not disjunctive heads), $\mathsf{Marked}(\Sigma)$ defines a set of full DTGDs Σ_M that completely captures the consequences of Σ. Therefore, we can replace Σ by Σ_M, while preserving the entailment of facts. Later we show that Σ_M can in turn be replaced by the desired Σ_{Horn}.

To define Σ_M, recall that $\mathbf{A}$ is the set of all atoms that can occur in *types*(Σ). Each type τ defines a full DTGD *rule*(τ) as follows:

$$\tau \to \bigvee_{\boldsymbol{\alpha} \in \mathbf{A} \setminus \tau} \boldsymbol{\alpha}$$

Consider a core instance D_c, which we would like to extend to satisfy Σ. Intuitively, if $\tau \in \mathsf{Marked}(\Sigma)$, such a rule expresses that if all the atoms in τ are true in D_c, then some atom not in τ must be true as well, as otherwise the marked type τ would be realized.

Formally, we can show the following. We remark that we formulate the lemma for TGDs, but it holds also for DTGDs.

▶ **Lemma 10.** *Let Σ be a set of guarded TGDs and let Σ_M be the set of all rule(τ) with $\tau \in \mathsf{Marked}(\Sigma)$. Then, for any given atom $\boldsymbol{\alpha}$ and database D, $(D, \Sigma) \not\models \boldsymbol{\alpha}$ iff $(D, \Sigma_M) \not\models \boldsymbol{\alpha}$.*

Proof. For the "⇒" direction, let I be an instance such that $I \models (D, \Sigma)$ and $\boldsymbol{\alpha} \notin I$. We show that $I \models (D, \Sigma_M)$. Let D_c be the restriction of I to constants in $\mathsf{dom}(D)$, that is D_c is a core instance for (D, Σ). The latter together with Theorem 6 imply that Bob has a non-losing strategy on D_c, which together with Theorem 7 imply that none of the types realized in D_c is marked by $\mathbf{Mark}(\Sigma)$. We claim that $D_c \models (D, \Sigma_M)$. Towards a contradiction assume there there exists a *rule*(τ) $\in \Sigma_M$ such that $D_c \not\models \tau \rightarrow \bigvee_{\boldsymbol{\alpha} \in \mathbf{A} \setminus \tau} \boldsymbol{\alpha}$. That D_c doesn't satisfy $\tau \rightarrow \bigvee_{\boldsymbol{\alpha} \in \mathbf{A} \setminus \tau} \boldsymbol{\alpha}$ intuitively means that one can match τ to D_c, but cannot match any of $\boldsymbol{\alpha}$ to D_c. The latter imply that τ must be realized in D_c, but this contradicts that $\tau \in \mathsf{Marked}(\Sigma)$ by definition of *rule*(τ).

For the "⇐" direction, let I be an instance such that $I \models (D, \Sigma_M)$ and $\boldsymbol{\alpha} \notin I$. Recall that no rule in Σ_M enforces the invention of nulls, that is w.l.o.g. we assume $\mathsf{terms}(I) = \mathsf{dom}(D)$. One could easily show that I is a core instance for (D, Σ). Observe that none of the types realized in I are marked by $\mathbf{Mark}(\Sigma)$ as otherwise, by definition of $\mathsf{Marked}(\Sigma)$, the corresponding type would appear in the body of a rule in Σ_M which together with $I \models \Sigma_M$ contradict the assumption that the corresponding type is realized in I. The latter and Theorem 7 imply Bob has a non-losing strategy on I which together with assumption $\boldsymbol{\alpha} \notin I$, and by Theorem 6 imply $(D, \Sigma) \not\models \boldsymbol{\alpha}$. ◀

In general, Σ_M is a set of full DTGDs, but if Σ is not disjunctive, then we can obtain a set of non-disjunctive full TGDs that achieves the same effect. The core intuition is that, for TGDs, the disjunctive heads express choices between atoms that are *irrelevant*, and one can disregard such choices, obtaining full TGDs that enforce only the necessary atoms. To identify the necessary and the irrelevant atoms, we rely on what we call *abstract types*, which are types augmented with a set τ_{ir} of irrelevant atoms.

▶ **Definition 11.** An *abstract type* (over Σ) is a pair (τ_c, τ_{ir}) of (possibly empty) disjoint subsets of $\mathbf{A}$.

Intuitively, an atom $\boldsymbol{\alpha}$ is irrelevant for τ_c, with $\boldsymbol{\alpha} \notin \tau_c$, if both τ_c and $\tau_c \cup \{\boldsymbol{\alpha}\}$ are marked. The algorithm **Horn** takes as input the set of all abstract types $(\tau, \emptyset)$ where $\tau \in \mathsf{Marked}(\Sigma)$, and outputs a set of abstract types. The pair (τ_c, τ_{ir}) will be returned by the algorithm if $\tau = \tau_c \cup \tau'_{ir}$ is marked for each $\tau'_{ir} \subseteq \tau_{ir}$, that is, if the presence of the atoms in τ_{ir} is irrelevant for the marking of τ_c. We use $\mathsf{Horn}(\Sigma)$ to denote the result of running Algorithm 2 on $\mathsf{Marked}(\Sigma)$ for a given theory Σ.

▶ **Lemma 12.** *Let (τ_c, τ_{ir}) be an abstract type. Then $(\tau_c, \tau_{ir}) \in \mathsf{Horn}(\Sigma)$ if and only if $\tau_c \cup \tau'_{ir} \in \mathsf{Marked}(\Sigma)$ for each $\tau'_{ir} \subseteq \tau_{ir}$.*

If $(\tau_c, \tau_{it}) \in \mathsf{Horn}(\Sigma)$, then the irrelevant atoms in τ_{ir} can be omitted from the head of *rule*(τ_c), as shown next.

▶ **Lemma 13.** *Let (τ_c, τ_{ir}) be an abstract type. Then $(\tau_c, \tau_{ir}) \in \mathsf{Horn}(\Sigma)$ if and only if $\Sigma \models \tau_c \rightarrow \bigvee_{\boldsymbol{\alpha} \in \mathbf{A} \setminus (\tau_c \cup \tau_{ir})} \boldsymbol{\alpha}$.*

Algorithm 2: Horn.

input : a set M_0 of types

output : set of abstract types

$N \leftarrow \{(\tau_c, \emptyset) \mid \tau_c \in M_0\}$

repeat

  if *there exist* $\{(\tau_c, \tau_{ir}), (\tau'_c, \tau_{ir})\} \subseteq N$ *with* $\tau_c = \tau'_c \cup \{\boldsymbol{\alpha}\}$ *for some atom* $\boldsymbol{\alpha} \notin \tau'_c$ **then**

    add the pair $(\tau'_c, \tau_{ir} \cup \{\boldsymbol{\alpha}\})$ to N

until *no further changes*

return N

These rules are still disjunctive, however, we can show that for every marked type, there exists a corresponding abstract type that covers all atoms over the signature of the theory, except for one, and hence it induces a full TGD with only one atom in the head. The proof of the next lemma uses the well-known property that (non-disjunctive) TGDs are convex, that is, $\Sigma \models \bigwedge_{i=1}^{k} \boldsymbol{\alpha}_i \rightarrow \bigvee_{j=1}^{n} \boldsymbol{\alpha}_j$ implies that there exists $l \in \{1, \ldots, n\}$ such that $\Sigma \models \bigwedge_{i=1}^{k} \boldsymbol{\alpha}_i \rightarrow \boldsymbol{\alpha}_l$ (see Lemma 2 in [3]).

▶ **Lemma 14.** *Let Σ be a set of TGDs, $M = \mathsf{Mark}(\Sigma)$, and let $N = \mathsf{Horn}(\Sigma)$. Then $\tau_c \in M$ implies that there exists a pair $(\tau_c, \tau_{ir}) \in N$ such that $\tau_c \cup \tau_{ir} \cup \{\boldsymbol{\alpha}\} = \mathbf{A}$ for some $\boldsymbol{\alpha} \in \mathbf{A}$.*

Now we state the central result of this section: from $\mathsf{Horn}(\Sigma)$ one can construct the desired set of full TGDs Σ_{Horn}, which has the same atomic consequences as the original Σ.

▶ **Theorem 15.** *Let Σ be a set of TGDs, and let Σ_{Horn} be the set of all full TGDs $\tau_c \rightarrow \boldsymbol{\alpha}$ such that there is an abstract type (τ_c, τ_{ir}) in $\mathsf{Horn}(\Sigma)$ with $\tau_c \cup \tau_{ir} \cup \{\boldsymbol{\alpha}\} = \mathbf{A}$. Then, for any given atom α and database D, we have $D \cup \Sigma \models \alpha$ iff $D \cup \Sigma_{\mathsf{Horn}} \models \alpha$.*

5.2 Translation into Datalog

Assume a normalized theory Σ of guarded TGDs. Also, assume $w = \mathit{width}(\Sigma)$, and n is the number of distinct predicate symbols occurring in $\mathsf{sch}(\Sigma)$. Similarly as we did in Section 4, we build next a program P such that the queries (Σ, q) and (P, q) have the same answers for all databases D over $\mathsf{sch}(\Sigma)$. As before, P is polynomial under the assumption that w is bounded by a constant. Crucially, in this case, P will be a plain DATALOG program.

First of all, P contains the rules described in **(I)**, **(III)**, **(IV)**, and **(V)** in Section 4. These rules that collect the constants and implement the marking algorithm from the previous translation do not use the rules with disjunction in **(II)**. Additionally, we include the following rules that implement the algorithm **Horn** and the inferences from Σ_{Horn}.

(VII) Generating abstract types from marked types. We add rules that construct the set $\mathsf{Horn}(\Sigma)$ from Algorithm 2, creating abstract types from the marked types. Recall that we represent (regular) types as m-sequences of 1 and 0, where the i-th value indicates whether the atom $\boldsymbol{\alpha}_i$ in the enumeration is contained in the type or not. We extend this representation using an additional constant 2 in position i to indicate that an atom $\boldsymbol{\alpha}_i$ is in the set τ_{ir} of an abstract type (τ_c, τ_{ir}). We will write $\vec{u}_{[i,j]}$ to denote the tuple $(u_i, u_{i+1}, \ldots, u_j)$; if $j < i$ then $\vec{u}_{[i,j]}$ refers to the empty tuple. For all $1 \leq i \leq m$, we add the rules:

$$\mathsf{Marked}(\vec{u}_{[1,i-1]}, 2, \vec{u}_{[i+1,m]}) \leftarrow \mathsf{Marked}(\vec{u}_{[1,i-1]}, 0, \vec{u}_{[i+1,m]}), \mathsf{Marked}(\vec{u}_{[1,i-1]}, 1, \vec{u}_{[i+1,m]})$$

(VIII) Constructing all horn types. We call a type *horn* if only one bit is set to 0, and the other bits to 1 or 2. Intuitively, horn types stand for abstract types of the form (τ_c, τ_{ir}) that can be converted into full TGDs of the form $\tau_c \to \boldsymbol{\alpha}$, where the atoms in τ_c are represented with 1, the atoms in τ_{ir} with 2, and atom $\boldsymbol{\alpha}$ with 0. First, we use a new m-ary predicate symbol Horn to construct all possible Horn types. To this end, we first add the following facts for all $0 \leq i < m$:

$$\mathsf{Horn}(\underbrace{1,\ldots,1}_{i},0,\underbrace{1,\ldots,1}_{m-i-1}) \leftarrow$$

To construct all possible alternations of 1 and 2, we add the following rule for all $1 \leq i \leq m$:

$$\mathsf{Horn}(\vec{u}_{[1,i-1]},2,\vec{u}_{[i+1,m]}) \leftarrow \mathsf{Horn}(\vec{u}_{[1,i-1]},1,\vec{u}_{[i+1,m]})$$

Finally, we use another m-ary relation $\mathsf{MarkedHorn}$ to collect all types from $\mathsf{Horn}(\Sigma)$ that are marked and that can be converted into full TGDs. The rule is as follows:

$$\mathsf{MarkedHorn}(\vec{u}) \leftarrow \mathsf{Marked}(\vec{u}),\mathsf{Horn}(\vec{u})$$

(IX) Implementing the full TGDs from Σ_{Horn}. Finally, we guarantee each $\tau_c \to \boldsymbol{\alpha} \in \Sigma_{\mathsf{Horn}}$ is satisfied. For each Horn type in $\mathsf{MarkedHorn}$, whenever the atoms in positions with 1 are made true by some instantiation of the variables with constants, we infer the atom in the 0 position. For all $1 \leq i,j \leq m$ we take a fresh $(j+1)$-ary predicate symbol Proj_i^j, where i indicates the position of the bit labelled with 0. For all $1 \leq i \leq m$, we add:

$$\mathsf{Proj}_i^m(\vec{\mathbf{x}},\vec{u}_{[1,i-1]},0,\vec{u}_{[i+1,m]}) \leftarrow \mathsf{MarkedHorn}(\vec{u}_{[1,i-1]},0,\vec{u}_{[i+1,m]}),\mathsf{const}^w(\vec{\mathbf{x}})$$

We will now project away bits from the Proj_i^j relations. We add the following rules for all $1 \leq j \leq m$ and for all $1 \leq i \leq m$:

$$\begin{aligned}
\mathsf{Proj}_i^{j-1}(\vec{\mathbf{x}},\vec{u}) &\leftarrow \mathsf{Proj}_i^j(\vec{\mathbf{x}},\vec{u},1),\boldsymbol{\alpha}_j\\
\mathsf{Proj}_i^{j-1}(\vec{\mathbf{x}},\vec{u}) &\leftarrow \mathsf{Proj}_i^j(\vec{\mathbf{x}},\vec{u},2)\\
\mathsf{Proj}_i^{j-1}(\vec{\mathbf{x}},\vec{u}) &\leftarrow \mathsf{Proj}_i^j(\vec{\mathbf{x}},\vec{u},0)
\end{aligned}$$

If we reach the last bit, we add the atom whose bit is set to 0. For all $1 \leq i \leq m$, we add:

$$\boldsymbol{\alpha}_i \leftarrow \mathsf{Proj}_i^0(\vec{\mathbf{x}})$$

This completes the polynomial translation of a query (Σ, q), where Σ is a set of guarded TGDs (without disjunction), to a (plain) Datalog query (P, q). Next, we provide a theorem that captures the correctness of the above translation.

▶ **Theorem 16.** *For a query (Σ, q), where Σ is a normalized theory of guarded TGDs, we can build a query (P, q), where P is a set of (plain)* Datalog *rules such that $\mathsf{ans}(\Sigma, q, D) = \mathsf{ans}(P, q, D)$ for any given database D over $\mathsf{sch}(\Sigma)$. Moreover, if width(Σ) is bounded, then (P, q) can be built in polynomial time.*

6 Discussion and Conclusions

In this paper, we have studied a very rich query language, where a query predicate is given together with a set of guarded (D)TGDs expressing domain knowledge. For queries (Σ, q) with q an instance query, we have proposed rewritings to Datalog$^\vee$ for Σ a set of guarded

DTGDs, and to (plain) Datalog for Σ a set of guarded (non-disjunctive) TGDs. To our best knowledge, these are the first such rewritings that are *polynomial* under the relevant, useful restriction that the maximal number of variables in the guarded (D)TGDs is bounded by a constant. If the number of variables per rule is not bounded, the rewriting is exponential in the size of the query. This is natural: answering guarded (D)TGD queries is 2ExpTime-hard [4], and cautious inference from Datalog$^\vee$ queries is coNExpTime-complete, and from Datalog is ExpTime-complete [12]. For the case with disjunction, a polynomial rewriting would imply coNExpTime = 2ExpTime, and for the case without disjunction a polynomial rewriting cannot exist since ExpTime $\neq$ 2ExpTime.

Our results can easily be generalized to guarded (D)TGDs with *acyclic conjunctive queries* or with *quantifier-free conjunctive queries*, that is conjunctive queries with no existential variables. Acyclic conjunctive queries can be rewritten in polynomial time into a set of rules that fall into the guarded (D)TGD fragment (see, e.g., the proof of Theorem 6.2 in [8]) and the quantifier-free conjunctive queries are syntactically restricted to ensure that the relevant variable assignments only map into the constants of the input database. Additionally, our techniques can be easily extended to support *nearly guarded (D)TGDs* [18], which are a strict extension of the guarded variants with non-guarded rules. Indeed, the latter rules can 'operate' only on constants from the input database and we just need to add a simple condition that ensures the satisfaction of these rules by the core instances. Identifying other cases that are in ExpTime and that allow for polynomial Datalog rewritings is an open research problem for future work.

Extending our approach to support other query languages is also an interesting direction for the future. We remark, however, that generalizing our translation to guarded DTGDs in the presence of arbitrary conjunctive queries while remaining polynomial is out of reach under common assumptions in complexity theory. Indeed, the associated query answering problem is 2ExpTime-hard in combined complexity already for a restricted fragment of arity two [24]. For the case of TGDs without disjunction a polynomial rewriting to Datalog for conjunctive queries may be feasible. However, although adapting the ideas in this work to conjunctive queries may be possible, and at the cost of an exponential blow-up for DTGDs, it seems a challenging task. The guarded DTGDs we studied in this paper do not allow for constants. Extending our translation to support constants remains for future work.

References

1 Shqiponja Ahmetaj, Magdalena Ortiz, and Mantas Simkus. Polynomial datalog rewritings for expressive description logics with closed predicates. In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 878–885. IJCAI/AAAI Press, 2016. URL: `http://www.ijcai.org/Abstract/16/129`.

2 Hajnal Andréka, István Németi, and Johan van Benthem. Modal languages and bounded fragments of predicate logic. *J. Philosophical Logic*, 27(3):217–274, 1998. `doi:10.1023/A:1004275029985`.

3 Vince Bárány, Michael Benedikt, and Balder ten Cate. Rewriting guarded negation queries. In *Proc. of MFCS' 13*, pages 98–110. ACM, 2013.

4 Vince Bárány, Georg Gottlob, and Martin Otto. Querying the guarded fragment. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*, pages 1–10. IEEE Computer Society, 2010. `doi:10.1109/LICS.2010.26`.

5 Vince Bárány, Balder ten Cate, and Martin Otto. Queries with guarded negation. *PVLDB*, 5(11):1328–1339, 2012. URL: http://vldb.org/pvldb/vol5/p1328_vincebarany_vldb2012.pdf.

6 Catriel Beeri and Moshe Y. Vardi. A proof procedure for data dependencies. *J. ACM*, 31(4):718–741, 1984. doi:10.1145/1634.1636.

7 Meghyn Bienvenu, Balder ten Cate, Carsten Lutz, and Frank Wolter. Ontology-based data access: A study through disjunctive datalog, csp, and MMSNP. *ACM Trans. Database Syst.*, 39(4):33:1–33:44, 2014. doi:10.1145/2661643.

8 Pierre Bourhis, Marco Manna, Michael Morak, and Andreas Pieris. Guarded-based disjunctive tuple-generating dependencies. *ACM Trans. Database Syst.*, 41(4):27:1–27:45, 2016. doi:10.1145/2976736.

9 Andrea Calì, Georg Gottlob, and Michael Kifer. Taming the infinite chase: Query answering under expressive relational constraints. *J. Artif. Intell. Res.*, 48:115–174, 2013. doi:10.1613/jair.3873.

10 Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. *J. Autom. Reasoning*, 39(3):385–429, 2007. doi:10.1007/s10817-007-9078-x.

11 Alin Deutsch and Val Tannen. Reformulation of XML queries and constraints. In Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors, *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*, volume 2572 of *Lecture Notes in Computer Science*, pages 225–241. Springer, 2003. doi:10.1007/3-540-36285-1_15.

12 Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive datalog. *ACM Trans. Database Syst.*, 22(3):364–418, 1997. doi:10.1145/261124.261126.

13 Thomas Eiter, Magdalena Ortiz, Mantas Simkus, Trung-Kien Tran, and Guohui Xiao. Query rewriting for horn-shiq plus rules. In Jörg Hoffmann and Bart Selman, editors, *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.* AAAI Press, 2012. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/4931.

14 Georg Gottlob, Stanislav Kikot, Roman Kontchakov, Vladimir V. Podolskii, Thomas Schwentick, and Michael Zakharyaschev. The price of query rewriting in ontology-based data access. *Artif. Intell.*, 213:42–59, 2014. doi:10.1016/j.artint.2014.04.004.

15 Georg Gottlob, Marco Manna, Michael Morak, and Andreas Pieris. On the complexity of ontological reasoning under disjunctive existential rules. In Branislav Rovan, Vladimiro Sassone, and Peter Widmayer, editors, *Mathematical Foundations of Computer Science 2012 - 37th International Symposium, MFCS 2012, Bratislava, Slovakia, August 27-31, 2012. Proceedings*, volume 7464 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2012. doi:10.1007/978-3-642-32589-2_1.

16 Georg Gottlob, Marco Manna, and Andreas Pieris. Polynomial combined rewritings for existential rules. In Chitta Baral, Giuseppe De Giacomo, and Thomas Eiter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014.* AAAI Press, 2014. URL: http://www.aaai.org/ocs/index.php/KR/KR14/paper/view/7973.

17 Georg Gottlob, Marco Manna, and Andreas Pieris. Polynomial rewritings for linear existential rules. In Qiang Yang and Michael Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 2992–2998. AAAI Press, 2015. URL: http://ijcai.org/Abstract/15/423.

18 Georg Gottlob, Sebastian Rudolph, and Mantas Simkus. Expressiveness of guarded existential rule languages. In Richard Hull and Martin Grohe, editors, *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, pages 27–38. ACM, 2014. doi:10.1145/2594538.2594556.

19 Georg Gottlob and Thomas Schwentick. Rewriting ontological queries into small nonrecursive datalog programs. In Gerhard Brewka, Thomas Eiter, and Sheila A. McIlraith, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-14, 2012*. AAAI Press, 2012. URL: http://www.aaai.org/ocs/index.php/KR/KR12/paper/view/4510.

20 Erich Grädel. On the restraining power of guards. *J. Symb. Log.*, 64(4):1719–1742, 1999.

21 Ullrich Hustadt, Boris Motik, and Ulrike Sattler. Reasoning in description logics by a reduction to disjunctive datalog. *J. Autom. Reasoning*, 39(3):351–384, 2007. doi:10.1007/s10817-007-9080-3.

22 Mark Kaminski, Yavor Nenov, and Bernardo Cuenca Grau. Datalog rewritability of disjunctive datalog programs and its applications to ontology reasoning. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 1077–1083. AAAI Press, 2014. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8201.

23 Roman Kontchakov, Carsten Lutz, David Toman, Frank Wolter, and Michael Zakharyaschev. The combined approach to ontology-based data access. In Toby Walsh, editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 2656–2661. IJCAI/AAAI, 2011. doi:10.5591/978-1-57735-516-8/IJCAI11-442.

24 Carsten Lutz. Inverse roles make conjunctive queries hard. In Diego Calvanese, Enrico Franconi, Volker Haarslev, Domenico Lembo, Boris Motik, Anni-Yasmin Turhan, and Sergio Tessaris, editors, *Proceedings of the 2007 International Workshop on Description Logics (DL2007), Brixen-Bressanone, near Bozen-Bolzano, Italy, 8-10 June, 2007*, volume 250 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007. URL: http://ceur-ws.org/Vol-250/paper_3.pdf.

25 Carsten Lutz, Inanç Seylan, and Frank Wolter. Ontology-based data access with closed predicates is inherently intractable(sometimes). In Francesca Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 1024–1030. IJCAI/AAAI, 2013. URL: http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6870.

26 Carsten Lutz, Inanç Seylan, and Frank Wolter. Ontology-based data access with closed predicates is inherently intractable(sometimes). In Francesca Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 1024–1030. IJCAI/AAAI, 2013. URL: http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6870.

27 Magdalena Ortiz, Sebastian Rudolph, and Mantas Simkus. Worst-case optimal reasoning for the horn-dl fragments of OWL 1 and 2. In Fangzhen Lin, Ulrike Sattler, and Miroslaw Truszczynski, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13, 2010*. AAAI Press, 2010. URL: http://aaai.org/ocs/index.php/KR/KR2010/paper/view/1296.

28 Héctor Pérez-Urbina, Boris Motik, and Ian Horrocks. Tractable query answering and rewriting under description logic constraints. *J. Applied Logic*, 8(2):186–209, 2010. doi:10.1016/j.jal.2009.09.004.

29 Despoina Trivela, Giorgos Stoilos, Alexandros Chortaras, and Giorgos B. Stamou. Optimising resolution-based rewriting algorithms for OWL ontologies. *J. Web Sem.*, 33:30–49, 2015. doi:10.1016/j.websem.2015.02.001.

A Appendix

A.1 Proofs of Section 3

To prove Theorem 6, we first introduce the following lemma.

▶ **Lemma 17.** *Assume a theory Σ, a database D and an assertion α. Then $(\Sigma, D) \not\models \alpha$ iff there exists a core instance D_c for (Σ, D) s.t.*
(1) *$\alpha \notin D_c$, and*
(2) *there exists an extension I of D_c s.t. $I \models (\Sigma, D)$.*

Proof. For the "$\Leftarrow$" direction, it is clear that I is the desired instance that satisfies (Σ, D) (condition *(2)*) and is such that $\alpha \notin I$. The latter is true from condition *(1)* and the fact that I is an extension of D_c. For the "$\Rightarrow$" direction, assume I with $\alpha \notin I$ is an instance that satisfies (Σ, D). W.l.o.g. assume that $\mathsf{sch}(I) \subseteq \mathsf{sch}(\Sigma)$. We construct a core database D_c such that $D_c = \{R(\vec{t}) \mid R(\vec{t}) \in I, \vec{t} \subseteq \mathsf{dom}(D)\}$. We claim that D_c is a core interpretation for (Σ, D). Condition *(c1)* of Definition 3 holds by construction of D_c. Also, D_c satisfies all the rules from $\Sigma_\forall \subseteq \Sigma$ as these rules can only create new atoms with terms that range over constants from $\mathsf{dom}(D)$, and thus condition *(c2)* is also satisfied by D_c. ◀

Proof of Theorem 6. By Lemma 17, we only need to show that, for any given core D_c, there is a non-losing strategy *str* for Bob on D_c if and only if D_c can be extended into an instance I that satisfies (Σ, D).

For the "$\Rightarrow$" direction, from an arbitrary non-losing *str* for D_c, we build I as follows. We write $\vec{c}_\tau$ to denote a tuple of constants realizing τ, that is $\tau = \mathit{type}(\vec{c}_\tau, D_c)$. In the following, we denote by $\mathit{rn}(D_c, \mathit{str})$ the set of all finite runs $\vec{c}_\tau \sigma_1 h_1 \tau_1 \sigma_2 h_2 \tau_2 \dots$ that follow *str*. Additionally, we will denote with $\mathit{fv}(\sigma)$ the set $\mathsf{vars}(B) \cap \mathsf{vars}(H)$ of variables where $\sigma \in \Sigma$ is a TGD of the form $B \to \exists \vec{y}.H \in \Sigma$.

We let I be the set of atoms $R(t_1, \dots, t_\ell)$ such that one of the following holds:
(a) there exists a run $r = \vec{c}_\tau$ in $\mathit{rn}(D_c, \mathit{str})$ with $R(\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_\ell}) \in \tau$ and $t_j = c_{i_j} \in \vec{c}_\tau$ for each $j \in [1, \ell]$, or
(b) there exists a run $r = \vec{c}_\tau \dots \sigma_i h_i \tau_i$ in $\mathit{rn}(D_c, \mathit{str})$ for $i \geq 1$ with $R(\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_\ell}) \in \tau_i$, where σ_i is a TGD of the form $B \to \exists \vec{y}.H \in \Sigma$ such that, for each t_j with $j \in [1, \ell]$, the following holds:
  (1) t_j is the labelled null $r'\mathbf{x}_{i_j}$, if
    (i) $r' = \vec{c}_\tau \dots \sigma_k h_k \tau_k$ and $r = r' \dots \sigma_i h_i \tau_i$ with $1 \leq k \leq i$,
    (ii) $\mathbf{x}_{i_j} \notin h_k(\mathit{fv}(\sigma_k))$, and
    (iii) $\mathbf{x}_{i_j} \in h_l(\mathit{fv}(\sigma_l))$ for each $k < l \leq i$ if $i \neq k$.
  (2) t_j is the constant c_{i_j} if:
    (i) $\mathbf{x}_{i_j} \in h_l(\mathit{fv}(\sigma_l))$ for each $l \in [1, i]$.

First, we show that I is an extension of D_c, that is for each predicate $R \in \mathsf{sch}(\Sigma)$ and for each tuple $\vec{c}$ of constants from $\mathsf{dom}(D_c)$, $R(\vec{c}) \in D_c$ if and only if $R(\vec{c}) \in I$. Observe that, by construction of I, condition (a), D_c is contained in I. That no other ground atom (that does not appear in D_c) is added to I is ensured by *(b)* point (2) and by definition of a strategy, more precisely by condition (C1). That is (C1) guarantees that each type τ_i in a run $r = \vec{c}_\tau \dots \sigma_i h_i \tau_i$ in $\mathit{rn}(D_c, \mathit{str})$ coincides with the type τ_{i-1} on the shared variables

$h_i(fv(\sigma_i))$. Thus τ_i coincides with τ on the variables propagated throughout the run up to i and that those variables are mapped to the same constants in the core is ensured by *(b)* point (2) in the model I constructed above.

We now show that I models (Σ, D). That $I \models D$ results by definition of a core instance D_c. To prove $I \models \Sigma$, we first show that I satisfies all guarded DTGDs in $\Sigma_\forall$. Assume an arbitrary guarded DTGD σ of the form $\varphi \to \bigvee_{i=1}^{n} H_i$ in $\Sigma_\forall$ and let h be an arbitrary substitution from $\mathsf{vars}(\varphi)$ to $\mathsf{terms}(I)$ such that $h(\varphi) \subseteq I$. Recall that the guard atom in φ contains all the variables that appear in σ. We show that there must exist an $i \in [n]$ with $h(H_i) \in I$. We distinguish the following two cases:

(i) If $h(\varphi) \subseteq D_c$, then by definition of a core instance, $D_c \models \Sigma_\forall$. In particular D_c satisfies σ, that is there exists an $i \in [n]$ with $h(H_i) \in D_c$. The latter together with $D_c \subseteq I$ implies $h(H_i) \in I$.

(ii) Otherwise, let the ℓ-ary atom $R(x_1, \ldots, x_\ell)$ be the guard atom in φ and let $h(x_i) = t_i$ for each $i \in [1, \ell]$, such that $R(t_1, \ldots, t_\ell) \in I$ and $R(t_1, \ldots, t_\ell) \notin D_c$, that is there exists an $i \in [1, \ell]$ such that t_i is a labelled null. Observe that each t_i is either a labelled null $r\mathbf{x}_{i_j}$, or a constant c_{i_j}, or a labelled null $r'\mathbf{x}_{i_j}$ where r' is such that $r = r' \ldots \sigma_i h_i \tau_i$. Let $t_j \in \{t_1, \ldots, t_\ell\}$ be the labelled null $r\mathbf{x}_{i_j}$ with the longest prefix run $r \in rn(D_c, str)$, that is there is no $k \in [1, \ell]$ such that $t_k = r'\mathbf{x}_{i_k}$, $r' = rr''$, and $r' \neq r$ and assume r is of the form $\vec{c}_\tau \ldots \sigma_i h_i \tau_i$ and note that $i \geq 1$. First we claim that the corresponding $R(\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_\ell}) \in \mathit{tail}(r) = \tau_i$, where each $\mathbf{x}_{i_j}$ is in the position of a labelled null $t_j = r\mathbf{x}_{i_j}$ or the labelled null $t_j = r'\mathbf{x}_{i_j}$ with $r = r'r''$ or the constant $c_{i_j} \in \vec{c}_\tau$. Towards a contradiction assume $R(\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_\ell}) \notin \tau_i$. Then by construction of I, there exists a run $r_l \in rn(D_c, str)$ of the form $r \ldots \sigma_l h_l \tau_l$ such that $R(\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_\ell}) \in \mathit{tail}(r_l)$ and each variable $\mathbf{x}_{i_1}$ with $t_j = r\mathbf{x}_{i_j}$ must have been shared throughout the run up to τ_l from τ_i and the other variables must come from earlier types in r. But then by condition (C1) of the game all types appearing in the run r_l from $\mathit{tail}(r) = \tau_i$ to $\mathit{tail}(r_l) = \tau_l$ must coincide on these shared variables and hence the atom $R(\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_\ell})$ must be in each type from τ_l to τ_i. The latter contradicts the assumption that $R(\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_\ell}) \notin \tau_i$.

That $R(x_1, \ldots, x_\ell)$ is the guard atom in φ and that $h(\varphi) \subseteq I$ imply $R'(t_{i_j}, \ldots, t_{i_k}) \in I$ for all atoms $R'(x_{i_j}, \ldots, x_{i_k})$ in φ where $\{t_{i_j}, \ldots, t_{i_k}\} \subseteq \{t_1, \ldots, t_\ell\}$. Assume an arbitrary atom $R'(x_{i_j}, \ldots, x_{i_k})$ in φ that is not the guard atom, that is $R'(t_{i_j}, \ldots, t_{i_k}) \in I$. We claim that the corresponding $R'(\mathbf{x}_{i_j}, \ldots, \mathbf{x}_{i_k})$ is also in τ_i. Let $t_{i_l} \in \{t_{i_j}, \ldots, t_{i_k}\}$ be $r'\mathbf{x}_{i_l}$ with the longest prefix run r' from the terms of $R'(t_{i_j}, \ldots, t_{i_k})$. Indeed, if $r' = r$ then $R'(\mathbf{x}_{i_l}, \ldots, \mathbf{x}_{i_k}) \in \tau_i$ by construction of I. Otherwise observe that $r = r' \ldots \sigma_i h_i \tau_i$ since $t_{i_l} \in \{t_1, \ldots, t_\ell\}$. Then it is clear by construction of I that the atom $R'(t_{i_l}, \ldots, t_{i_k})$ is added to I because $R'(\mathbf{x}_{i_l}, \ldots, \mathbf{x}_{i_k}) \in \mathit{tail}(r')$. We claim that $R'(\mathbf{x}_{i_l}, \ldots, \mathbf{x}_{i_k}) \in \tau_j$ for each type τ_j occurring in the run r from $\mathit{tail}(r')$ to τ_i. By assumption that the guard $R(\mathbf{x}_{i_l}, \ldots, \mathbf{x}_{i_\ell}) \in \tau_i$ and by construction points (1) and (2), each $\mathbf{x}_{i_j}$ appears in each type throughout the run from $\mathit{tail}(r')$ to τ_i which together with condition (C2) imply $R'(\mathbf{x}_{i_l}, \ldots, \mathbf{x}_{i_k})$ must have be carried throughout the run and in particular $R'(\mathbf{x}_{i_l}, \ldots, \mathbf{x}_{i_k})$ must be in τ_i. Finally, with a similar argument one prove the claim if each t_{i_j} is a constant from the database.

The above implies that there is a substitution $h' : \{x_1, \ldots, x_\ell\} \to \{\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_\ell}\}$ such that $h'(\varphi) \subseteq \tau_i$. Since by construction each $r \in rn(D_c, str)$ follows the strategy str which is non-losing, thus $\tau_i \in \mathsf{LC}(\Sigma, I)$, that is τ_i satisfies the condition $\mathsf{LC}_\forall$, thus there exists $i \in [n]$ with $H_i(\mathbf{x}_{i_j}, \ldots, \mathbf{x}_{i_k}) \in \tau_i$, hence by construction of I also the corresponding $H_i(t_j, \ldots, t_k) \in I$ and $H_i(t_j, \ldots, t_k) = h(H_i(x_j, \ldots, x_k))$, which shows the claim.

It is left to show that I satisfies all the TGDs σ in $\Sigma_\exists$ of the form $B \to \exists \vec{y}.H$. Recall that $fv(\sigma)$ is the set of shared variables $\mathsf{vars}(B) \cap \mathsf{vars}(H)$. Let h be an arbitrary substitution from $\mathsf{vars}(B)$ to $\mathsf{terms}(I)$ such that $h(B) \in I$. We show that there exists a substitution h' from $\mathsf{vars}(H)$ to $\mathsf{terms}(I)$ such that $h'(H) \in I$ and $h'(\mathbf{x}) = h(\mathbf{x})$ for all $\mathbf{x} \in fv(\sigma)$. Intuitively, that B is mapped to I implies there is a run with the corresponding atom in the tail type and hence there is a substitution from B to the type. This makes the TGD σ applicable to the type, and by the strategy there exists a new type that satisfies the head preserving the atoms over the shared variables, and hence there is an h' that satisfies the above properties. Formally, let $h(B)$ be the atom $B(t_1, \ldots, t_\ell) \in I$ such that $B(t_1, \ldots, t_\ell) \notin D_c$, that is at least one of $t_1, \ldots, t_\ell$ is a labelled null. The case when each t_i is a constant, that is $B(t_1, \ldots, t_\ell) \in D_c$, is analogous. Let $t_j \in \{t_1, \ldots, t_\ell\}$ be the labelled null $r\mathbf{x}_{i_j}$ that has the longest prefix run $r \in rn(D_c, str)$, that is there is no $k \in [1, \ell]$ such that $t_k = r'\mathbf{x}_{i_k}$, $r' = rr''$, and $r' \neq r$. Then by construction of I there exists a run $r \in rn(D_c, str)$ of the form $r = \vec{c}_\tau \ldots \sigma_i h_i \tau_i$ with $i \geq 1$, such that $B(\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_\ell}) \in \tau_i$ where each $\mathbf{x}_{i_j}$ is in the position of the labelled null $t_j = r'\mathbf{x}_{i_j}$ or the constant c_{i_j} in $R(t_1, \ldots, t_\ell)$. Then let h_σ be the substitution from $\mathsf{vars}(B)$ to $\{\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_\ell}\}$ such that for each $x \in \mathsf{vars}(B)$, $h_\sigma(x) = \mathbf{x}_{i_j}$ if $h(x)$ is the labelled null $r'\mathbf{x}_{i_j}$ or the constant $c_{i_j} \in \vec{c}_\tau$. Observe that $h_\sigma(B) = B(\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_\ell}) \in \tau_i$. Since str is a non-losing strategy, $str(\tau_i, \sigma, h_\sigma)$ is defined and hence there exists a type $\tau_i' = str(\tau_i, \sigma, h_\sigma)$ and a substitution $h'_\sigma : \mathsf{vars}(H) \to \mathbf{X}$ such that $h'_\sigma(H) \in \tau_i'$. Thus the run $r' = r\sigma h_\sigma \tau_i'$ is in $rn(D_c, str)$ with $h'_\sigma(H) \in \tau_i'$. Then, from $h_\sigma(fv(\sigma)) = h'_\sigma(fv(\sigma))$ (condition (C2) in the game) and by construction of I, $h'(H) \in I$, where h' is the substitution which coincided with h on the shared variables $fv(\sigma)$ and maps the other variables $x \in \mathsf{vars}(H) \setminus fv(\sigma)$ to $r'h'_\sigma(x)$.

For "$\Leftarrow$", assume an arbitrary model I of (Σ, D) that is an extension of D_c. We can extract from it a non-losing strategy for Bob as follows. First, let $rlz(I)$ be a the set of all types realized in I, and observe that $rlz(I) \subseteq \mathsf{LC}(\Sigma, D_c)$ and in particular that, for all $R(\vec{c}) \in D_c$, $type(\vec{c}, I) \in rlz(I)$. Then it suffices to set, for each type $\tau \in rlz(I)$ and each TGD $\sigma \in \Sigma_\exists$ of the form $B \to \exists \vec{y}.H$ with $h(B) \in \tau$, $str(\tau, \sigma, h) = \tau'$ for an arbitrarily chosen type $\tau' \in rlz(I)$ that satisfies (C1) and (C2) which exists because I is a model, hence it satisfies all DTGDs in Σ. This str is a non-losing strategy for Bob. ◀

Proof. (Proof of Theorem 7.) For the "$\Rightarrow$" direction, let str be a non-losing strategy on D_c. It suffices to show that if a type is marked then it cannot occur in a run that follows str, which implies that if a type realized in D_c is marked by the algorithm **Mark**(Σ), then there is no non-losing strategy for D_c. The proof is by induction in the number of iterations needed to mark a type. For the base case, $k = 0$, assume a type $\tau \in types(\Sigma)$ is marked by ($\mathrm{M}_\forall$) in the algorithm **Mark**(Σ), then τ doesn't satisfy ($\mathrm{LC}_\forall$, that is $\tau \notin \mathsf{LC}(\Sigma, D_c)$. Hence there cannot exist a run r with $tail(r) = \tau$ since this contradicts the definition of non-losing str.

For the induction hypothesis, assume the claim holds for the first k iterations, that is assume that every type marked by the algorithm **Mark**(Σ) from ($\mathrm{M}_\forall$) or in the first k iterations of ($\mathrm{M}_\exists$) doesn't occur in a run that follows str. We show that the claim holds also for the next type that gets marked by (M_Σ). Thus for the inductive case, let $\tau \in types(\Sigma)$ be the first type marked by the algorithm after some iteration $j > k$ of ($\mathrm{M}_\exists$) and let $\sigma \in \Sigma_\exists$ of the form $B \to \exists \vec{y}.H$ and $h : \mathsf{vars}(B) \to \mathbf{X}$ with $h(B) \in \tau$, be the TGD and the substitution, respectively, chosen at that step. Let $fv(\sigma) = \mathsf{vars}(B) \cap \mathsf{vars}(H)$. Towards a contradiction, assume there exists a run r that follows str and $tail(r) = \tau$. By assumption and definition of non-losing str, then there must exist a type $\tau' = str(\tau, \sigma, h)$ s.t. $\tau' \in \mathsf{LC}(\Sigma, D_c)$, and it satisfies conditions (C1) and (C2) of the game. But since τ gets marked then for each $\tau_i \in types(\Sigma)$, either τ_i is marked at some earlier step in the algorithm **Mark**(Σ), and then by hypothesis this type cannot appear in a run, or τ_i doesn't satisfy conditions (C1) or (C2).

It follows that there exists no τ_i that can define $str(\tau, \sigma, h)$ which contradicts that str is a non-losing strategy.

For the "$\Leftarrow$" direction, let $good(\Sigma) \subseteq types(\Sigma)$ be the set of all types that are not marked by the algorithm $\mathbf{Mark}(\Sigma)$, that is $good(\Sigma) \subseteq \mathsf{LC}(\Sigma, D_c)$. By assumption the types realized in D_c are not marked. We can build a strategy str on D_c which maps all pairs of a type $\tau \in good(\Sigma)$ and each TGD $\sigma \in \Sigma_\exists$ of the form $B \to \exists \vec{y}.H$ and substitution $h : \mathsf{vars}(B) \to \mathbf{X}$ with $h(B) \in \tau$ to an arbitrary type $\tau' \in good(\Sigma)$ such that (C1) and (C2) hold, that is $\tau|_{h(fv(\sigma))} = \tau'|_{h(fv(\sigma))}$, and there exist a substitution $h' : \mathsf{vars}(H) \to \mathbf{X}$ with $h(fv(\sigma)) = h'(fv(\sigma))$ such that $h'(H) \in \tau'$. Such construction of str is possible since if there would not exists a $\tau' = str(\tau, \sigma, h)$ then τ would be marked by the algorithm as some iteration of ($\mathrm{M}_\exists$). We claim that str is a non-losing strategy on D_c. Towards a contradiction, assume str is not a non-losing strategy on D_c, that is there exists a finite run r with $tail(r) = \tau$ that follows str, such that either $tail(r) \notin \mathsf{LC}(\Sigma, D_c)$ or there exists some TGD $\sigma \in \Sigma_\exists$ of the form $B \to \exists \vec{y}.H$ and a substitution $h : \mathsf{vars}(B) \to \mathbf{X}$ with $h(B) \in \tau$ and $str(\tau, \sigma, h)$ is not defined. By construction of str, $tail(r) \in good(\Sigma)$, hence it cannot be the case that $tail(r) \notin \mathsf{LC}(\Sigma, D_c)$, because then τ doesn't satisfy ($\mathrm{LC}_\forall$) and as a consequence τ would be marked by ($\mathrm{M}_\forall$) in the algorithm, which then contradicts our assumption that τ is not marked. For the other case, using similar reasoning as for the "$\Rightarrow$" direction, if $str(\tau, \sigma, h)$ is not defined then there exists some iteration of ($\mathrm{M}_\exists$) that would mark $tail(r)$, which again contradicts our assumption. Hence, str constructed as above is a non-losing strategy for Bob on D_c. ◀

A.2 Proofs of Section 4

Proof. (Proof of Proposition 9.) This proof is similar to the proof of Theorem 3 in [1]. Program P can be partitioned into programs P_1 and P_2 as follows:

- P_1 consists of all rules in **(I)** and **(II)**. P_1 is a positive disjunctive program with at most w variables in each rule.
- P_2 consists of the remaining rules, and is disjunction-free.

Note that P_2 does not define any relations used in P_1, i.e. none of the relation symbols of P_1 occurs in the head of a rule in P_2. Assume a set F of facts over the signature of P_1. Due to the above properties, the successful runs of the following non-deterministic procedure generate the set of all models of $P \cup F$:

(S1) Compute a minimal model I_1 of $P_1 \cup F$.

(S2) Compute the least model I_2 of $I_1 \cup P_2^{I_1}$. If I_2 does not exist due to a constraint violation, then return *failure*.

Since P_1 has at most w-variables in every rule, each minimal model I_1 of $P_1 \cup F$ is of polynomial size in the size of $P_1 \cup F$, and the set of all such models can be traversed in polynomial space. For a given I_1, performing step (S2) is feasible in (deterministic) exponential time, because $P_2^{I_1}$ is a ground disjunction-free positive program of exponential size. It follows that computing the answers to (P, q) for any given input database D over the schema $\mathsf{sch}(\Sigma)$ requires single deterministic exponential time and is coNP in data complexity. ◀

A.3 Proofs of Section 5

Proof. (Proof of Lemma 12.) To show the "$\Rightarrow$" direction, we consider an arbitrary run of the algorithm that starts at $N_0 = \{(\tau_c, \emptyset) \mid \tau_c \in M\}$ with $M = \mathsf{Marked}(\Sigma)$, and we let N_i be the set N after the i-th iteration of the algorithm. (Note that although different runs may generate different sequences of N_is, this is irrelevant for the proof, and the final $\mathsf{Horn}(\Sigma)$ is unique). We show that $(\tau_c, \tau_{ir}) \in N_i$ implies that $\tau_c \cup \tau'_{ir} \in M$ for each $\tau'_{ir} \subseteq \tau_{ir}$, by

induction on the minimal i such that $(\tau_c, \tau_{ir}) \in N_i$. The claim holds trivially for $i = 0$. Next, assume $(\tau'_c, \tau'_{ir}) \in N_{i+1}$ but $(\tau'_c, \tau'_{ir}) \notin N_i$. Then there are $\{(\tau_c, \tau_{ir}), (\tau'_c, \tau_{ir})\} \subseteq N_i$ with $\tau_c = \tau'_c \cup \{\boldsymbol{\alpha}\}$ for some $\boldsymbol{\alpha} \notin \tau_c$, and $\tau'_{ir} = \tau_{ir} \cup \{\boldsymbol{\alpha}\}$. By hypothesis $\tau'_c \in M$ and $\tau_c = \tau'_c \cup \{\boldsymbol{\alpha}\} \in M$. Moreover, for any subset $\tau''_{ir} \subseteq \tau'_{ir}$: if $\{\boldsymbol{\alpha}\} \notin \tau''_{ir}$ then $\tau''_{ir} \subseteq \tau_{ir}$ and by hypothesis $\tau'_c \cup \tau''_{ir} \in M$; otherwise if $\{\boldsymbol{\alpha}\} \in \tau''_{ir}$ then $\tau'_c \cup \{\boldsymbol{\alpha}\} = \tau_c$ and $\tau''_{ir} \setminus \{\boldsymbol{\alpha}\} \subseteq \tau_{ir}$ and again by hypothesis $\tau_c \cup (\tau''_{ir} \setminus \{\boldsymbol{\alpha}\}) \in M$.

We show the "$\Leftarrow$" direction by induction on the size of τ_{ir}. Let $M = \mathsf{Marked}(\Sigma)$ and $N = \mathsf{Horn}(\Sigma)$. For $\tau_{ir} = \emptyset$, by construction $(\tau_c, \emptyset) \in N$. Assume the claim holds for all abstract types with τ_{ir} of size $k \leq |\mathbf{A}| - |\tau_c| - 1$. We show it for $k+1$. Assume $\tau_c \cup \tau'_{ir} \in M$ for each $\tau'_{ir} \subseteq \tau_{ir}$ with $|\tau_{ir}| = k+1$. In particular, it must hold for an arbitrary $\tau'_{ir} \subseteq \tau_{ir}$ with $|\tau'_{ir}| = k$. Thus by hypothesis $(\tau_c, \tau'_{ir}) \in N$. Now, let $\{\boldsymbol{\alpha}\} = \tau_{ir} \setminus \tau'_{ir}$. By assumption $\tau_c \cup \{\boldsymbol{\alpha}\} \cup \tau''_{ir} \in M$ for each $\tau''_{ir} \subseteq \tau'_{ir}$. As $|\tau'_{ir}| = k$, again by hypothesis $(\tau_c \cup \{\boldsymbol{\alpha}\}, \tau'_{ir})$ must also be in the set N. But then $(\tau_c \cup \{\boldsymbol{\alpha}\}, \tau'_{ir}) \in N$ and $(\tau_c, \tau'_{ir}) \in N$ implies $(\tau_c, \tau'_{ir} \cup \{\boldsymbol{\alpha}\}) \in N$ by algorithm Horn, that is $(\tau_c, \tau_{ir}) \in N$. ◀

Let φ be a disjunction of atoms. In what follows, for an instance I, we write $I \models \varphi$ if some atom from φ is in I, and $I \not\models \varphi$ otherwise.

Proof. (Proof of Lemma 13.) Let $M = \mathsf{Marked}(\Sigma)$ and $N = \mathsf{Horn}(\Sigma)$. For convenience, we denote the atoms in τ_{ir} by $\boldsymbol{\alpha}_1^{ir}, \ldots, \boldsymbol{\alpha}_l^{ir}$, and the disjunction of the atoms in $\mathbf{A} \setminus (\tau_c \cup \tau_{ir})$ by $\mathbf{A}^H_\vee$. For the "$\Rightarrow$" direction, as $(\tau_c, \tau_{ir}) \in N$, we know from Lemma 12 that $\tau_c \cup \tau_{ir} \in M$, and by Lemma 10, $\Sigma \models \tau_c, \tau_{ir} \rightarrow \mathbf{A}^H_\vee$. To show that $\Sigma \models \tau_c \rightarrow \mathbf{A}^H_\vee$, we argue that one can always remove any of the atoms $\boldsymbol{\alpha}_i^{ir}$ with $1 \leq i \leq \ell$ from the rule and the new rule obtained will still be entailed by the theory. The claim is trivial for $l = 0$. Let $\boldsymbol{\alpha}_i^{ir}$ be an arbitrary atom in τ_{ir} and let $\tau'_{ir} = \tau_{ir} \setminus \{\boldsymbol{\alpha}_i^{ir}\}$. From Lemma 12, we know that $\tau_c \cup \tau'_{ir} \in M$, thus $\Sigma \models \tau_c, \tau'_{ir} \rightarrow \mathbf{A}^H_\vee \vee \boldsymbol{\alpha}_i^{ir}$ (*). But by assumption $\Sigma \models \tau_c, \tau'_{ir}, \boldsymbol{\alpha}_i^{ir} \rightarrow \mathbf{A}^H_\vee$ (**). Then one can eliminate the atom $\boldsymbol{\alpha}_i^{ir}$ and the resulting rule will is still entailed by the theory, that is $\Sigma \models \tau_c, \tau'_{ir} \rightarrow \mathbf{A}^H_\vee$. Assume towards a contradiction that there exists a model I of Σ, and a substitution h such that $h(\tau_c \cup \tau'_{ir}) \subseteq I$, but $I \not\models h(\mathbf{A}^H_\vee)$. From this and (*), it follows that $h(\boldsymbol{\alpha}_i^{ir}) \subseteq I$, but then by (**), $I \not\models h(\mathbf{A}^H_\vee(a))$, a contradiction. So $\Sigma \models \tau_c, \tau'_{ir} \rightarrow \mathbf{A}^H_\vee$.

For the "$\Leftarrow$" direction, $\Sigma \models \tau_c \rightarrow \mathbf{A}^H_\vee$ implies $\Sigma \models \tau_c \rightarrow \tau_{ir} \vee \mathbf{A}^H_\vee$. But then by Lemma 10, $\tau_c \in M$. Now, it easy to see that for every $\tau'_{ir} \subseteq \tau_{ir}$, we have $\Sigma \models \tau_c, \tau'_{ir} \rightarrow \mathbf{A}^H_\vee \vee \tau''_\vee$ (***), where $\tau''_\vee$ is the disjunction of all atoms in $\tau_{ir} \setminus \tau'_{ir}$. Indeed, if this were not the case, then there would exist a model I of Σ and a substitution h such that $h(\tau_c, \tau'_{ir}) \subseteq I$ and $I \not\models h(\mathbf{A}^H_\vee \vee \tau''_\vee)$. But then $I \not\models h(\mathbf{A}^H_\vee)$, which contradicts our assumption. Now, (***) implies $\tau_c \cup \tau'_{ir}$ is a marked type for each $\tau'_{ir} \subseteq \tau_{ir}$, and thus by Lemma 12, $(\tau_c, \tau_{ir}) \in N$ follows. ◀

Proof. (Proof of Lemma 14.) Let $|\mathbf{A}| = n$. First, notice that $|\tau_c| \geq n - 1$, implies that $(\tau_c, \emptyset) \in M$ is the desired abstract type. Now, assume $|\tau_c| < n - 1$. Then by Lemma 10, $\Sigma \models \tau_c \rightarrow \boldsymbol{\alpha}_1 \vee \ldots \vee \boldsymbol{\alpha}_\ell$ with $\{\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n\} = \mathbf{A} \setminus \tau_c$. Since Σ is a set of (non-disjunctive) TGDs, by the property of convexity there exists $i \in \{1, \ldots, \ell\}$ such that $\Sigma \models \tau_c \rightarrow \boldsymbol{\alpha}_i$. Then, by Lemma 13, we have $(\tau_c, \{\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_\ell\} \setminus \{\boldsymbol{\alpha}_i\}) \in N$, which proves the claim. ◀

Proof. (Proof of Theorem 15.) Let $M = \mathsf{Marked}(\Sigma)$ and $N = \mathsf{Horn}(\Sigma)$, and let Σ_M be the set of all $\mathit{rule}(\tau)$ with $\tau \in M$. By Lemma 10, it is sufficient to show that $\Sigma \models \Sigma_M$ if and only if $\Sigma \models \Sigma_{\mathsf{Horn}}$. For the "$\Rightarrow$" direction, it suffices to show that for every rule in Σ_M there exists a more restricted rule in Σ_{Horn} that is also entailed by the theory. To this end, let $\tau_c \rightarrow \mathbf{A}^H_\vee$ be an arbitrary rule in Σ_M, where $\tau_c \in M$ and $\mathbf{A}^H_\vee = \bigvee_{\boldsymbol{\alpha} \in \mathbf{A} \setminus \tau_c} \boldsymbol{\alpha}$. By Lemma 14 there exists an abstract type (τ_c, τ_{ir}) in M with $|\tau| + |\tau_{ir}| \geq |\mathbf{A}| - 1$. Hence $\tau_c \rightarrow \boldsymbol{\alpha} \in \Sigma_{\mathsf{Horn}}$ for $\{\boldsymbol{\alpha}\} = \mathbf{A} \setminus (\tau \cup \tau_{ir})$, and by Lemma 13, $\Sigma \models \tau_c \rightarrow \boldsymbol{\alpha}$. For the "$\Leftarrow$" direction, it is easy to see

that for every rule in Σ_{Horn} there exists a more general rule in Σ_M. That is, for an arbitrary $\tau_c \to \mathbf{A}^H_\vee$ in Σ_{Horn} with $\mathbf{A}^H_\vee$ a disjunction over the set of atoms $\mathbf{A}^H$, by Lemma 13, we have $(\tau_c, \tau_{ir}) \in N$ for $\tau_{ir} = \mathbf{A} \setminus (\tau_c \cup \mathbf{A}^H)$. Then by Lemma 12, $\tau_c \in M$, hence $\tau_c \to \mathbf{A}^H_\vee \vee \tau''_\vee$ is in Σ_M, where $\tau''_\vee$ is the disjunction of the atoms in τ_{ir}. ◀

Enumeration on Trees under Relabelings

Antoine Amarilli
LTCI, Télécom ParisTech, Université Paris-Saclay, Paris, France

Pierre Bourhis
CRIStAL, CNRS UMR 9189 & Inria Lille, Lille, France

Stefan Mengel
CNRS, CRIL UMR 8188, Lens, France

Abstract

We study how to evaluate MSO queries with free variables on trees, within the framework of enumeration algorithms. Previous work has shown how to enumerate answers with linear-time preprocessing and delay linear in the size of each output, i.e., constant-delay for free first-order variables. We extend this result to support *relabelings*, a restricted kind of update operations on trees which allows us to change the node labels. Our main result shows that we can enumerate the answers of MSO queries on trees with linear-time preprocessing and delay linear in each answer, while supporting node relabelings in logarithmic time. To prove this, we reuse the circuit-based enumeration structure from our earlier work, and develop techniques to maintain its index under node relabelings. We also show how enumeration under relabelings can be applied to evaluate practical query languages, such as aggregate, group-by, and parameterized queries.

2012 ACM Subject Classification Theory of computation → Logic and databases

Keywords and phrases enumeration, trees, updates, MSO, circuits, knowledge compilation

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.5

Funding For the full version with proofs, see [5], `https://arxiv.org/abs/1709.06185`.

1 Introduction

Enumeration algorithms are a common way to compute large query results on databases, see, e.g., [28]. Instead of computing all results, these algorithms compute results one after the other, while ensuring that the time between two successive results (the *delay*) remains small. Ideally, the delay should be *linear* in the size of each produced solution, and independent of the size of the input database. To make this possible, enumeration algorithms can build an index structure on the database during a *preprocessing phase* that ideally runs in linear time.

Most enumeration algorithms assume that the input database will not change. If we *update* the database, we must re-run the preprocessing phase from scratch, which is unreasonable in practice. Losemann and Martens [24] proposed the first enumeration algorithm that addresses this issue: they study monadic second-order (MSO) query evaluation on trees, and show that the index structure for enumeration can be maintained under updates. More precisely, they can update the index in time polylogarithmic in the input tree T (much better than re-running the linear preprocessing). The tradeoff is that their delay is also polylogarithmic in T, whereas the delay can be independent of T when there are no updates [8].

This result of [24] leads to a natural question: does the support for updates inherently increase the delay of enumeration algorithms? This is not always the case: e.g., when evaluating first-order queries (plus modulo-counting quantifiers) on bounded-degree databases, updates can be applied in constant time [12] and the delay is constant, as in the case without

updates [18, 22]. However, when evaluating conjunctive queries (CQs) on arbitrary databases, supporting updates has a cost: under complexity-theoretic assumptions, the class of CQs with efficient enumeration under updates [11] is a strict subclass of the class of CQs for the case without updates [9]. Could the same be true of MSO on trees, as [24] would suggest?

In this work, we answer this question in the negative, for a restricted update language. Specifically, we show an enumeration algorithm for MSO on trees with the same delay as in the case without updates [8], while supporting updates with a better complexity than [24] (see detailed comparison of results in Section 3). The tradeoff is that we only allow updates that change the labels of nodes, called *relabelings*, unlike [24] where updates can also insert and delete leaves. We still show how these relabelings are useful to evaluate practical query languages, such as *parameterized* queries and *group-by queries with aggregates*. A *parameterized* query allows the user to specify some parameters for the evaluation (e.g., select some positions on the tree). Our results support such queries: we can model the parameters as labels and apply relabeling updates when the user changes the parameters. A *group-by query with aggregates* partitions the set of results into groups based on an attribute, and computes some aggregate quantity on each group (e.g., a sum). We show how to enumerate the results of such queries. For groups, our techniques can handle them with one single enumeration structure using relabelings to switch groups. For aggregates, we can efficiently compute and maintain them in arbitrary semirings; this problem was left open by [24] even for counting, and is practically relevant in its own right [26]. Of course, by Courcelle's theorem [15], our results generalize to MSO queries on bounded-treewidth data (see [3]), where relabelings mean adding or removing unary facts (i.e., the tree decomposition is unchanged).

The proof of our main result follows the approach of [4] and is inspired by knowledge compilation in artificial intelligence and by factorized representations in database theory. Specifically, we encode knowledge (in our case, the query result) as a circuit in a restricted class, and we then use the circuit for efficient reasoning and for aggregates as in [17]. In [4], we have used this circuit-based approach to recapture existing enumeration results for MSO on trees [8, 23]. In this work, we refine the approach and show that it can support updates. Our key new ingredient are *hybrid circuits*: they have both *set-valued* gates that represent the values to enumerate, and *Boolean* gates that encode the tree labels which can be updated. We first show that we can efficiently compute such circuits to capture the possible results of an MSO query under all possible labelings of a tree. Second, we show how to efficiently enumerate the set of assignments captured by these circuits, also supporting updates that toggle the Boolean gates affected by a relabeling. We also introduce some standalone tools, e.g., a lemma to *balance* the input trees to MSO queries (Lemma 4.3), ensuring that hybrid circuits have logarithmic depth so that changes can be propagated quickly; and a constant-delay enumeration algorithm for reachability in forests under updates (Section 7).

Paper structure. We start with preliminaries in Section 2, and define our problem and give our main result in Section 3. In Section 4, we review the set-valued provenance circuits of [4], and show our balancing lemma. We introduce hybrid circuits in Section 5, and show in Section 6 how to use them for enumeration under updates, using a standalone reachability indexing scheme on forests given in Section 7. Having shown our main result, we outline its consequences for application-oriented query languages in Section 8 and conclude in Section 9.

2 Preliminaries

Trees, queries, answers, assignments. In this work, unless otherwise specified, a *tree* is always binary, rooted, ordered, and full. Let Γ be a finite set called a *tree alphabet*. A Γ-*tree* (T, λ) is a pair of a tree T and of a *labeling function* λ that maps each node n of T to a *set*

of labels $\lambda(n) \subseteq \Gamma$. We often abuse notation and identify T to its node set, e.g., write λ as a function from T to the powerset 2^Γ of Γ; we may also omit λ and write the Γ-tree as just T.

We consider queries in *monadic second-order logic* (MSO) on the *signature* of Γ-trees: it features two binary relations E_1 and E_2 denoting the first and second child of each internal node, and a unary relation P_l for each $l \in \Gamma$ denoting the nodes that carry label l (i.e., nodes n for which $l \in \lambda(n)$). MSO extends *first-order logic*, which builds formulas from atoms of this signature and from equality atoms, using the Boolean connectives and existential and universal quantification over nodes. Formulas in MSO can also use second-order quantification over sets of nodes, written as second-order variables. For instance, on $\Gamma = \{l_1, l_2, l_3\}$, we can express in MSO that every node carrying labels l_1 and l_2 has a descendant carrying label l_3.

In this work, we study MSO *queries*, i.e., MSO formulas with free variables. The free variables can be first-order or second-order, but we can rewrite any MSO query $Q(\mathbf{x}, \mathbf{Y})$ to ensure that all free variables are second-order: for instance as $Q'(\mathbf{X}, \mathbf{Y}) : \exists \mathbf{x} \bigwedge_i \mathrm{Sing}(X_i, x_i) \wedge Q(\mathbf{x}, \mathbf{Y})$, where $\mathrm{Sing}(X, x)$ asserts that X is exactly the singleton set $\{x\}$. Hence, we usually assume without loss of generality that MSO queries only have second-order free variables.

Given a Γ-tree T and an MSO query $Q(X_1, \ldots, X_m)$, an m-tuple $\mathbf{B} = B_1, \ldots, B_m$ of subsets of T is an *answer* of Q on T, written $T \models Q(\mathbf{B})$, if T satisfies $Q(\mathbf{B})$ in the usual logical sense. It will be more convenient to represent each answer as an *assignment*, which is a set of pairs called *singletons* that indicate that an element is in the interpretation of a variable. Formally, given an m-tuple $\mathbf{B}$ of subsets of T, the corresponding assignment is $\{\langle X_i : n\rangle \mid 1 \leq i \leq m \text{ and } n \in B_i\}$. We can convert each assignment in linear time to the corresponding answer and vice-versa, so we will use the assignment representation throughout this work. Our goal is to compute the set of assignments of Q on T, which we call the *output* of Q on T; we abuse notation and write it $Q(T)$. We measure the complexity of this task in *data complexity*, i.e., as a function of the input tree T, with the query Q being fixed.

Enumeration. The output of an MSO query can be huge, so we work in the setting of *enumeration algorithms* [31, 28] which we present following [4]. As usual for enumeration algorithms [28], we work in the RAM model with uniform cost measure (see, e.g., [1]), where pointers, numbers, labels for elements and facts, etc., have constant size.

An *enumeration algorithm with linear-time preprocessing* for a fixed MSO query $Q(\mathbf{X})$ on Γ-trees takes as input a Γ-tree T and computes the output $Q(T)$ of Q on T. It consists of two phases. First, the *preprocessing phase* takes T as input and produces in *linear time* a data structure J called the *index*, and an initial *state* s. Second, the *enumeration phase* repeatedly calls an algorithm $\mathcal{A}$. Each call to $\mathcal{A}$ takes as input the index J and the current state s, and returns one assignment and a new state s': a special state value indicates that the enumeration is over so $\mathcal{A}$ should not be called again. The assignments produced by the successive calls to $\mathcal{A}$ must be exactly the elements of $Q(T)$, with no duplicates.

We say that the enumeration algorithm has *linear delay* if the time to produce each new assignment A is linear in its cardinality $|A|$, and is independent of T. In particular, if all answers to Q are tuples of singleton sets (for instance, if Q is the translation of a MSO query where all free variables are first-order), then the cardinality of each assignment is constant (it is the arity of Q). In this case, the enumeration algorithm must produce each assignment with constant delay: this is called *constant-delay enumeration*. The *memory usage* of an enumeration algorithm is the maximum number of memory cells used during the enumeration phase (not counting the index J, which resides in read-only memory), expressed as a function of the size of the largest assignment (as in [8]): we say that the enumeration algorithm has *linear memory* if its memory usage is linear in the size of the largest assignment.

Previous works have studied enumeration for MSO on trees. Bagan [8] showed that for any fixed MSO query $Q(\mathbf{X})$, given a Γ-tree T, we can enumerate the output of Q on T with linear delay and memory, i.e., constant delay and memory when all free variables are first-order. This result was re-proven by Kazana and Segoufin [23] via a result of Colcombet [14], and a third proof via provenance circuits was recently proposed by the present authors [4].

3 Problem Statement and Main Result

Our goal is to address a limitation of these existing results, namely, the assumption that the input Γ-tree T will never change. Indeed, if T is updated, these results must discard the index J and re-run the preprocessing phase on the new tree. To improve on this, we want our enumeration algorithm to support *update operations* on T, and to update J accordingly instead of recomputing it from scratch. Specifically, an algorithm for *enumeration under updates* on a tree T has a preprocessing phase that produces the index J as usual, but has two algorithms during the enumeration phase: (i.) an enumeration algorithm $\mathcal{A}$ as presented before, and (ii.) an *update algorithm* $\mathcal{U}$. When we want to change the tree T, we call $\mathcal{U}$ with a description of the changes: $\mathcal{U}$ modifies T accordingly, updates the index J, and resets the enumeration state (so enumeration starts over on the new tree, and all working memory of the enumeration phase is freed). The *update time* of the enumeration algorithm is the complexity of $\mathcal{U}$: like preprocessing, but unlike delay, it is a function of the size of the (current) tree T.

To our knowledge, the only published result on enumeration for MSO queries under updates is the work of Losemann and Martens [24], which applies to words and to trees, for MSO queries with only free first-order variables. They show an enumeration algorithm with linear-time preprocessing: on words, the update complexity and delay is $O(\log |T|)$; on trees, these complexities become $O(\log^2 |T|)$. Thus the delay is worse than in the case without updates [8], and in particular it is no longer independent from T.

Main result. In this work, we show that enumeration under updates for MSO queries on trees can be performed with a better complexity that matches the case without updates: linear-time preprocessing, linear delay and memory (in the assignments), and update time in $O(\log |T|)$. This improves on the bounds of [24] (and uses entirely different techniques). However, in exchange for the better complexity, we only support a weaker update language: we can change the labels of tree nodes, called a *relabeling*, but we cannot insert or delete leaf nodes as in [24], which we leave for future work (see the conclusion in Section 9). We show in Section 8 that relabelings are still useful to derive results for some practical query languages.

Formally, a relabeling on a Γ-tree T is a pair of a node $n \in T$ and a label $l \in \Gamma$. To apply it, we change the label $\lambda(n)$ of n by adding l if $l \notin \lambda(n)$, and removing it if $l \in \lambda(n)$. In other words, the tree T never changes, and updates only modify λ. Our main result is then:

▶ **Theorem 3.1.** *For any fixed tree alphabet Γ and MSO query $Q(\mathbf{X})$ on Γ-trees, given a Γ-tree T, we can enumerate the output $Q(T)$ of Q on T with linear-time preprocessing, linear delay and memory, and logarithmic update time for relabelings.*

In other words, after preprocessing T in time $O(|T|)$ to compute the index J, we can:

- Enumerate the assignments of Q on T, using J, with delay linear in the size of each assignment, so constant if the assignments to Q have constant size.
- Toggle a label of a node of T, update J, and reset the enumeration, in time $O(\log |T|)$.

We show this result in Sections 4–7, and then give consequences of this result in Section 8.

4 Provenance Circuits

Our general technique for enumeration follows our earlier work [4]: from the query and input tree, we compute in linear time a structure called a *provenance circuit* to represent the results to enumerate, we observe that it falls in a restricted circuit class, and we conclude by showing a general enumeration result for circuits of this class. In this section, we review our construction of provenance circuits in [4], with some additional observations that will be useful for updates. In particular, we show an independent *balancing lemma* on input trees, which allows us to bound a parameter of the circuit called *dependency size*. We will extend the formalism of this section to so-called *hybrid circuits* in the next section; and we will show our enumeration result for such circuits in Sections 6 and 7.

Set circuits. We start with some preliminaries about circuits. A *circuit* $C = (G, W, g_0, \mu)$ is a directed acyclic graph (G, W) whose vertices G are called *gates*, whose edges W are called *wires*, where $g_0 \in G$ is the *output gate*, and where μ is a function giving a *type* to each gate of G (the possible types depend on the kind of circuit). The *inputs* to a gate $g \in G$ are $\mathrm{inp}(g) := \{g' \in G \mid (g', g) \in W\}$ and the *fan-in* of g is its number of inputs $|\mathrm{inp}(g)|$.

We define *set-valued circuits*, which are an equivalent rephrasing of the *circuits in zero-suppressed semantics* used in [4]. They can also be seen to be isomorphic to arithmetic circuits, and generalize factorized representations used in database theory [27]. The type function μ of a set-valued circuit maps each gate to one of $\cup$, $\times$, var. We require that $\times$-gates have fan-in 0 or 2, and that var-gates have fan-in 0: the latter are called the *variables* of C, with C_{var} denoting the set of variables. Each gate g of C *captures* a set $\mathrm{S}(g)$ of *assignments*, where each *assignment* is a subset of C_{var}. These sets are defined bottom-up as follows:

- For a variable gate g, we have $\mathrm{S}(g) := \{\{g\}\}$.
- For a $\cup$-gate g, we have $\mathrm{S}(g) := \bigcup_{g' \in \mathrm{inp}(g)} \mathrm{S}(g')$. In particular, if $\mathrm{inp}(g) = \emptyset$ then $\mathrm{S}(g) = \emptyset$.
- For a $\times$-gate g with no inputs, we have $\mathrm{S}(g) := \{\{\}\}$.
- For a $\times$-gate g with two inputs g_1 and g_2, we have $\mathrm{S}(g) := \{A_1 \cup A_2 \mid (A_1, A_2) \in \mathrm{S}(g_1) \times \mathrm{S}(g_2)\}$, which we write $\mathrm{S}(g) := \mathrm{S}(g_1) \times_{\mathrm{rel}} \mathrm{S}(g_2)$ (this is the relational product).

The set $\mathrm{S}(C)$ *captured* by C is $\mathrm{S}(g_0)$ for g_0 the output gate of C. Note that each assignment of $\mathrm{S}(C)$ is a satisfying assignment of C when seen in the usual semantics of monotone circuits.

Structural requirements. Before defining our provenance circuits, we introduce some structural restrictions that they will respect, and that will be useful for enumeration.

The first requirement is that the circuit is a *d-DNNF*. Our definition of d-DNNF is inspired by [16] but applies to set-valued circuits, as in [4] (see also the z-st-d-DNNFs of [30]). For each gate g of a set-valued circuit C, we define the *domain* $\mathsf{dom}(g)$ of g as the variable gates having a directed path to g. In particular, for $g \in C_{\mathrm{var}}$, we have $\mathsf{dom}(g) = \{g\}$, and if $\mathrm{inp}(g) = \emptyset$ then $\mathsf{dom}(g) = \emptyset$. We now call a $\times$-gate g *decomposable* if it has no inputs or if, letting $g'_1 \neq g'_2$ be its two inputs, the domains $\mathsf{dom}(g'_1)$ and $\mathsf{dom}(g'_2)$ are disjoint. This ensures that no variable of C occurs both in an assignment of $\mathrm{S}(g'_1)$ and in an assignment of $\mathrm{S}(g'_2)$. We call a $\cup$-gate g *deterministic* if, for any two inputs $g'_1 \neq g'_2$ of g, the sets $\mathrm{S}(g'_1)$ and $\mathrm{S}(g'_2)$ are disjoint, i.e., there is no assignment that occurs in both sets. We call C a *d-DNNF* if every $\times$-gate g is decomposable and every $\cup$-gate g is deterministic. Under this assumption, we can tractably compute the cardinality of the set $\mathrm{S}(C)$ captured by C.

The second requirement on circuits is called *upwards-determinism* and was introduced in [3]. In that paper, it was used to show an improved memory bound; in the present paper, we will always be able to enforce it. A wire (g, g') in a set-valued circuit C is called *pure* if:

- g' is a $\cup$-gate; or
- g' is a $\times$-gate and, letting g'' be the other input of g', we have $\{\} \in \mathrm{S}(g'')$, i.e., g'' captures the empty assignment.

We say that a gate g is *upwards-deterministic* if there is at most one gate g' such that (g, g') is pure. We call C *upwards-deterministic* if every gate of C is.

The third requirement concerns the *maximal fan-in* of circuits, which is simply defined for a set-valued circuit C as the maximal fan-in of a gate of C. We will require that the maximal fan-in is bounded by a constant.

The fourth and last requirement concerns a new parameter called *dependency size*. To introduce this, we define the *dependent gates* $\Delta(g')$ of a gate g' in a set-valued circuit C as the gates g such that there is a directed path from g' to g. Intuitively, the set $\mathrm{S}(g)$ captured by g may then depend on the set $\mathrm{S}(g')$ captured by g'. The *dependency size* of C is $\Delta(C) := \max_{g \in C} |\Delta(g)|$, i.e., the maximal number of gates that are dependent on any given gate g. We will require this parameter to be connected to the height of the input tree.

Set-valued provenance circuits. We can now define provenance circuits like in [4]. A set-valued circuit C is a *provenance circuit* of a MSO query $Q(X_1, \ldots, X_m)$ on a Γ-tree T if:

- The variables of C correspond to the possible singletons, formally: $C_{\mathrm{var}} = \{\langle X_i : n\rangle \mid 1 \leq i \leq m \text{ and } n \in T\}$; and
- The set of assignments captured by C is the output of Q on T, formally: $\mathrm{S}(C) = Q(T)$. Equivalently, for any tuple $\mathbf{B} = (B_1, \ldots, B_m)$ of subsets of T, we have $T \models Q(\mathbf{B})$ iff the assignment $\{\langle X_i : n\rangle \mid 1 \leq i \leq m \text{ and } n \in B_i\}$ is in $\mathrm{S}(C)$.

▶ **Example 4.1.** Consider the unlabeled tree T of Figure 1, the alphabet $\Gamma = \{B\}$, and the MSO query $Q(x)$ with one free first-order variable asking for the leaf nodes whose B-annotation is different from that of its parent (i.e., the node carries label B and the parent does not, or vice-versa). Consider the labeling λ mapping 1 to $\{B\}$ and 2 and 3 to $\emptyset$. A set-valued circuit capturing the provenance of Q on (T, λ) is given in Figure 2.

We then know from [3] that provenance circuits can be computed efficiently, and they can be made to respect our structural requirements:

▶ **Theorem 4.2** (from [4], Theorem 7.3)**.** *For any fixed MSO query $Q(\mathbf{X})$ on Γ-trees, given a Γ-tree T, we can compute in time $O(|T|)$ a set-valued provenance circuit C of Q on T. Further, C is a d-DNNF, it is upwards-deterministic, its maximal fan-in is constant, and its dependency size is in $O(\mathrm{h}(T))$, where* h *denotes the height of T.*

Proof sketch. We recall the main proof technique: we convert Q to a bottom-up deterministic tree automaton A on Γ-trees, and we add nodes to T to describe the possible valuations of variables. The provenance circuit C then captures the possible ways that A can read T depending on the valuation: we compute it with the construction of [6], and is a d-DNNF thanks to automaton determinism (see [2]). Upwards-determinism is shown like in [3].

The bounds on fan-in and dependency size are not stated in [4, 3] but already hold there. Specifically, the maximal fan-in is a function of the transition function of A, i.e., it does not depend on T. The bound on dependency size holds because C is constructed following the structure of T: we create for each tree node a gadget whose size depends only on A, and we connect these gadgets precisely following the structure of T, so that $\Delta(g)$ for any gate g of C can only contain gates from the node n of g or from ancestors of n in the tree. ◀

In the context of updates, the bound of dependency size will be crucial: intuitively, it describes how many gates need to be updated when an update operation modifies a gate

of the circuit. As this bound depends on the height of the input tree, we will conclude this section by a *balancing lemma* that ensures that this height can always be made logarithmic (which matches our desired update complexity). We will then add support for updates in the next section by extending circuits to *hybrid circuits*.

Balancing lemma. Our balancing lemma is a general observation on MSO query evaluation on trees, and is in fact completely independent from provenance circuits. It essentially says that the input tree can be assumed to be balanced. Formally, we will show that we can rewrite any MSO query Q on Γ-trees to an MSO query Q' on a larger tree alphabet Γ' so that any input tree T for Q can be rewritten in linear time to a balanced tree T' on which Q' returns exactly the same output. Because we intend to support update operations, the input tree T will be unlabeled, and the rewritten tree T' will work for any labeling of T. Formally:

▶ **Lemma 4.3.** *For any tree alphabet Γ and MSO query $Q(\mathbf{X})$ on Γ-trees, we can compute a tree alphabet $\Gamma' \supseteq \Gamma$ and MSO query $Q'(\mathbf{X})$ on Γ'-trees such that the following holds. Given any unlabeled tree T with node set N, we can compute in linear time a Γ'-tree (T', λ') with node set $N' \supseteq N$, such that $\mathrm{h}(T') = O(\log |T|)$ and such that, for any labeling function $\lambda : T \to 2^\Gamma$, we have $Q(\lambda(T)) = Q'(\lambda''(T'))$, where $\lambda''(n)$ maps $n \in T'$ to $\lambda(n)$ if $n \in T$ and $\lambda'(n)$ otherwise.*

Proof sketch. We prove Lemma 4.3 by seeing the input tree T as a relational structure I of treewidth 1, and invoking the result by Bodlaender [13] to compute in linear time a constant-width tree decomposition of I which is of logarithmic height. We then translate the query Q to a MSO query Q' on tree encodings of this width, and compute from T the tree encoding T' corresponding to the tree decomposition (we rename some nodes of T' to ensure that the nodes of T are reflected in T'). Note that the balanced tree decompositions of [13] were already used for similar purposes elsewhere, e.g., in [19], end of Section 2.3. ◀

5 Hybrid Circuits for Updates

In this section, we extend set-valued circuits to support updates, defining *hybrid circuits*. We then extend Theorem 4.2 for these circuits. Last, we introduce a new structural notion of *homogenization* of hybrid circuits and show how to enforce it. We close the section by stating our main enumeration result on hybrid circuits, which implies our main theorem (Theorem 3.1), and is proved in the two next sections.

Hybrid circuits. A hybrid circuit is intuitively similar to a set-valued circuit, but it additionally has *Boolean variables* (which can be toggled when updating), Boolean gates ($\wedge$, $\vee$, $\neg$), and gates labeled $\boxtimes$ which keep or discard a set of assignments depending on a Boolean value. Formally, a *hybrid circuit* $C = (G, W, g_0, \mu)$ is a circuit where the possible gate types are svar (*set-valued variables*), bvar (*Boolean variables*), $\cup$, $\times$, $\boxtimes$, $\wedge$, $\vee$, and $\neg$. We call a gate *Boolean* if its type is bvar, $\wedge$, $\vee$, or $\neg$; and *set-valued* otherwise. We require that the output gate g_0 is set-valued and that the following conditions hold:

- svar-gates and bvar-gates have fan-in exactly 0;
- All inputs to $\wedge$-gates, $\vee$-gates, and $\neg$-gates are Boolean, and $\neg$-gates have fan-in exactly 1;
- All inputs to $\cup$ and $\times$-gates are set-valued, and $\times$-gates have fan-in either 0 or 2;
- $\boxtimes$-gates have one set-valued input and one Boolean input (so they have fan-in exactly 2).

We write C_{bvar} to denote the gates of C of type bvar, called the *Boolean variables*, and define likewise the *set-valued variables* C_{svar}. An example hybrid circuit is illustrated in Figure 3.

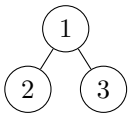


Figure 1 Example unlabeled tree.

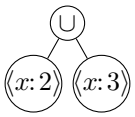


Figure 2 Example set circuit.

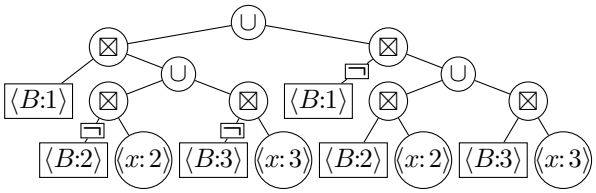


Figure 3 Example hybrid circuit. Boolean gates are squared, set-valued gates are circled, and variables are repeated.

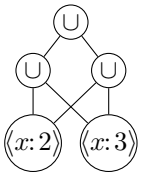


Figure 4 Example switchboard.

Unlike set-valued circuits, which capture only one set of assignments, hybrid circuits capture several different sets of assignments, depending on the value of the Boolean variables (intuitively corresponding to the tree labels). This value is given by a *valuation* of C, i.e., a function $\nu : C_{\text{bvar}} \to \{0, 1\}$. Given such a valuation ν, each Boolean gate g *captures* a Boolean value $\mathrm{V}_\nu(g) \in \{0, 1\}$, computed bottom-up in the usual way: we set $\mathrm{V}_\nu(g) := \nu(g)$ for $g \in C_{\text{bvar}}$, and otherwise $\mathrm{V}_\nu(g)$ is the result of the Boolean operation given by the type $\mu(g)$ of g, applied to the Boolean values $\mathrm{V}_\nu(g')$ captured by the inputs g' of g (in particular, a $\wedge$-gate with no inputs always has value 1, and a $\vee$-gate with no inputs always has value 0).

We then define the *evaluation* of C under ν as the set-valued circuit $\nu(C)$ obtained as follows. First, replace each Boolean gate g of C by a $\times$-gate with no inputs (capturing $\{\{\}\}$) if $\mathrm{V}_\nu(g) = 1$, and by a $\cup$-gate with no inputs (capturing $\emptyset$) if $\mathrm{V}_\nu(g) = 0$. Second, relabel each $\boxtimes$-gate g of C to be a $\times$-gate. Using $\nu(C)$, for each set-valued gate g of C, we define the set *captured by g under ν*: it is the set of assignments (subsets of C_{svar}) that g captures in $\nu(C)$. The set $\mathrm{S}_\nu(C)$ *captured* by C under ν is then $\mathrm{S}_\nu(g_0)$, for g_0 the output gate of C.

We last lift the structural definitions from set-valued circuits to hybrid circuits. The *maximal fan-in* and *dependency size* of a hybrid circuit are defined like before (these definitions do not depend on the kind of circuit). A hybrid circuit C is a *d-DNNF*, resp. is *upwards-deterministic*, if for every valuation ν of C, the set-valued circuit $\nu(C)$ has the same property. For instance, the hybrid circuit in Figure 3 is upwards-deterministic and is a d-DNNF.

Hybrid provenance circuits. We can now use hybrid circuits to define provenance with support for updates. The set-valued variables of the circuit will correspond to singletons as before, describing the interpretation of the *free variables* of the query; and the Boolean variables stand for a different kind of singletons, describing which *labels* are carried by each node. To describe this formally, we will consider an *unlabeled* tree T, and define a *labeling assignment* of T for a tree alphabet Γ as a set of singletons of the form $\langle l : n \rangle$ where $l \in \Gamma$ and $n \in T$. Given a labeling assignment α, we can define a labeling function λ_α for T, which maps each node $n \in T$ to $\lambda(n) := \{l \in \Gamma \mid \langle l : n \rangle \in \alpha\}$. Now, we say that a hybrid circuit C is a *provenance circuit* of a MSO query $Q(X_1, \ldots, X_m)$ on an *unlabeled* tree T if:

- The set-valued variables of C correspond to the possible singletons in an assignment, formally $C_{\text{svar}} = \{\langle X_i : n \rangle \mid 1 \leq i \leq m \text{ and } n \in T\}$;
- The Boolean variables of C correspond to the possible singletons in an update assignment, formally $C_{\text{bvar}} = \{\langle l : n \rangle \mid l \in \Gamma \text{ and } n \in T\}$;

- For any labeling assignment α, let ν_α be the Boolean valuation of C_{bvar} mapping each $\langle l : n\rangle$ to 0 or 1 depending on whether $\langle l : n\rangle \in \alpha$ or not, and let λ_α be the labeling function on T defined as above. Then we require that the set of assignments $\mathrm{S}_{\nu_\alpha}(C)$ captured by C under ν_α is exactly the output of Q on $\lambda_\alpha(T)$, formally, $\mathrm{S}_{\nu_\alpha}(C) = Q(\lambda_\alpha(T))$.

In other words, for each labeling λ of the tree T, considering the valuation ν that sets the Boolean variables of C accordingly, then $\nu(C)$ is a provenance circuit for Q on $\lambda(T)$.

▶ **Example 5.1.** Recall the query $Q(x)$ and alphabet $\Gamma = \{B\}$ of Example 4.1, and the tree T of Figure 1. A hybrid circuit C capturing the provenance of Q on T is given in Figure 3 (with variable gates being drawn at multiple places for legibility): square leaves correspond to Boolean variables testing node labels, and circle leaves correspond to set-valued variables capturing a singleton of the form $\langle x{:}n\rangle$ for some $n \in T$. In particular, for the labeling λ of Example 4.1, the corresponding valuation ν maps $\langle B{:}1\rangle$ to 1 and $\langle B{:}2\rangle$ and $\langle B{:}3\rangle$ to 0, and the evaluation $\nu(C)$ of C under ν captures the same set as the circuit of Figure 2.

We can now extend Theorem 4.2 to compute a hybrid provenance circuit as follows:

▶ **Theorem 5.2.** *For any fixed MSO query $Q(\mathbf{X})$ on Γ-trees, given an unlabeled tree T, we can compute in time $O(|T|)$ a hybrid provenance circuit C which is a d-DNNF, is upwards-deterministic, has constant maximal fan-in, and has dependency size in $O(\mathrm{h}(T))$.*

Proof sketch. The proof is analogous to that of Theorem 4.2. The only difference is that the automaton now reads the label of each node as if it were a variable, so that the provenance circuit C also reflects these label choices as Boolean variables. ◀

Homogenization. We will make enumeration simpler by imposing one last requirement on hybrid circuits. A hybrid circuit C is *homogenized* if there is no valuation ν of C and set-valued gate g of C such that $\{\} \in \mathrm{S}_\nu(g)$. Note that the requirement does not apply to the Boolean gates of C, nor to the gates that replace them in evaluations $\nu(C)$ of C, so it equivalently means that C does not contain $\times$-gates with no inputs. Intuitively, set-valued gates in C that capture the empty assignment would waste time in the enumeration. We will show that we can rewrite circuits in linear time to make them homogenized, while preserving our requirements; but we need to change our definitions slightly to ensure that the circuit can still capture the empty assignment overall. To do so, we add the possibility of distinguishing a Boolean gate g_1 of a hybrid circuit C as its *secondary output*; in this case, given a valuation ν of C, the set $\mathrm{S}_\nu(C)$ captured by C under ν is $\mathrm{S}_\nu(C)$ plus the empty assignment $\{\}$ if the secondary output g_1 evaluates to 1, i.e., if $\mathrm{V}_\nu(g_1) = 1$. We say that two hybrid circuits C and C' (with or without secondary outputs) are *equivalent* if $C_{\text{bvar}} = C'_{\text{bvar}}$, $C_{\text{svar}} = C'_{\text{svar}}$, and for any valuation ν of C, we have $\mathrm{S}_\nu(C) = \mathrm{S}_\nu(C')$. We then have:

▶ **Lemma 5.3.** *For any hybrid circuit C, we can build in linear time a hybrid circuit C' with a secondary output g_1, such that C' is homogenized and it is equivalent to C. Further, if C is a d-DNNF and is upwards-deterministic, then so is C'; if C has bounded fan-in then the same holds of C'; and we have $\Delta(C') = O(\Delta(C))$.*

Proof sketch. This is shown analogously to homogenization in [4], which follows the technique of Strassen [29] (only done for two "layers", namely, empty and non-empty assignments). ◀

Hence, up to linear-time processing, we can additionally assume that the circuits of Theorem 5.2 are homogenized. We can now use this theorem, the lemma above, and Lemma 4.3, to reduce enumeration for MSO on trees (as in our main theorem, Theorem 3.1) to the task of enumerating the set captured by a hybrid circuit satisfying some structural properties. The result that we need is the following (we prove it in the next two sections):

▶ **Theorem 5.4.** *Given an upwards-deterministic, d-DNNF, homogenized hybrid circuit C with constant fan-in, given an initial Boolean valuation ν of C_{bvar}, there is an enumeration algorithm with linear-time preprocessing to enumerate the set $\mathrm{S}_\nu(C)$ captured by C under ν, with linear delay and memory in each produced assignment, and with update time in $O(\Delta(C))$: an* update *consists here of toggling one value in ν.*

6 Enumerating Assignments of Hybrid Circuits

In this section and the next, we prove Theorem 5.4 by giving an algorithm for enumeration under updates. We start by describing the preprocessing phase, computing two simple structures: a *shortcut function* and a *partial evaluation*; we also explain how this index can be efficiently updated. We then describe an algorithm for the enumeration phase, which needs an additional index structure to achieve the required delay. We close the section by presenting the missing index, called a *switchboard*. The switchboard must support a kind of reachability queries with a specific algorithm for enumeration under updates: we give a self-contained presentation of this scheme in the next section.

Preprocessing phase: shortcuts and partial evaluation. The first index structure that we precompute on our hybrid circuit C consists of a *shortcut function* to avoid wasting time in chains of $\boxtimes$-gates. For each $\boxtimes$-gate g, we precompute the one set-valued gate, called $\delta(g)$ which is not a $\boxtimes$-gate and which has a directed path to g going only through $\boxtimes$-gates. The function δ can clearly be computed in a linear-time bottom-up pass during the preprocessing, and it will never need to be updated (it does not depend on ν). For notational convenience, we extend δ by setting $\delta(g) := g$ for any set-valued gate g which is not a $\boxtimes$-gate.

The second index structure that we precompute is a *partial evaluation*, which depends on the valuation ν: it is a function ω_ν from the gates of C to $\{0, 1\}$ satisfying the following:

- For every Boolean gate g, we have $\omega_\nu(g) = \mathrm{V}_\nu(g)$.
- For every set-valued gate g, we have $\omega_\nu(g) = 1$ iff $\mathrm{S}_\nu(g)$ is non-empty.

The function ω_ν is intuitively an evaluation of the Boolean gates in the circuit, extended to the set-valued gates to determine whether their set is empty or not. We can easily compute ω_ν bottom-up from ν. Further, whenever ν is changed on a Boolean variable gate g, we can update ω_ν by recomputing it bottom-up on $\Delta(g)$. Formally:

▶ **Lemma 6.1.** *Given a hybrid circuit C of constant fan-in, given a valuation ν of C, we can compute ω_ν in linear time from ν and C. Further, for any $g \in C_{\mathrm{bvar}}$, letting ν' be the result of toggling the value of ν on g, we can update ω_ν to $\omega_{\nu'}$ in time $O(\Delta(g))$.*

Hence, we can compute ω_ν and δ in the preprocessing and maintain them under updates.

Enumeration phase. We can use the shortcut function and partial evaluation to enumerate the assignments in the set $\mathrm{S}_\nu(C)$ of our hybrid circuit C. Of course, if $\omega_\nu(g_0) = 0$ then we detect in constant time that there is nothing to enumerate. Otherwise, the enumeration scheme proceeds essentially like in [4]; to achieve the right delay bounds, it will need an additional index that we will present later. We start by enumerating $\mathrm{S}_\nu(g_0)$, and describe what happens when we try to enumerate $\mathrm{S}_\nu(g)$ for a set-valued gate g; we will always ensure that $\omega_\nu(g) = 1$. The base case is when g is a set-valued variable, in which case the only assignment to enumerate is $\{g\}$. There are three induction cases: $\times$-gates, $\boxtimes$-gates, and $\cup$-gates.

First, assume that g is a $\times$-gate. As C is homogenized, g has two inputs g_1 and g_2. Then we have $\mathrm{S}_\nu(g) = \mathrm{S}_\nu(g_1) \times_{\mathrm{rel}} \mathrm{S}_\nu(g_2)$. Hence, we can simply enumerate $\mathrm{S}_\nu(g)$ as the lexicographic product of $\mathrm{S}_\nu(g_1)$ and $\mathrm{S}_\nu(g_2)$. In particular, as $\omega_\nu(g) = 1$, we have $\omega_\nu(g_1) = \omega_\nu(g_2) = 1$, so neither set is empty. Formally, we have the following lemma:

▶ **Lemma 6.2.** *For any $\times$-gate g with inputs g_1 and g_2, if we can enumerate $\mathrm{S}_\nu(g_1)$ and $\mathrm{S}_\nu(g_2)$ with delay and memory respectively θ_1 and θ_2, then we can enumerate $\mathrm{S}_\nu(g)$ with delay and memory $\theta_1 + \theta_2 + c$ for some constant c.*

Note that the constant c paid at the $\times$-gate is not a problem to achieve linear delay and memory, because it is paid at most $n - 1$ times when enumerating an assignment A of size n. Indeed, C is homogenized, so A is always split non-trivially at each $\times$-gate, and g is decomposable in $\nu(C)$, so the two sub-assignments never share any variable.

Second, assume that g is a $\boxtimes$-gate. As $\omega_\nu(g) = 1$, we clearly have $\mathrm{S}_\nu(g) = \mathrm{S}_\nu(\delta(g))$. Hence, we can simply follow the pointer to $\delta(g)$ and enumerate $\mathrm{S}_\nu(\delta(g))$. Intuitively, the cost of this operation can be covered by that of g, because $\delta(g)$ can no longer be a $\boxtimes$-gate.

▶ **Lemma 6.3.** *For any $\boxtimes$-gate g, if we can enumerate $\mathrm{S}_\nu(\delta(g))$ with delay and memory θ, then we can enumerate $\mathrm{S}_\nu(g)$ with delay and memory $\theta + c$ for some constant c.*

Third, assume that g is a $\cup$-gate g. Naively, we can enumerate $\mathrm{S}_\nu(g)$ as the union of the $\mathrm{S}_\nu(g')$ for the inputs g' of g for which $\omega_\nu(g') = 1$ (this union is disjoint thanks to determinism). This is correct, but does not satisfy the delay bounds, because g' may be another $\cup$-gate. A more clever scheme is to to "jump" to the $\times$-gate or set-valued variable gates on which $\mathrm{S}_\nu(g)$ depends. Let us accordingly call *exits* the gates of these two types. The set $\mathrm{S}_\nu(g)$ can then be expressed as a union of $\mathrm{S}_\nu(g')$ for the exits g' that have a directed path of $\cup$-gates and $\boxtimes$-gates to g. We introduce definitions to "collapse" these paths.

The first definition collapses paths of $\boxtimes$-gates. There is a *$\boxtimes$-path* from a set-valued gate g' to a set-valued gate $g \neq g'$, written $g' \to^*_\boxtimes g$, if there is a directed path $g' = g_1 \to \cdots \to g_n = g$ in C such that $g_2, \ldots, g_{n-1}$ are all $\boxtimes$-gates. In particular, a wire (g', g) between set-valued gates implies $g' \to^*_\boxtimes g$ (take $n = 2$), and $\delta(g) \to^*_\boxtimes g$ whenever $\delta(g) \neq g$. When g is a $\cup$-gate, there are two cases, depending on ν. First, we may have $\omega_\nu(g_{n-1}) = 1$, and then $\omega_\nu(g') = 1$ and $\mathrm{S}_\nu(g')$ contributes to $\mathrm{S}_\nu(g)$: we call the path *live under ν*. Second, we may have $\omega_\nu(g_{n-1}) = 0$, and then $\mathrm{S}_\nu(g')$ does not contribute to $\mathrm{S}_\nu(g)$ via this path.

The second definition collapses paths of $\cup$-gates. An *$\cup$-path* from a set-valued gate g' to a set-valued gate $g \neq g'$ is a sequence $g' = g_1 \to^*_\boxtimes \cdots \to^*_\boxtimes g_n = g$ in C, where $g_2, \ldots, g_{n-1}$ are all $\cup$-gates and there is a $\boxtimes$-path between any two consecutive gates. The path is *live under ν* if there is a *live* $\boxtimes$-path under ν between any two consecutive gates.

We now use these definitions to express $\mathrm{S}_\nu(g)$ as a function of the set of *exits under ν* of g in C, written D^ν_g, which is the set of exits g' having a live $\cup$-path to g under ν in C:

▶ **Lemma 6.4.** *For any valuation ν and $\cup$-gate g, we have $\mathrm{S}_\nu(g) = \bigcup_{g' \in D^\nu_g} \mathrm{S}_\nu(g')$. Further, this union is disjoint and all its terms are nonempty.*

Hence, we can enumerate $\mathrm{S}_\mu(g)$ for a $\cup$-gate g by enumerating D^ν_g and the set $\mathrm{S}_\nu(g')$ for each g' in D^ν_g. Note that g' is an exit, i.e., a variable or a $\times$-gate; so we make progress.

▶ **Lemma 6.5.** *For any $\cup$-gate g, if we can enumerate D^ν_g with delay and memory c, and can enumerate $\mathrm{S}_\nu(g')$ for every $g' \in D^\nu_g$ with delay and memory θ, then we can enumerate $\mathrm{S}_\nu(g)$ with delay and memory $\theta + c + c'$ for some constant c'.*

We have described our enumeration scheme in Lemmas 6.2, 6.3, and 6.5. The only missing piece is to enumerate, for each $\cup$-gate g, the set D_g^ν of exits under ν of g, with *constant* delay and memory. To do so, we will need additional preprocessing. We will rely on upwards-determinism, and extend the tree-based index of [3] to support updates. We first present an additional structure, called the *switchboard*, that we compute in the preprocessing; and we explain in the next section an indexing scheme that we perform on this structure.

Switchboard. Our third index component in the preprocessing is called the *switchboard*. It consists of a directed graph $B = (V, E)$ called the *panel*, which does not depend on ν (so it does not need to be updated), and a valuation $\beta_\nu : E \to \{0, 1\}$ called the *wiring*. The *panel* $B = (V, E)$ is defined as follows: V consists of all $\cup$-gates, $\times$-gates, and svar-gates, and $E \subseteq V \times V$ contains the edge $(\delta(g'), g)$ for each wire (g', g) of C such that g is a $\cup$-gate. This implies that the maximal fan-in of B is no greater than that of C, and it implies that B is a DAG. The *wiring* β_ν maps every edge (g', g) of B to 1 if there is a $\boxtimes$-path from g' to g in C which is live under ν, and 0 otherwise. We can use ω_ν to compute the switchboard, and to update it in time $O(\Delta(C))$ whenever ν is updated by toggling a gate of C_{bvar}. Formally:

▶ **Lemma 6.6.** *The switchboard can be computed in linear time given C and ν, and we can update it in time $O(\Delta(C))$ when toggling any gate in ν.*

We now explain how we use the switchboard to enumerate, given a $\cup$-gate g, the set D_g^ν of the exits g' having a *live* $\cup$-path to g under ν. In terms of the switchboard, we must enumerate the exits g' that have a path to g in B whose edges are all mapped to 1 by β_ν. Hence, we must solve the following enumeration task on the switchboard: letting $\beta_\nu(B)$ be the DAG of edges of B mapped to 1 by β_ν, we are given a gate g of B, and we must enumerate all exit gates g' of B (i.e., the $\times$-gates or svar-gates) that have a directed path to g in $\beta_\nu(B)$. Further, we must be able to handle updates on $\beta_\nu(B)$, as given by updates on ν. Fortunately, thanks to upwards-determinism, this problem is easier than it looks:

▶ **Claim 6.7.** *For any valuation ν of the hybrid circuit C, the DAG $\beta_\nu(B)$ is a forest.*

▶ **Example 6.8.** Figure 4 describes the switchboard for the hybrid circuit C of Figure 3. The edges of the switchboard correspond to $\boxtimes$-paths. The switchboard itself is not a forest; however, for every valuation of C, the $\boxtimes$-paths that are live must always form a forest.

Thus, what we need is a constant-delay reachability index on forests that can be updated efficiently when adding and removing edges to the forest. This is the focus of the next section.

7 Reachability Indexing under Updates

In this section, we present our indexing scheme for reachability on forests under updates. The construction in this section is independent from what precedes. For convenience, we will orient the edges of the forest downwards, i.e., the reverse of the previous section (so g is the parent of g' in the forest if there is an edge from g' to g in the switchboard). We first define the problem and state the enumeration result, and then sketch the proof.

Definitions and main result. A *reachability forest* $F = (V, E, X)$ is a directed graph (V, E) where V is the *vertex set*, $E \subseteq V \times V$ are the *edges*, and $X \subseteq V$ is a subset of vertices called *exits*. When $(v, v') \in E$, we call v a *parent* of v', and v' a *child* of v. We impose three requirements on F: (i.) the graph (V, E) is a forest, i.e., each vertex of V has at most one parent; (ii.) there is a constant *degree bound* $c \in \mathbb{N}$ such that every vertex has at most c

children; (iii.) every exit $v \in X$ is a *leaf*, i.e., a vertex with no children. We will call *trees* the connected components of F. For convenience, we assume that F is *ordered*, i.e., there is some total order $<$ on the children of every node.

Given a reachability forest $F = (V, E, X)$ and a vertex $v \in V$, we write $\mathrm{reach}(v)$ for the set of exits reachable in F from v, i.e., the vertices of X to which v has a directed path. These are the sets that we wish to enumerate efficiently, allowing two kinds of *updates* on the edges E of F. First, a *delete operation* is written $-E'$ for a set $E' \subseteq E$, and $F = (V, E, X)$ is updated to $F - E' := (V, E \setminus E', X)$; it is still a reachability forest. Second, an *insert operation* is written $+E'$ for some $E' \subseteq V \times V$, and we require that the update result $F + E' := (V, E \cup E', X)$ still satisfies the three requirements above (with the same degree bound). In terms of the order $<$ on children, when we remove edges, we take the restriction of $<$ in the expected way, and when we insert edges, we add each new child at an arbitrary position in $<$. We then introduce *ancestry* to measure the impact of updates (analogously to dependency size): the *ancestry* $\mathcal{A}_F(v)$ of $v \in V$ is the set of vertices of F that have a directed path to v, and the *ancestry* $\mathcal{A}_F(E')$ for $E' \subseteq V \times V$ is $\bigcup_{(v,w)\in E'} \mathcal{A}_F(v)$. We then have:

▶ **Theorem 7.1.** *Given a reachability forest F, there is an enumeration algorithm with linear-time preprocessing such that: (i.) given any $v \in V$, we can enumerate* $\mathrm{reach}(v)$ *with constant delay and memory; (ii.) given an update $\pm E'$, we can apply it (replacing F by $F \pm E'$ and updating the index) with update time in $O(\mathcal{A}_F(E'))$.*

Note how we can insert (or delete) many edges at the same time, paying only once the price $\mathcal{A}_F(E')$: this point is used in the proof of Theorem 5.4 to bound the total cost of each update on the circuit. We sketch the proof of Theorem 7.1 in the rest of this section.

Construction for Theorem 7.1. Our index structure follows the one used to prove Proposition F.4 of [3]: it maps every $v \in V$ to a pointer $\mathsf{first}_F(v)$ and a pointer $\mathsf{last}_F(v)$, called the first and last *pointer*; and maps every exit $v \in X$ to a pointer $\mathsf{next}_F(v)$ called the next *pointer*. These pointers are defined using the order $<'$ given by a preorder traversal of F following $<$. Specifically, $\mathsf{first}_F(v)$ is the first exit $v' \in \mathrm{reach}_F(v)$ according to $<'$, and $\mathsf{last}_F(v)$ is the last such exit; if $\mathrm{reach}_F(v) = \emptyset$ then both pointers are null. Now, $\mathsf{next}_F(v)$ for $v \in X$ is the exit $v' \in X$ in the tree of v which is the successor of v according to $<'$; if v is the last exit of its tree, then $\mathsf{next}_F(v)$ is null. If we know these pointers, we can enumerate $\mathrm{reach}_F(v)$ for any $v \in V$ with constant delay and memory as in [3]: if $\mathsf{first}_F(v)$ is null then there is nothing to enumerate, otherwise start at $v_- := \mathsf{first}_F(v)$, memorize $v_+ := \mathsf{last}_F(v)$, and enumerate the reachable exits following the next pointers from v_- until reaching v_+. Hence, to conclude the proof of Theorem 7.1, it suffices to compute and update these pointers efficiently:

▶ **Lemma 7.2.** *Given a reachability forest F, we can compute the* first, last, *and* next *pointers of all vertices in time $O(|F|)$. Further, for any update $\pm E'$, we can apply it and update the pointers in time $O(\mathcal{A}_F(E'))$.*

Proof sketch. The first and last pointers are computed bottom-up in linear time: for a leaf v, they either point to v if $v \in X$ or to null otherwise; for an internal vertex v, we set $\mathsf{first}_F(v)$ as $\mathsf{first}_F(v')$ for the smallest child v' of v in the order $<'$ with a non-null first pointer (or null if all first pointers of children are null), and we set $\mathsf{last}_F(v)$ analogously, using the last pointer of the largest child of v in the order $<'$ for which the last pointer is non-null. Further, given an update $\pm E'$, the first and last pointers need only to be updated in $\mathcal{A}_F(E')$, and we can recompute them there with the same bottom-up scheme.

The next pointers are also computed bottom-up in linear time: at each internal vertex v, we go over its children and stitch together the sequences of next pointers of their subtrees.

Specifically, when $\mathsf{last}_F(v_1)$ is not null for a child v_1, we find the next child v_2 for which $\mathsf{first}_F(v_2)$ is not null, and set $\mathsf{next}_F(\mathsf{last}_F(v_1)) := \mathsf{first}_F(v_2)$. Again, for an update $\pm E'$, we recompute the next pointers by processing $\mathcal{A}_F(E')$ bottom-up in a similar fashion. ◀

8 Applications

We have finished the proof of our main result (Theorem 3.1), and now explain how it applies to query languages motivated by applications. Specifically, we show how to extend our techniques to support *aggregate* queries in arbitrary semirings, following the ideas of semiring provenance [21] and provenance circuits [17]. We then extend this to *group-by queries*, and last explain how updates are useful to support *parameterized queries*. Throughout this section, unlike the rest of the paper, we only study MSO queries with free first-order variables.

Aggregate queries. We will describe aggregation operators using a general structure called a *semiring* (always assumed to be commutative). It consists of a *set* K (finite or infinite), two binary operations $\oplus$ and $\otimes$, and distinguished elements $0_K, 1_K \in K$. We require that $(K, \oplus)$ and $(K, \otimes)$ are commutative monoids with neutral elements respectively 0_K and 1_K; that $\otimes$ distributes over $\oplus$, and that 0_K is absorptive for $\otimes$, i.e., $0_K \otimes a = 0_K$ for all $a \in K$. We always assume that evaluating $\oplus$ or $\otimes$ take constant time, and that elements from K take constant space. Examples of semirings include the natural numbers $\mathbb{N}$ with usual addition and product (assumed to take unit time in the RAM model); or the *security semiring* [20], the *tropical semiring* [17], etc. Note that sets of assignments with union and relational product are also a semiring, but one that does not satisfy our constant-space assumption.

To define aggregation in a semiring K on a tree T, we consider a mapping $\rho : T \to K$ giving a value in K to each node. We extend ρ to tuples $\mathbf{b}$ of T by setting $\rho(\mathbf{b}) := \bigotimes_{n \in \mathbf{b}} \rho(n)$; to assignments A on some first-order variable set $\mathbf{x}$ by setting $\rho(A) := \bigotimes_{\langle x_i : n \rangle \in A} \rho(n)$; and to sets S of assignments by setting $\rho(S) := \bigoplus_{A \in S} \rho(A)$. An *aggregate query* on Γ-trees consists of a semiring K (satisfying our assumptions) and of a MSO query $Q(\mathbf{x})$ on Γ-trees. Given a Γ-tree T and a mapping $\rho : T \to K$, the *aggregate output* $Q_\rho(T)$ of Q on T under ρ is $\rho(Q(T))$, where $Q(T)$ is the output of Q on T as we studied so far, i.e., the set of assignments A such that $T \models Q(A)$. Aggregate MSO queries on trees were already studied, e.g., by Arnborg and Lagergren [7], but our techniques allow us to handle updates:

▶ **Theorem 8.1.** *For any aggregate query $Q(\mathbf{x})$ on Γ-trees with semiring K, given a Γ-tree T and mapping $\rho : T \to K$, we can compute $Q_\rho(T)$ in time $O(|T|)$, and recompute it in time $O(\log |T|)$ after any update that relabels a node of T or that changes $\rho(n)$ for a node n of T.*

Proof sketch. We adapt hybrid circuits by replacing set-valued gates by K-valued gates. Now, the set $\mathrm{S}_\nu(g)$ captured by a gate g under a Boolean valuation ν is an element of K, so we can simplify our linear-time preprocessing by making ω_ν compute exactly $\mathrm{S}_\nu(g)$ for each gate g. We can then handle updates to ν as before, and handle updates to ρ by recomputing ω_ν bottom-up. All of this still relies on the balancing lemma (Lemma 4.3). ◀

One important application of this result is *maintaining* the number of query answers under updates, a question left open by [24]. We answer the question for relabeling updates (and in the set semantics), using the semiring $\mathbb{N}$ and mapping each node to 1 with ρ:

▶ **Corollary 8.2.** *For any MSO query $Q(\mathbf{x})$ on Γ-trees, given a Γ-tree T, we can compute the number $|Q(T)|$ of answers of Q on T in time $O(|T|)$, and we can update it in time $O(\log |T|)$ after a relabeling of T.*

However, we can also use Theorem 8.1 for more complex aggregation semirings:

▶ **Example 8.3.** Let $\Gamma = \{A, B\}$, let $Q(x)$ be a MSO query with one variable that selects some tree nodes (e.g., select the B-labeled nodes which are descendants of some A-labeled node), let (T, λ) be a Γ-tree, and let χ be a function that maps each node of T to an element of the set $\mathbb{D}$ of floating-point numbers (with fixed precision). We can compute in linear time the *average* of $\chi(n)$ for the nodes n such that $T \models Q(n)$, and update it in logarithmic time when relabeling a node of T or changing a value of χ. This follows from Theorem 8.1: we use the semiring of pairs in $\mathbb{N} \times \mathbb{D}$ and the mapping $\rho : n \mapsto (1, \chi(n))$ to compute and maintain the number of selected nodes and the sum of their χ-images, from which we can deduce the average in constant time.

Group-by. We have adapted our techniques to show results for aggregate queries under updates. However, supporting updates is also useful for *group-by queries*. A *group-by query* consists of a MSO query $Q(\mathbf{x}, \mathbf{y})$ on Γ-trees with two tuples of first-order variables, and of a semiring K. A *group* on a Γ-tree T is a set of tuples $\mathcal{G}(\mathbf{b}) := \{(\mathbf{b}, \mathbf{c}) \mid T \models Q(\mathbf{b}, \mathbf{c})\}$ for some tuple $\mathbf{b}$ of nodes of T. The *output* $Q_\rho(T)$ of Q on T under a mapping $\rho : T \to K$ contains one pair $(\mathbf{b}, \rho(\mathcal{G}(\mathbf{b})))$ for each tuple $\mathbf{b}$ such that $\mathcal{G}(\mathbf{b})$ is non-empty.

▶ **Example 8.4.** Consider a MSO query $Q(x, y)$ and the semiring $\mathbb{N}$. The output of Q on a Γ-tree T under a mapping ρ contains one pair per $n \in T$, annotated with the sum of $\rho(n')$ for $n' \in T$ such that $T \models Q(n, n')$, where we exclude the nodes n for which the sum is empty.

▶ **Theorem 8.5.** *For any group-by query $Q(\mathbf{x}, \mathbf{y})$ and semiring K, given a Γ-tree T and $\rho : T \to K$, we can enumerate $Q_\rho(T)$ with linear-time preprocessing and delay in $O(\log |T|)$*

Proof sketch. We use two enumeration structures. First, we prepare the structure of Theorem 8.1 for $Q(\mathbf{x}, \mathbf{y})$ but writing the valuation of $\mathbf{x}$ as part of the tree label. Second, we enumerate the non-empty groups with constant delay using Theorem 3.1 on $\exists \mathbf{y}\ Q(\mathbf{x}, \mathbf{y})$. For each tuple $\mathbf{b}$ in the output of the second structure, letting $\mathcal{G}(\mathbf{b})$ be the corresponding group, we update the first structure to compute $\rho(\mathcal{G}(\mathbf{b}))$ in time $O(\log |T|)$. ◀

Parameterized queries. We conclude by presenting another kind of practical queries that we can support thanks to updates. A *parameterized* MSO query $Q(\mathbf{x}, \mathbf{y})$ on Γ-trees has two kinds of first-order variables, like group-by: we call $\mathbf{x}$ the *parameters*. The idea is that, given a Γ-tree T, the user chooses a tuple $\mathbf{b}$ to instantiate the parameters $\mathbf{x}$, and we must enumerate efficiently the results of $Q(\mathbf{b}, \mathbf{y})$; however the user can change their mind and modify $\mathbf{b}$ to change the value of the parameters. We know by Theorem 3.1 that we can support these queries efficiently: after a linear-time preprocessing of T, we can enumerate the results of $Q(\mathbf{b}, \mathbf{y})$ with constant delay; and we can react to changes to $\mathbf{b}$ in time $O(\log |T|)$ by performing an update on the enumeration structure.

9 Conclusion

We have studied MSO queries on trees under *relabeling* updates, and shown how to enumerate their answers with linear-time preprocessing, delay and memory linear in each valuation, and update time logarithmic in the input tree. We have shown this by extending our circuit-based approach [4] to hybrid circuits, and we have deduced consequences for practical query languages, in particular for efficient aggregation. Our results have another technical property

that we have not presented in the main text: like those of [24], they are also tractable in the size of the query when representing it as a deterministic automaton.

The main direction for future work would be to extend our result to support insertions and deletions of leaves, like [24], hopefully preserving our improved bounds: while deletions can be emulated with relabelings, insertions are trickier. Such a result was very recently shown in [25] for the case of words rather than trees. We believe that many of our constructions on trees should adapt to insertions and deletions. The main challenge is to extend Lemma 4.3, which we believe to be an interesting question in its own right: the technique of [10] may be applicable here, although it would lead to an $O(\log^2 n)$ update time.

References

1 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

2 Antoine Amarilli. *Leveraging the structure of uncertain data*. PhD thesis, Télécom ParisTech, 2016. URL: `https://tel.archives-ouvertes.fr/tel-01345836`.

3 Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. A circuit-based approach to efficient enumeration. *CoRR*, abs/1702.05589, 2017. `arXiv:1702.05589`.

4 Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. A circuit-based approach to efficient enumeration. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 111:1–111:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. `doi:10.4230/LIPIcs.ICALP.2017.111`.

5 Antoine Amarilli, Pierre Bourhis, and Stefan Mengel. Enumeration on trees under relabelings. *CoRR*, abs/1709.06185, 2017. `arXiv:1709.06185`.

6 Antoine Amarilli, Pierre Bourhis, and Pierre Senellart. Provenance circuits for trees and treelike instances (extended version). *CoRR*, abs/1511.08723, 2015. `arXiv:1511.08723`.

7 Stefan Arnborg, Jens Lagergren, and Detlef Seese. Easy problems for tree-decomposable graphs. *J. Algorithms*, 12(2), 1991.

8 Guillaume Bagan. MSO queries on tree decomposable structures are computable with linear delay. In *CSL*, 2006.

9 Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2007. `doi:10.1007/978-3-540-74915-8_18`.

10 Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. Incremental validation of XML documents. *ACM Trans. Database Syst.*, 29(4):710–751, 2004. `doi:10.1145/1042046.1042050`.

11 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 303–318. ACM, 2017. `doi:10.1145/3034786.3034789`.

12 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering FO+MOD queries under updates on bounded degree databases. In Michael Benedikt and Giorgio Orsi, editors, *20th International Conference on Database Theory, ICDT 2017, March 21-24, 2017,*

Venice, Italy, volume 68 of *LIPIcs*, pages 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.ICDT.2017.8.

13 Hans L. Bodlaender and Torben Hagerup. Parallel algorithms with optimal speedup for bounded treewidth. *SIAM J. Comput.*, 27(6):1725–1746, 1998. doi:10.1137/S0097539795289859.

14 Thomas Colcombet. A combinatorial theorem for trees. In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, volume 4596 of *Lecture Notes in Computer Science*, pages 901–912. Springer, 2007. doi:10.1007/978-3-540-73420-8_77.

15 Bruno Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990. doi:10.1016/0890-5401(90)90043-H.

16 Adnan Darwiche. On the tractable counting of theory models and its application to truth maintenance and belief revision. *Journal of Applied Non-Classical Logics*, 11(1-2):11–34, 2001. doi:10.3166/jancl.11.11-34.

17 Daniel Deutch, Tova Milo, Sudeepa Roy, and Val Tannen. Circuits for datalog provenance. In Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014.*, pages 201–212. OpenProceedings.org, 2014. doi:10.5441/002/icdt.2014.22.

18 Arnaud Durand and Etienne Grandjean. First-order queries on structures of bounded degree are computable with constant delay. *ACM Trans. Comput. Log.*, 8(4):21, 2007. doi:10.1145/1276920.1276923.

19 David Eppstein and Denis Kurz. K-best solutions of MSO problems on tree-decomposable graphs. *CoRR*, abs/1703.02784, 2017. arXiv:1703.02784.

20 J. Nathan Foster, Todd J. Green, and Val Tannen. Annotated XML: queries and provenance. In Maurizio Lenzerini and Domenico Lembo, editors, *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2008, June 9-11, 2008, Vancouver, BC, Canada*, pages 271–280. ACM, 2008. doi:10.1145/1376916.1376954.

21 Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In Leonid Libkin, editor, *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*, pages 31–40. ACM, 2007. doi:10.1145/1265530.1265535.

22 Wojciech Kazana and Luc Segoufin. First-order query evaluation on structures of bounded degree. *Logical Methods in Computer Science*, 7(2), 2011. doi:10.2168/LMCS-7(2:20)2011.

23 Wojciech Kazana and Luc Segoufin. Enumeration of monadic second-order queries on trees. *ACM Trans. Comput. Log.*, 14(4):25:1–25:12, 2013. doi:10.1145/2528928.

24 Katja Losemann and Wim Martens. MSO queries on trees: enumerating answers under updates. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 67:1–67:10. ACM, 2014. doi:10.1145/2603088.2603137.

25 Matthias Niewerth and Luc Segoufin. Enumeration of MSO queries on strings with constant delay and logarithmic updates. In *PODS*, 2018. To appear.

26 Milos Nikolic and Dan Olteanu. Incremental maintenance of regression models over joins, 2017. arXiv:1703.07484.

27 Dan Olteanu and Jakub Závodný. Size bounds for factorised representations of query results. *ACM Trans. Database Syst.*, 40(1):2:1–2:44, 2015. doi:10.1145/2656335.

28 Luc Segoufin. A glimpse on constant delay enumeration (invited talk). In Ernst W. Mayr and Natacha Portier, editors, *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014), STACS 2014, March 5-8, 2014, Lyon, France*, volume 25 of *LIPIcs*, pages 13–27. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014. doi:10.4230/LIPIcs.STACS.2014.13.

29 Volker Strassen. Vermeidung von divisionen. *Journal für die reine und angewandte Mathematik*, 264, 1973. URL: https://eudml.org/doc/151394.

30 Teruji Sugaya, Masaaki Nishino, Norihito Yasuda, and Shin-Ichi Minato. Fast compilation of s-t paths on a graph for counting and enumeration. In *AMBN*, volume 73 of *PMLR*, 2017. URL: http://proceedings.mlr.press/v73/teruji-sugaya17a.html.

31 Kunihiro Wasa. Enumeration of enumeration algorithms. *CoRR*, abs/1605.05102, 2016. arXiv:1605.05102.

Connecting Width and Structure in Knowledge Compilation

Antoine Amarilli
LTCI, Télécom ParisTech, Université Paris-Saclay

Mikaël Monet
LTCI, Télécom ParisTech, Université Paris-Saclay, and Inria Paris, Paris, France

Pierre Senellart
LTCI, Télécom ParisTech, Université Paris-Saclay, and DI ENS, ENS, CNRS, PSL Research University, and Inria Paris, Paris, France

Abstract

Several query evaluation tasks can be done via *knowledge compilation*: the query result is compiled as a *lineage circuit* from which the answer can be determined. For such tasks, it is important to leverage some width parameters of the circuit, such as bounded treewidth or pathwidth, to convert the circuit to structured classes, e.g., deterministic structured NNFs (d-SDNNFs) or OBDDs. In this work, we show how to connect the width of circuits to the size of their structured representation, through upper and lower bounds. For the upper bound, we show how bounded-treewidth circuits can be converted to a d-SDNNF, in time linear in the circuit size. Our bound, unlike existing results, is constructive and only singly exponential in the treewidth. We show a related lower bound on monotone DNF or CNF formulas, assuming a constant bound on the arity (size of clauses) and degree (number of occurrences of each variable). Specifically, any d-SDNNF (resp., SDNNF) for such a DNF (resp., CNF) must be of exponential size in its treewidth; and the same holds for pathwidth when compiling to OBDDs. Our lower bounds, in contrast with most previous work, apply to *any* formula of this class, not just a well-chosen family. Hence, for our language of DNF and CNF, pathwidth and treewidth respectively characterize the efficiency of compiling to OBDDs and (d-)SDNNFs, that is, compilation is singly exponential in the width parameter. We conclude by applying our lower bound results to the task of query evaluation.

2012 ACM Subject Classification Theory of computation Incomplete, inconsistent, and uncertain databases

Keywords and phrases knowledge compilation, probabilistic databases, treewidth, circuits

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.6

Related Version Full proofs are given in the extended version [8], `https://arxiv.org/abs/1709.06188`.

Acknowledgements We acknowledge Chandra Chekuri for his helpful comments at `https://cstheory.stackexchange.com/a/38943/`, as well as Florent Capelli for pointing out the connection to [19, Corollary 6.35] and [40].

1 Introduction

Uncertainty and errors in data can be modeled using *probabilistic databases* [39], annotating every tuple with a probability of existence. Query evaluation on probabilistic databases must then handle the uncertainty by computing the probability that each query result holds. A common technique to evaluate queries on probabilistic databases is the *intensional approach*:

first compute a representation of the *lineage* of the query on the database, which intuitively describes how the query depends on the possible database facts; then use this lineage to compute probabilities efficiently. Specifically, the lineage can be computed as a *circuit* [32], and efficient probability computation can be achieved by restricting to tractable circuit classes via *knowledge compilation*. Thus, to evaluate queries on probabilistic databases, we can use knowledge compilation algorithms to translate circuits to tractable classes; conversely, lower bounds in knowledge compilation can identify the limits of the intensional approach.

In this paper, we study the relationship between two kinds of tractable circuit classes in knowledge compilation: *width-based* classes, specifically, bounded-treewidth and bounded-pathwidth circuits; and *structure-based* classes, specifically, OBDDs (ordered binary decision diagrams [17], following a *variable order*) and d-SDNNFs (structured deterministic decomposable negation normal forms [35], following a *v-tree*). Circuits of bounded treewidth can be obtained as a result of practical query evaluation [4, 6, 30], whereas OBDDs and d-DNNFs have been studied to show theoretical characterizations of the query lineages they can represent [31]. Both classes enjoy tractable probabilistic computation: for width-based classes, using *message passing* [33], in time linear in the circuit and exponential in the treewidth; for OBDDs and d-SDNNFs, in linear time by definition of the class [22]. Hence the question that we study: can we compile width-based classes efficiently into structure-based classes?

We first study how to perform this transformation, and show corresponding *upper bounds*. Existing work has already studied the compilation of bounded-pathwidth circuits to OBDDs [32], which can be made constructive [7, Lemma 6.9]. Accordingly, we focus on compiling *bounded-treewidth circuits* to *d-SDNNF circuits*. Our first contribution, stated in Section 3 and proved in Section 4, is to show the following:

▶ **Result 1** (Theorem 5). *Given as input a Boolean circuit C of treewidth k, we can compute a d-SDNNF equivalent to C in time $O(|C| \times f(k))$ where f is singly exponential.*

The algorithm transforms the input circuit bottom-up, considering all possible valuations of the gates in each bag of the tree decomposition, and keeping track of additional information to remember which guessed values have been substantiated by a corresponding input. Our result relates to a recent theorem of Bova and Szeider in [16], except that our bound depends on $|C|$ (the circuit size) whereas their bound depends on the number of variables of C. In exchange for this, we improve on their result in two ways. First, our result is constructive, whereas [16] only shows a bound on the size of the d-SDNNF, without bounding the complexity of effectively computing it. Second, our bound is singly exponential in k, whereas [16] is doubly exponential; this allows us to be competitive with message passing (also singly exponential in k), and we believe it can be useful for practical applications. Indeed, beyond probabilistic query evaluation, our result implies that all tractable tasks on d-SDNNFs (e.g., enumeration [2] and MAP inference [27]) are also tractable on bounded-treewidth circuits.

Second, we study *lower bounds* on how efficiently we can convert from width-based to structure-based classes. Our bounds already apply to a weaker formalism of width-based circuits, namely monotone CNFs and DNFs of bounded width, so we focus on them. Our second contribution (in Section 5) concerns pathwidth and OBDD representations: we show that, up to factors in the formula arity (maximal size of clauses) and degree (maximal number of variable occurrences), any OBDD for a monotone CNF or DNF must be of width exponential in the pathwidth of the formula. Formally:

▶ **Result 2** (Theorem 15). *Let φ be a monotone DNF or monotone CNF, let $a := \mathrm{arity}(\varphi)$ and $d := \mathrm{degree}(\varphi)$. Then any OBDD for φ has width $2^{\Omega\left(\frac{\mathrm{pw}(\varphi)}{a^3 \times d^2}\right)}$.*

This result generalizes several existing lower bounds in knowledge compilation that exponentially separate CNFs from OBDDs, such as [25] and [15, Theorem 19].

Our third contribution (Section 6) is to show an analogue for treewidth and (d-)SDNNFs:

▶ **Result 3** (Theorem 25). *Let φ be a monotone DNF (resp., monotone CNF), let $a := \mathrm{arity}(\varphi)$ and $d := \mathrm{degree}(\varphi)$. Then any d-SDNNF (resp., SDNNF) for φ has size $2^{\Omega\left(\frac{\mathrm{tw}(\varphi)}{a^3 \times d^2}\right)}$.*

Our two lower bounds contribute to a vast landscape of knowledge compilation results giving lower bounds on compiling specific Boolean functions to restricted circuits classes, e.g., [15, 25, 37] to OBDDs, [18] to *decision* structured DNNF, [9] to *sentential decision diagrams* (SDDs), [13, 36] to d-SDNNF, [13, 19, 20] to d-DNNFs and DNNFs. However, all those lower bounds (with the exception of some results in [19, 20] discussed in Section 6) apply to well-chosen families of Boolean functions (usually CNF), whereas Result 2 and 3 apply to *any* monotone CNF and DNF. Together with Result 1, these generic lower bounds point to a strong relationship between width parameters and structure representations, on monotone CNFs and DNFs of constant arity and degree. Specifically, the smallest width of OBDD representations of any such formula φ is in $2^{\Theta(\mathrm{pw}(\varphi))}$, i.e., precisely singly exponential in the pathwidth; and an analogous bound applies to d-SDNNF size and treewidth of DNFs.

To prove our lower bounds, we rephrase pathwidth and treewidth to new notions of *pathsplitwidth* and *treesplitwidth*, which intuitively measure the performance of a variable ordering or v-tree. We also use the *disjoint non-covering prime implicant sets* (dncpi-sets), a tool introduced in [7, 1] by some of the present authors, and generalizing *subfunction width* [15]. These dncpi-sets allow us to derive lower bounds on OBDD width directly using [1]. We show how they can also imply lower bounds on d-SDNNF size, using the recent communication complexity approach of Bova, Capelli, Mengel and Slivovsky [13].

Our fourth contribution (Section 7) applies our lower bounds to intensional query evaluation on relational databases. We reuse the notion of *intricate* queries of [7], and show that d-SDNNF representations of the lineage of these queries have size exponential in the treewidth of *any* input instance. This extends the result of [7] from OBDDs to d-SDNNFs:

▶ **Result 4** (Theorem 33). *There is a constant $d \in \mathbb{N}$ such that the following is true. Let σ be an arity-2 signature, and Q be a connected* $\mathrm{UCQ}^{\neq}$ *which is intricate on σ. For any instance I on σ, any d-SDNNF representing the lineage of Q on I has size $\geqslant 2^{\Omega(\mathrm{tw}(I)^{1/d})}$.*

As in [7], this result shows that, on arity-2 signatures and under constructibility assumptions, treewidth is the right parameter on instance families to ensure that all queries (in monadic second-order) have tractable d-SDNNF lineage representations.

We start in Section 2 with preliminaries.

2 Preliminaries

Hypergraphs, treewidth, pathwidth. A *hypergraph* $H = (V, E)$ consists of a finite set of *nodes* (or *vertices*) V and of a set E of *hyperedges* (or simply *edges*) which are non-empty subsets of V. We always assume that hypergraphs have at least one edge. For a node v of H, we write $E(v)$ for the set of edges of H that contain v. The *arity* of H, written $\mathrm{arity}(H)$, is the maximal size of an edge of H. The *degree* of H, written $\mathrm{degree}(H)$, is the maximal number of edges to which a vertex belongs, i.e., $\max_{v \in V} |E(v)|$.

A *tree decomposition* of a hypergraph $H = (V, E)$ is a finite, rooted tree T, whose nodes b (called *bags*) are labeled by a subset $\lambda(b)$ of V, and which satisfies:

1. for every fact $e \in E$, there is a bag $b \in T$ with $e \subseteq \lambda(b)$;

2. for all $v \in V$, the set of bags $\{b \in T \mid v \in \lambda(b)\}$ is a connected subtree of T.

For brevity, we identify a bag b with its domain $\lambda(b)$. The *width* of T is $\max_{b \in T} |\lambda(b)| - 1$. The *treewidth* of H is the minimal width of a tree decomposition of H. Pathwidth is defined similarly but with *path decompositions*, where T is a path rather than a tree.

It is NP-hard to determine the treewidth of a hypergraph, but we can compute a tree decomposition in linear time when parametrizing by the treewidth:

▶ **Theorem 1** ([10]). *Given a hypergraph H and an integer $k \in \mathbb{N}$ we can check in time $O(|H| \times g(k))$ whether H has treewidth $\leqslant k$, and if yes output a tree decomposition of H of width $\leqslant k$, where g is a fixed function in $O(2^{(32+\varepsilon)k^3})$ for any $\varepsilon > 0$.*

For simplicity, we will often assume that a tree decomposition is *nice*, meaning that:

1. it is a full binary tree, i.e., each node has exactly zero or two children;
2. for every internal bag b with children b_l, b_r we have $b \subseteq b_l \cup b_r$;
3. for every leaf bag b we have $|b| \leqslant 1$;
4. for every non-root bag b with parent b', we have $|b \setminus b'| \leqslant 1$;
5. for the root bag b we have $|b| \leqslant 1$.

▶ **Lemma 2.** *Given a tree decomposition T of width k having n nodes, we can compute in time $O(k \times n)$ a nice tree decomposition T' of width k having $O(k \times n)$ nodes.*

Boolean circuits and functions. A (Boolean) *valuation* of a set V is a function $\nu : V \to \{0, 1\}$. A *Boolean function* φ on variables V is a mapping that associates to each valuation ν of V a Boolean value in $\{0, 1\}$ called the *evaluation* of φ according to ν.

A *(Boolean) circuit* $C = (G, W, g_{\mathsf{output}}, \mu)$ is a directed acyclic graph (G, W) whose vertices G are called *gates*, whose edges W are called *wires*, where $g_{\mathsf{output}} \in G$ is the *output gate*, and where each gate $g \in G$ has a *type* $\mu(g)$ among var (a *variable gate*), not, or, and. The *inputs* of a gate $g \in G$ are the gates $g' \in G$ such that $(g', g) \in W$; the *fan-in* of g is its number of inputs. We require not-gates to have fan-in 1 and var-gates to have fan-in 0. The *treewidth* of C, and its *size*, are those of the graph (G, W). The set C_{var} of *variable gates* of C are those of type var. Given a valuation ν of C_{var}, we extend it to an *evaluation* of C by mapping each variable $g \in C_{\mathsf{var}}$ to $\nu(g)$, and evaluating the other gates according to their type. The Boolean function on C_{var} *captured* by the circuit is the one that maps ν to the evaluation of g_{output} under ν. Two circuits are *equivalent* if they capture the same function.

We recall restricted circuit classes from knowledge compilation. We say that C is in *negation normal form* (NNF) if the inputs of not-gates are always variable gates. For a gate g in a Boolean circuit C, we write $\mathsf{Vars}(g)$ for the set of variable gates of C_{var} that have a directed path to g in C. An and-gate g of C is *decomposable* if for every two input gates $g_1 \neq g_2$ of g we have $\mathsf{Vars}(g_1) \cap \mathsf{Vars}(g_2) = \emptyset$. We call C *decomposable* if each and-gate is.

A stronger requirement than decomposability is *structuredness*. A *v-tree* [35] over a set V is a rooted ordered binary tree T whose leaves are in bijection with V; we identify each leaf with the associated element of V. For $n \in T$, we denote by T_n the subtree of T rooted at n, and for a subset $U \subseteq T$ of nodes of T, we denote by $\mathsf{Leaves}(U)$ the leaves that are in U, i.e., $U \cap V$. We say that T *structures* a Boolean circuit C (and call it a *v-tree for C*) if T is over the set C_{var} and if, for every and-gate g of C with inputs $g_1, \ldots, g_m$ and $m > 0$, there is a node $n \in T$ that *structures* g, i.e., n has m children $n_1, \ldots, n_m$ and we have $\mathsf{Vars}(g_i) \subseteq \mathsf{Leaves}(T_{n_i})$ for all $1 \leqslant i \leqslant m$. We call C *structured* if some v-tree structures it. Note that structured Boolean circuits are always decomposable, and their and-gates have at most two inputs because T is binary.

A last requirement on circuits is *determinism*. An or-gate g of C is *deterministic* if there is no pair $g_1 \neq g_2$ of input gates of g and valuation ν of C_{var} such that g_1 and g_2 both evaluate to 1 under ν. A Boolean circuit is *deterministic* if each or-gate is.

The main structured class of circuits that we study in this work are *deterministic structured decomposable NNFs*, which we denote d-SDNNF for brevity as in [35].

DNFs and CNFs. We also study other representations of Boolean functions, namely, Boolean formulas in *conjunctive normal form* (*CNFs*) and in *disjunctive normal form* (*DNFs*). A DNF (resp., CNF) φ on a set of variables V is a disjunction (resp., conjunction) of *clauses*, each of which is a conjunction (resp., disjunction) of *literals* on V, i.e., variables of V (a *positive* literal) or their negation (a *negative* literal). A *monotone DNF* (resp., monotone CNF) is one where all literals are positive, in which case we often identify a clause to the set of variables that it contains. We always assume that monotone DNFs and monotone CNFs are *minimized*, i.e., no clause is a subset of another. This ensures that every monotone Boolean function has a unique representation as a monotone DNF, and likewise for CNF. We assume that CNFs and DNFs always contain at least one non-empty clause (in particular, they cannot represent constant functions). Monotone DNFs and CNFs φ are isomorphic to hypergraphs: the vertices are the variables of φ, and the hyperedges are the clauses of φ. We often identify φ to its hypergraph. In particular, the *pathwidth* and *treewidth* of φ, and its *arity* and *degree*, are defined as that of its hypergraph.

3 Upper Bounds

Our upper bound result studies how to compile a Boolean circuit to a d-SDNNF, parametrized by the treewidth of the input circuit. To present it, we first review the independent result that was recently shown by Bova and Szeider [16] about these circuit classes:

▶ **Theorem 3** ([16, Theorem 3 and Equation (22)]). *Given a Boolean circuit C with n variables and of treewidth $\leqslant k$, there exists an equivalent d-SDNNF of size $O(f(k) \times n)$, where f is doubly exponential.*

An advantage of their result is that it depends only on the *number of variables* of the circuit (and on the width parameter), not on the *size* of the circuit. None of our results will have this advantage, and we will always measure complexity as a function of the size of the input circuit. In exchange for this advantage, their result has two drawbacks: (i) the doubly exponential dependency on the width; and (ii) its nonconstructive aspect, because [16] gives no time bound on the computation, leaving open the question of effectively compiling bounded-treewidth circuits to d-SDNNFs.

Naive constructive bound. We first address the second drawback by showing an easy constructive result. The argument is very simple and appeals to techniques from our earlier works on provenance circuits [6, 7]; it is independent from the techniques of [16].

▶ **Theorem 4.** *Given any circuit C of treewidth k, we can compute an equivalent d-SDNNF in linear time parametrized by k, i.e., in time $O(|C| \times f(k))$ for some computable function f.*

Proof sketch. We encode in linear time the input circuit C to a relational instance I with same treewidth. We use [7, Theorem 6.11] to construct in linear time a provenance representation C' on I of a fixed MSO formula that describes Boolean circuit evaluation. This allows us to obtain in linear time from C' the desired equivalent d-SDNNF representation. ◀

This result shows that we can effectively compile in linear time parametrized by the treewidth k, but does not address the first drawback, namely, the dependency in k.

Improved bound. Our main upper bound result subsumes the naive bound above, with a more elaborate proof, again independent of the techniques of [16]. It addresses both drawbacks and shows that we can effectively compile in time singly exponential in k; formally:

▶ **Theorem 5.** *Given as input a Boolean circuit C and tree decomposition T of width k, we can compute a d-SDNNF equivalent to C with its v-tree, in $O\left(|T| \times 2^{(4+\varepsilon)k}\right)$ for any $\varepsilon > 0$.*

We prove Theorem 5 in the next section. Observe how we assume the tree decomposition to be given as part of the input. If it is not, we can compute one with Theorem 1, but this becomes the bottleneck: the complexity becomes $O\left(|C| \times 2^{(32+\varepsilon)k^3}\right)$ for any $\varepsilon > 0$.

Applications. Theorem 5 implies several consequences for bounded-treewidth circuits. The first one deals with *probability computation*: we are given a *probability valuation* π mapping each variable $g \in C_{\mathsf{var}}$ to a probability that g is true (independently from other variables), and we wish to compute the probability $\pi(C)$ that C evaluates to true under π, assuming that arithmetic operations (sum and product) take unit time. This problem is #P-hard for arbitrary circuits, but it is tractable for d-SDNNF [22]. Hence, our result implies the following, where $|\pi|$ denotes the size of writing the probability valuation π:

▶ **Corollary 6.** *Given a Boolean circuit C, a tree decomposition T of width k of C, and a probability valuation π of C, we can compute $\pi(C)$ in $O\left(|\pi| + |T| \times 2^{(4+\varepsilon)k}\right)$ for any $\varepsilon > 0$.*

This improves the bound obtained when applying message passing techniques [33] directly on the bounded-treewidth input circuit (as presented, e.g., in [5, Theorem D.2]). Indeed, message passing applies to *moralized* representations of the input: for each gate, the tree decomposition must contain a bag containing all inputs of this gate *simultaneously*, which is problematic for circuits of large fan-in. Indeed, if the original circuit has a tree decomposition of width k, rewriting it to make it moralized results in a tree decomposition of width $3k^2$ (see [3, Lemmas 53 and 55]), and the bound of [5, Theorem D.2] then yields an overall complexity of $O\big(|\pi| + |T| \times 2^{3k^2}\big)$ for message passing. Our Corollary 6 achieves a more favorable bound because Theorem 5 uses directly the associativity of and and or. We note that the connection between message-passing techniques and structured circuits has also been investigated by Darwiche, but his result [23, Theorem 6] produces arithmetic circuits rather than d-DNNFs, and it also needs the input to be moralized.

A second consequence concerns the task of *enumerating* the accepting valuations of circuits, i.e., producing them one after the other, with small *delay* between each accepting valuation. The valuations are concisely represented as *assignments*, i.e., as a set of variables that are set to true, omitting those that are set to false. This task is of course NP-hard on arbitrary circuits (as it implies that we can check whether an accepting valuation exists), but was recently shown in [2] to be feasible on d-SDNNFs with linear-time preprocessing and delay linear in the Hamming weight of each produced assignment. Hence, we have:

▶ **Corollary 7.** *Given a Boolean circuit C and a tree decomposition T of width k of C, we can enumerate the accepting assignments of C with preprocessing in $O\left(|T| \times 2^{(4+\varepsilon)k}\right)$ and delay linear in the size of each produced assignment.*

Other applications of Theorem 5 include counting the number of satisfying valuations of the circuit (a special case of probability computation), MAP inference [27] or random sampling of possible worlds (which can be done on the d-SDNNF in an easy manner).

4 Proof of the Main Upper Bound Result

In this section, we present the construction used to prove Theorem 5. We start with prerequisites, and then describe how to build the d-SDNNF equivalent to the input bounded-treewidth circuit. Last, we sketch the correctness proof.

Prerequisites. Let C be the input circuit, and T the input tree decomposition. By Lemma 2, we assume that T is nice. Further, up to adding a constant number of bags and re-rooting T, we can assume that the root bag of T contains only the output gate g_{output}. For any bag b of T, we define $\mathrm{VarT}(b)$ to be the set of variable gates such that b is the topmost bag in which they appear; as T is nice, $\mathrm{VarT}(b)$ is either empty or is a singleton $\{g\}$, in which case we call b *responsible for the variable gate* g. We can explicitly compute the function VarT in $O(|T|)$, i.e., compute $\mathrm{VarT}(b)$ for each $b \in T$; see for instance [28, Lemma 3.1].

To abstract away the type of gates and their values in the construction, we will talk of *strong* and *weak* values. Intuitively, a value is *strong* for a gate g if any input g' of g which carries this value determines the value of g; and *weak* otherwise. Formally:

▶ **Definition 8.** Let g be a gate and $c \in \{0,1\}$:
- If g is an and-gate, we say that $c = 0$ is *strong* for g and $c = 1$ is *weak* for g;
- If g is an or-gate, we say that $c = 1$ is *strong* for g and $c = 0$ is *weak* for g;
- If g is a not-gate, $c = 0$ and $c = 1$ are both *strong* for g;
- For technical convenience, if g is a var-gate, $c = 0$ and $c = 1$ are both *weak* for g.

If we take any valuation $\nu : C_{\mathsf{var}} \to \{0,1\}$ of the circuit $C = (G, W, g_{\mathsf{output}}, \mu)$, and extend it to an evaluation $\nu : G \to \{0,1\}$, then ν will respect the semantics of gates. In particular, it will *respect strong values*: for any gate g of C, if g has an input g' for which $\nu(g')$ is a strong value, then $\nu(g)$ is determined by $\nu(g')$, specifically, it is $\nu(g')$ if g is an or- or an and-gate, and $1 - \nu(g')$ if g is a not-gate. In our construction, we will need to guess how gates of the circuit are evaluated, focusing on a subset of the gates (as given by a bag of T); we will then call *almost-evaluation* an assignment of these gates that respects strong values. Formally:

▶ **Definition 9.** Let U be a set of gates of C. We call $\nu : U \to \{0,1\}$ a (C,U)-*almost-evaluation* if it *respects strong values*, i.e., for every gate $g \in U$, if there is an input g' of g in U and $\nu(g')$ is a strong value for g, then $\nu(g)$ is determined from $\nu(g')$ in the sense above.

Respecting strong values is a necessary condition for such an assignment to be extensible to a valuation of the entire circuit. However, it is not sufficient: an almost-evaluation ν may map a gate g to a strong value even though g has no input that can justify this value. This is hard to avoid: when we focus on the set U, we do not know about other inputs of g. For now, let us call *unjustified* the gates of U that carry a strong value that is not justified by ν:

▶ **Definition 10.** Let U be a set of gates of a circuit C and ν a (C,U)-almost-evaluation. We call $g \in U$ *unjustified* if $\nu(g)$ is a strong value for g, but, for every input g' of g in U, the value $\nu(g')$ is weak for g; otherwise, g is *justified*. The set of unjustified gates is written $\mathrm{Unj}(\nu)$.

Let us start to explain how to construct the d-SDNNF circuit D equivalent to the input circuit C. We do so by traversing T bottom-up, and for each bag b of T we create gates $G_b^{\nu,S}$ in D, where ν is a (C,b)-almost-evaluation and S is a subset of $\mathrm{Unj}(\nu)$ which we call the *suspicious gates* of $G_b^{\nu,s}$. We will connect the gates of D created for each internal bag b with the gates created for its children in T, in a way that we will explain later. Intuitively, for a gate $G_b^{\nu,S}$ of D, the *suspicious gates* g in the set S are gates of b whose strong value is not

justified by ν (i.e., $g \in \mathrm{Unj}(\nu)$), and is not justified either by any of the almost-evaluations at descendant bags of b to which $G_b^{\nu,S}$ is connected. We call *innocent* the other gates of b; they are the gates that are justified in ν (in particular, those who carry weak values), and the gates that are unjustified in ν but have been justified by an almost-evaluation at a descendant bag b' of b. Crucially, in the latter case, the gate g' justifying the strong value in b' may no longer appear in b, making g unjustified for ν; this is why we remember the set S.

We still have to explain how we connect the gates $G_b^{\nu,S}$ of D to the gates $G_{b_l}^{\nu_l,S_l}$ and $G_{b_r}^{\nu_r,S_r}$ created for the children b_l and b_r of b in T. The first condition is that ν_l and ν_r must *mutually agree*, i.e., $\nu_l(g) = \nu_r(g)$ for all $g \in b_l \cap b_r$, and ν must then be the union of ν_l and ν_r, restricted to b. Remember that T is nice, so b is a subset of $b_l \cup b_r$, and it is easy to verify that ν is then a (C, b)-almost-evaluation. We impose a second condition to prohibit suspicious gates from escaping before they have been justified, which we formalize as *connectibility* of a pair (ν, S) at bag b to the parent bag of b.

▶ **Definition 11.** Let b be a non-root bag, b' its parent bag, and ν a (C, b)-almost-evaluation. For any set $S \subseteq \mathrm{Unj}(\nu)$, we say that (ν, S) is *connectible* to b' if $S \subseteq b'$, i.e., the suspicious gates of ν must still appear in b'.

If a gate $G_b^{\nu,S}$ is such that (ν, S) is not connectible to the parent bag b', then this gate will not be used as input to any other gate (but we do not try to preemptively remove these useless gates in the construction). We are now ready to give the formal definition that will be used to explain how gates are connected:

▶ **Definition 12.** Let b be an internal bag with children b_l and b_r, let ν_l and ν_r be respectively (C, b_l) and (C, b_r)-almost-evaluations that mutually agree, and $S_l \subseteq \mathrm{Unj}(\nu_l)$ and $S_r \subseteq \mathrm{Unj}(\nu_r)$ be sets of suspicious gates such that both (ν_l, S_l) and (ν_r, S_r) are connectible to b. The *result* of (ν_l, S_l) and (ν_r, S_r) is then defined as the pair (ν, S) where:

- ν is a (C, b)-almost-evaluation defined as the restriction of $\nu_l \cup \nu_r$ to b.
- $S \subseteq \mathrm{Unj}(\nu)$ is the new set of suspicious gates, defined as follows. A gate $g \in b$ is innocent (i.e., $g \in b \setminus S$) if it is justified for ν or if it is innocent for some child. Formally, $b \setminus S := (b \setminus \mathrm{Unj}(\nu)) \cup \big[b \cap [(b_l \setminus S_l) \cup (b_r \setminus S_r)]\big]$.

Construction. We now use these definitions to present the construction formally. For every variable gate g of C, we create a corresponding variable gate $G^{g,1}$ of D, and we create $G^{g,0} := \mathsf{not}(G^{g,1})$. For every internal bag b of T, for each (C, b)-almost-evaluation ν and set $S \subseteq \mathrm{Unj}(\nu)$ of suspicious gates of ν, we create one and-gate $G_b^{\nu,S}$ and one or-gate $G_{b,\mathsf{children}}^{\nu,S}$ which is an input of $G_b^{\nu,S}$. For every leaf bag b of T, we create one gate $G_b^{\nu,S}$ for every (C, b)-almost-evaluation ν, where we set $S := \mathrm{Unj}(\nu)$; intuitively, in a leaf bag, an unjustified gate is always suspicious (it cannot have been justified at a descendant bag).

Now, for each internal bag b of T with children b_l, b_r, for each pair of gates $G_{b_l}^{\nu_l,S_l}$ and $G_{b_r}^{\nu_r,S_r}$ that are both connectible to b and where ν_l and ν_r mutually agree, letting (ν, S) be the result of (ν_l, S_l) and (ν_r, S_r), we create a gate $G_b^{\nu_l,S_l,\nu_r,S_r} = \mathsf{and}(G_{b_l}^{\nu_l,S_l}, G_{b_r}^{\nu_r,S_r})$ and make it an input of $G_{b,\mathsf{children}}^{\nu,S}$. Last, for each bag b which is responsible for a variable gate g, for each (C, b)-almost-evaluation ν and set of suspicious gates $S \subseteq \mathrm{Unj}(\nu)$, we set the gate $G^{g,\nu(g)}$ to be the second input of $G_b^{\nu,S}$. The output gate of D is the gate $G_{b_{\mathrm{root}}}^{\nu,\emptyset}$ where b_{root} is the root of T and ν maps g_{output} to 1 (remember that b_{root} contains only g_{output}).

Correctness. We have formally described the construction of our d-SDNNF D. The construction clearly works in linear time, and we can prove that the dependency on k of the

running time is as stated. Further, we easily see that D is structured by a v-tree constructed from the tree decomposition T. To show that D is equivalent to C, one direction is easier: any valuation χ that satisfies C also satisfies D, because we can construct an *accepting trace* in D using the gates $G_b^{\nu,S}$ for ν the restriction of the evaluation χ to b, and for $S := \mathrm{Unj}(\chi_{|T_b})$ where T_b denotes the gates of C occurring in the bags of the subtree of T rooted at b. The converse is trickier: we show that any accepting trace of D describes an evaluation of C that respects strong values by definition of almost-evaluations, and eventually justifies every gate which is given a strong value thanks to our bookkeeping of suspicious gates. Last, we show that D is deterministic: this is unexpected because we freely guess the values of gates of C at leaf bags, but it holds because, when we know the valuation of the variable gates, knowing the valuation of all gates of a bag b uniquely fixes the valuation at the subtree rooted at b. This concludes the proof sketch of Theorem 5; see the extended version [8] for the full proof.

5 Lower Bounds on OBDDs

We now move to lower bounds on the size of structured representations of Boolean functions, in terms of the width of a circuit for that function. Our end goal is to obtain a lower bound for (d-)SDNNFs, that will form a counterpart to the upper bound of Theorem 5. We will do so in Section 6. For now, we consider a weaker class of lineage representations: *OBDDs*.

▶ **Definition 13.** An *ordered binary decision diagram* (or *OBDD*) on a set of variables $V = \{v_1, \ldots, v_n\}$ is a rooted DAG O whose leaves are labeled by 0 or 1, and whose internal nodes are labeled with a variable of V and have two outgoing edges labeled 0 and 1. We require that there exists a total order $\mathbf{v} = v_{i_1}, \ldots, v_{i_n}$ on the variables such that, for every path from the root to a leaf, the sequence of variables which labels the internal nodes of the path is a subsequence of $\mathbf{v}$ and does not contain duplicate variables. The OBDD O *captures* a Boolean function on V defined by mapping each valuation ν to the value of the leaf reached from the root by following the path given by ν. The *size* $|O|$ of O is its number of nodes, and the *width* w of O is the maximum number of nodes at every *level*, where a level is defined for a prefix of $\mathbf{v}$ as the set of nodes reached by enumerating all possible valuations of this prefix.

Our upper bound in the previous section applied to arbitrary Boolean circuits; however, our lower bounds in this section and the next one will already apply to much weaker formalisms for Boolean functions, namely, monotone DNFs and monotone CNFs (recall their definition from Section 2). Some lower bounds are already known for the compilation of monotone CNFs into OBDDs: Bova and Slivovsky have constructed a family of CNFs of bounded degree whose OBDD width is exponential in their number of variable occurrences [15, Theorem 19], following an earlier result of this type by Razgon [37, Corollary 1]. The result is as follows:

▶ **Theorem 14** ([15, Theorem 19])**.** *There is a class of monotone CNF formulas of bounded degree and arity such that every formula φ in this class has OBDD size at least $2^{\Omega(|\varphi|)}$.*

We adapt some of these techniques to show a more general result: our lower bound applies to *any* monotone DNF or monotone CNF, not to one specific family. Specifically, we show:

▶ **Theorem 15.** *Let φ be a monotone DNF or monotone CNF, let $a := \mathrm{arity}(\varphi)$ and $d := \mathrm{degree}(\varphi)$. Then any OBDD for φ has width $\geqslant 2^{\left\lfloor \frac{\mathrm{pw}(\varphi)}{a^3 \times d^2} \right\rfloor}$.*

From our Theorem 15, we can easily derive Theorem 14 using the fact (also used in the proof of [15, Theorem 19]) that there exists a family of monotone CNFs of bounded degree and arity whose treewidth (hence pathwidth) is linear in their size, namely, the CNFs built

from *expander graphs* (see [29, Theorem 5 and Proposition 1]). Note that expander graphs can also be used to show lower bounds for *DNNFs* for a CNF formula [12]; our lower bound on SDNNFs of Section 6 does not capture this result (because we need structuredness).

We observe that, for a family of formulas with bounded arity and degree, the bound of Theorem 15 is optimal, up to constant factors in the exponent. Indeed, following earlier work [26, 37], Bova and Slivovsky have shown that any CNF φ can be compiled to OBDDs of width $2^{\mathrm{pw}(\varphi)+2}$ [15, Theorem 4 and Lemma 9]. (Their upper bound result also applies to DNFs, and does not assume monotonicity nor a bound on the arity or degree.) In other words, for any monotone DNF or monotone CNF of bounded arity and degree, pathwidth *characterizes* the width of an OBDD for the formula, in the following sense:

▶ **Corollary 16.** *For any constant c, for any monotone DNF (or monotone CNF) φ with arity and degree bounded by c, the smallest width of an OBDD for φ is $2^{\Theta(\mathrm{pw}(\varphi))}$.*

This corollary talks about the pathwidth of φ measured as that of its hypergraph, but note that the same result would hold when measuring the pathwidth of the incidence graph or dual hypergraph of φ. Indeed, all these pathwidths are within a constant factor of one another when the degree and arity are bounded by a constant.

We prove Theorem 15 in the rest of this section. We present the proof in the case of monotone DNFs to reuse existing lower bound techniques from [7, 1], but explain at the end of this section how the proof adapts to monotone CNFs. We first present *pathsplitwidth*, a new notion which intuitively measures the performance of a variable ordering for an OBDD on the monotone DNF φ, and connect it to the pathwidth of φ. Second, we recall the definition of *dncpi-sets* introduced in [7, 1] to show lower bounds from the structure of Boolean functions. Last, we conclude the proof by connecting pathsplitwidth to the size of dncpi-sets.

Pathsplitwidth. The first step of the proof is to rephrase the bound on pathwidth, arity, and degree, in terms of a bound on the performance of variable orderings. Intuitively, a good variable ordering is one which does not *split* too many clauses. Formally:

▶ **Definition 17.** Let $H = (V, E)$ be a hypergraph, and $\mathbf{v} = v_1, \ldots, v_{|V|}$ be an ordering on the variables of V. For $1 \leqslant i \leqslant |V|$, we define $\mathrm{Split}_i(\mathbf{v}, H)$ as the set of hyperedges e of H that contain both a variable at or before v_i, and a variable strictly after v_i, formally: $\mathrm{Split}_i(\mathbf{v}, H) := \{e \in E \mid \exists l \in \{1, \ldots, i\} \text{ and } \exists r \in \{i+1, \ldots, |V|\} \text{ such that } \{v_l, v_r\} \subseteq e\}$. Note that $\mathrm{Split}_{|V|}(\mathbf{v}, H)$ is always empty. The *pathsplitwidth* of $\mathbf{v}$ relative to H is the maximum size of the split, formally, $\mathrm{psw}(\mathbf{v}, H) := \max_{1 \leqslant i \leqslant |V|} |\mathrm{Split}_i(\mathbf{v}, H)|$. The *pathsplitwidth* $\mathrm{psw}(H)$ of H is then the minimum of $\mathrm{psw}(\mathbf{v}, H)$ over all variable orderings $\mathbf{v}$ of V.

In other words, $\mathrm{psw}(H)$ is the smallest integer $n \in \mathbb{N}$ such that, for any variable ordering $\mathbf{v}$ of the nodes of H, there is a moment at which n hyperedges of H are *split*, i.e., for n hyperedges e, we have begun enumerating the nodes of e but we have not enumerated all of them yet. We note that the pathsplitwidth of H is exactly the *linear branch-width* [34] of the dual hypergraph of H, but we introduced pathsplitwidth because it fits our proofs better.

For a monotone DNF φ with hypergraph H, the quantity $\mathrm{psw}(H)$ is intuitively connected to the quantity of information that an OBDD will have to remember when evaluating φ following any variable ordering, which we will formalize via dncpi-sets. This being said, the definition of pathsplitwidth is also reminiscent of that of pathwidth, and we can indeed connect the two (up to a factor of the arity):

▶ **Lemma 18.** *For any hypergraph $H = (V, E)$, we have* $\mathrm{pw}(H) \leqslant \mathrm{arity}(H) \times \mathrm{psw}(H)$.

Proof sketch. From a variable ordering $\mathbf{v}$, we construct a path decomposition of H by creating $|V|$ bags in sequence, each of which containing v_i plus $\bigcup \mathrm{Split}_i(\mathbf{v}, H)$. The width is $\leqslant \mathrm{arity}(H) \times \mathrm{psw}(H)$, and we check the two conditions of path decompositions. First, each hyperedge of H is contained in a bag where it is split. Second, each vertex v_i occurs in the corresponding bag b_i and at all positions where the edges containing v are split, which forms a segment of $\mathbf{v}$: thus, the connectedness condition of tree decompositions is respected. ◀

Thanks to Lemma 18, it suffices to show that an OBDD for φ has width $\geqslant 2^{\left\lfloor \frac{\mathrm{psw}(\varphi)}{(a \times d)^2} \right\rfloor}$, which we will do in the rest of this section.

dncpi-sets. To show this lower bound, we use the technical tool of *dncpi-sets* [7, 1]. We recall the definitions here, adapting the notation slightly. Remember that our monotone DNFs are assumed to be minimized. Note that dncpi-sets are reminiscent of *subfunction width* in [15] (see Theorem 17 in [15]), but the latter notion is only defined for graph CNFs.

▶ **Definition 19** ([1, Definition 6.4.6])**.** Given a monotone DNF φ on variables V, a *disjoint non-covering prime implicant set* (dncpi-set) of φ is a set S of clauses of φ which:

- are pairwise disjoint: for any $D_1 \neq D_2$ in S, we have $D_1 \cap D_2 = \emptyset$.
- are *non-covering* in the following sense: for any clause D of φ, if $D \subseteq \bigcup S$, then $D \in S$.

The *size* of S is the number of clauses that it contains.

Given a variable ordering $\mathbf{v}$ of V, we say that $\mathbf{v}$ *shatters* a dncpi-set S if there exists $1 \leqslant i \leqslant |V|$ such that $S \subseteq \mathrm{Split}_i(\mathbf{v}, H)$, where H is the hypergraph of φ.

We recall the main result on dncpi-sets:

▶ **Lemma 20** ([1, Lemma 6.4.7])**.** *Let φ be a monotone DNF on variables V and $n \in \mathbb{N}$. Assume that, for every variable ordering $\mathbf{v}$ of V, there is some dncpi-set S of φ with $|S| \geqslant n$, such that $\mathbf{v}$ shatters S. Then any OBDD for φ has width $\geqslant 2^n$.*

Proof sketch. Considering the point at which the dncpi-set is shattered, the OBDD must remember exactly the status of each clause of the set: any valuation that satisfies a subset of these clauses gives rise to a different continuation function. This is where we use the fact that the DNF is monotone: it ensures that we can freely choose a valuation of the variables that do not occur in the dncpi-set without making the formula true. ◀

Concluding the proof. We conclude the proof of Theorem 15 by showing that any variable ordering of the variables of a monotone DNF φ shatters a dncpi-set of the right size. The formal statement is as follows, and it is the last result to prove:

▶ **Lemma 21.** *Let φ be a monotone DNF, H its hypergraph, and $\mathbf{v}$ an enumeration of its variables. Then there is a dncpi-set S of φ shattered by $\mathbf{v}$ such that $|S| \geqslant \left\lfloor \frac{\mathrm{psw}(H)}{(\mathrm{arity}(H) \times \mathrm{degree}(H))^2} \right\rfloor$.*

We prove this result in the rest of the section. Our goal is to construct a dncpi-set, which intuitively consists of clauses that are disjoint and which do not cover another clause. We can do so by picking clauses sufficiently "far apart". Let the *exclusion graph* of $H = (V, E)$ be the graph on E where two edges $e \neq e'$ are adjacent if there is an edge e'' of E with which they both share a node: this is in particular the case when e and e' intersect as we can take $e'' := e$. Formally, the exclusion graph is $G_H = (E, \{\{e, e'\} \in E^2 \mid e \neq e' \wedge \exists e'' \in E, (e \cap e'') \neq \emptyset \wedge (e' \cap e'') \neq \emptyset\})$. In other words, two hyperedges are adjacent in G_H iff they are different and are at distance at most 4 in the incidence graph of H.

Remember that an *independent set* in the graph G_H is a subset S of E such that no two elements of S are adjacent in G_H. The definition of G_H then ensures:

▶ **Lemma 22.** *For any monotone DNF φ, letting H be its hypergraph, any independent set of the exclusion graph G_H is a dncpi-set of φ.*

In other words, our goal is to compute a large independent set of the exclusion graph. To do this, we will use the following straightforward lemma about independent sets:

▶ **Lemma 23.** *Let $G = (V, E)$ be a graph and let $V' \subseteq V$. Then G has an independent set $S \subseteq V'$ of size at least $\left\lfloor \frac{|V'|}{\mathrm{degree}(G)+1} \right\rfloor$.*

Moreover, we can bound the degree of G_H using the degree and arity of H:

▶ **Lemma 24.** *Let H be a hypergraph. Then* $\mathrm{degree}(G_H) \leqslant (\mathrm{arity}(H) \times \mathrm{degree}(H))^2 - 1$.

Proof sketch. The bound on the arity and degree of H implies a bound on the number of edges that can be at distance $\leqslant 4$ of another edge in the incidence graph of H, hence bounding the degree of the exclusion graph. ◀

We are now ready to conclude the proof of Lemma 21, which combined with Lemma 18 and Lemma 20 concludes the proof of Theorem 15.

Proof of Lemma 21. Let φ be a monotone DNF, $H = (V, E)$ its hypergraph, and $\mathbf{v}$ an enumeration of its variables. By definition of pathsplitwidth, there is $v_i \in V$ such that, for $E' := \mathrm{Split}_i(\mathbf{v}, H)$, we have $|E'| \geqslant \mathrm{psw}(H)$. Now, by Lemma 23, G_H has an independent set $S \subseteq E'$ of size at least $\left\lfloor \frac{|E'|}{\mathrm{degree}(G_H)+1} \right\rfloor$ which is $\geqslant \left\lfloor \frac{\mathrm{psw}(H)}{(\mathrm{arity}(H)\times\mathrm{degree}(H))^2} \right\rfloor$ by Lemma 24. Hence, S is a dncpi-set by Lemma 22, has the desired size, and is shattered since $S \subseteq E'$. ◀

From DNFs to CNFs. We now argue that Theorem 15 also holds for monotone CNFs. Let φ be a monotone CNF, $a := \mathrm{arity}(\varphi)$ and $d := \mathrm{degree}(\varphi)$, and suppose for a contradiction that there is an OBDD O for φ of width $< 2^{\left\lfloor \frac{\mathrm{pw}(\varphi)}{a^3 \times d^2} \right\rfloor}$. Consider the monotone DNF φ' built from φ by replacing each $\wedge$ by a $\vee$ and each $\vee$ by a $\wedge$. Now, let O' be the OBDD built from O by replacing the label $b \in \{0, 1\}$ of each edge by $1 - b$, and replacing the label b of each leaf by $1 - b$. It is clear, by De Morgan's laws, that O' is an OBDD for φ' of size $< 2^{\left\lfloor \frac{\mathrm{pw}(\varphi)}{a^3 \times d^2} \right\rfloor}$, which contradicts Theorem 15 applied to monotone DNFs.

6 Lower Bounds on d-SDNNFs

In the previous section, we have shown that *pathwidth* measures how concisely an OBDD can represent a monotone DNF or CNF formula with bounded degree and arity. In this section, we move from OBDDs to (d-)SDNNFs, and show that *treewidth* plays a similar role to pathwidth in this setting. Formally, we show the following analogue of Theorem 15:

▶ **Theorem 25.** *Let φ be a monotone DNF (resp., monotone CNF), let $a :=$* $\mathrm{arity}(\varphi)$ *and $d :=$* $\mathrm{degree}(\varphi)$*. Then any d-SDNNF (resp., SDNNF) for φ has size $\geqslant 2^{\left\lfloor \frac{\mathrm{tw}(\varphi)}{3 \times a^3 \times d^2} \right\rfloor} - 1$.*

Combined with Theorem 5 (or with existing results specific to CNF formulas such as [14, Corollary 1]), this yields an analogue of Corollary 16. However, its statement is less neat: unlike OBDDs, (d-)SDNNFs have no obvious notion of width, so the lower bound above refers to size rather than width, and it does not exactly match our upper bound. We obtain:

▶ **Corollary 26.** *For any constant c, for any monotone DNF (resp., monotone CNF) φ with arity and degree bounded by c, there is a d-SDNNF for φ having size $|\varphi| \times 2^{O(\mathrm{tw}(\varphi))}$, and any d-SDNNF (resp., SDNNF) for φ has size $2^{\Omega(\mathrm{tw}(\varphi))}$.*

Our proof of Theorem 25 will follow the same overall structure as in the previous section. We present the proof for monotone DNFs and d-DNNFs: see the extended version [8] for the extension to monotone CNFs and SDNNFs. Recall that d-SDNNFs are structured by *v-trees*, which generalize variable orders. We first introduce *treesplitwidth*, a width notion that measures the performance of a v-tree by counting how many clauses it splits; and we connect treesplitwidth to treewidth. We use again dncpi-sets, and argue that a d-SDNNF structured by a v-tree must shatter a dncpi-set whose size follows the treesplitwidth of the v-tree. We then show that shattering a dncpi-set forces d-SDNNFs to be large: instead of the easy OBDD result of the previous section (Lemma 20), we will need a much deeper result of Pipatsrisawat and Darwiche [36, Theorem 3], rephrased in the setting of communication complexity by Bova, Capelli, Mengel, and Slivovsky [13].

Note that [13], by a similar approach, shows an exponential lower bound on the size of d-SDNNF which is reminiscent of ours. However, their bound again applies to one well-chosen family of Boolean functions. In essence, our result is shown by observing that the family of functions used in their lower bound occurs "within" any bounded-degree, bounded-arity monotone DNF. Also note that a result similar to the lower bound of Corollary 26 is proven by Capelli [19, Corollary 6.35] as an auxiliary statement to separate structured DNNFs and FBDDs. The result uses *MIM-width*, but Theorem 4.2.5 of [40], as degree and arity are bounded, implies that we could rephrase it to treewidth; further, the result assumes arity-2 formulas, but it could be extended to arbitrary arity as in [20, Theorem 12]. More importantly, the result applies only to monotone CNFs and not to DNFs .

Treesplitwidth. Informally, treesplitwidth is to v-trees what pathsplitwidth is to variable orders: it bounds the "best performance" of any v-tree.

▶ **Definition 27.** Let $H = (V, E)$ be a hypergraph, and T be a v-tree over V. For any node n of T, we define $\mathrm{Split}_n(T, H)$ as the set of hyperedges e of H that contain both a variable in T_n and one outside T_n (recall that T_n denotes the subtree of T rooted at n). Formally: $\mathrm{Split}_n(T, H) := \{e \in E \mid \exists v_\mathrm{i} \in \mathsf{Leaves}(T_n)$ and $\exists v_\mathrm{o} \in \mathsf{Leaves}(T \setminus T_n)$ such that $\{v_\mathrm{i}, v_\mathrm{o}\} \subseteq e\}$.

The *treesplitwidth* of T relative to H is $\mathrm{tsw}(T, H) := \max_{n \in T} |\mathrm{Split}_n(T, H)|$. The *treesplitwidth* $\mathrm{tsw}(H)$ of H is then the minimum of $\mathrm{tsw}(T, H)$ over all v-trees T of V.

Again, the treesplitwidth of H is exactly the *branch-width* [38] of the dual hypergraph of H, but treesplitwidth is more convenient for our proofs. As with pathsplitwidth and pathwidth (Lemma 18), we can bound the treewidth of a hypergraph by its treesplitwidth:

▶ **Lemma 28.** *For any hypergraph $H = (V, E)$, we have $\mathrm{tw}(H) \leqslant 3 \times \mathrm{arity}(H) \times \mathrm{tsw}(H)$.*

Moreover, using the same techniques that we used in the last section, we can show the analogue of Lemma 21. Specifically, given a monotone DNF φ on variables V, a v-tree T over V, and a dncpi-set S of φ, we say that T *shatters* S if there is a node n in T such that $S \subseteq \mathrm{Split}_n(T, \varphi)$. Reusing Lemmas 22, 23, and 24, we can show that any v-tree over V must shatter a large dncpi-set (depending on the treewidth, degree, and arity):

▶ **Lemma 29.** *Let φ be a monotone DNF, H its hypergraph, and T be a v-tree over its variables. Then there is a dncpi-set S of φ shattered by T such that $|S| \geqslant \left\lfloor \frac{\mathrm{tsw}(H)}{(\mathrm{arity}(H) \times \mathrm{degree}(H))^2} \right\rfloor$.*

Hence, to prove Theorem 25, the only missing ingredient is a lower bound on the size of d-SDNNFs that shatter large dncpi-sets. Specifically, we need an analogue of Lemma 20:

▶ **Lemma 30.** *Let φ be a monotone DNF on variables V and $n \in \mathbb{N}$. Assume that, for every v-tree T over V, there is some dncpi-set S of φ with $|S| \geqslant n$, such that T shatters S. Then any d-SDNNF for φ has size $\geqslant 2^n - 1$.*

We will prove Lemma 30 in the rest of this section using a recent lower bound by Bova, Capelli, Mengel, and Slivovsky [13]. They bound the size of any d-SDNNF for the *set intersection* function, defined as $\mathrm{SINT}_n := (x_1 \wedge y_1) \vee \ldots \vee (x_n \wedge y_n)$. This bound is useful for us: a dncpi-set intuitively isolates some variables on which φ computes exactly SINT_n:

▶ **Lemma 31.** *Let φ be a DNF with variables V, and let $S = \{D_1, \ldots, D_n\}$ be a dncpi-set of φ where every clause has size $\geqslant 2$. Pick two variables $x_i \neq y_i$ in D_i for each $1 \leqslant i \leqslant n$, and let $V' := \{x_1, y_1, \ldots, x_n, y_n\}$. Then there is a partial valuation ν of V with domain $V \setminus V'$ such that $\nu(\varphi) = \mathrm{SINT}_n$.*

Proof sketch. The valuation ν sets to 1 the variables V'' which are in $\bigcup S$ but not in V', and sets to 0 all remaining variables. This amounts to discarding the clauses not in the dncpi-set, and discarding the variables of V'' in the dncpi-set: what remains of the DNF is then precisely SINT_n. Note that this result relies on monotonicity, and on the fact that φ is a DNF. (However, in the full version [8], we show a dual result for monotone CNF.) ◀

This observation allows us to leverage the bound of [13] on the size of d-SDNNFs that compute SINT_n, assuming that they are structured by an "inconvenient" v-tree:

▶ **Proposition 32** ([13, Proposition 14])**.** *Let $X_n = \{x_1, \ldots, x_n\}$ and $Y_n = \{y_1, \ldots, y_n\}$ for $n \in \mathbb{N}$, and let T be a v-tree over $X_n \sqcup Y_n$ such that there exists a node $n \in T$ with $X_n \subseteq \mathsf{Leaves}(T_n)$ and $Y_n \subseteq \mathsf{Leaves}(T \setminus T_n)$. Then any d-SDNNF structured by T computing SINT_n has size $\geqslant 2^n - 1$.*

In our setting, an "inconvenient" v-tree for a dncpi-set is one that shatters it: each clause of the dncpi-set is then partitioned in two non-empty subsets where we can pick x_i and y_i for Lemma 31. Hence, when every v-tree shatters a large dncpi-set of φ, Proposition 32 allows us to deduce the lower bound on the size of every d-SDNNF for φ. We have thus shown Lemma 30, and this concludes the proof of Theorem 25 (in the DNF case).

7 Application to Query Lineages

In this section, we adapt the lower bound of the previous section to the computation of query lineages on relational instances. Like in [7], for technical reasons, we must assume a graph signature. We first recall some preliminaries and then state our result.

We fix a *graph signature* σ of relation names and arities in $\{1, 2\}$, with at least one relation of arity 2. An *instance* I on σ is a finite set of *facts* of the form $R(a_1, \ldots, a_n)$ for n the arity of R; we call $a_1, \ldots, a_n$ *elements* of I. An instance I' is a *subinstance* of I if $I' \subseteq I$. The *treewidth* $\mathrm{tw}(I)$ of I is that of its *Gaifman graph*, which has the elements of I as vertices and has one edge between each pair of elements that co-occur in some fact of I.

A *Boolean conjunctive query* (CQ) is an existentially quantified conjunction of *atoms* of the form $R(x_1, \ldots, x_n)$ where the x_i are *variables*. A *UCQ* is a disjunction of CQs, and a $\mathrm{UCQ}^{\neq}$ also allows atoms of the form $x \neq y$. A $\mathrm{UCQ}^{\neq}$ is *connected* if the Gaifman graph of each disjunct (seen as an instance, and ignoring $\neq$-atoms) is connected. For instance, letting σ_R consist of one arity-2 relation R, the following connected $\mathrm{UCQ}^{\neq}$ tests if there are

two facts that share one element: $Q_{\text{p}} : \exists xyz\ (R(x,y) \vee R(y,x))) \wedge (R(y,z) \vee R(z,y)) \wedge x \neq z$. (While Q_{p} is not given as a disjunction of CQs, it can be rewritten to one using distributivity.)

The *lineage* of a UCQ$^{\neq}$ Q over I is a Boolean formula $\varphi(Q, I)$ on the facts of I that maps each Boolean valuation $\nu : I \to \{0,1\}$ to 1 or 0 depending on whether I_ν satisfies Q or not, where $I_\nu := \{F \in I \mid \nu(F) = 1\}$. The lineage intuitively represents which facts of I suffice to satisfy Q. Lineages are useful to evaluate queries on *probabilistic databases* [39]: we can obtain the probability of the query from an OBDD or d-DNNF representing its lineage.

We study when query lineages can be computed efficiently in *data complexity*, i.e., as a function of the input instance, with the query being fixed. A first question asks which *queries* have *tractable lineages* on all instances: Jha and Suciu [32, Theorem 3.9] showed that *inversion-free* UCQ$^{\neq}$ queries admit OBDD representations in this sense, and Bova and Szeider [16, Theorem 5] have recently shown that UCQ$^{\neq}$ queries with inversions do not even have tractable d-SDNNF lineages. A second question asks which *instance classes* ensure that *all queries* have tractable lineages on them. This was studied for OBDD representations in [7]: bounded-treewidth instances have tractable OBDD lineage representations for any MSO query ([7, Theorem 6.5], using [32]); conversely there are *intricate* queries (a class of connected UCQ$^{\neq}$ queries) whose lineages never have tractable OBDD representations in the instance treewidth [7, Theorem 8.7]. The query Q_{p} above is an example of an intricate query on the signature σ_R (refer to [7, Definition 8.5] for the formal definition of intricate queries). This result shows that we must bound instance treewidth for all queries to have tractable OBDDs, but leaves the question open for more expressive lineage representations.

Our bound in the previous section allows us to extend Theorem 8.7 of [7] from OBDDs to d-SDNNFs, yielding the following:

▶ **Theorem 33.** *There is a constant $d \in \mathbb{N}$ such that the following is true. Let σ be an arity-2 signature, and Q a connected* UCQ$^{\neq}$ *which is intricate on σ. For any instance I on σ, any d-SDNNF representing the lineage of Q on I has size $2^{\Omega(\text{tw}(I)^{1/d})}$.*

Proof sketch. As in [7], we use a result of Chekuri and Chuzhoy [21] to show that the Gaifman graph of I has a degree-3 topological minor S of treewidth $\Omega(\text{tw}(I)^{1/d})$ for some constant $d \in \mathbb{N}$; we also ensure that S has sufficiently high *girth* relative to Q. We focus on a subinstance I' of I that corresponds to S: this suffices to show our lower bound, because we can always compute a tractable representation of $\varphi(Q, I')$ from one of $\varphi(Q, I)$. Now, we can represent $\varphi(Q, I')$ as a minimized DNF ψ by enumerating its minimal matches: ψ has constant arity because the number of atoms of Q is fixed, and it has constant degree because S has constant degree and Q is connected. Further, as Q is intricate and I' has high girth relative to Q, we can ensure that this DNF has treewidth $\Omega(\text{tw}(I'))$. We conclude by Theorem 25: d-SDNNFs representing $\varphi(Q, I')$, hence $\varphi(Q, I)$, have size $2^{\Omega(\text{tw}(I)^{1/d})}$. ◀

To summarize, given an instance family $\mathcal{I}$ satisfying the constructibility requirement of Theorem 8.1 of [7], there are two regimes: (i.) $\mathcal{I}$ has bounded treewidth and then all MSO queries have d-SDNNF lineages on instances of $\mathcal{I}$ that are computable in linear time; or (ii.) the treewidth is unbounded and then there are UCQ$^{\neq}$ queries (the intricate ones) whose lineages on instances of $\mathcal{I}$ have no d-SDNNF representations polynomial in the instance size.

8 Conclusion

We have shown tight connections between structured circuit classes and width measures on circuits. We constructively rewrite bounded-treewidth circuits to d-SDNNFs in time linear in the circuit and singly exponential in the treewidth, and show matching lower bounds for

arbitrary monotone CNFs or DNFs under degree and arity assumptions; we also show a lower bound for pathwidth and OBDDs. Our results have applications to rich query evaluation: probabilistic query evaluation, computation of lineages, enumeration, etc.

Our work also raises a number of open questions. First, the d-SDNNF obtained in the proof of Theorem 5 does *not* respect the definition of a *sentential decision diagram* (SDD) [24]. Can this be fixed, and Theorem 5 extended to SDDs? Or is it impossible, which could solve the open question [11] of separating SDDs and d-SDNNFs? Second, can we weaken the hypotheses of bounded degree and arity in Corollaries 16 and 26, and can we rephrase the latter to a notion of (d-)SDNNF width to match more closely the statement of the former? Last, Section 7 shows that d-SDNNF representations of the lineages of intricate queries are exponential in the treewidth; we conjecture a similar result for pathwidth and OBDDs, but this would require a pathwidth analogue of the minor extraction results of [21].

References

1 Antoine Amarilli. *Leveraging the Structure of Uncertain Data*. PhD thesis, Télécom ParisTech, 2016.

2 Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. A circuit-based approach to efficient enumeration. In *ICALP*, 2017.

3 Antoine Amarilli, Pierre Bourhis, Mikaël Monet, and Pierre Senellart. Combined tractability of query evaluation via tree automata and cycluits (Extended version). *CoRR*, abs/1612.04203, 2017. Extended version of [4].

4 Antoine Amarilli, Pierre Bourhis, Mikaël Monet, and Pierre Senellart. Combined tractability of query evaluation via tree automata and cycluits. In *ICDT*, 2017.

5 Antoine Amarilli, Pierre Bourhis, and Pierre Senellart. Provenance circuits for trees and treelike instances (Extended version). *CoRR*, abs/1511.08723, 2015. Extended version of [6].

6 Antoine Amarilli, Pierre Bourhis, and Pierre Senellart. Provenance circuits for trees and treelike instances. In *ICALP*, 2015.

7 Antoine Amarilli, Pierre Bourhis, and Pierre Senellart. Tractable lineages on treelike instances: limits and extensions. In *PODS*, 2016.

8 Antoine Amarilli, Mikaël Monet, and Pierre Senellart. Connecting width and structure in knowledge compilation (Extended version). *CoRR*, abs/1709.06188, 2017.

9 Paul Beame and Vincent Liew. New limits for knowledge compilation and applications to exact model counting. In *UAI*, 2015.

10 Hans L Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6), 1996.

11 Simone Bova. SDDs are exponentially more succinct than OBDDs. In *AAAI*, 2016.

12 Simone Bova, Florent Capelli, Stefan Mengel, and Friedrich Slivovsky. A strongly exponential separation of DNNFs from CNF formulas. *CoRR*, abs/1411.1995, 2015.

13 Simone Bova, Florent Capelli, Stefan Mengel, and Friedrich Slivovsky. Knowledge compilation meets communication complexity. In *IJCAI*, 2016.

14 Simone Bova and Friedrich Slivovsky. On compiling structured CNFs to OBDDs. In *CSR*, 2015.

15 Simone Bova and Friedrich Slivovsky. On compiling structured CNFs to OBDDs. *TCS*, 61(2), 2017.

16 Simone Bova and Stefan Szeider. Circuit treewidth, sentential decision, and query compilation. In *PODS*, 2017.

17 Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3), 1992.

18 Andrea Calì, Florent Capelli, and Igor Razgon. Non-FPT lower bounds for structural restrictions of decision DNNF. *CoRR*, abs/1708.07767, 2017.

19 Florent Capelli. *Structural restrictions of CNF-formulas: applications to model counting and knowledge compilation*. PhD thesis, Université Paris-Diderot, 2016.

20 Florent Capelli. Understanding the complexity of #SAT using knowledge compilation. In *LICS*, 2017.

21 Chandra Chekuri and Julia Chuzhoy. Polynomial bounds for the grid-minor theorem. In *STOC*, 2014.

22 Adnan Darwiche. On the tractable counting of theory models and its application to truth maintenance and belief revision. *J. Applied Non-Classical Logics*, 11(1-2), 2001.

23 Adnan Darwiche. A differential approach to inference in Bayesian networks. *JACM*, 50(3), 2003.

24 Adnan Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *IJCAI*, 2011.

25 Srinivas Devadas. Comparing two-level and ordered binary decision diagram representations of logic functions. *IEEE TCAD*, 12(5), 1993.

26 Andrea Ferrara, Guoqiang Pan, and Moshe Y Vardi. Treewidth in verification: Local vs. global. In *LPAR*, 2005.

27 Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *TPLP*, 15(3), 2015.

28 Jörg Flum, Markus Frick, and Martin Grohe. Query evaluation via tree-decompositions. *J. ACM*, 49(6), 2002.

29 Martin Grohe and Dániel Marx. On tree width, bramble size, and expansion. *J. Combinatorial Theory, Series B*, 99(1), 2009.

30 Abhay Kumar Jha, Dan Olteanu, and Dan Suciu. Bridging the gap between intensional and extensional query evaluation in probabilistic databases. In *EDBT*, 2010.

31 Abhay Kumar Jha and Dan Suciu. Knowledge compilation meets database theory: compiling queries to decision diagrams. In *ICDT*, 2011.

32 Abhay Kumar Jha and Dan Suciu. On the tractability of query compilation and bounded treewidth. In *ICDT*, 2012.

33 Steffen L. Lauritzen and David J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *J. Royal Statistical Society. Series B*, 1988.

34 Joakim Alme Nordstrand. Exploring graph parameters similar to tree-width and path-width. Master's thesis, University of Bergen, 2017.

35 Knot Pipatsrisawat and Adnan Darwiche. New compilation languages based on structured decomposability. In *AAAI*, 2008.

36 Knot Pipatsrisawat and Adnan Darwiche. A lower bound on the size of decomposable negation normal form. In *AAAI*, 2010.

37 Igor Razgon. On OBDDs for CNFs of bounded treewidth. In *KR*, 2014.

38 Neil Robertson and P.D Seymour. Graph minors X. Obstructions to tree-decomposition. *Journal of Combinatorial Theory, Series B*, 52(2), 1991.

39 Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. *Probabilistic Databases*. Morgan & Claypool, 2011.

40 Martin Vatshelle. *New width parameters of graphs*. PhD thesis, University of Bergen, 2012.

A More General Theory of Static Approximations for Conjunctive Queries

Pablo Barceló

Millenium Institute for Foundational Research on Data, DCC, University of Chile, Santiago, Chile
pbarcelo@dcc.uchile.cl

Miguel Romero

University of Oxford, Oxford, UK
miguel.romero@cs.ox.ac.uk

Thomas Zeume

TU Dortmund, Dortmund, Germany
thomas.zeume@cs.tu-dortmund.de

Abstract

Conjunctive query (CQ) evaluation is NP-complete, but becomes tractable for fragments of bounded hypertreewidth. If a CQ is hard to evaluate, it is thus useful to evaluate an approximation of it in such fragments. While underapproximations (i.e., those that return correct answers only) are well-understood, the dual notion of overapproximations that return complete (but not necessarily sound) answers, and also a more general notion of approximation based on the symmetric difference of query results, are almost unexplored. In fact, the decidability of the basic problems of evaluation, identification, and existence of those approximations, is open.

We develop a connection with existential pebble game tools that allows the systematic study of such problems. In particular, we show that the evaluation and identification of overapproximations can be solved in polynomial time. We also make progress in the problem of existence of overapproximations, showing it to be decidable in 2EXPTIME over the class of acyclic CQs. Furthermore, we look at when overapproximations do not exist, suggesting that this can be alleviated by using a more liberal notion of overapproximation. We also show how to extend our tools to study symmetric difference approximations. We observe that such approximations properly extend under- and over-approximations, settle the complexity of its associated identification problem, and provide several results on existence and evaluation.

2012 ACM Subject Classification Information systems → Structured Query Language, Theory of computation → Database query languages (principles), Theory of computation → Database theory → Database query processing and optimization (theory)

Keywords and phrases conjunctive queries, hypertreewidth, approximations, pebble games

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.7

Acknowledgements Barceló and Romero have been funded by Millennium Nucleus Center for Semantic Web Research under Grant NC120004. Barceló has also been funded by Fondecyt grant 1170109. Zeume acknowledges the financial support by the European Research Council (ERC), grant agreement No 683080. Romero and Zeume thank the Simons Institute for the Theory of Computing for hosting them.

1 Introduction

Context. Due to the growing number of scenarios in which exact query evaluation is infeasible – e.g., when the volume of the data being queried is very large, or when queries are inherently complex – approximate query answering has become an important area of study in databases (see, e.g. [15, 20, 24, 12, 13]). Here we focus on approximate query answering for the fundamental class of conjunctive queries (CQs), for which exact evaluation is NP-complete. (Recall that CQ evaluation is the problem of given a CQ q, a database $\mathcal{D}$, and a tuple $\bar{a}$ of constants in $\mathcal{D}$, check if $\bar{a}$ belongs to $q(\mathcal{D})$, the *result* of q over $\mathcal{D}$).

It is known that the complexity of evaluation of a CQ depends on its *degree of acyclicity*, which can be formalized using different notions. One of the most general and well-studied such notions corresponds to *generalized hypertreewidth* [17]. Notably, the classes of CQs of bounded generalized hypertreewidth can be evaluated in polynomial time (see [16] for a survey). Following recent work on approximate query answering for CQs and some related query languages [5, 6], we study the process of approximating a CQ as one of bounded generalized hypertreewidth. This provides us with a certificate of efficiency for the cost of evaluating such an approximation.

It is worth noticing that our approximations are *static*, in the sense that they depend only on the CQ q and not on the underlying database $\mathcal{D}$. This has clear benefits in terms of the cost of the approximation process, as q is often orders of magnitude smaller than $\mathcal{D}$ and an approximation that has been computed once can be used for all databases. Moreover, it allows us to construct a principled approach to CQ approximation based on the well-studied notion of CQ containment [8]. Recall that a CQ q is *contained* in a CQ q', written $q \subseteq q'$, if $q(\mathcal{D}) \subseteq q'(\mathcal{D})$ over each database $\mathcal{D}$. This notion constitutes the theoretical basis for the study of several CQ optimization problems [1].

We denote by $\mathsf{GHW}(k)$ the class of CQs of generalized hypertreewidth at most k, for $k \geq 1$. As mentioned above, we look for an approximation of a CQ q in $\mathsf{GHW}(k)$. A formalization of this notion was first introduced in [4], based on the following partial order $\sqsubseteq_q$ over the set of CQs in $\mathsf{GHW}(k)$: if $q', q'' \in \mathsf{GHW}(k)$, then $q' \sqsubseteq_q q''$ iff over every database $\mathcal{D}$ the symmetric difference between $q(\mathcal{D})$ and $q''(\mathcal{D})$ is contained in the symmetric difference between $q(\mathcal{D})$ and $q'(\mathcal{D})$. Intuitively, this states that q'' is a better $\mathsf{GHW}(k)$-approximation of q than q'. The $\mathsf{GHW}(k)$-approximations of q then correspond to maximal elements with respect to $\sqsubseteq_q$ among a distinguished class of CQs in $\mathsf{GHW}(k)$. Three notions of approximation were introduced in [4], by imposing different "reasonable" conditions on such a class. These are:

- *Underapproximations:* In this case we look for approximations in the set of CQs q' in $\mathsf{GHW}(k)$ that are contained in q, i.e., $q' \subseteq q$. This ensures that the evaluation of such approximations always produce correct (but not necessarily complete) answers to q. A $\mathsf{GHW}(k)$-underapproximation of q is then a CQ q' amongst these CQs that is *maximal* with respect to the partial order defined by $\sqsubseteq_q$. Noticeably, the latter coincides with being maximal with respect to the containment partial order $\subseteq$ among the CQs in $\mathsf{GHW}(k)$ that are contained in q; i.e., no other CQ in such a set strictly contains q'.
- *Overapproximations:* This is the dual notion of underapproximations, in which we look for minimal elements in the class of CQs q' in $\mathsf{GHW}(k)$ that contain q, i.e., $q \subseteq q'$. Hence, $\mathsf{GHW}(k)$-overapproximations produce complete (but not necessarily correct) answers to q.
- *Symmetric difference approximations:* While underapproximations must be contained in the original query, and overapproximations must contain it, symmetric difference approximations do not impose any constraint on approximations with respect to the partial order $\subseteq$. Thus, a symmetric difference $\mathsf{GHW}(k)$-approximation of q – or simply

$\mathsf{GHW}(k)$-Δ-approximation from now on – is a maximal CQ in $\mathsf{GHW}(k)$ with respect to the partial order $\sqsubseteq_q$.

The approximations presented above provide "qualitative" guarantees for evaluation, as they are as close as possible to q among all CQs in $\mathsf{GHW}(k)$ of a certain kind. In particular, under- and overapproximations are dual notions which provide lower and upper bounds for the exact evaluation of a CQ, while Δ-approximations can give us useful information when the quality of the result of the under- and overapproximations is poor. Then, in order to develop a robust theory of bounded hypertreewidth static approximations for CQs, it is necessary to have a good understanding of all three notions.

The notion of underapproximation is by now well-understood [5]. Indeed, it is known that for each $k \geq 1$ the $\mathsf{GHW}(k)$-underapproximations have good properties that justify their application: (a) they always exist, and (b) evaluating all $\mathsf{GHW}(k)$-underapproximations of a CQ q over a database $\mathcal{D}$ is *fixed-parameter tractable* with the size of q as parameter. This is an improvement over general CQ evaluation for which the latter is believed not to hold [26].

The notions of $\mathsf{GHW}(k)$-overapproximations and $\mathsf{GHW}(k)$-Δ-approximations, while already introduced in [4], are much less understood. No general tools have been identified so far for studying the decidability of basic problems such as:

- *Existence:* Does CQ q have a $\mathsf{GHW}(k)$-overapproximation (or $\mathsf{GHW}(k)$-Δ-approximation)?
- *Identification:* Is q' a $\mathsf{GHW}(k)$-overapproximation (or $\mathsf{GHW}(k)$-Δ-approximation) of q?
- *Evaluation:* Given a CQ q, a database $\mathcal{D}$, and a tuple $\bar{a}$ in $\mathcal{D}$, is it the case that $\bar{a} \in q'(\mathcal{D})$, for some $\mathsf{GHW}(k)$-overapproximation (resp., $\mathsf{GHW}(k)$-Δ-approximation) q' of q?

Partial results were obtained in [4], but based on ad-hoc tools. It has also been observed that some CQs have no $\mathsf{GHW}(k)$-overapproximations (in contrast to underapproximations, that always exist), which was seen as a negative result.

Contributions. We develop tools for the study of overapproximations and Δ-approximations. While we mainly focus on the former, we provide a detailed account of how our techniques can be extended to deal with the latter. In the context of $\mathsf{GHW}(k)$-overapproximations, we apply our tools to pinpoint the complexity of evaluation and identification, and make progress in the problem of existence. We also study when overapproximations do not exist and suggest how this can be alleviated. Our contributions are as follows:

1. *Link to existential pebble games.* We establish a link between $\mathsf{GHW}(k)$-overapproximations and *existential pebble games* [22]. Such games have been used to show that CQs of bounded width can be evaluated efficiently [11, 9]. Using the fact that the existence of winning conditions in the existential pebble game can be checked in PTIME [9], we show that identification and evaluation for $\mathsf{GHW}(k)$-overapproximations are tractable problems.
2. *A more liberal notion of overapproximation.* We observe that non-existence of overapproximations is due to the fact that in some cases overapproximations require expressing conjunctions of infinitely many atoms. By relaxing our notion, we get that each CQ q has a (potentially infinite) $\mathsf{GHW}(k)$-overapproximation q'. This q' is unique (up to equivalence). Further, it can be evaluated efficiently – in spite of being potentially infinite – by checking a winning condition for the existential k-pebble game on q and $\mathcal{D}$.
3. *Existence of overapproximations.* It is still useful to check if a CQ q has a *finite* $\mathsf{GHW}(k)$-overapproximation q', and compute it if possible. This might allow us to optimize q' before evaluating it. There is also a difference in complexity, as existential pebble game techniques are PTIME-complete in general [21], and thus inherently sequential, while evaluation of CQs in $\mathsf{GHW}(k)$ is highly parallelizable (Gottlob et al. [17]).

Table 1 Summary of results on under- and overapproximations of bounded generalized hypertreewidth. The complexity of identification coincides with that of evaluation in both cases. New results are marked with (*). All remaining results follow from [4, 5].

	Existence?	**Unique?**	**Evaluation**	**Existence check**
GHW(k)-*underapp.*	always	not always	NP-hard	N/A
GHW(k)-*overapp.*	not always	always	PTIME*	For $k = 1$: 2EXPTIME* PTIME* on binary schemas For $k > 1$: Open

By exploiting automata techniques, we show that checking if a CQ q has a (finite) GHW(1)-overapproximation q' is in 2EXPTIME. Also, when such q' exists it can be computed in 3EXPTIME. This is important since GHW(1) coincides with the well-known class of *acyclic* CQs [27]. If the arity of the schema is fixed, these bounds drop to EXPTIME and 2EXPTIME, respectively. Also, we look at the case of binary schemas, such as the ones used in *graph databases* [3] and *description logics* [2]. In this case, GHW(1)-overapproximations can be computed efficiently via a greedy algorithm. This is optimal, as over ternary schemas we prove an exponential lower bound for the size of GHW(1)-overapproximations. We do not know if the existence problem is decidable for $k > 1$. However, we show that it can be recast as an unexplored boundedness condition for the existential pebble game. Understanding the decidability boundary for such conditions is often difficult [25, 7].

Table 1 shows a summary of these results in comparison with underapproximations.

Our contributions for GHW(k)-Δ-approximations are as follows. As a preliminary step, we show that GHW(k)-under and GHW(k)-overapproximations are particular cases of GHW(k)-Δ-approximations, but not vice versa. Afterwards, as for GHW(k)-overapproximations, we provide a link between GHW(k)-Δ-overapproximations and the existential pebble game, and use it to characterize when a CQ q has at least one GHW(k)-Δ-approximation that is neither a GHW(k)-underapproximation nor a GHW(k)-overapproximation (a so-called *incomparable* GHW(k)-Δ-*approximation*). This allows us to show that the identification problem for such Δ-approximations is coNP-complete. As for the problem of checking for the existence of incomparable GHW(k)-Δ-approximations, we extend our automata techniques to prove that it is in 2EXPTIME for $k = 1$ (and in EXPTIME for fixed-arity schemas). In case such a GHW(1)-Δ-approximation exists, we can evaluate it using a fixed-parameter tractable algorithm. We also provide results on existence and evaluation of infinite incomparable GHW(1)-Δ-approximations.

Organization. Section 2 contains preliminaries. Basic properties of overapproximations are presented in Section 3, while the existence of overapproximations is studied in Section 4. In Section 5 we deal with Δ-approximations, and conclude in Section 6 with final remarks.

2 Preliminaries

Relational databases and homomorphisms. A *relational schema* σ is a finite set of relation symbols, each one of which has an arity $n > 0$. A *database* $\mathcal{D}$ over σ is a finite set of atoms of the form $R(\bar{a})$, where R is a relation symbol in σ of arity n and $\bar{a}$ is an n-tuple of constants. We often abuse notation and write $\mathcal{D}$ also for the set of elements in $\mathcal{D}$.

Let $\mathcal{D}$ and $\mathcal{D}'$ be databases over σ. A *homomorphism* from $\mathcal{D}$ to $\mathcal{D}'$ is a mapping h from $\mathcal{D}$ to $\mathcal{D}'$ such that for every atom $R(\bar{a})$ in $\mathcal{D}$ it is the case that $R(h(\bar{a})) \in \mathcal{D}'$. If $\bar{a}$

and $\bar{b}$ are n-ary tuples ($n \geq 0$) in $\mathcal{D}$ and $\mathcal{D}'$, respectively, we write $(\mathcal{D}, \bar{a}) \to (\mathcal{D}', \bar{b})$ if there is a homomorphism h from $\mathcal{D}$ to $\mathcal{D}'$ such that $h(\bar{a}) = \bar{b}$. Checking if $(\mathcal{D}, \bar{a}) \to (\mathcal{D}', \bar{b})$ is a well-known NP-complete problem.

Conjunctive queries. A *conjunctive query* (CQ) over schema σ is a formula q of the form $\exists \bar{y} \bigwedge_{1 \leq i \leq m} R_i(\bar{x}_i)$, where each $R_i(\bar{x}_i)$ is an atom over σ ($1 \leq i \leq m$). We often write this as $q(\bar{x})$ to denote that $\bar{x}$ are the *free variables* of q, i.e., those that are not existentially quantified in $\bar{y}$. If $\bar{x}$ is empty, then q is *Boolean*. We define the evaluation of CQs in terms of homomorphisms. Recall that the *canonical database* $\mathcal{D}_q$ of a CQ $q = \exists \bar{y} \bigwedge_{1 \leq i \leq m} R_i(\bar{x}_i)$ consists precisely of the atoms $R_i(\bar{x}_i)$, for $1 \leq i \leq m$. The *result of q over $\mathcal{D}$*, denoted $q(\mathcal{D})$, is the set of all tuples $\bar{a}$ such that $(\mathcal{D}_q, \bar{x}) \to (\mathcal{D}, \bar{a})$. We often do not distinguish between a CQ q and its canonical database $\mathcal{D}_q$ (i.e., we write q for $\mathcal{D}_q$).

Evaluation and tractable classes of CQs. The *evaluation problem for CQs* is as follows: Given a CQ q, a database $\mathcal{D}$, and a tuple $\bar{a}$ in $\mathcal{D}$, is $\bar{a} \in q(\mathcal{D})$? Since this problem corresponds to checking if $(q, \bar{x}) \to (\mathcal{D}, \bar{a})$, it is NP-complete [8]. This led to a flurry of activity for finding classes of CQs for which evaluation is tractable.

Here we deal with one of the most studied such classes: CQs of bounded *generalized hypertreewidth* [17], also called *coverwidth* [9]. We adopt the definition of [9] which is better suited for working with non-Boolean queries. A *tree decomposition* of a CQ $q = \exists \bar{y} \bigwedge_{1 \leq i \leq m} R_i(\bar{x}_i)$ is a pair (T, χ), where T is a tree and χ is a mapping that assigns a subset of the existentially quantified variables in $\bar{y}$ to each node $t \in T$, such that:

1. For each $1 \leq i \leq m$, the variables in $\bar{x}_i \cap \bar{y}$ are contained in $\chi(t)$, for some $t \in T$.
2. For each variable y in $\bar{y}$, the set of nodes $t \in T$ for which y occurs in $\chi(t)$ is connected.

The *width* of node t in (T, χ) is the minimal size of an $I \subseteq \{1, \ldots, m\}$ such that $\bigcup_{i \in I} \bar{x}_i$ covers $\chi(t)$. The width of (T, χ) is the maximal width of the nodes of T. The *generalized hypertreewidth* of q is the minimum width of its tree decompositions.

For a fixed $k \geq 1$, we denote by $\mathsf{GHW}(k)$ the class of CQs of generalized hypertreewidth at most k. The CQs in $\mathsf{GHW}(k)$ can be evaluated in polynomial time; see [16].

Containment of CQs. A CQ q is *contained* in a CQ q', written as $q \subseteq q'$, if $q(\mathcal{D}) \subseteq q'(\mathcal{D})$ over every database $\mathcal{D}$. Two CQs q and q' are *equivalent*, denoted $q \equiv q'$, if $q \subseteq q'$ and $q' \subseteq q$.

It is known that CQ containment and CQ evaluation are, essentially, the same problem [8]. In particular, let $q(\bar{x})$ and $q'(\bar{x})$ be CQs. Then

$$q \subseteq q' \iff \bar{x} \in q'(\mathcal{D}_q) \iff (\mathcal{D}_{q'}, \bar{x}) \to (\mathcal{D}_q, \bar{x}). \tag{1}$$

Thus, $q \subseteq q'$ and $(q', \bar{x}) \to (q, \bar{x})$ (i.e., $(\mathcal{D}_{q'}, \bar{x}) \to (\mathcal{D}_q, \bar{x})$) are used interchangeably.

Approximations of CQs. Fix $k \geq 1$. Let q be a CQ. The approximations of q in $\mathsf{GHW}(k)$ are defined with respect to a partial order $\sqsubseteq_q$ over the set of CQs in $\mathsf{GHW}(k)$. Formally, for any two CQs q', q'' in $\mathsf{GHW}(k)$ we have

$$q' \sqsubseteq_q q'' \iff \Delta(q(\mathcal{D}), q''(\mathcal{D})) \subseteq \Delta(q(\mathcal{D}), q'(\mathcal{D})), \text{ for every database } \mathcal{D},$$

where $\Delta(A, B)$ denotes the symmetric difference between sets A and B. Thus, $q' \sqsubseteq_q q''$, whenever the "error" of q'' with respect to q – measured in terms of the symmetric difference between $q''(\mathcal{D})$ and $q(\mathcal{D})$ – is contained in that of q' for each database $\mathcal{D}$. As usual, we write $q' \sqsubset_q q''$ if $q' \sqsubseteq_q q''$ but $q'' \not\sqsubseteq_q q'$.

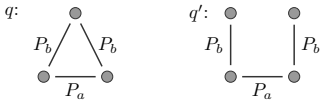


Figure 1 The CQ q and its $\mathsf{GHW}(1)$-overapproximation q' from Example 2.

The approximations of q in $\mathsf{GHW}(k)$ always correspond to maximal elements, with respect to the partial order $\sqsubseteq_q$, over a class of CQs in $\mathsf{GHW}(k)$ that satisfies certain conditions. The following three basic notions of approximation were identified in [4]:

- *Underapproximations:* Let q, q' be CQs such that $q' \in \mathsf{GHW}(k)$. Then q' is a $\mathsf{GHW}(k)$-*underapproximation* of q if it is maximal, with respect to $\sqsubseteq_q$, among all CQs in $\mathsf{GHW}(k)$ that are contained in q. That is, $q' \subseteq q$, and there is no CQ $q'' \in \mathsf{GHW}(k)$ such that $q'' \subseteq q$ and $q' \sqsubset_q q''$. In particular, the $\mathsf{GHW}(k)$-underapproximations of q produce correct (but not necessarily complete) answers with respect to q over every database $\mathcal{D}$.
- *Overapproximations:* Analogously, q' is a $\mathsf{GHW}(k)$-*overapproximation* of q if it is maximal, with respect to $\sqsubseteq_q$, among all CQs in $\mathsf{GHW}(k)$ that contain q. That is, the $\mathsf{GHW}(k)$-overapproximations of q produce complete (but not necessarily correct) answers with respect to q over every database $\mathcal{D}$.
- Δ-*approximations:* In this case we impose no restriction on q'. That is, q' is a $\mathsf{GHW}(k)$-Δ-*approximation* of q if it is maximal with respect to the partial order $\sqsubseteq_q$, i.e., there is no $q'' \in \mathsf{GHW}(k)$ such that $q' \sqsubset_q q''$.

Underapproximations and overapproximations admit an equivalent, but arguably simpler, characterization as maximal (resp., minimal) elements, with respect to the containment partial order $\subseteq$, among all CQs in $\mathsf{GHW}(k)$ that are contained in q (resp., contain q):

▶ **Proposition 1.** *[4] Fix $k \geq 1$. Let q, q' be CQs such that $q' \in \mathsf{GHW}(k)$. Then:*

- *q' is a $\mathsf{GHW}(k)$-underapproximation of q iff $q' \subseteq q$ and there is no CQ $q'' \in \mathsf{GHW}(k)$ such that $q' \subset q'' \subseteq q$.*
- *q' is a $\mathsf{GHW}(k)$-overapproximation of q iff $q \subseteq q'$ and there is no CQ $q'' \in \mathsf{GHW}(k)$ such that $q \subseteq q'' \subset q'$.*

As mentioned before, $\mathsf{GHW}(k)$-underapproximations are by now well-understood. We concentrate on $\mathsf{GHW}(k)$-overapproximations and $\mathsf{GHW}(k)$-Δ-approximations in this paper. We start by studying the former.

3 Overapproximations

Recall that $\mathsf{GHW}(k)$-overapproximations are minimal elements (in terms of $\subseteq$) in the set of CQs in $\mathsf{GHW}(k)$ that contain q. We show an example of a $\mathsf{GHW}(1)$-approximation below:

▶ **Example 2.** Figure 1 shows a CQ q and its $\mathsf{GHW}(1)$-overapproximation q'. The schema consists of binary symbols P_a and P_b. Dots represent variables, and an edge labeled P_a between x and y represents the presence of atoms $P_a(x, y)$ and $P_a(y, x)$. (Same for P_b). All variables are existentially quantified. Clearly, $q \subseteq q'$ (as $q' \to q$). In addition, there is no CQ $q'' \in \mathsf{GHW}(1)$ such that $q \subseteq q'' \subset q'$. We provide an explanation for this later. ◀

We start in Section 3.1 by stating some basic properties on existence and uniqueness of $\mathsf{GHW}(k)$-overapproximations. Later in Section 3.2 we establish a connection between $\mathsf{GHW}(k)$-overapproximations and the existential pebble game, which allows us to show that

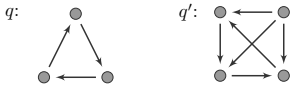


Figure 2 The CQ q is in $\mathsf{GHW}(2)$ but has no $\mathsf{GHW}(1)$-overapproximations, while q' is in $\mathsf{GHW}(3)$ but has no $\mathsf{GHW}(\ell)$-overapproximations for $\ell \in \{1, 2\}$.

both the identification and evaluation problems for $\mathsf{GHW}(k)$-overapproximations are tractable. Finally, in Section 3.4 we look at the case when $\mathsf{GHW}(k)$-overapproximations do not exist, and suggest how this can be alleviated by allowing infinite overapproximations.

3.1 Existence and uniqueness of overapproximations

As shown in [4], existence of overapproximations is not a general phenomenon. In fact, for every $k > 1$ there is a Boolean CQ q in $\mathsf{GHW}(k)$ that has no $\mathsf{GHW}(1)$-overapproximation. Using the characterization given later in Theorem 21, we can strengthen this further:

▶ **Proposition 3.** *For each $k > 1$, there is a Boolean CQ $q \in \mathsf{GHW}(k)$ without $\mathsf{GHW}(\ell)$-overapproximations for any $1 \leq \ell < k$.*

Figure 2 depicts examples of CQs in $\mathsf{GHW}(k)$, for $k = 2$ and $k = 3$, respectively, without $\mathsf{GHW}(\ell)$-overapproximations for any $1 \leq \ell < k$.

Interestingly, when $\mathsf{GHW}(k)$-overapproximations do exist, they are unique (up to equivalence). This follows since, in this case, $\mathsf{GHW}(k)$-overapproximations are not only the minimal elements, but also the lower bounds of the set of CQs in $\mathsf{GHW}(k)$ that contain q:

▶ **Proposition 4.** *Let q, q' be CQs such that $q' \in \mathsf{GHW}(k)$. The following are equivalent:*

1. *q' is a $\mathsf{GHW}(k)$-overapproximation of q.*
2. *(i) $q \subseteq q'$, and (ii) for every CQ $q'' \in \mathsf{GHW}(k)$, it is the case that $q \subseteq q''$ implies $q' \subseteq q''$.*

Proof. We only prove the nontrivial direction (1) $\Rightarrow$ (2). By contradiction, suppose that there is a CQ $q'' \in \mathsf{GHW}(k)$ such that $q \subseteq q''$ but $q' \not\subseteq q''$. Note that we can assume that q' and q'' have the same free variables $\bar{x}$; otherwise we can rename them accordingly. Let $(q' \wedge q'')$ be the *conjunction* of q' and q'', i.e., the CQ which is obtained by first renaming each existentially quantified variable in q' and q'' with a different fresh variable, and then taking the conjunction of the atoms in q' and q''. The tuple of free variables of $(q' \wedge q'')$ is $\bar{x}$. Observe that $(q' \wedge q'')$ is in $\mathsf{GHW}(k)$. Also, by the definition of $(q' \wedge q'')$ we have that $q \subseteq (q' \wedge q'') \subseteq q'$. But q' is a $\mathsf{GHW}(k)$-overapproximation of q, and thus $q' \subseteq (q' \wedge q'')$. On the other hand, we have that $(q' \wedge q'') \subseteq q''$, and then $q' \subseteq q''$. This is a contradiction. ◀

As a corollary, we immediately obtain the following:

▶ **Corollary 5.** *If a CQ q has $\mathsf{GHW}(k)$-overapproximations q_1 and q_2, then $q_1 \equiv q_2$.*

The previous results show the stark difference between $\mathsf{GHW}(k)$-overapproximations and $\mathsf{GHW}(k)$-underapproximations: $\mathsf{GHW}(k)$-overapproximations do not necessarily exist, but when they do they are unique; $\mathsf{GHW}(k)$-underapproximations always exist but there can be exponentially many incomparable ones [5].

3.2 A link with the existential pebble game

We characterize $\mathsf{GHW}(k)$-overapproximations in terms of the existential pebble game. We use a version of such a game, known as *existential cover game*, that is tailored for CQs of bounded generalized hypertreewidth [9]. Let $k \geq 1$. The existential k-cover game is played by *Spoiler* and *Duplicator* on pairs $(\mathcal{D}, \bar{a})$ and $(\mathcal{D}', \bar{b})$, where $\mathcal{D}$ and $\mathcal{D}'$ are databases and $\bar{a}$ and $\bar{b}$ are n-ary ($n \geq 0$) tuples over $\mathcal{D}$ and $\mathcal{D}'$, respectively. The game proceeds in rounds. In each round, Spoiler places (resp., removes) a pebble on (resp., from) an element of $\mathcal{D}$, and Duplicator responds by placing (resp., removing) its corresponding pebble on an element of (resp., from) $\mathcal{D}'$. The number of pebbles is not bounded, but Spoiler is constrained as follows: At any round p of the game, if $c_1, \ldots, c_\ell$ ($\ell \leq p$) are the elements marked by Spoiler's pebbles in $\mathcal{D}$, there must be a set of at most k atoms in $\mathcal{D}$ that contain all such elements (this is why the game is called k-cover, as pebbled elements are *covered* by no more than k atoms).

Duplicator wins if she has a *winning strategy*, i.e., she can indefinitely continue playing the game in such a way that after each round, if $c_1, \ldots, c_\ell$ are the elements that are marked by Spoiler's pebbles in $\mathcal{D}$ and $d_1, \ldots, d_\ell$ are the elements marked by the corresponding pebbles of Duplicator in $\mathcal{D}'$, then $\big((c_1, \ldots, c_\ell, \bar{a}), (d_1, \ldots, d_\ell, \bar{b})\big)$ is a *partial homomorphism* from $\mathcal{D}$ to $\mathcal{D}'$. That is, for every atom $R(\bar{c}) \in \mathcal{D}$, where each element c of $\bar{c}$ appears in $(c_1, \ldots, c_\ell, \bar{a})$, it is the case that $R(\bar{d}) \in \mathcal{D}'$, where $\bar{d}$ is the tuple obtained from $\bar{c}$ by replacing each element c of $\bar{c}$ by its corresponding element d in $(d_1, \ldots, d_\ell, \bar{b})$. We write $(\mathcal{D}, \bar{a}) \to_k (\mathcal{D}', \bar{b})$ if Duplicator has a winning strategy.

Notice that $\to_k$ "approximates" $\to$ as follows: $\to \subset \cdots \subset \to_{k+1} \subset \to_k \subset \cdots \subset \to_1$. These approximations are convenient complexity-wise: Checking whether $(\mathcal{D}, \bar{a}) \to (\mathcal{D}', \bar{b})$ is NP-complete, but $(\mathcal{D}, \bar{a}) \to_k (\mathcal{D}', \bar{b})$ can be solved efficiently.

▶ **Proposition 6.** *[9] Fix $k \geq 1$. Checking whether $(\mathcal{D}, \bar{a}) \to_k (\mathcal{D}', \bar{b})$ is in polynomial time.*

Moreover, there is a connection between $\to_k$ and the evaluation of CQs in $\mathsf{GHW}(k)$ that we heavily exploit in our work:

▶ **Proposition 7.** *[9] Fix $k \geq 1$. Then $(\mathcal{D}, \bar{a}) \to_k (\mathcal{D}', \bar{b})$ iff for each CQ $q(\bar{x})$ in $\mathsf{GHW}(k)$ we have that if $(q, \bar{x}) \to (\mathcal{D}, \bar{a})$ then $(q, \bar{x}) \to (\mathcal{D}', \bar{b})$.*

In particular, if $q(\bar{x}) \in \mathsf{GHW}(k)$ then for every $\mathcal{D}$ and $\bar{a}$:

$$\bar{a} \in q(\mathcal{D}) \quad \iff \quad (q, \bar{x}) \to (\mathcal{D}, \bar{a}) \quad \iff \quad (q, \bar{x}) \to_k (\mathcal{D}, \bar{a}). \tag{2}$$

That is, the "approximation" of $\to$ provided by $\to_k$ is sufficient for evaluating CQs in $\mathsf{GHW}(k)$. Together with Proposition 6, this proves that CQs in $\mathsf{GHW}(k)$ can be evaluated efficiently.

The characterization. Existential cover games can be applied to obtain a semantic characterization of $\mathsf{GHW}(k)$-overapproximations:

▶ **Theorem 8.** *Fix $k \geq 1$. Let q, q' be CQs with $q' \in \mathsf{GHW}(k)$. Then $q'(\bar{x})$ is the $\mathsf{GHW}(k)$-overapproximation of $q(\bar{x})$ iff $(q', \bar{x}) \to_k (q, \bar{x})$ and $(q, \bar{x}) \to_k (q', \bar{x})$.*

Proof. Assume that $q'(\bar{x})$ is the $\mathsf{GHW}(k)$-overapproximation of $q(\bar{x})$. Then $(q', \bar{x}) \to (q, \bar{x})$, and thus $(q', \bar{x}) \to_k (q, \bar{x})$ since $\to \subset \to_k$. We prove now that $(q, \bar{x}) \to_k (q', \bar{x})$. From Proposition 7, we need to prove that if $q''(\bar{x})$ is a CQ in $\mathsf{GHW}(k)$ such that $(q'', \bar{x}) \to (q, \bar{x})$, then also $(q'', \bar{x}) \to (q', \bar{x})$. This follows directly from Proposition 4.

Assume now that $(q', \bar{x}) \to_k (q, \bar{x})$ and $(q, \bar{x}) \to_k (q', \bar{x})$. Since $q' \in \mathsf{GHW}(k)$, we have that $q \subseteq q'$ by Equation (2). From Proposition 7, if $q \subseteq q''$ and $q'' \in \mathsf{GHW}(k)$ then $q' \subseteq q''$, i.e., there is no q'' in $\mathsf{GHW}(k)$ such that $q \subseteq q'' \subset q'$. Hence q' is a $\mathsf{GHW}(k)$-overapproximation. ◀

▶ **Example 9.** (Example 2 cont.) It is now easy to see that the CQ q' in Figure 1 is a $\mathsf{GHW}(1)$-overapproximation of q. In fact, since $q' \to q$, we only need to show that $q \to_1 q'$. The latter is simple and left to the reader. ◀

Next we show that this characterization allows us to show that the identification and evaluation problems for $\mathsf{GHW}(k)$-overapproximations can be solved in polynomial time.

3.3 Identification and evaluation of $\mathsf{GHW}(k)$-overapproximations

A direct corollary of Proposition 6 and Theorem 8 is that the *identification* problem for $\mathsf{GHW}(k)$-overapproximations is in polynomial time:

▶ **Corollary 10.** *Fix $k \geq 1$. Given CQs q, q' such that $q' \in \mathsf{GHW}(k)$, checking if q' is the $\mathsf{GHW}(k)$-overapproximation of q can be solved in polynomial time.*

This corresponds to a *promise version* of the problem, as it is given to us that q' is in fact in $\mathsf{GHW}(k)$. Checking the latter is NP-complete for every fixed $k \geq 2$ [18, 14].

Assume now that we are given the *promise* that q has a $\mathsf{GHW}(k)$-overapproximation q' (but q' itself is not given). How hard is it to evaluate q' over a database $\mathcal{D}$? We could try to compute q', but so far we have no techniques to do that. Notably, we can use existential cover games to show that $\mathsf{GHW}(k)$-overapproximations can be evaluated efficiently, without even computing them. This is based on the next result, which states that evaluating q' over $\mathcal{D}$ boils down to checking $(q, \bar{x}) \to_k (\mathcal{D}, \bar{a})$ for the tuples $\bar{a}$ over $\mathcal{D}$.

▶ **Theorem 11.** *Fix $k \geq 1$. Let $q(\bar{x})$ be a CQ with a $\mathsf{GHW}(k)$-overapproximation $q'(\bar{x})$. Then for every $\mathcal{D}$ and $\bar{a}$:*

$$\bar{a} \in q'(\mathcal{D}) \iff (q', \bar{x}) \to (\mathcal{D}, \bar{a}) \iff (q, \bar{x}) \to_k (\mathcal{D}, \bar{a}).$$

Proof. Assume first that $(q, \bar{x}) \to_k (\mathcal{D}, \bar{a})$. Since q' is a $\mathsf{GHW}(k)$-overapproximation of q, we have that $(q', \bar{x}) \to (q, \bar{x})$. Since winning strategies for Duplicator compose and $\to \subset \to_k$ then, $(q', \bar{x}) \to_k (\mathcal{D}, \bar{a})$. But $q' \in \mathsf{GHW}(k)$, and thus $(q', \bar{x}) \to (\mathcal{D}, \bar{a})$ from Equation (2). Assume now that $(q', \bar{x}) \to (\mathcal{D}, \bar{a})$. From Theorem 8, we have that $(q, \bar{x}) \to_k (q', \bar{x})$. By composition and $\to \subset \to_k$, it follows that $(q, \bar{x}) \to_k (\mathcal{D}, \bar{a})$ holds. ◀

As a corollary to Theorem 11 and Proposition 6 we obtain:

▶ **Corollary 12.** *Fix $k \geq 1$. Checking if $\bar{a} \in q'(\mathcal{D})$, given a CQ q that has a $\mathsf{GHW}(k)$-overapproximation q', a database $\mathcal{D}$, and a tuple $\bar{a}$ in $\mathcal{D}$, can be solved in polynomial time by checking if $(q, \bar{x}) \to_k (\mathcal{D}, \bar{a})$. Moreover, this can be done without even computing q'.*

3.4 More liberal $\mathsf{GHW}(k)$-overapproximations

CQs may not have $\mathsf{GHW}(k)$-overapproximations, for some $k \geq 1$. We observe in this section that this anomaly can be solved by extending the language of queries over which overapproximations are to be found.

An *infinite* CQ is as a finite one, save that now the number of atoms is countably infinite. We assume that there are finitely many free variables in an infinite CQ. The evaluation of an infinite CQ $q(\bar{x})$ over a database $\mathcal{D}$ is defined analogously to the evaluation of a finite one. Similarly, the generalized hypertreewidth of an infinite CQ is defined as in the finite case, but now tree decompositions can be infinite. We write $\mathsf{GHW}(k)^\infty$ for the class of all CQs, finite and infinite ones, of generalized hypertreewidth at most k. The next result states a crucial relationship between the existential k-cover game and the class $\mathsf{GHW}(k)^\infty$:

▶ **Lemma 13.** *Fix $k \geq 1$. For every CQ q there is a q' in $\mathsf{GHW}(k)^\infty$ such that for every database $\mathcal{D}$ and tuple $\bar{a}$ of constants in $\mathcal{D}$:*

$$\bar{a} \in q'(\mathcal{D}) \iff (q', \bar{x}) \to (\mathcal{D}, \bar{a}) \iff (q, \bar{x}) \to_k (\mathcal{D}, \bar{a}).$$

This holds even for countably infinite databases $\mathcal{D}$.

The proof of this result follows from techniques in [22]. The basic idea is that q' has an (infinite) generalized hypertree decomposition of width k that represents all possible moves of Spoiler in the existential k-cover game played from q.

Since we now deal with infinite CQs and databases, we cannot apply Proposition 7 directly in our analysis of $\mathsf{GHW}(k)^\infty$-overapproximations. Instead, we use the following suitable reformulation of it, which we obtain by inspection of its proof:

▶ **Proposition 14.** *Fix $k \geq 1$. Consider countably infinite databases $\mathcal{D}$ and $\mathcal{D}'$. Then $(\mathcal{D}, \bar{a}) \to_k (\mathcal{D}', \bar{b})$ iff for each CQ $q(\bar{x})$ in $\mathsf{GHW}(k)^\infty$, if $(q, \bar{x}) \to (\mathcal{D}, \bar{a})$ then $(q, \bar{x}) \to (\mathcal{D}', \bar{b})$.*

$\mathsf{GHW}(k)^\infty$-overapproximations. We expand the notion of overapproximation by allowing infinite CQs. Let $q' \in \mathsf{GHW}(k)^\infty$. Then q' is a $\mathsf{GHW}(k)^\infty$-overapproximation of CQ q, if $q \subseteq q'$ and there is no $q'' \in \mathsf{GHW}(k)^\infty$ such that $q \subseteq q'' \subset q'$. (Here, $\subseteq$ is still defined with respect to finite databases only). In $\mathsf{GHW}(k)^\infty$, we can provide each CQ q an overapproximation:

▶ **Theorem 15.** *Fix $k \geq 1$. For every CQ q there is a CQ in $\mathsf{GHW}(k)^\infty$ that is a $\mathsf{GHW}(k)^\infty$-overapproximation of q.*

Proof. We prove that q', as given in Lemma 13, is a $\mathsf{GHW}(k)^\infty$-overapproximation of q. Notice that $(q', \bar{x}) \to (q, \bar{x})$ (by choosing $(\mathcal{D}, \bar{a})$ as $(q, \bar{x})$ in Lemma 13). Therefore, $q \subseteq q'$ since this direction of Equation (1) continues to hold for countably infinite CQs. Observe now that $(q, \bar{x}) \to_k (q', \bar{x})$ (by choosing $(\mathcal{D}, \bar{a})$ as $(q', \bar{x})$ in Lemma 13). Proposition 14 tells us that for every $q''(\bar{x})$ in $\mathsf{GHW}(k)^\infty$, if $(q'', \bar{x}) \to (q, \bar{x})$ then $(q'', \bar{x}) \to (q', \bar{x})$. But then $q \subseteq q''$ implies that $q' \subseteq q''$, since Equation (1) continues to hold if q (but not necessarily q') is finite. Thus, q' is a $\mathsf{GHW}(k)^\infty$-overapproximation of q. ◀

Despite the non-computable nature of $\mathsf{GHW}(k)^\infty$-overapproximations, we get from Proposition 6 and the proof of Theorem 15 that they can be evaluated efficiently:

▶ **Corollary 16.** *Fix $k \geq 1$. Checking whether $\bar{a} \in q'(\mathcal{D})$, given a CQ q with $\mathsf{GHW}(k)^\infty$-overapproximation q', a database $\mathcal{D}$, and a tuple $\bar{a}$ in $\mathcal{D}$, boils down to checking if $(q, \bar{x}) \to_k (\mathcal{D}, \bar{a})$, and thus it can be solved in polynomial time.*

4 Deciding existence of $\mathsf{GHW}(k)$-overapproximations

CQs always have $\mathsf{GHW}(k)^\infty$-overapproximations, but not necessarily finite ones. Here we study when a CQ q has a finite overapproximation. We start with the case $k = 1$, which we show to be decidable in 2ExpTime (we do not know if this is optimal). For $k > 1$ we leave the decidability open, but provide some explanation about where the difficulty lies.

4.1 The acyclic case

We start with the case of $\mathsf{GHW}(1)$-overapproximations. Recall that $\mathsf{GHW}(1)$ is an important class, as it consists precisely of the well-known acyclic CQs. Our main result is the following:

▶ **Theorem 17.** *There is a* 2Exptime *algorithm that checks if a CQ q has a* GHW(1)*-overapproximation and, if one exists, it computes one in triple-exponential time.*

If the arity of the schema is fixed, there is an Exptime *algorithm that does this and computes a* GHW(1)*-overapproximation of q in double-exponential time.*

We sketch the proof for nonfixed arities. From a CQ q we build a *two-way alternating tree automaton* [10], or 2ATA, $\mathcal{A}_q$, such that the language $L(\mathcal{A}_q)$ of trees accepted by $\mathcal{A}_q$ is nonempty iff q has a GHW(1)-overapproximation. Intuitively, $\mathcal{A}_q$ accepts those trees that encode a GHW(1)-overapproximation q' of q. Formally:

▶ **Proposition 18.** *There exists a double-exponential time algorithm that takes as input a CQ q and returns a 2ATA $\mathcal{A}_q$ with exponentially many states, such that q has a* GHW(1)*-overapproximation iff $L(\mathcal{A}_q) \neq \emptyset$. Furthermore, from every tree T in $L(\mathcal{A}_q)$ one can construct in polynomial time a* GHW(1)*-overapproximation of q.*

Proof sketch. For simplicity we assume that q is Boolean. Before describing the construction of $\mathcal{A}_q$, we explain how input trees for $\mathcal{A}_q$ encode CQs in GHW(1). To this end let q' be a CQ in GHW(1) and $(T_{q'}, \chi)$ a tree decomposition of q'. The CQ q' can have unbounded many variables. Yet, in each node of $T_{q'}$ at most r variables appear, where r is the maximum arity of an atom in q. Thus, by reusing variables, $(T_{q'}, \chi)$ can be encoded by using $2r$ variables in such a way that it can then be decoded, i.e. a variable name u_i is used in two neighboring nodes v and v' of the encoding iff the corresponding variables of the tree decomposition also occur in neighboring nodes. The encoding $\mathsf{Enc}(T_{q'}, \chi)$ of $(T_{q'}, \chi)$ is a tree labeled by (a) the variables $\{u_1, \ldots, u_{2r}\}$ as described, and (b) the atoms of q' covered by those variables.

The 2ATA $\mathcal{A}_q$ checks that the CQ q' encoded by $T' = \mathsf{Enc}(T_{q'}, \chi)$ is a GHW(1)-overapproximation of q. From Theorem 8, we need to check: (1) $q' \to_1 q$, and (2) $q \to_1 q'$. The 2ATA $\mathcal{A}_q$ will be defined as the intersection of 2ATAs $\mathcal{A}_1$ and $\mathcal{A}_2$, that check conditions (1) and (2), respectively. Condition (1) is equivalent to $q' \to q$ (since $q' \in \mathsf{GHW}(1)$). A 2ATA $\mathcal{A}_1$ can guess and verify a homomorphism from q' to q. In particular, $\mathcal{A}_1$ requires no alternation and has at most exponentially many states.

We now sketch how the automaton $\mathcal{A}_2$ works. First, $q \to_1 q'$ can be restated as Duplicator having a *compact winning strategy* [9] as follows. The set of variables appearing in an atom of q constitute a 1-*union* of q. Then $q \to_1 q'$ iff there is a non-empty family $\mathcal{F}$ of partial homomorphisms from q to q' such that: (a) the domain of each $f \in \mathcal{F}$ is a 1-union of q, and (b) if U and U' are 1-unions of q, then each $f \in \mathcal{F}$ with domain U can be *extended* to U', i.e., there is $f' \in \mathcal{F}$ with domain U' such that $f(x) = f'(x)$ for every $x \in U \cap U'$.

The 2ATA $\mathcal{A}_2$ assumes an annotation of $T' = \mathsf{Enc}(T_{q'}, \chi)$ that encodes the intended strategy $\mathcal{F}$. This annotation labels each node t' of T' by the set of partial mappings from q to q' whose domain is a 1-union of q, and whose range is contained in the variables from $\{u_1, \ldots, u_{2r}\}$ labeling t'. It can be easily checked from the labelings of T' if each mapping in this annotation is a partial homomorphism. To check condition (2), the 2ATA $\mathcal{A}_2$ makes a universal transition for each pair (U, U') of 1-unions and partial mapping g with domain U annotating a node t' of T'. Then it checks the existence of a node t' in T' that is annotated with a mapping g' that extends g to U'. The latter means that, for each $x \in U \cap U'$, both $g(x)$ and $g'(x)$ are the same variable of q', that is, $g(x)$ and $g(x')$ are connected occurrences of the same variable in $\{u_1, \ldots, u_{2r}\}$. Thus to check the consistency of g and g', the automaton can store the variables in $\{g(x) \mid x \in U \cap U'\}$, and check that these are present in the label of each node guessed before reaching t'. As this is a polynomial amount of information, $\mathcal{A}_2$ can be implemented using exponentially many states. ◀

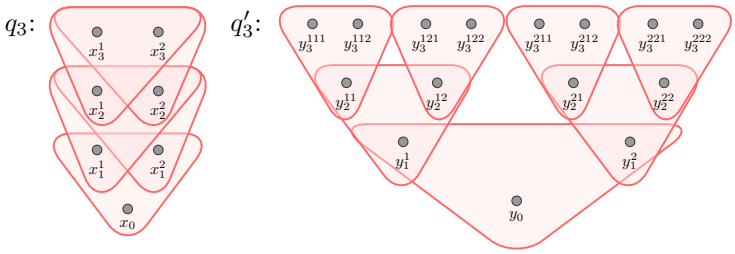


Figure 3 Illustration of the CQs q_3 and q_3' from Proposition 20. Each triple of variables represents two atoms in the query; e.g., $\{y_0, y_1^1, y_1^2\}$ represents atoms $R(y_0, y_1^1, y_1^2)$ and $R(y_0, y_1^2, y_1^1)$ in q_3'.

It is easy to see how Theorem 17 follows from Proposition 18. Checking if a CQ q has a $\mathsf{GHW}(1)$-overapproximation amounts to checking if $L(\mathcal{A}_q) \neq \emptyset$. The latter can be done in exponential time in the number of states of $\mathcal{A}_q$ [10], and thus in double-exponential time in the size of q. If $L(\mathcal{A}_q) \neq \emptyset$, one can construct a tree $T \in L(\mathcal{A}_q)$ in double-exponential time in the size of $\mathcal{A}_q$, and thus in triple-exponential time in the size of q. From T one then gets in polynomial time (i.e., in 3EXPTIME in the size of q) a $\mathsf{GHW}(1)$-overapproximation of q.

The case of binary schemas. For schemas of arity two the existence and computation of $\mathsf{GHW}(1)$-overapproximations can be solved in polynomial time. This is of practical importance since data models such as *graph databases* [3] and description logic *ABoxes* [2] can be represented using schemas of this kind. Note that in this context $\mathsf{GHW}(1)$ coincides with the class of CQs of treewidth one [11]. Then:

▶ **Theorem 19.** *There is a* PTIME *algorithm that checks if a CQ q over a schema of maximum arity two has a $\mathsf{GHW}(1)$-overapproximation q', and computes such a q' if it exists.*

The proof of this result can be found in the appendix.

Size of overapproximations. Over binary schemas $\mathsf{GHW}(1)$-overapproximations are of polynomial size. This is optimal as over schemas of arity three there is an exponential lower bound for the size of $\mathsf{GHW}(1)$-overapproximations:

▶ **Proposition 20.** *There is a schema σ with a single ternary relation symbol and a family $(q_n)_{n\geq 1}$ of Boolean CQs over σ, such that (1) q_n is of size $O(n)$, and (2) the size of every $\mathsf{GHW}(1)$-overapproximation of q_n is $\Omega(2^n)$.*

Proof. The CQ q_n contains the atoms $R(x_0, x_1^1, x_1^2)$, $R(x_0, x_1^2, x_1^1)$, as well as $R(x_i^j, x_{i+1}^1, x_{i+1}^2)$ and $R(x_i^j, x_{i+1}^2, x_{i+1}^1)$, for each $1 \leq i \leq n-1$ and $j \in \{1, 2\}$. Consider now the CQ q_n' with the atoms $R(y_0, y_1^1, y_1^2)$, $R(y_0, y_1^2, y_1^1)$, as well as $R(y_{|w|}^w, y_{|w|+1}^{w1}, y_{|w|+1}^{w2})$ and $R(y_{|w|}^w, y_{|w|+1}^{w2}, y_{|w|+1}^{w1})$, for each word w over $\{1, 2\}$ of length $1 \leq |w| \leq n-1$. Figure 3 depicts q_3 and q_3'.

The query q_n' indeed is an overapproximation of q_n. The mapping $h : q_n' \to q_n$ defined as $h(y_0) = x_0$ and $h(y_{|w|+1}^{wj}) = x_{|w|+1}^j$, for each word w over $\{1, 2\}$ of length $0 \leq |w| \leq n-1$ and $j \in \{1, 2\}$, is a homomorphism. On the other hand, a compact winning strategy for Duplicator can be obtained by basically "inverting" the homomorphism h.

The size of q_n' is $\Omega(2^n)$. We claim that q_n' is the smallest $\mathsf{GHW}(1)$-overapproximation of q_n, which proves the proposition. A straightforward case-by-case analysis shows that q_n' is a *core* [8, 19]. Now assume, towards a contradiction, that q' is a $\mathsf{GHW}(1)$-overapproximation of q_n with fewer atoms than q_n'. Then $q_n' \equiv q'$ by Corollary 5. Composing the homomorphisms $h_1 : q_n' \to q'$ and $h_2 : q' \to q_n'$ yields a homomorphism from q_n' to a proper subset of the atoms of q_n'. This is a contradiction to q_n' being a core. ◀

4.2 Beyond acyclicity

Theorem 8 characterizes when a CQ has a $\mathsf{GHW}(k)$-overapproximation. We provide an alternative characterization in terms of a *boundedness* condition for the existential cover game. This helps understanding where lies the difficulty of determining the decidability status of the problem of existence of $\mathsf{GHW}(k)$-overapproximations, for $k > 1$.

We write $(\mathcal{D}, \bar{a}) \to_k^c (\mathcal{D}, \bar{b})$, for $k \geq 1$ and $c \geq 0$, if Duplicator has a winning strategy *in the first c rounds* of the existential k-cover game on $(\mathcal{D}, \bar{a})$ and $(\mathcal{D}, \bar{b})$. The next result establishes that a CQ q has a $\mathsf{GHW}(k)$-overapproximation iff the existential k-cover game played from q is "bounded", i.e., if there is a constant $c \geq 0$ that bounds the number of rounds this game needs to be played in order to determine if Duplicator wins.

▶ **Theorem 21.** *Fix $k \geq 1$. The CQ $q(\bar{x})$ has a $\mathsf{GHW}(k)$-overapproximation iff there is an integer $c \geq 0$ such that $(q, \bar{x}) \to_k (\mathcal{D}, \bar{a})$ iff $(q, \bar{x}) \to_k^c (\mathcal{D}, \bar{a})$, for each database $\mathcal{D}$ and $\bar{a} \in \mathcal{D}$.*

Boundedness conditions are a difficult area of study, with a delicate decidability boundary. For *least fixed point logic* (LFP), undecidability results for boundedness abound with the exception of a few restricted fragments [25, 7]. The existence of winning Duplicator strategies in existential pebble games is expressible in LFP [23], yet results from this context are not directly applicable to determine the decidability status of the condition from Theorem 21.

5 Beyond under and overapproximations: Δ-approximations

We now turn to $\mathsf{GHW}(k)$-Δ-approximations. Recall that a $\mathsf{GHW}(k)$-Δ-approximation of q is a maximal element in $\mathsf{GHW}(k)$ with respect to the order $\sqsubseteq_q$, where $q' \sqsubseteq_q q''$, for CQs $q', q'' \in \mathsf{GHW}(k)$, iff $\Delta(q(\mathcal{D}), q''(\mathcal{D})) \subseteq \Delta(q(\mathcal{D}), q'(\mathcal{D}))$ for all databases $\mathcal{D}$. It is not surprising that $\mathsf{GHW}(k)$-Δ-approximations generalize over- and underapproximations.

▶ **Proposition 22.** *Fix $k \geq 1$. Let q, q' be CQs such that $q' \in \mathsf{GHW}(k)$. If $q \subseteq q'$ (resp., $q' \subseteq q$), then q' is a $\mathsf{GHW}(k)$-Δ-approximation of q if and only if q' is a $\mathsf{GHW}(k)$-overapproximation (resp., $\mathsf{GHW}(k)$-underapproximation) of q.*

Thus, we concentrate on the study of $\mathsf{GHW}(k)$-Δ-approximations that are neither $\mathsf{GHW}(k)$-under- nor $\mathsf{GHW}(k)$-overapproximations. Evaluating such Δ-approximations can give us useful information when the quality of $\mathsf{GHW}(k)$-under- and $\mathsf{GHW}(k)$-overapproximations is poor. But, are there $\mathsf{GHW}(k)$-Δ-approximations that are neither $\mathsf{GHW}(k)$-under- nor $\mathsf{GHW}(k)$-overapproximations? In the rest of this section, we settle this question and study complexity questions associated with such $\mathsf{GHW}(k)$-Δ-approximations.

5.1 Incomparable $\mathsf{GHW}(k)$-Δ-approximations

Let q be a CQ. In view of Proposition 22, the $\mathsf{GHW}(k)$-Δ-approximations q' of q that are neither $\mathsf{GHW}(k)$-overapproximations nor $\mathsf{GHW}(k)$-underapproximations must be *incomparable* with q in terms of containment; i.e., both $q \not\subseteq q'$ and $q' \not\subseteq q$ must hold. Incomparable $\mathsf{GHW}(k)$-Δ-approximations do not necessarily exist, even when approximating in the set of infinite CQs $\mathsf{GHW}(k)^\infty$. A trivial example is any CQ q in $\mathsf{GHW}(k)$, as its only $\mathsf{GHW}(k)$-Δ-approximation (up to equivalence) is q itself. The following characterization will help us to find CQs with incomparable $\mathsf{GHW}(k)$-Δ-approximations.

▶ **Theorem 23.** *Fix $k \geq 1$. Let $q(\bar{x}), q'(\bar{x})$ be CQs such that $q' \in \mathsf{GHW}(k)$. Then q' is an incomparable $\mathsf{GHW}(k)$-Δ-approximation of q iff $(q, \bar{x}) \to_k (q', \bar{x})$, and both $q \not\subseteq q'$ and $q' \not\subseteq q$.*

Proof. Suppose that q' is an incomparable $\mathsf{GHW}(k)$-Δ-approximation of q and assume, by contradiction, that $(q, \bar{x}) \not\to_k (q', \bar{x})$. By Proposition 7, there is a $q'' \in \mathsf{GHW}(k)$ such that $q \subseteq q''$ and $q' \not\subseteq q''$. We show that $q' \sqsubset_q (q'' \wedge q')$, which is a contradiction as $(q'' \wedge q') \in \mathsf{GHW}(k)$. Assume that $\bar{a} \in \Delta(q(\mathcal{D}), (q'' \wedge q')(\mathcal{D}))$, for some $\mathcal{D}$ and $\bar{a} \in \mathcal{D}$. If $\bar{a} \notin q(\mathcal{D})$, then $\bar{a} \in (q'' \wedge q')(\mathcal{D}) \subseteq q'(\mathcal{D})$, and thus, $\bar{a} \in \Delta(q(\mathcal{D}), q'(\mathcal{D}))$. Otherwise, $\bar{a} \in q(\mathcal{D})$ and $\bar{a} \notin (q'' \wedge q')(\mathcal{D})$. Since $q \subseteq q''$, we have $\bar{a} \notin q'(\mathcal{D})$, and then $\bar{a} \in \Delta(q(\mathcal{D}), q'(\mathcal{D}))$. Hence $q' \sqsubseteq_q (q'' \wedge q')$. Now, since $q' \not\subseteq q''$, there is a database $\mathcal{D}^*$ such that $q'(\mathcal{D}^*) \not\subseteq q''(\mathcal{D}^*)$, i.e., $\bar{a} \in q'(\mathcal{D}^*)$ but $\bar{a} \notin q''(\mathcal{D}^*)$, for some tuple $\bar{a}$ in $\mathcal{D}^*$. In particular $\bar{a} \in \Delta(q(\mathcal{D}^*), q'(\mathcal{D}^*))$ and $\bar{a} \notin \Delta(q(\mathcal{D}^*), (q'' \wedge q')(\mathcal{D}^*))$, and thus $(q'' \wedge q') \not\sqsubseteq_q q'$. For the converse, we need the following:

▶ **Lemma.** Fix $k \geq 1$. Let $q(\bar{x}), q'(\bar{x}), q''(\bar{x})$ be CQs such that $q'' \in \mathsf{GHW}(k)$. Suppose that $(q, \bar{x}) \to_k (q', \bar{x})$. Then $(q'', \bar{x}) \to (q' \wedge q, \bar{x})$ implies $(q'', \bar{x}) \to (q', \bar{x})$.

Assume that $q \not\subseteq q'$, $q' \not\subseteq q$, and $(q, \bar{x}) \to_k (q', \bar{x})$. By contradiction, suppose that there is a CQ $q'' \in \mathsf{GHW}(k)$ such that $q' \sqsubset_q q''$. We show that $q' \equiv q''$, which is a contradiction. Recall that $\mathcal{D}_{(q' \wedge q)}$ denotes the canonical database of $(q' \wedge q)$. Clearly, $\bar{x} \in q(\mathcal{D}_{(q' \wedge q)})$ and $\bar{x} \in q'(\mathcal{D}_{(q' \wedge q)})$. It follows that $\bar{x} \notin \Delta(q(\mathcal{D}_{(q' \wedge q)}), q'(\mathcal{D}_{(q' \wedge q)}))$, and by hypothesis, $\bar{x} \notin \Delta(q(\mathcal{D}_{(q' \wedge q)}), q''(\mathcal{D}_{(q' \wedge q)}))$. Hence, $\bar{x} \in q''(\mathcal{D}_{(q' \wedge q)})$. By the lemma above, we have $(q'', \bar{x}) \to (q', \bar{x})$, that is, $q' \subseteq q''$. For $q'' \subseteq q'$, note that $\bar{x} \notin q(\mathcal{D}_{q''})$; otherwise, $q'' \subseteq q$ would hold, implying that $q' \subseteq q$, which is a contradiction. Since $\bar{x} \in q''(\mathcal{D}_{q''})$, we have $\bar{x} \in \Delta(q(\mathcal{D}_{q''}), q''(\mathcal{D}_{q''}))$. This implies that $\bar{x} \in \Delta(q(\mathcal{D}_{q''}), q'(\mathcal{D}_{q''}))$, and then $\bar{x} \in q'(\mathcal{D}_{q''})$, i.e., $q'' \subseteq q'$. Hence, $q' \equiv q''$. ◀

▶ **Example 24.** Consider again the CQ $q = \exists x \exists y \exists z (E(x, y) \wedge E(y, z) \wedge E(z, x))$ from Figure 2. Then q has a unique $\mathsf{GHW}(1)$-underapproximation $q' = \exists x E(x, x)$. As mentioned in Section 3.1, q has no $\mathsf{GHW}(1)$-overapproximations. Does q have incomparable $\mathsf{GHW}(1)$-Δ-approximations? By applying Theorem 23, we can give a positive answer to this question: the CQ $q'' = \exists x \exists y (E(x, y) \wedge E(y, x))$ is an incomparable $\mathsf{GHW}(1)$-Δ-approximation of q. ◀

Therefore, as Example 24 shows, incomparable $\mathsf{GHW}(k)$-Δ-approximations may exist for some CQs. However, in contrast with overapproximations, they are not unique in general:

▶ **Proposition 25.** *There is a CQ with infinitely many (non-equivalent) incomparable* $\mathsf{GHW}(1)$-Δ-*approximations. In fact, this holds for the CQ q in Figure 1.*

Identification, existence and evaluation. A direct consequence of Theorem 23 is that the identification problem, i.e., checking if $q' \in \mathsf{GHW}(k)$ is an incomparable $\mathsf{GHW}(k)$-Δ-approximation of a CQ q, is in coNP. It suffices to check that $q \not\subseteq q'$ and $q' \not\subseteq q$ – which are in coNP – and $(q, \bar{x}) \to_k (q', \bar{x})$ – which is in PTIME from Proposition 6. This is optimal:

▶ **Proposition 26.** *Fix $k \geq 1$. Checking if a given CQ $q' \in \mathsf{GHW}(k)$ is an incomparable* $\mathsf{GHW}(k)$-Δ-*approximation of a given CQ q, is* coNP*-complete.*

As in the case of $\mathsf{GHW}(k)$-overapproximations, we do not know how to check existence of incomparable $\mathsf{GHW}(k)$-Δ-approximations, for $k > 1$. Nevertheless, for $k = 1$ we can exploit the automata techniques developed in Section 4 and obtain an analogous decidability result:

▶ **Proposition 27.** *There is a* 2EXPTIME *algorithm that checks if a CQ q has a incomparable* $\mathsf{GHW}(1)$-Δ-*approximation and, if one exists, it computes one in triple exponential time. The bounds become* EXPTIME *and* 2EXPTIME*, respectively, if the arity of the schema is fixed.*

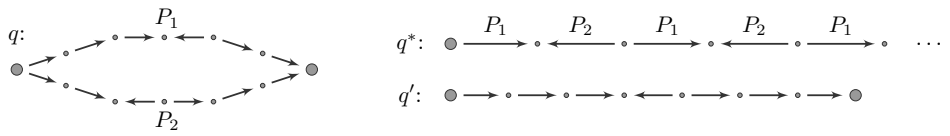


Figure 4 The CQ $q \in \mathsf{GHW}(2)$ from Example 29. The CQ $(q^* \wedge q')$ is an incomparable $\mathsf{GHW}(1)^\infty$-Δ-approximation of q. On the other hand, q has no incomparable $\mathsf{GHW}(1)$-Δ-approximations.

Now we study evaluation. Recall that, unlike $\mathsf{GHW}(k)$-overapproximations, incomparable $\mathsf{GHW}(k)$-Δ-approximations of a CQ q are not unique. In fact, there can be infinitely many (see Proposition 25). Thus, it is reasonable to start by trying to evaluate *at least one* of them. It would be desirable, in addition, if the one we evaluate depends only on q (i.e., it is independent of the underlying database $\mathcal{D}$). Proposition 27 allows us to do so as follows. Given a CQ q with at least one incomparable $\mathsf{GHW}(1)$-Δ-approximation, we can compute in 3Exptime one such an incomparable $\mathsf{GHW}(1)$-Δ-approximation q'. We can then evaluate q' over a database $\mathcal{D}$ in time $O(|\mathcal{D}| \cdot |q'|)$ [27], which is $O(|\mathcal{D}| \cdot f(|q|))$, for f a triple-exponential function. This means that the evaluation of such a q' over $\mathcal{D}$ is *fixed-parameter tractable*, i.e., it can be solved by an algorithm that depends polynomially on the size of the large database $\mathcal{D}$, but more loosely on the size of the small CQ q. (This is a desirable property for evaluation, which does not hold in general for the class of all CQs [26]). Formally, then:

▶ **Theorem 28.** *There is a fixed-parameter tractable algorithm that, given a CQ q that has incomparable $\mathsf{GHW}(1)$-Δ-approximations, a database $\mathcal{D}$, and a tuple $\bar{a}$, checks whether $\bar{a} \in q'(\mathcal{D})$, for some incomparable $\mathsf{GHW}(1)$-Δ-approximation q' of q that depends only on q.*

It is worth noticing that the automata techniques are essential for proving this result, and thus for evaluating incomparable $\mathsf{GHW}(1)$-Δ-approximations. This is in stark contrast with $\mathsf{GHW}(k)$-overapproximations, which can be evaluated in polynomial time by simply checking if $(q, \bar{x}) \to_k (\mathcal{D}, \bar{a})$. It is not at all clear whether such techniques can be extended to allow for the efficient evaluation of incomparable $\mathsf{GHW}(k)$-Δ-approximations.

The infinite case. All the previous results continue to apply for the class of infinite CQs in $\mathsf{GHW}(k)^\infty$. The following example shows that, as in the case of $\mathsf{GHW}(k)$-overapproximations, considering $\mathsf{GHW}(k)^\infty$ helps us to obtain better incomparable $\mathsf{GHW}(k)$-Δ-approximations.

▶ **Example 29.** Consider the CQ q that asks for the existence of the two oriented paths P_1 and P_2, as shown in Figure 4. Theorem 23 can be used to show that q has no incomparable $\mathsf{GHW}(1)$-Δ-approximation. However, q has an incomparable $\mathsf{GHW}(1)^\infty$-Δ-approximation. In fact, let q^* be the $\mathsf{GHW}(1)^\infty$-overapproximation of q which is depicted in Figure 4 (a P_1-labeled edge represents a copy of the oriented path P_1, similarly for P_2). Also, let q' be an arbitrary CQ in $\mathsf{GHW}(1)$ which is incomparable with q (one such a q' is shown in Figure 4). Applying the extension of Theorem 23 to the class $\mathsf{GHW}(k)^\infty$, we can prove that $(q^* \wedge q')$ is an incomparable $\mathsf{GHW}(1)^\infty$-Δ-approximation of q. ◀

Example 29 also illustrates the following fact: If there is a CQ $q' \in \mathsf{GHW}(k)$ which is incomparable with q, then $(q^* \wedge q')$ is an incomparable $\mathsf{GHW}(k)^\infty$-Δ-approximation of q, where q^* is the $\mathsf{GHW}(k)^\infty$-overapproximation of q. Given a database $\mathcal{D}$ and a tuple $\bar{a}$ in $\mathcal{D}$, we can check whether $\bar{a}$ belongs to the evaluation of such a Δ-approximation $(q^* \wedge q')$ over $\mathcal{D}$ as follows: First we compute q', and then we check both $(q, \bar{x}) \to_k (\mathcal{D}, \bar{a})$ and $\bar{a} \in q'(\mathcal{D})$. In other words, we evaluate $(q^* \wedge q')$ via the existential k-cover game, as for the

$\mathsf{GHW}(k)^\infty$-overapproximation, and then use the incomparable CQ q' to filter out some tuples in the answer. Interestingly, we can easily exploit automata techniques and compute such an incomparable q' (in case one exists). Thus we have the following:

▶ **Theorem 30.** *Fix $k \geq 1$. There is a fixed-parameter tractable algorithm that given a CQ q that has an incomparable q' in $\mathsf{GHW}(k)$, a database $\mathcal{D}$, and a tuple $\bar{a}$ in $\mathcal{D}$, decides whether $\bar{a} \in \hat{q}(\mathcal{D})$, for some incomparable $\mathsf{GHW}(k)^\infty$-Δ-approximation $\hat{q}$ of q that depends only on q.*

6 Final Remarks

Several problems remain open: is the existence of $\mathsf{GHW}(k)$-overapproximations decidable for $k > 1$? What is the precise complexity of checking for the existence of $\mathsf{GHW}(1)$-overapproximations? In particular, can we improve the 2Exptime upper bound from Theorem 17? What is an optimal upper bound on the size of $\mathsf{GHW}(1)$-overapproximations?

In the future we plan to study how our notions of approximation can be combined with other techniques to obtain quantitative guarantees. One possibility is to exploit semantic information about the data – e.g., in the form of integrity constraints – in order to ensure that certain bounds on the size of the result of the approximation hold. Another possibility is to try to obtain probabilistic guarantees for approximations based on reasonable assumptions about the distribution of the data.

References

1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2 Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.
3 Pablo Barceló. Querying graph databases. In *PODS*, pages 175–188, 2013.
4 Pablo Barceló, Leonid Libkin, and Miguel Romero. Efficient approximations of conjunctive queries. In *PODS*, pages 249–260, 2012.
5 Pablo Barceló, Leonid Libkin, and Miguel Romero. Efficient approximations of conjunctive queries. *SIAM J. Comput.*, 43(3):1085–1130, 2014.
6 Pablo Barceló, Miguel Romero, and Moshe Y. Vardi. Semantic acyclicity on graph databases. *SIAM J. Comput.*, 45(4):1339–1376, 2016.
7 Achim Blumensath, Martin Otto, and Mark Weyer. Decidability results for the boundedness problem. *Logical Methods in Computer Science*, 10(3), 2014.
8 Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.
9 Hubie Chen and Víctor Dalmau. Beyond hypertree width: Decomposition methods without decompositions. In *CP*, pages 167–181, 2005.
10 Stavros S. Cosmadakis, Haim Gaifman, Paris C. Kanellakis, and Moshe Y. Vardi. Decidable optimization problems for database logic programs (preliminary report). In *STOC*, pages 477–490, 1988.
11 Víctor Dalmau, Phokion G. Kolaitis, and Moshe Y. Vardi. Constraint satisfaction, bounded treewidth, and finite-variable logics. In *CP*, pages 310–326, 2002.
12 Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. Graph pattern matching: From intractable to polynomial time. *PVLDB*, 3(1):264–275, 2010.
13 Robert Fink and Dan Olteanu. On the optimal approximation of queries using tractable propositional languages. In *ICDT*, pages 174–185, 2011.

14 Wolfgang Fischl, Georg Gottlob, and Reinhard Pichler. General and fractional hypertree decompositions: Hard and easy cases. *CoRR*, abs/1611.01090, 2016. `arXiv:1611.01090`.

15 Minos Garofalakis and Phillip Gibbon. Approximate query processing: taming the terabytes. In *VLDB*, page 725, 2001.

16 Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. Hypertree decompositions: Questions and answers. In *PODS*, pages 57–74, 2016.

17 Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002.

18 Georg Gottlob, Zoltán Miklós, and Thomas Schwentick. Generalized hypertree decompositions: NP-hardness and tractable variants. *J. ACM*, 56(6), 2009.

19 Pavol Hell and Jaroslav Nesetril. The core of a graph. *Discrete Mathematics*, 109(1-3):117–126, 1992.

20 Yannis Ioannidis. Approximations in database systems. In *ICDT*, pages 16–30, 2003.

21 Phokion G. Kolaitis and Jonathan Panttaja. On the complexity of existential pebble games. In *CSL*, pages 314–329, 2003.

22 Phokion G. Kolaitis and Moshe Y. Vardi. On the expressive power of datalog: Tools and a case study. *J. Comput. Syst. Sci.*, 51(1):110–134, 1995.

23 Phokion G. Kolaitis and Moshe Y. Vardi. Conjunctive-query containment and constraint satisfaction. *J. Comput. Syst. Sci.*, 61(2):302–332, 2000.

24 Qing Liu. Approximate query processing. In *Encyclopedia of Database Systems*, pages 113–119, 2009.

25 Martin Otto. The boundedness problem for monadic universal first-order logic. In *LICS*, pages 37–48, 2006.

26 Christos H. Papadimitriou and Mihalis Yannakakis. On the complexity of database queries. *J. Comput. Syst. Sci.*, 58(3):407–427, 1999.

27 Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94, 1981.

7 Appendix

▶ **Theorem 19** (restated). *There is a* PTIME *algorithm that checks if a CQ q over a schema of maximum arity two has a* GHW(1)*-overapproximation q', and computes such q' if it exists.*

The idea is to show that if a CQ q has a GHW(1)-overapproximation then it can be extracted from q in a simple way: it is a subquery of q or it is a subquery of a CQ $q_u \# q_v$ constructed from q and two distinguished variables u, v in q. The algorithm then greedily searches through the subqueries of q and $q_u \# q_v$ to find an overapproximation of q.

We need to introduce some notation. The *Gaifman graph* of a CQ q, denoted by $\mathcal{G}(q)$, is the undirected graph whose set of nodes is the set of variables of q and where there is an edge $\{z, z'\}$ whenever z and z' are distinct variables that appear together in an atom of q. The *existential* Gaifman graph of q, denoted by $\mathcal{G}_\exists(q)$, is the subgraph of $\mathcal{G}(q)$ induced by the existentially quantified variables of q.

▶ **Claim 31.** Let $q(\bar{x})$ be a CQ. Then $q \in \mathsf{GHW}(1)$ iff $\mathcal{G}_\exists(q)$ is an acyclic graph.

A CQ q is *connected* if $\mathcal{G}(q)$ is connected. Using Theorem 8, we have the following:

▶ **Lemma 32.** *Let q be a connected CQ. If q has a* GHW(1)*-overapproximation, then it has one that is connected.*

We start by proving Theorem 19 for Boolean CQs; thus, until stated otherwise, we assume all CQs to be Boolean. First we show Theorem 19 for connected CQs, then we extend it to the non-connected case and finally prove the general non-Boolean statement.

7.1 The connected case

We start with the following technical lemma. Let q be a CQ. We say that u and v are *adjacent* if $\{u, v\}$ is an edge in $\mathcal{G}(q)$, that is, if u and v appear together in an atom of q.

▶ **Lemma 33.** *Let q be a connected CQ in* GHW(1)*. If q is a core then for all variables u and v in q there is at most one endomorphism of q that maps u to v.*

Proof. Assume that there are two distinct endomorphisms h_1 and h_2 with $h_1(u) = h_2(u) = v$. Recall that, since q is a core, h_1 and h_2 are isomorphisms. We root $\mathcal{G}(q)$ at u. Let x be a variable in q with $h_1(x) \neq h_2(x)$ whose distance to u is minimal in $\mathcal{G}(q)$. Then there is a unique y such that $h_1(x) = h_2(y)$. We claim that x and y have the same parent in $\mathcal{G}(q)$. Let z be the parent of x. Since h_1 is an isomorphism, $h_1(x)$ and $h_1(z)$, and therefore also $h_2(y)$ and $h_2(z)$, are adjacent in $\mathcal{G}(q)$. Thus, also y and z are adjacent. However, y cannot be the parent of z since h_1 and h_2 agree on all variables above z in $\mathcal{G}(q)$. Therefore z is the parent of y.

Construct a new endomorphism h that maps variables w in the subtree $\mathcal{G}(q)$ rooted at y to $h_2(w)$, and all other variables w' to $h_1(w')$. Then h is an endomorphism, but not injective as both x and y are mapped to $h_1(x) = h_2(y)$. This is a contradiction to q being a core, and therefore there are no two distinct endomorphisms h_1 and h_2 mapping u to v. ◀

Suppose $q \in \mathsf{GHW}(1)$ is connected and u, v are adjacent. If we remove all the atoms that mention u, v, we obtain two connected CQs, one containing u and the other containing v. We denote these CQs by t_u^q and t_v^q, respectively.

Suppose $q \in \mathsf{GHW}(1)$ is a connected core. If an endomorphism h of q maps u to v where u and v are adjacent, then it is the case that $h(u) = v$ and $h(v) = u$, and h swaps the

subqueries t_u^q and t_v^q. We call such an h a *swapping endomorphism* for u and v. Note that Lemma 33 tells us that if such a swapping homomorphism for u and v exists, then it is unique.

▶ **Lemma 34.** *Let q be a connected core in* GHW(1)*. Then q has at most one endomorphism besides the identity mapping. If this endomorphism exists, it is a swapping endomorphism.*

Proof. Let $P = x_0, x_1, ...x_m$ be a simple path of maximal length in $\mathcal{G}(q)$. For each endomorphism h of q, the path $P' = y_1 \dots y_m$ where $y_i = h(x_i)$ is a simple path of the same length (as q is a core and therefore h is an isomorphism). Furthermore, P and P' share a vertex. Indeed, if this not the case, since q is connected, one can pick w in P and w' in P' such that w and w' are connected by a path P'' vertex-disjoint (except for w and w') from P and P', and construct a longer path than P.

Now, if $|P|$ is even, then its middle vertex $u = x_{m/2}$ is in the intersection of P and P' (as otherwise q would contain a path longer than P). But then $h(u) = u$ and then h must be the identity mapping by Lemma 33.

If $|P|$ is odd, then $u = x_{\lfloor m/2 \rfloor}$ and $v = x_{\lceil m/2 \rceil}$ are in the intersection of P and P' (again, as otherwise q would contain a path longer than P). If $h(u) = u$, again we have that h is the identity. Otherwise, if $h(u) = v$, since u and v are adjacent, we have that h must be the swapping endomorphism for u and v. ◀

For a CQ q and variables u, v in q the CQ $q_u \# q_v$ is defined as follows. Denote by $q \setminus v$ the CQ obtained from q by removing all atoms that contain v. Let q_u be the query constructed from $q \setminus v$ by replacing each variable z by a fresh variable z_u. Similarly let q_v be the CQ where each variable z in $q \setminus u$ is replaced by a fresh variable z_v. The CQ $q_u \# q_v$ is the union of q_u and q_v plus all atoms $R(u_u, v_v)$ when $R(u, v)$ is an atom in q. Likewise for atoms $R(v, u)$. By construction, we have the following:

▶ **Claim 35.** For each CQ q and variables u, v in q, it is the case that $q_u \# q_v \to q$.

Before immersing into the proof of Theorem 19 for connected CQs, we need some notation and properties of GHW(1)-overapproximations. Suppose $\hat{q}, \hat{q}'$ are CQs and X, Y are set of variables from $\hat{q}$ and $\hat{q}'$, respectively. We denote by $(\hat{q}, X) \to_1 (\hat{q}', Y)$ the fact that Duplicator has a winning strategy in the existential 1-cover game on $\hat{q}$ and $\hat{q}'$ with the property that whenever Spoiler places a pebble on an element of X in $\hat{q}$, then Duplicator responds with some element of Y in $\hat{q}'$. Checking whether $(\hat{q}, X) \to_1 (\hat{q}', Y)$ can still be done in polynomial time.

▶ **Lemma 36.** *Suppose q is a CQ and suppose q' is a connected core that is a* GHW(1)*-overapproximation of q. Then we have the following:*

- *If the only endomorphism of q' is the identity, then any homomorphism from q' to q is injective. In particular, q' is a subquery of q.*
- *If q' has a swapping endomorphism for u' and v', then for any homomorphism h from q' to q, we have that*
  - *$(q, \{h(u'), h(v')\}) \to_1 (q', \{u', v'\})$, and*
  - *h is "almost" injective, more precisely, $h(z') \neq h(z'')$ for all pairs of variables z', z'', except maybe for $z' \neq u'$ in $t_{u'}^{q'}$ and $z' \neq v'$ in $t_{v'}^{q'}$. In particular, q' is a subquery of $q_{h(u')} \# q_{h(v')}$.*

Proof. Suppose the only endomorphism of q' is the identity and towards a contradiction, suppose there is a non-injective homomorphism h from q' to q. Then we have $h(z') = h(z'')$,

for distinct variables z', z'' in q'. Using the fact that $q \to_1 q'$, it is easy to see that there is a Duplicator winning strategy on q' and q' such that z'' is a possible response of Duplicator when Spoiler starts playing on z'. Since $q' \in \mathsf{GHW}(1)$, we can define an endomorphism g of q' that maps z' to z''. Then g is an endomorphism different from the identity, which is a contradiction.

Suppose now that q' has a swapping endomorphism for u' and v', and let h be a homomorphism from q' to q. First, assume by contradiction that Duplicator's strategy witnessing $q \to_1 q'$ is such that for $h(u')$ (the case for $h(v')$ is analogous), Duplicator responds with $z' \notin \{u', v'\}$. By composing h with this strategy, and using the fact that $q' \in \mathsf{GHW}(1)$, it follows that there is an endomorphism g of q' that maps u' to z'. This endomorphism is different from the identity and from the swapping endomorphism for u' and v', which contradicts Lemma 34. Finally, suppose towards a contradiction that $h(z') = h(z'')$, where $z' \neq z''$ and $z' = u'$ and z'' is in $t^{q'}_{v'}$ (the other case is analogous). Again by composing h with the strategy witnessing $q \to_1 q'$ and the fact that $q' \in \mathsf{GHW}(1)$, it is easy to derive an endomorphism of q' that is neither the identity nor the swapping endomorphism for u' and v'. ◀

As a corollary of Lemma 36 and Lemma 34, we have that whenever q' is a connected core, and it is a $\mathsf{GHW}(1)$-overapproximation of q, then q' is a subquery of q or a subquery of $q_u \# q_v$, for some variables u, v in q.

Proof of Theorem 19 for connected, Boolean CQs. We assume that the given CQ q is connected. The algorithm first checks whether a subquery of q is a $\mathsf{GHW}(1)$-overapproximation. This is Step 1. In Step 2, the algorithm checks whether a subquery of $q_u \# q_v$ is a $\mathsf{GHW}(1)$-overapproximation, for some u and v in q. If neither step succeed then the algorithm rejects. Step 1 is as follows:

1. Set q_0 to be q.
2. While $q_i \notin \mathsf{GHW}(1)$, search for an atom e such that $q_i \to_1 q_i \setminus e$. If there is no such atom then continue with Step 2. Otherwise, set q_{i+1} to be $q_i \setminus e$.
3. If $q_i \in \mathsf{GHW}(1)$, for some i, then accept and output q_i.

For Step 2, let $\mathcal{P}$ be an enumeration of the pairs (u, v) such that u, v are adjacent in q and $q \to_1 q_u \# q_v$. Step 2 is as follows:

1. Let (u, v) be the first pair in $\mathcal{P}$.
2. Set q_0 to be $q_u \# q_v$.
3. While $q_i \notin \mathsf{GHW}(1)$, search for an atom e that does not mention u_u and v_v simultaneously such that $(q_i, \{u_u, v_v\}) \to_1 (q_i \setminus e, \{u_u, v_v\})$. If there is no such atom, let (u, v) be the next pair in $\mathcal{P}$ and repeat from item 2. Otherwise, set q_{i+1} to be $q_i \setminus e$.
4. If $q_i \in \mathsf{GHW}(1)$, for some i, then accept and output q_i.

Notice that the described algorithm can be implemented in polynomial time. Below we argue that it is correct.

Suppose first that the algorithm, on input q, accepts with output q^*. By construction $q^* \in \mathsf{GHW}(1)$. Assume first that the algorithm accepts in the m-th iteration of Step 1, and thus $q^* = q_m$. By construction, for each $0 \leq i < m$, we have that $q_i \to_1 q_{i+1}$ and $q_{i+1} \to_1 q_i$. In particular, $q \to_1 q^*$ and $q^* \to_1 q$, and thus q^* is a $\mathsf{GHW}(1)$-overapproximation of q. Suppose now that the algorithm accepts in Step 2 for a pair $(u, v) \in \mathcal{P}$, in the m-th iteration. Again we have that $q_i \to_1 q_{i+1}$ and $q_{i+1} \to_1 q_i$, for each $0 \leq i < m$, and thus $q_u \# q_v \to_1 q^*$ and $q^* \to_1 q_u \# q_v$. Since $(u, v) \in \mathcal{P}$, it follows that $q \to_1 q_u \# q_v$, and then $q \to_1 q^*$. Using the fact that $q_u \# q_v \to q$, we have that $q^* \to_1 q$. Hence, q^* is a $\mathsf{GHW}(1)$-overapproximation of q.

It remains to show that if q has a $\mathsf{GHW}(1)$-overapproximation q' then the algorithm accepts. Since q is connected, we can assume that q' also is. Moreover, we can assume w.l.o.g. that q' is a core. By Lemma 34, we have two cases: (1) the only endomorphism of q' is the identity, or (2) q' has two endomorphisms, namely, the identity and the swapping endomorphism for some variables u' and v'.

First suppose case (1) applies. We show that the algorithm accepts in Step 1. By definition, $q_i \to_1 q_{i+1}$ and $q_{i+1} \to_1 q_i$ (actually $q_{i+1} \to q_i$), for each $0 \leq i \leq m$, where m is the number of iteration in Step 1. It follows that $q_0 = q \to_1 q_m$ and $q_m \to_1 q$. Since the relation $\to_1$ composes, q' is a $\mathsf{GHW}(1)$-overapproximation of q_m and by using Lemma 36, q' is a subquery of q_m. Now for the sake of contradiction assume that the algorithm does not accept in Step 1. Then $q_m \notin \mathsf{GHW}(1)$ and there is no edge e in q_m such that $q_m \to_1 q_m \setminus e$. Since q' is $\mathsf{GHW}(1)$-overapproximation of q_m, we have that $q_m \to_1 q'$ and, since $q' \in \mathsf{GHW}(1)$, q' is a proper subquery of q_m. It follows that there is an edge e in q_m such that $q_m \to_1 q_m \setminus e$, which is a contradiction.

Suppose case (2) holds. In this case the algorithm accepts in Step 2. Let h be a homomorphism from q' to q, and let $u = h(u')$ and $v = h(v')$. By Lemma 36, $u \neq v$ and then u and v are adjacent. Also, by Lemma 36, q' is a subquery of $q_u \# q_v$. Since $q \to_1 q'$, it follows that $q \to_1 q_u \# q_v$, and then $(u, v) \in \mathcal{P}$. We claim that the algorithm accepts when (u, v) is chosen from $\mathcal{P}$. First, note that q' is a $\mathsf{GHW}(1)$-overapproximation of q_m. Indeed, by definition, $q_m \to q_u \# q_v$, $q_u \# q_v \to q$ (Claim 35), and $q \to_1 q'$. It follows that $q_m \to_1 q'$. On the other hand, we have that $(q', (u', v')) \to (q_u \# q_v, (u_u, v_v))$ (q' is a subquery of $q_u \# q_v$) and $(q_u \# q_v, \{u_u, v_v\}) \to_1 (q_m, \{u_u, v_v\})$. It follows that $(q', \{u', v'\}) \to_1 (q_m, \{u_u, v_v\})$, which implies that $(q', (u', v')) \to (q_m, (u_u, v_v))$ via a homomorphism g. Then q' is a $\mathsf{GHW}(1)$-overapproximation of q_m. By applying Lemma 36 to q_m, q' and g, we obtain that $(q_m, \{u_u, v_v\}) \to_1 (q', \{u', v'\})$, and g is "almost" injective. Observe that $g(z') \neq g(z'')$ for all $z' \neq u'$ in $t_{u'}^{q'}$ and $z'' \neq v'$ in $t_{v'}^{q'}$, since $\{u_u, v_v\}$ is a bridge of $\mathcal{G}(q_m)$, that is, its removal disconnect $\mathcal{G}(q_m)$. We conclude that g is injective and then q' is a subquery of q_m.

Towards a contradiction, assume that the algorithm do not accept when (u, v) is chosen from $\mathcal{P}$. Then $q_m \notin \mathsf{GHW}(1)$ and there is no edge e that does not mention both u_u, v_v such that $(q_m, \{u_u, v_v\}) \to_1 (q_m \setminus e, \{u_u, v_v\})$. Since $(q_m, \{u_u, v_v\}) \to_1 (q', \{u', v'\})$, $(q', (u', v')) \to (q_m, (u_u, v_v))$ via the injective homomorphism g and $q' \in \mathsf{GHW}(1)$, it follows that there is an edge e that does not mention both u_u, v_v such that $(q_m, \{u_u, v_v\}) \to_1 (q_m \setminus e, \{u_u, v_v\})$. This is a contradiction. ◀

7.2 The unconnected case

Now we consider the non-connected case. A *connected component* of a CQ is a maximal connected subquery. Given a CQ q with connected components $q_1, \ldots, q_m$, the algorithm proceeds as follows:

1. Start by simplifying q: Compute a minimal subset of CQs $\mathcal{Q}$ in $\{q_1, \ldots, q_m\}$ such that for each $1 \leq i \leq m$, there is a $p \in \mathcal{Q}$ with $q_i \to_1 p$.
2. Check whether each $p \in \mathcal{Q}$ has a $\mathsf{GHW}(1)$-overapproximation p' using the algorithm described for connected CQs. If this is the case then accept and output $\bigwedge_{p \in \mathcal{Q}} p'$.

Clearly, the algorithm can be implemented in polynomial time. For the correctness, suppose first that the algorithm accepts and outputs $q' = \bigwedge_{p \in \mathcal{Q}} p'$. Then $q' \to \bigwedge_{p \in \mathcal{Q}} p \to q$. We also have that $q \to_1 \bigwedge_{p \in \mathcal{Q}} p$ (by definition of $\mathcal{Q}$), and $\bigwedge_{p \in \mathcal{Q}} p \to_1 q'$. This implies that q' is a $\mathsf{GHW}(1)$-overapproximation of q.

Suppose now that q has a $\mathsf{GHW}(1)$-overapproximation q'. Since $q \to_1 \bigwedge_{p\in\mathcal{Q}} p$ and $\bigwedge_{p\in\mathcal{Q}} p \to_1 q$, it follows that q' is also a $\mathsf{GHW}(1)$-overapproximation of $\bigwedge_{p\in\mathcal{Q}} p$. By the minimality of $\mathcal{Q}$, we have that $p \not\to_1 \hat{p}$, for each pair of distinct CQs $p, \hat{p} \in \mathcal{Q}$. Let p be a CQ in $\mathcal{Q}$. Since $p \to_1 q'$ and p is connected, it follows that $p \to_1 p^*$, where p^* is a connected component of q'. Also, since $q' \to \bigwedge_{p\in\mathcal{Q}} p$, there is $p_0 \in \mathcal{Q}$ such that $p^* \to p_0$. In particular, $p \to_1 p_0$. It follows that $p_0 = p$, and then p^* is a $\mathsf{GHW}(1)$-overapproximation of p. We conclude that each $p \in \mathcal{Q}$ has a $\mathsf{GHW}(1)$-overpproximation, and thus the algorithm accepts.

7.3 The non-Boolean case

Finally, we consider the general case that includes non-Boolean queries. Let $q(\bar{x})$ be a CQ. We denote by q^B the Boolean CQ obtained from $q(\bar{x})$ by existentially quantifying the free variables $\bar{x}$. Recall that $\mathcal{G}(q)$ is the Gaifman graph of q, while $\mathcal{G}_\exists(q)$ denotes the restriction of $\mathcal{G}(q)$ to the existentially quantified variables of q. Recall also that $q(\bar{x})$ is connected if $\mathcal{G}(q)$ is connected, and a connected component of q is a maximal connected subquery. If $q(\bar{x})$ is connected, $q'(\bar{x})$ is a *part* of q if it is a maximal subquery of q with $\mathcal{G}_\exists(q')$ connected.

Let $q(\bar{x})$ be a CQ. Let $q_1 \ldots, q_m$ be the connected components of q. Let $\mathcal{C}^{\mathtt{free}}$ be the CQs in $\{q_1 \ldots, q_m\}$ that contain a free variable from $\bar{x}$, and let $\mathcal{C}^{\exists}$ be the rest of the CQs. The algorithm proceeds as follows:

1. Simplify $q(\bar{x})$: Compute a minimal subset of CQs $\mathcal{Q}$ in $\{q_1, \ldots, q_m\}$ such that $\mathcal{C}^{\mathtt{free}} \subseteq \mathcal{Q}$ and for each $1 \leq i \leq m$, there is a $p \in \mathcal{Q}$ with $q_i^B \to_1 p^B$.
2. Check whether each $p \in \mathcal{Q}$ has a $\mathsf{GHW}(1)$-overapproximation p'. If this is the case then accept and output $\bigwedge_{p\in\mathcal{Q}} p'$. To check if $p \in \mathcal{Q}$ has a $\mathsf{GHW}(1)$-overapproximation, for a $p \in \mathcal{C}^{\exists}$, we simply apply the algorithm for the Boolean and connected case described previously. In case $p(\bar{z}) \in \mathcal{C}^{\mathtt{free}} \cap \mathcal{Q}$, where $\bar{z}$ are the free variables from $\bar{x}$ present in p, the algorithm does the following:
   a. Simplify $p(\bar{z})$: Compute a minimal subset $\mathcal{S}$ of the parts of $p(\bar{z})$ such that for each part $p'(\bar{z})$ of $p(\bar{z})$ there is a part $p''(\bar{z}) \in \mathcal{S}$ such that $(p', \bar{z}) \to_1 (p'', \bar{z})$.
   b. Check whether each part $p'(\bar{z}) \in \mathcal{S}$ has a $\mathsf{GHW}(1)$-overapproximation $p'_*(\bar{z})$. If this is the case then accept and output $\bigwedge_{p'\in\mathcal{S}} p'_*(\bar{z})$.

It remains to explain how the algorithm checks the existence of $\mathsf{GHW}(1)$-overapproximations for a part $p'(\bar{z})$ of a connected CQ $p(\bar{z})$. This is done by applying an adaptation of the algorithm described for the connected and Boolean case. For $p'(\bar{z})$ and two existentially quantified variables u, v adjacent in $\mathcal{G}_\exists(p')$, we define $p'_u \# p'_v(\bar{z})$ as the CQ obtained from $p'(\bar{z})$ as follows: the free variables are $\bar{z}$ and the atoms in $p'_u \# p'_v(\bar{z})$ mentioning only variables in $\bar{z}$ are exactly those in $p'(\bar{z})$. The CQ induced by the existentially quantified variables of $p'_u \# p'_v(\bar{z})$ is the Boolean CQ $p''_u \# p''_v$, where p'' is the subquery of p' induced by the existential variables. Finally, if there is an atom in p' mentioning a free variable and an existential variable w, then the same atom appears in $p'_u \# p'_v(\bar{z})$ but replacing w by its "copies", that is, by w_u or w_v, if $w = u$ or $w = v$ respectively, or by w_u and w_v, if $w \notin \{u, v\}$.

Observe that Lemma 3–5, Claim 35 and Lemma 36 hold for the non-Boolean case, when we consider the adapted definition for $q_u \# q_v$ (exactly the same arguments apply). Using this, we have that the algorithm developed for Boolean and connected CQs still works for non-Boolean CQs $p'(\bar{z})$ that are parts of connected CQs.

This finishes the proof of Theorem 19.

Answering UCQs under Updates and in the Presence of Integrity Constraints

Christoph Berkholz
Humboldt-Universität zu Berlin, Germany
berkholz@informatik.hu-berlin.de

Jens Keppeler
Humboldt-Universität zu Berlin, Germany
keppelej@informatik.hu-berlin.de

Nicole Schweikardt
Humboldt-Universität zu Berlin, Germany
schweikn@informatik.hu-berlin.de

Abstract

We investigate the query evaluation problem for fixed queries over fully dynamic databases where tuples can be inserted or deleted. The task is to design a dynamic data structure that can immediately report the new result of a fixed query after every database update. We consider unions of conjunctive queries (UCQs) and focus on the query evaluation tasks *testing* (decide whether an input tuple $\overline{a}$ belongs to the query result), *enumeration* (enumerate, without repetition, all tuples in the query result), and *counting* (output the number of tuples in the query result).

We identify three increasingly restrictive classes of UCQs which we call *t-hierarchical*, *q-hierarchical*, and *exhaustively q-hierarchical* UCQs. Our main results provide the following dichotomies: If the query's homomorphic core is t-hierarchical (q-hierarchical, exhaustively q-hierarchical), then the testing (enumeration, counting) problem can be solved with constant update time and constant testing time (delay, counting time). Otherwise, it cannot be solved with sublinear update time and sublinear testing time (delay, counting time), unless the OV-conjecture and/or the OMv-conjecture fails.

We also study the complexity of query evaluation in the dynamic setting in the presence of integrity constraints, and we obtain similar dichotomy results for the special case of small domain constraints (i.e., constraints which state that all values in a particular column of a relation belong to a fixed domain of constant size).

2012 ACM Subject Classification Theory of computation → Database query languages (principles), Theory of computation → Logic and databases

Keywords and phrases dynamic query evaluation, union of conjunctive queries, constant-delay enumeration, counting problem, testing

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.8

Funding Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SCHW 837/5-1

Acknowledgements We are very grateful to the anonymous reviewers — their valuable feedback helped to significantly improve the paper. We also acknowledge the financial support by the German Research Foundation DFG under grant SCHW 837/5-1.

1 Introduction

Dynamic query evaluation refers to a setting where a fixed query q has to be evaluated against a database that is constantly updated [20]. In this paper, we study dynamic query evaluation for unions of conjunctive queries (UCQs) on relational databases that may be updated by inserting or deleting tuples. A dynamic algorithm for evaluating a query q receives an initial database and performs a preprocessing phase which builds a data structure that contains a suitable representation of the database and the result of q on this database. After every database update, the data structure is updated so that it suitably represents the new database D and the result $q(D)$ of q on this database.

To solve the *counting problem*, such an algorithm is required to quickly report the number $|q(D)|$ of tuples in the current query result, and the *counting time* is the time used to compute this number. To solve the *testing problem*, the algorithm has to be able to check for an arbitrary input tuple if it belongs to the current query result, and the *testing time* is the time used to perform this check. To solve the *enumeration problem*, the algorithm has to enumerate $q(D)$ without repetition and with a bounded *delay* between the output tuples. The *update time* is the time used for updating the data structure after having received a database update. We regard the counting (testing, enumeration) problem of a query q to be *tractable under updates* if it can be solved by a dynamic algorithm with linear preprocessing time, constant update time, and constant counting time (testing time, delay).

This setting has been studied for conjunctive queries (CQs) in our previous paper [5], which identified a class of CQs called *q-hierarchical* that precisely characterises the tractability frontier of the counting problem and the enumeration problem for CQs under updates: For every q-hierarchical CQ, the counting problem and the enumeration problem can be solved with linear preprocessing time, constant update time, constant counting time, and constant delay. And for every CQ that is not equivalent to a q-hierarchical CQ, the counting problem (and for the case of self-join-free queries, the enumeration problem) cannot be solved with sublinear update time and sublinear counting time (delay), unless the OMv-conjecture or the OV-conjecture (the OMv-conjecture) fails. The latter are well-known algorithmic conjectures on the hardness of the Boolean online matrix-vector multiplication problem (OMv) and the Boolean orthogonal vectors problem (OV) [19, 1], and "sublinear" means $O(n^{1-\epsilon})$, where $\epsilon > 0$ and n is the size of the active domain of the current database.

Our contribution. We identify a new subclass of CQs which we call *t-hierarchical*, which contains and properly extends the class of q-hierarchical CQs, and which precisely characterises the tractability frontier of the testing problem for CQs under updates (see Theorem 3.4): For every t-hierarchical CQ, the testing problem can be solved by a dynamic algorithm with linear preprocessing time, constant update time, and constant testing time. And for every CQ that is not equivalent to a t-hierarchical CQ, the testing problem cannot be solved with arbitrary preprocessing time, sublinear update time, and sublinear testing time, unless the OMv-conjecture fails.

Furthermore, we transfer the notions of t-hierarchical and q-hierarchical queries to unions of conjunctive queries (UCQs) and identify a further class of UCQs which we call *exhaustively q-hierarchical*, yielding three increasingly restricted subclasses of UCQs. In a nutshell, our main contribution concerning UCQs shows that these notions precisely characterise the tractability frontiers of the testing problem, the enumeration problem, and the counting problem for UCQs under updates (see the Theorems 4.1, 4.2, 4.5): For every t-hierarchical (q-hierarchical, exhaustively q-hierarchical) UCQ, the testing (enumeration, counting) problem

can be solved with linear preprocessing time, constant update time, and constant testing time (delay, counting time). And for every UCQ that is not equivalent to a t-hierarchical (q-hierarchical, exhaustively q-hierarchical) UCQ, the testing (enumeration, counting) problem cannot be solved with sublinear update time and sublinear testing time (delay, counting time). To be precise, the lower bound for enumeration is obtained only for self-join-free queries, the lower bounds for testing and enumeration are conditioned on the OMv-conjecture, and the lower bound for counting is conditioned on the OMv-conjecture and the OV-conjecture.

Finally, we transfer our results to a scenario where databases are required to satisfy a set of *small domain constraints* (i.e., constraints stating that all values which occur in a particular column of a relation belong to a fixed domain of constant size), leading to a precise characterisation of the UCQs for which the testing (enumeration, counting) problem under updates is tractable in this scenario (see Theorem 5.3).

Further related work. The complexity of evaluating CQs and UCQs in the *static* setting (i.e., without database updates) is well-studied. In particular, there are characterisations of "tractable" queries known for Boolean queries [17, 16, 24] as well as for the task of counting the result tuples [12, 8, 13, 15, 9]. In [3], the fragment of self-join-free CQs that can be enumerated with constant delay after linear preprocessing time has been identified, but almost nothing is known about the complexity of the enumeration problem for UCQs on static databases. Very recent papers also studied the complexity of CQs with respect to a given set of integrity constraints [14, 21, 4]. The *dynamic* query evaluation problem has been considered from different angles, including *descriptive dynamic complexity* [27, 28, 29] and, somewhat closer to what we are aiming for, *incremental view maintenance* [18, 10, 22, 23, 26]. In [20], the enumeration and testing problem under updates has been studied for q-hierarchical and (more general) acyclic CQs in a setting that is very similar to our setting and the setting of [5]; the *Dynamic Constant-delay Linear Representations* (DCLR) of [20] are data structures that use at most linear update time and solve the enumeration problem and the testing problem with constant delay and constant testing time.

Outline. The rest of the paper is structured as follows. Section 2 provides basic notations concerning databases, queries, and dynamic algorithms for query evaluation. Section 3 is devoted to CQs and proves our dichotomy result concerning the testing problem for CQs. Section 4 focuses on UCQs and proves our dichotomies concerning the testing, enumeration, and counting problem for UCQs. Section 5 is devoted to integrity constraints. Due to space restrictions, some proof details had to be deferred to the paper's full version [6].

2 Preliminaries

Basic notation. We write $\mathbb{N}$ for the set of non-negative integers and let $\mathbb{N}_{\geqslant 1} := \mathbb{N} \setminus \{0\}$ and $[n] := \{1, \ldots, n\}$ for all $n \in \mathbb{N}_{\geqslant 1}$. By 2^S we denote the power set of a set S. We write $\vec{v}_i$ to denote the i-th component of an n-dimensional vector $\vec{v}$, and we write $M_{i,j}$ for the entry in row i and column j of a matrix M. By $()$ we denote the empty tuple, i.e., the unique tuple of arity 0. For an r-tuple $t = (t_1, \ldots, t_r)$ and indices $i_1, \ldots, i_m \in \{1, \ldots, r\}$ we write $\pi_{i_1,\ldots,i_m}(t)$ to denote the projection of t to the components $i_1, \ldots, i_m$, i.e., the m-tuple $(t_{i_1}, \ldots, t_{i_m})$, and in case that $m = 1$ we identify the 1-tuple (t_{i_1}) with the element t_{i_1}. For a set T of r-tuples we let $\pi_{i_1,\ldots,i_m}(T) := \{\pi_{i_1,\ldots,i_m}(t) \ : \ t \in T\}$.

Databases. We fix a countably infinite set **dom**, the *domain* of potential database entries. Elements in **dom** are called *constants*. A *schema* is a finite set σ of relation symbols, where each $R \in \sigma$ is equipped with a fixed *arity* $\mathrm{ar}(R) \in \mathbb{N}$ (note that here we explicitly allow relation symbols of arity 0). Let us fix a schema $\sigma = \{R_1, \ldots, R_s\}$, and let $r_i := \mathrm{ar}(R_i)$ for $i \in [s]$. A *database* D of schema σ (σ-db, for short), is of the form $D = (R_1^D, \ldots, R_s^D)$, where R_i^D is a finite subset of $\mathbf{dom}^{r_i}$. The *active domain* $\mathrm{adom}(D)$ of D is the smallest subset A of **dom** such that $R_i^D \subseteq A^{r_i}$ for all $i \in [s]$.

Queries. We fix a countably infinite set **var** of *variables*. We allow queries to use variables and constants. An *atom* ψ of schema σ is of the form $Rv_1 \cdots v_r$ with $R \in \sigma$, $r = \mathrm{ar}(R)$, and $v_1, \ldots, v_r \in \mathbf{var} \cup \mathbf{dom}$. A *conjunctive formula* of schema σ is of the form

$$\exists y_1 \cdots \exists y_\ell \left(\psi_1 \wedge \cdots \wedge \psi_d \right) \qquad (*)$$

where $\ell \geqslant 0$, $d \geqslant 1$, ψ_j is an atom of schema σ for every $j \in [d]$, and $y_1, \ldots, y_\ell$ are distinct elements in **var**. For a conjunctive formula φ of the form $(*)$ we let $\mathrm{vars}(\varphi)$ ($\mathrm{cons}(\varphi)$, respectively) be the set of all variables (and constants, respectively) occurring in φ. The set of *free* variables of φ is $\mathrm{free}(\varphi) := \mathrm{vars}(\varphi) \setminus \{y_1, \ldots, y_\ell\}$. For every variable $x \in \mathrm{vars}(\varphi)$ we let $\mathrm{atoms}_\varphi(x)$ (or $\mathrm{atoms}(x)$, if φ is clear from the context) be the set of all atoms ψ_j of φ such that $x \in \mathrm{vars}(\psi_j)$. The formula φ is called *quantifier-free* if $\ell = 0$, and it is called *self-join-free* if no relation symbol occurs more than once in φ.

For $k \geqslant 0$, a *k-ary conjunctive query* (k-ary CQ, for short) is of the form

$$\{ (u_1, \ldots, u_k) \ : \ \varphi \} \qquad (**)$$

where φ is a conjunctive formula of schema σ, $u_1, \ldots, u_k \in \mathrm{free}(\varphi) \cup \mathbf{dom}$, and $\{u_1, \ldots, u_k\} \cap \mathbf{var} = \mathrm{free}(\varphi)$. We often write $q_\varphi(\overline{u})$ for $\overline{u} = (u_1, \ldots, u_k)$ (or q_φ if $\overline{u}$ is clear from the context) to denote such a query. We let $\mathrm{vars}(q_\varphi) := \mathrm{vars}(\varphi)$, $\mathrm{free}(q_\varphi) := \mathrm{free}(\varphi)$, and $\mathrm{cons}(q_\varphi) := \mathrm{cons}(\varphi) \cup (\{u_1, \ldots, u_k\} \cap \mathbf{dom})$. For every $x \in \mathrm{vars}(q_\varphi)$ we let $\mathrm{atoms}_{q_\varphi}(x) := \mathrm{atoms}_\varphi(x)$, and if q_φ is clear from the context, we omit the subscript and simply write $\mathrm{atoms}(x)$. The CQ q_φ is called *quantifier-free* (*self-join-free*) if φ is quantifier-free (self-join-free).

The semantics are defined as usual: A *valuation* is a mapping $\beta : \mathrm{vars}(q_\varphi) \cup \mathbf{dom} \to \mathbf{dom}$ with $\beta(a) = a$ for every $a \in \mathbf{dom}$. A valuation β is a *homomorphism* from q_φ to a σ-db D if for every atom $Rv_1 \cdots v_r$ in q_φ we have $\big(\beta(v_1), \ldots, \beta(v_r)\big) \in R^D$. We sometimes write $\beta : q_\varphi \to D$ to indicate that β is a homomorphism from q_φ to D. The *query result* $q_\varphi(D)$ of a k-ary CQ $q_\varphi(u_1, \ldots, u_k)$ on the σ-db D is defined as the set $\{ \big(\beta(u_1), \ldots, \beta(u_k)\big) \ : \ \beta \text{ is a homomorphism from } q_\varphi \text{ to } D\}$. If $\overline{x} = (x_1, \ldots, x_k)$ is a list of the free variables of φ and $\overline{a} \in \mathbf{dom}^k$, we sometimes write $D \models \varphi[\overline{a}]$ to indicate that there is a homomorphism $\beta : q \to D$ with $\overline{a} = \big(\beta(x_1), \ldots, \beta(x_k)\big)$, for the query $q = q_\varphi(x_1, \ldots, x_k)$.

A *k-ary union of conjunctive queries* (UCQ) is of the form $q_1(\overline{u}_1) \cup \cdots \cup q_d(\overline{u}_d)$ where $d \geqslant 1$ and $q_i(\overline{u}_i)$ is a k-ary CQ of schema σ for every $i \in [d]$. The query result of such a UCQ q on a σ-db D is $q(D) := \bigcup_{i=1}^{d} q_i(D)$. For example, $\{(x, y) : Exy\} \cup \{(x, x) : Exy\} \cup \{(y, y) : Exy\}$ is a 2-ary UCQ q with $q(D) = E^D \cup \{ (a, a) \ : \ a \in \mathrm{adom}(D) \}$ for every $\{E\}$-db D.

For a k-ary query q we write $\mathrm{vars}(q)$ ($\mathrm{cons}(q)$) to denote the set of all variables (and constants) that occur in q. Clearly, $q(D) \subseteq (\mathrm{adom}(D) \cup \mathrm{cons}(q))^k$. A *Boolean* query is a query of arity $k = 0$. As usual, for Boolean queries q we will write $q(D) = \texttt{yes}$ instead of $q(D) \neq \emptyset$, and $q(D) = \texttt{no}$ instead of $q(D) = \emptyset$. Two k-ary queries q and q' are *equivalent* ($q \equiv q'$, for short) if $q(D) = q'(D)$ for every σ-db D.

Homomorphisms. We use standard notation concerning homomorphisms (cf., e.g. [2]). The notion of a homomorphism $\beta : q \to D$ from a CQ q to a database D has already been defined above. A homomorphism $g : D \to q$ from a database D to a CQ q is a mapping from $\text{adom}(D)$ to $\text{vars}(q) \cup \text{cons}(q)$ such that $g(a) = a$ for all $a \in \text{adom}(D) \cap \text{cons}(q)$ and whenever $(a_1, \ldots, a_r)$ is a tuple in some relation R^D of D, then $Rg(a_1) \cdots g(a_r)$ is an atom of q.

Let $q(u_1, \ldots, u_k)$ and $q'(v_1, \ldots, v_k)$ be two k-ary CQs. A *homomorphism* from q to q' is a mapping $h\colon \text{vars}(q) \cup \mathbf{dom} \to \text{vars}(q') \cup \mathbf{dom}$ with $h(a) = a$ for all $a \in \mathbf{dom}$ and $h(u_i) = v_i$ for all $i \in [k]$ such that for every atom $Rw_1 \cdots w_r$ in q there is an atom $Rh(w_1) \cdots h(w_r)$ in q'. We sometimes write $h : q \to q'$ to indicate that h is a homomorphism from q to q'. Note that by [7] there is a homomorphism from q to q' if and only if for every database D it holds that $q(D) \supseteq q'(D)$. A CQ q is a *homomorphic core* if there is no homomorphism from q into a proper subquery of q. Here, a *subquery* of a CQ $q_\varphi(\overline{u})$ where φ is of the form $(*)$ is a CQ $q_{\varphi'}(\overline{u})$ where φ' is of the form $\exists y_{i_1} \cdots \exists y_{i_m}\, (\psi_{j_1} \wedge \cdots \wedge \psi_{j_n})$ with $i_1, \ldots, i_m \in [\ell]$, $j_1, \ldots, j_n \in [d]$, and $\text{free}(\varphi') = \text{free}(\varphi)$.

We say that a UCQ is a *homomorphic core* if every CQ in the union is a homomorphic core and there is no homomorphism between two distinct CQs. It is well-known that every CQ and every UCQ is equivalent to a unique (up to renaming of variables) homomorphic core, which is therefore called *the core of* the query (cf., e.g., [2]).

Sizes and Cardinalities. The *size* $\|\sigma\|$ of a schema σ is $|\sigma| + \sum_{R \in \sigma} \text{ar}(R)$. The size $\|q\|$ of a query q of schema σ is the length of q when viewed as a word over the alphabet $\sigma \cup \mathbf{var} \cup \mathbf{dom} \cup \{\wedge, \exists, (,), \{, \}, :, \cup\} \cup \{,\}$. For a k-ary query q and a σ-db D, the *cardinality of the query result* is the number $|q(D)|$ of tuples in $q(D)$. The *cardinality* $|D|$ of a σ-db D is defined as the number of tuples stored in D, i.e., $|D| := \sum_{R \in \sigma} |R^D|$. The *size* $\|D\|$ of D is defined as $\|\sigma\| + |\text{adom}(D)| + \sum_{R \in \sigma} \text{ar}(R) \cdot |R^D|$ and corresponds to the size of a reasonable encoding of D.

The following notions concerning updates, dynamic algorithms for query evaluation, and algorithmic conjectures are taken almost verbatim from [5].

Updates. We allow to update a σ-db by inserting or deleting tuples as follows. An *insertion* command is of the form $\mathsf{insert}\ R(a_1, \ldots, a_r)$ for $R \in \sigma$, $r = \text{ar}(R)$, and $a_1, \ldots, a_r \in \mathbf{dom}$. When applied to a σ-db D, it results in the updated σ-db D' with $R^{D'} := R^D \cup \{(a_1, \ldots, a_r)\}$ and $S^{D'} := S^D$ for all $S \in \sigma \setminus \{R\}$. A *deletion* command is of the form $\mathsf{delete}\ R(a_1, \ldots, a_r)$ for $R \in \sigma$, $r = \text{ar}(R)$, and $a_1, \ldots, a_r \in \mathbf{dom}$. When applied to a σ-db D, it results in the updated σ-db D' with $R^{D'} := R^D \setminus \{(a_1, \ldots, a_r)\}$ and $S^{D'} := S^D$ for all $S \in \sigma \setminus \{R\}$. Note that both types of commands may change the database's active domain.

Dynamic algorithms for query evaluation. Following [11], we use Random Access Machines (RAMs) with $O(\log n)$ word-size and a uniform cost measure to analyse our algorithms. We will assume that the RAM's memory is initialised to 0. In particular, if an algorithm uses an array, we will assume that all array entries are initialised to 0, and this initialisation comes at no cost (in real-world computers this can be achieved by using the *lazy array initialisation technique*, cf. [25]). A further assumption is that for every fixed dimension $k \in \mathbb{N}_{\geqslant 1}$ we have available an unbounded number of k-ary arrays $\mathtt{A}$ such that for given $(n_1, \ldots, n_k) \in \mathbb{N}^k$ the entry $\mathtt{A}[n_1, \ldots, n_k]$ at position $(n_1, \ldots, n_k)$ can be accessed in constant time (while this can be accomplished easily in the RAM-model, for an implementation on real-world computers one would probably have to resort to replacing our use of arrays by using suitably designed hash functions). For our purposes it will be convenient to assume that $\mathbf{dom} = \mathbb{N}_{\geqslant 1}$.

Our algorithms will take as input a k-ary query q and a σ-db D_0. For all query evaluation problems considered in this paper, we aim at routines **preprocess** and **update** which achieve the following. Upon input of q and D_0, the **preprocess** routine builds a data structure `D` which represents D_0 (and which is designed in such a way that it supports the evaluation of q on D_0). Upon input of a command $\mathsf{update}\ R(a_1, \ldots, a_r)$ (with $\mathsf{update} \in \{\mathsf{insert}, \mathsf{delete}\}$), calling **update** modifies the data structure `D` such that it represents the updated database D. The *preprocessing time* t_p is the time used for performing **preprocess**. The *update time* t_u is the time used for performing an **update**, and in this paper we aim at algorithms where t_u is independent of the size of the current database D. By **init** we denote the particular case of the routine **preprocess** upon input of a query q and the *empty* database $D_\emptyset$, where $R^{D_\emptyset} = \emptyset$ for all $R \in \sigma$. The *initialisation time* t_i is the time used for performing **init**. In all algorithms presented in this paper, the **preprocess** routine for input of q and D_0 will carry out the **init** routine for q and then perform a sequence of $|D_0|$ update operations to insert all the tuples of D_0 into the data structure. Consequently, $t_p = t_i + |D_0| \cdot t_u$.

In the following, D will always denote the database that is currently represented by the data structure `D`. To solve the *enumeration problem under updates*, apart from the routines **preprocess** and **update**, we aim at a routine **enumerate** such that calling **enumerate** invokes an enumeration of all tuples, *without repetition*, that belong to the query result $q(D)$. The *delay* t_d is the maximum time used during a call of **enumerate**

- until the output of the first tuple (or the end-of-enumeration message `EOE`, if $q(D) = \emptyset$),
- between the output of two consecutive tuples, and
- between the output of the last tuple and the end-of-enumeration message `EOE`.

To *test* if a given tuple belongs to the query result, instead of **enumerate** we aim at a routine **test** which upon input of a tuple $\overline{a} \in \mathbf{dom}^k$ checks whether $\overline{a} \in q(D)$. The *testing time* t_t is the time used for performing a **test**. To solve the *counting problem under updates*, we aim at a routine **count** which outputs the cardinality $|q(D)|$ of the query result. The *counting time* t_c is the time used for performing a **count**. To *answer* a *Boolean* query under updates, we aim at a routine **answer** that produces the answer `yes` or `no` of q on D. The *answer time* t_a is the time used for performing **answer**. Whenever speaking of a *dynamic algorithm*, we mean an algorithm that has routines **preprocess** and **update** and, depending on the problem at hand, at least one of the routines **answer**, **test**, **count**, and **enumerate**.

When writing *poly*(n) we mean $n^{O(1)}$, and for a query q we often write *poly*(q) instead of *poly*$(\|q\|)$. We will often adopt the view of *data complexity* and suppress factors that may depend on the query q but not on the database D. E.g., "linear preprocessing time" means $t_p \leqslant f(q) \cdot \|D_0\|$ and "constant update time" means $t_u \leqslant f(q)$, for some function f.

Algorithmic conjectures. Similarly to [5] we obtain hardness results that are conditioned on algorithmic conjectures concerning the hardness of the following problems. These problems deal with *Boolean* matrices and vectors, i.e., matrices and vectors over $\{0, 1\}$, and all the arithmetic is done over the Boolean semiring, where multiplication means conjunction and addition means disjunction.

The *orthogonal vectors problem* (OV-problem) is the following decision problem. Given two sets U and V of n Boolean vectors of dimension d, decide whether there are vectors $\vec{u} \in U$ and $\vec{v} \in V$ such that $\vec{u}^\mathsf{T}\vec{v} = 0$. The *OV-conjecture* states that there is no $\epsilon > 0$ such that the OV-problem for $d = \lceil \log^2 n \rceil$ can be solved in time $O(n^{2-\epsilon})$, see [1].

The *online matrix-vector multiplication problem* (OMv-problem) is the following algorithmic task. At first, the algorithm gets a Boolean $n \times n$ matrix M and is allowed to do some preprocessing. Afterwards, the algorithm receives n vectors $\vec{v}^1, \ldots, \vec{v}^n$ one by one and

has to output $M\vec{v}^t$ before it has access to $\vec{v}^{t+1}$ (for each $t < n$). The running time is the overall time the algorithm needs to produce the output $M\vec{v}^1, \ldots, M\vec{v}^n$. The *OMv-conjecture* [19] states that there is no $\epsilon > 0$ such that the OMv-problem can be solved in time $O(n^{3-\epsilon})$.

A related problem is the *OuMv-problem* where the algorithm, again, is given a Boolean $n \times n$ matrix M and is allowed to do some preprocessing. Afterwards, the algorithm receives a sequence of pairs of n-dimensional Boolean vectors $\vec{u}^t, \vec{v}^t$ for each $t \in [n]$, and the task is to compute $(\vec{u}^t)^\intercal M\vec{v}^t$ before accessing $\vec{u}^{t+1}, \vec{v}^{t+1}$. The *OuMv-conjecture* states that there is no $\epsilon > 0$ such that the OuMv-problem can be solved in time $O(n^{3-\epsilon})$. It was shown in [19] that the OuMv-conjecture is equivalent to the OMv-conjecture, i.e., the OuMv-conjecture fails if, and only if, the OMv-conjecture fails.

3 Conjunctive queries

This section's aim is twofold: Firstly, we observe that the notions and results of [5] generalise to CQs with constants in a straightforward way. Secondly, we identify a new subclass of CQs which precisely characterises the CQs for which *testing* can be done efficiently under updates.

The definition of *q-hierarchical* CQs can be taken verbatim from [5]:

▶ **Definition 3.1.** A CQ q is *q-hierarchical* if for any two variables $x, y \in \text{vars}(q)$ we have

(i) $\text{atoms}(x) \subseteq \text{atoms}(y)$ or $\text{atoms}(y) \subseteq \text{atoms}(x)$ or $\text{atoms}(x) \cap \text{atoms}(y) = \emptyset$, and

(ii) if $\text{atoms}(x) \subsetneq \text{atoms}(y)$ and $x \in \text{free}(q)$, then $y \in \text{free}(q)$.

Obviously, it can be checked in time $poly(q)$ whether a given CQ q is q-hierarchical. It is straightforward to see that if a CQ is q-hierarchical, then so is its homomorphic core. In particular, a CQ is equivalent to a q-hierarchical CQ iff its homomorphic core is q-hierarchical. Using the main results of [5], it is not difficult to show the following; for details see [6].

▶ **Theorem 3.2.**

(a) *There is a dynamic algorithm that receives a q-hierarchical k-ary CQ q and a σ-db D_0, and computes within $t_p = poly(q) \cdot O(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = poly(q)$ and allows to*

- **(i)** *compute the cardinality $|q(D)|$ in time $t_c = O(1)$,*
- **(ii)** *enumerate $q(D)$ with delay $t_d = poly(q)$,*
- **(iii)** *test for an input tuple $\overline{a} \in \boldsymbol{dom}^k$ if $\overline{a} \in q(D)$ within time $t_t = poly(q)$,*
- **(iv)** *and when given a tuple $\overline{a} \in q(D)$, the tuple $\overline{a}'$ (or the message* `EOE`*) that the enumeration procedure of* (aii) *would output directly after having output $\overline{a}$, can be computed within time $poly(q)$.*

(b) *Let $\epsilon > 0$ and let q be a CQ that is not equivalent to a q-hierarchical CQ.*

- **(i)** *If q is Boolean, then there is no dynamic algorithm with arbitrary preprocessing time and $t_u = O(n^{1-\varepsilon})$ update time that answers $q(D)$ in time $t_a = O(n^{2-\varepsilon})$, unless the OMv-conjecture fails.*
- **(ii)** *There is no dynamic algorithm with arbitrary preprocessing time and $t_u = O(n^{1-\varepsilon})$ update time that computes the cardinality $|q(D)|$ in time $t_c = O(n^{1-\varepsilon})$, unless the OMv-conjecture or the OV-conjecture fails.*
- **(iii)** *If q is self-join-free, then there is no dynamic algorithm with arbitrary preprocessing time and $t_u = O(n^{1-\varepsilon})$ update time that enumerates $q(D)$ with delay $t_d = O(n^{1-\varepsilon})$, unless the OMv-conjecture fails.*

All lower bounds remain true if we restrict ourselves to the class of databases that map homomorphically into q.

Note that neither the results of [5] nor Theorem 3.2 provide a precise characterisation of the CQs for which *testing* can be done efficiently under updates. Of course, according to Theorem 3.2 (aiii), the testing problem can be solved with constant update time and constant testing time for every *q-hierarchical* CQ. But the same holds true for the non-q-hierarchical CQ $p_{S\text{-}E\text{-}T} := \{ (x, y) : Sx \wedge Exy \wedge Ty \}$. The corresponding dynamic algorithm simply uses 1-dimensional arrays $\mathtt{A}_S$ and $\mathtt{A}_T$ and a 2-dimensional array $\mathtt{A}_E$ such that for all $a, b \in \mathbf{dom}$ we have $\mathtt{A}_E[a, b] = 1$ if $(a, b) \in E^D$, and $\mathtt{A}_E[a, b] = 0$ otherwise, and $\mathtt{A}_R[a] = 1$ if $a \in R^D$, and $\mathtt{A}_R[a] = 0$ otherwise, for $R \in \{S, T\}$. When given an update command, the arrays can be updated within constant time. And when given a tuple $(a, b) \in \mathbf{dom}^2$, the **test** routine simply looks up the array entries $\mathtt{A}_S[a]$, $\mathtt{A}_E[a, b]$, $\mathtt{A}_T[b]$ and returns the correct query result accordingly. To characterise the conjunctive queries for which testing can be done efficiently under updates, we introduce the following notion of *t-hierarchical* CQs.

▶ **Definition 3.3.** A CQ q is *t-hierarchical* if the following is satisfied:

(i) for all $x, y \in \mathrm{vars}(q) \setminus \mathrm{free}(q)$, we have
$\mathrm{atoms}(x) \subseteq \mathrm{atoms}(y)$ or $\mathrm{atoms}(y) \subseteq \mathrm{atoms}(x)$ or $\mathrm{atoms}(x) \cap \mathrm{atoms}(y) = \emptyset$, and

(ii) for all $x \in \mathrm{free}(q)$ and all $y \in \mathrm{vars}(q) \setminus \mathrm{free}(q)$, we have
$\mathrm{atoms}(x) \cap \mathrm{atoms}(y) = \emptyset$ or $\mathrm{atoms}(y) \subseteq \mathrm{atoms}(x)$.

Obviously, it can be checked in time *poly*(q) whether a given CQ q is t-hierarchical. Note that every q-hierarchical CQ is t-hierarchical, and a *Boolean* query is t-hierarchical if and only if it is q-hierarchical. The queries $p_{S\text{-}E\text{-}T}$ and $p_{E\text{-}E\text{-}R} := \{ (x, y) : \exists v_1 \exists v_2 \exists v_3 (Exv_1 \wedge Eyv_2 \wedge Rxyv_3) \}$ are examples for queries that are t-hierarchical but not q-hierarchical. It is straightforward to verify that if a CQ is t-hierarchical, then so is its homomorphic core. This section's main result shows that the *t-hierarchical* CQs precisely characterise the CQs for which the *testing* problem can be solved efficiently under updates:

▶ **Theorem 3.4.**

(a) *There is a dynamic algorithm that receives a t-hierarchical k-ary CQ q and a σ-db D_0, and computes within $t_p = \mathit{poly}(q) \cdot O(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = \mathit{poly}(q)$ and allows to test for an input tuple $\overline{a} \in \mathbf{dom}^k$ if $\overline{a} \in q(D)$ within time $t_t = \mathit{poly}(q)$.*

(b) *Let $\epsilon > 0$ and let q be a k-ary CQ that is not equivalent to a t-hierarchical CQ. There is no dynamic algorithm with arbitrary preprocessing time and $t_u = O(n^{1-\epsilon})$ update time that can test for any input tuple $\overline{a} \in \mathbf{dom}^k$ if $\overline{a} \in q(D)$ within testing time $t_t = O(n^{1-\epsilon})$, unless the OMv-conjecture fails. The lower bound remains true if we restrict ourselves to the class of databases that map homomorphically into q.*

Proof. To avoid notational clutter, and without loss of generality, we restrict attention to queries $q_\varphi(u_1, \ldots, u_k)$ where $(u_1, \ldots, u_k)$ is of the form $(z_1, \ldots, z_k)$ for pairwise distinct variables $z_1, \ldots, z_k$. For the proof of (a), we combine the array construction described above for the example query $p_{S\text{-}E\text{-}T}$ with the dynamic algorithm provided by Theorem 3.2 (a) and the following Lemma 3.5. To formulate the lemma, we need the following notation. A k-ary *generalised CQ* is of the form $\{ (z_1, \ldots, z_k) : \varphi_1 \wedge \cdots \wedge \varphi_m \}$ where $k \geqslant 0$, $z_1, \ldots, z_k$ are pairwise distinct variables, $m \geqslant 1$, φ_j is a conjunctive formula for each $j \in [m]$, $\mathrm{free}(\varphi_1) \cup \cdots \cup \mathrm{free}(\varphi_m) = \{z_1, \ldots, z_k\}$, and the quantified variables of φ_j and $\varphi_{j'}$ are pairwise disjoint for all $j, j' \in [m]$ with $j \neq j'$ and disjoint from $\{z_1, \ldots, z_k\}$. For each $j \in [m]$ let $\overline{z}^{(j)}$ be the sublist of $\overline{z} := (z_1, \ldots, z_k)$ that only contains the variables in $\mathrm{free}(\varphi_j)$. I.e., $\overline{z}^{(j)}$ is obtained from $\overline{z}$ by deleting all variables that do not belong to $\mathrm{free}(\varphi_j)$. Accordingly, for a tuple $\overline{a} = (a_1, \ldots, a_k) \in \mathbf{dom}^k$ by $\overline{a}^{(j)}$ we denote the tuple that contains exactly those a_i

where z_i belongs to $\overline{z}^{(j)}$. The query result of q on a σ-db D is the set

$$q(D) \;:=\; \{\, \overline{a} \in \mathbf{dom}^k \;:\; D \models \varphi_j[\overline{a}^{(j)}] \;\text{ for each } j \in [m] \,\},$$

where $D \models \varphi_j[\overline{a}^{(j)}]$ means that there is a homomorphism $\beta_j : q_j \to D$ for the query $q_j := \{\, \overline{z}^{(j)} \;:\; \varphi_j \,\}$, with $\beta_j(z_i) = a_i$ for every i with $z_i \in \text{free}(\varphi_j)$. For example, $p'_{E\text{-}E\text{-}R} := \{\, (x,y) \;:\; \exists v_1\, Exv_1 \;\wedge\; \exists v_2\, Eyv_2 \;\wedge\; \exists v_3\, Rxyv_3 \,\}$ is a generalised CQ that is equivalent to the CQ $p_{E\text{-}E\text{-}R}$. The proof of the following lemma can be found in the appendix.

▶ **Lemma 3.5.** *Every t-hierarchical CQ $q_\varphi(z_1,\ldots,z_k)$ is equivalent to a generalised CQ $q' = \{\, (z_1,\ldots,z_k) : \varphi_1 \wedge \cdots \wedge \varphi_m \,\}$ such that for each $j \in [m]$ the CQ $q_j := \{\, \overline{z}^{(j)} : \varphi_j \,\}$ is q-hierarchical or quantifier-free. Furthermore, there is an algorithm which decides in time poly(q_φ) whether q_φ is t-hierarchical, and if so, outputs an according q'.*

The proof of Theorem 3.4 (a) now follows easily: When given a t-hierarchical CQ $q_\varphi(z_1,\ldots,z_k)$, use the algorithm provided by Lemma 3.5 to compute an equivalent generalised CQ q' of the form $\{(z_1,\ldots,z_k) \;:\; \varphi_1 \wedge \cdots \wedge \varphi_m\}$ and let $q_j := \{\overline{z}^{(j)} \;:\; \varphi_j\}$ for each $j \in [m]$. W.l.o.g. assume that there is an $m' \in \{0,\ldots,m\}$ such that q_j is q-hierarchical for each $j \leqslant m'$ and q_j is quantifier-free for each $j > m'$. We use in parallel, for each $j \leqslant m'$, the data structures provided by Theorem 3.2 (a) for the q-hierarchical CQ q_j. In addition to this, we use an r-dimensional array $\mathtt{A}_R$ for each relation symbol $R \in \sigma$ of arity $r := \text{ar}(R)$, and we ensure that for all $\overline{b} \in \mathbf{dom}^r$ we have $\mathtt{A}_R[\overline{b}] = 1$ if $\overline{b} \in R^D$, and $\mathtt{A}_R[\overline{b}] = 0$ otherwise. When receiving an update command $\mathsf{update}\, R(\overline{b})$, we let $\mathtt{A}_R[\overline{b}] := 1$ if $\mathsf{update} = \mathsf{insert}$, and $\mathtt{A}_R[\overline{b}] := 0$ if $\mathsf{update} = \mathsf{delete}$, and in addition to this, we call the **update** routines of the data structure for $q^{(j)}$ for each $j \leqslant m'$. Upon input of a tuple $\overline{a} \in \mathbf{dom}^k$, the **test** routine proceeds as follows. For each $j \leqslant m'$, it calls the **test** routine of the data structure for $q^{(j)}$ upon input $\overline{a}^{(j)}$. Additionally, it uses the arrays $\mathtt{A}_R$ for all $R \in \sigma$ to check if for each $j > m'$ the quantifier-free query $q^{(j)}$ is satisfied by the tuple $\overline{a}^{(j)}$. All this is done within time *poly*(q), and we know that $\overline{a} \in q(D)$ if, and only if, all these tests succeed. This completes the proof of part (a) of Theorem 3.4.

Let us now turn to the proof of part (b) of Theorem 3.4. We are given a query $q := q_\varphi(z_1,\ldots,z_k)$, and without loss of generality we assume that q is a homomorphic core and q is not t-hierarchical. Thus, q violates condition (i) or (ii) of Definition 3.3. In case that it violates condition (i), the proof is virtually identical to the proof of Theorem 3.4 in [5] (see [6] for a proof). Let us consider the case where q violates condition (ii) of Definition 3.3. In this case, there are two variables $x \in \text{free}(q)$ and $y \in \text{vars}(q) \setminus \text{free}(q)$ and two atoms $\psi^{x,y}$ and ψ^y of q with $\text{vars}(\psi^{x,y}) \cap \{x,y\} = \{x,y\}$ and $\text{vars}(\psi^y) \cap \{x,y\} = \{y\}$. The easiest example of a query for which this is true is $q_{E\text{-}T} := \{\, (x) \;:\; \exists y\, (\, Exy \wedge Ty \,) \,\}$. Here, we illustrate the proof idea for the particular query $q_{E\text{-}T}$; a proof for the general case can be found in [6].

Assume that there is a dynamic algorithm that solves the testing problem for $q_{E\text{-}T}$ with update time $t_u = O(n^{1-\epsilon})$ and testing time $t_t = O(n^{1-\epsilon})$ on databases whose active domain is of size $O(n)$. We show how this algorithm can be used to solve the OuMv-problem.

For the OuMv-problem, we receive as input an $n \times n$ matrix M. We start the preprocessing phase of our testing algorithm for $q_{E\text{-}T}$ with the empty database $D = (E^D, T^D)$ where $E^D = T^D = \emptyset$. As this database has constant size, the preprocessing is finished in constant time. We then apply $O(n^2)$ update steps to ensure that $E^D = \{(i,j) \;:\; M_{i,j} = 1\}$. All this takes time at most $O(n^2) \cdot t_u = O(n^{3-\epsilon})$. Throughout the remainder of the construction, we will never change E^D, and we will always ensure that $T^D \subseteq [n]$.

When we receive two vectors $\vec{u}^t$ and $\vec{v}^t$ in the dynamic phase of the OuMv-problem, we proceed as follows. First, we perform the update commands $\mathsf{delete}\, T(j)$ for each $j \in [n]$ with

$\vec{v}_j^t = 0$, and the update commands $\mathsf{insert}\, T(j)$ for each $j \in [n]$ with $\vec{v}_j^t = 1$. This is done within time $n \cdot t_u = O(n^{2-\epsilon})$. By construction of D we know that for every $i \in [n]$ we have

$$i \in q_{E\text{-}T}(D) \quad \Longleftrightarrow \quad \text{there is a } j \in [n] \text{ such that } M_{i,j} = 1 \text{ and } \vec{v}_j^t = 1\,.$$

Thus, $(\vec{u}^t)^\mathsf{T} M \vec{v}^t = 1 \iff$ there is an $i \in [n]$ with $\vec{u}_i^t = 1$ and $i \in q_{E\text{-}T}(D)$. Therefore, after having called the **test** routine for $q_{E\text{-}T}$ for each $i \in [n]$ with $\vec{u}_i^t = 1$, we can output the correct result of $(\vec{u}^t)^\mathsf{T} M \vec{v}^t$. This takes time at most $n \cdot t_t = O(n^{2-\epsilon})$. I.e., for each $t \in [n]$ after receiving the vectors $\vec{u}^t$ and $\vec{v}^t$, we can output $(\vec{u}^t)^\mathsf{T} M \vec{v}^t$ within time $O(n^{2-\epsilon})$. Consequently, the overall running time for solving the OuMv-problem is bounded by $O(n^{3-\epsilon})$.

Using the technical machinery of [5], this can be generalised from $q_{E\text{-}T}$ to all queries q that violate condition (ii) of Definition 3.3. This completes the proof of Theorem 3.4. ◀

4 Unions of conjunctive queries

In this section we consider dynamic query evaluation for UCQs. To transfer our notions of *hierarchical* queries from CQs to UCQs, we say that a UCQ $q(\overline{u})$ of the form $q_1(\overline{u}_1) \cup \cdots \cup q_d(\overline{u}_d)$ is q-hierarchical (t-hierarchical) if every CQ $q_i(\overline{u}_i)$ in the union is q-hierarchical (t-hierarchical). Note that for *Boolean* queries (CQs as well as UCQs) the notions of being q-hierarchical and being t-hierarchical coincide, and for a k-ary UCQ q it can be checked in time $poly(q)$ if q is q-hierarchical or t-hierarchical.

Testing. The following theorem generalises the statement of Theorem 3.4 from CQs to UCQs. Its proof follows easily from the Theorems 3.4 and 3.2; see [6] for details.

▶ **Theorem 4.1.**

(a) *There is a dynamic algorithm that receives a t-hierarchical k-ary UCQ q and a σ-db D_0, and computes within $t_p = poly(q) \cdot O(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = poly(q)$ and allows to test for an input tuple $\overline{a} \in \boldsymbol{dom}^k$ if $\overline{a} \in q(D)$ within time $t_t = poly(q)$. Furthermore, the algorithm allows to answer a t-hierarchical Boolean UCQ within time $t_a = O(1)$.*

(b) *Let $\epsilon > 0$ and let q be a k-ary UCQ that is not equivalent to a t-hierarchical UCQ. There is no dynamic algorithm with arbitrary preprocessing time and $t_u = O(n^{1-\epsilon})$ update time that can test for any input tuple $\overline{a} \in \boldsymbol{dom}^k$ if $\overline{a} \in q(D)$ within testing time $t_t = O(n^{1-\epsilon})$, unless the OMv-conjecture fails. Furthermore, if $k = 0$ (i.e., q is a Boolean UCQ), then there is no dynamic algorithm with arbitrary preprocessing time and $t_u = O(n^{1-\varepsilon})$ update time that answers $q(D)$ in time $t_a = O(n^{2-\varepsilon})$, unless the OMv-conjecture fails.*

Enumerating. It turns out that q-hierarchical UCQs, like q-hierarchical CQs, allow for efficient enumeration under updates. This, and the according lower bound, is stated in the following Theorem 4.2. In contrast to Theorem 4.1, the result does not follow immediately from the tractability of the enumeration problem for q-hierarchical CQs, because one has to ensure that tuples from result sets of two different CQs are not reported twice while enumerating their union.

▶ **Theorem 4.2.**

(a) *There is a dynamic algorithm that receives a q-hierarchical k-ary UCQ q and a σ-db D_0, and computes within $t_p = poly(q) \cdot O(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = poly(q)$ and allows to enumerate $q(D)$ with delay $t_d = poly(q)$.*

(b) *Let $\epsilon > 0$ and let q be a k-ary UCQ whose homomorphic core is not q-hierarchical and is a union of self-join-free CQs. There is no dynamic algorithm with arbitrary preprocessing time and $t_u = O(n^{1-\varepsilon})$ update time that enumerates $q(D)$ with delay $t_d = O(n^{1-\varepsilon})$, unless the OMv-conjecture fails.*

To prove Theorem 4.2 (a), we first develop a general method for enumerating the union of sets. We say that a data structure for a set T *allows to skip* if it is possible to test whether $t \in T$ in constant time and for some ordering $t_1, \ldots, t_n$ of the elements in T there are a function start, which returns t_1 in constant time, and a function $\mathsf{next}(t_i)$, which returns t_{i+1} (if $i < n$) or $\mathtt{EOE}$ (if $i = n$) in constant time. Note that a data structure that allows to skip enables constant delay enumeration of t_i, t_{i+1}, $\ldots$, t_n starting from an arbitrary element $t_i \in T$ (but we do not have control over the underlying order). An example of such a data structure is an explicit representation of the elements of T in a linked list with constant access. Another example is the data structure of the enumeration algorithm for the result $T := q(D)$ of a q-hierarchical CQ q, provided by Theorem 3.2 (aii)&(aiv). The next lemma states that we can use these data structures for sets T_j to enumerate the union $\bigcup_j T_j$ with constant delay and without repetition.

▶ **Lemma 4.3.** *Let $\ell \geqslant 1$ and let $T_1, \ldots, T_\ell$ be sets such that for each $j \in [\ell]$ there is a data structure for T_j that allows to skip. Then there is an algorithm that enumerates, without repetition, all elements in $T_1 \cup \cdots \cup T_\ell$ with $O(\ell)$ delay.*

Proof. For each $i \in [\ell]$ let start^i and next^i be the start element and the iterator for the set T_i. The main idea for enumerating the union $T_1 \cup \cdots \cup T_\ell$ is to first enumerate all elements in T_1, and then $T_2 \setminus T_1$, $T_3 \setminus (T_1 \cup T_2)$, $\ldots$, $T_\ell \setminus (T_1 \cup \cdots \cup T_{\ell-1})$. In order to do this we have to exclude all elements that have already been reported from all subsequent sets. As we want to ensure constant delay enumeration, we cannot just ignore the elements in $T_i \cap (T_1 \cup \cdots \cup T_{i-1})$ while enumerating T_i. As a remedy, we use an additional pointer to jump from an element that has already been reported to the least element that needs to be reported next. To do this we use arrays skip^i (for all $i \in [\ell]$) to jump over excluded elements. For technical reasons we add for each set T_i a dummy element $\mathtt{EOE}$ at the end of its list representation. The algorithm preserves the following invariant: "If $t_r, \ldots, t_s$ is a maximal interval of elements in T_i that have already been reported, then $\mathsf{skip}^i[t_r] = t_{s+1}$." For technical reasons we also need the array $\mathsf{skipback}^i$ which represents the inverse pointer, i.e., $\mathsf{skipback}^i[t_{s+1}] = t_r$. It follows from the invariant that $\mathsf{skip}^i[t] \neq \mathtt{nil}$ implies $\mathsf{skip}^i[\mathsf{skip}^i[t]] = \mathtt{nil}$. As a consequence, Algorithm 1 enumerates elements with constant delay. It uses the procedure $\textsc{exclude}^j$ described in Algorithm 2 to update the arrays whenever an element t has been reported (note that every element is excluded at most once). See Figure 1 in the appendix for an illustration. It is straightforward to verify that these algorithms provide the desired functionality within the claimed time bounds. ◀

Proof of Theorem 4.2. The upper bound follows immediately from combining Lemma 4.3 with Theorem 3.2 (aiv). For the lower bound let q_i be a self-join-free non-q-hierarchical CQ in the homomorphic core q' of the UCQ q. For every database D that maps homomorphically into q_i it holds that $q_j(D) = \emptyset$ for every other CQ q_j in q' (with $j \neq i$), since otherwise there would be a homomorphism from q_j to D and hence to q_i, contradicting that q' is a homomorphic core. It follows that every dynamic algorithm that enumerates the result of q on a database D which maps homomorphically into q_i also enumerates $q_i(D) = q(D)$, contradicting Theorem 3.2 (biii). ◀

Algorithm 1 Enumeration algorithm for $T_1 \cup \cdots \cup T_\ell$.

Input: Data structures for sets T_j with first element start^j and iterator next^j.
Pointer $\mathsf{skip}^j[t] = \mathsf{skipback}^j[t] = \texttt{nil}$ for all $j \in [\ell]$ and $t \in T_j$.

for $i = 1, \ldots, \ell$ **do**
  $t = \mathsf{start}^i$
  while $t \neq \texttt{EOE}$ **do**
    if $\mathsf{skip}^i[t] == \texttt{nil}$ **then**
      Output element t
      for $j = i + 1 \ \rightarrow \ \ell$ **do**
        EXCLUDE$^j(t)$
      $t = \mathsf{next}^i(t)$
    else
      $t = \mathsf{skip}^i[t]$
  Output the end-of-enumeration message EOE.

Algorithm 2 Procedure EXCLUDE^j for excluding t from T_j.

if $t \in T_j$ **then**
  if $\mathsf{skipback}^j[t] \neq \texttt{nil}$ **then**
    $t^- = \mathsf{skipback}^j[t]$
    $\mathsf{skipback}^j[t] = \texttt{nil}$
  else
    $t^- = t$
  if $\mathsf{skip}^j[\mathsf{next}^j(t)] \neq \texttt{nil}$ **then**
    $t^+ = \mathsf{skip}^j[\mathsf{next}^j(t)]$
    $\mathsf{skip}^j[\mathsf{next}^j(t)] = \texttt{nil}$
  else
    $t^+ = \mathsf{next}^j(t)$
  $\mathsf{skip}^j[t^-] = t^+; \quad \mathsf{skipback}^j[t^+] = t^-$

Counting. Note that according to Theorem 3.2, for CQs the enumeration problem as well as the counting problem can be solved by efficient dynamic algorithms if, and (modulo algorithmic conjectures) only if, the query is q-hierarchical. In contrast to this, it turns out that for UCQs computing the number of output tuples can be much harder than enumerating the query result. To characterise the UCQs that allow for efficient dynamic counting algorithms, we use the following notation. For two k-ary CQs $q_\varphi(u_1, \ldots, u_k)$ and $q_\psi(v_1, \ldots, v_k)$ we define the intersection $q := q_\varphi \cap q_\psi$ to be the following k-ary query. If there is an $i \in [k]$ such that u_i and v_i are distinct elements from **dom**, then $q := \emptyset$ (and this query is q-hierarchical by definition). Otherwise, we let $w_1, \ldots, w_k$ be elements from $\mathbf{var} \cup \mathbf{dom}$ which satisfy the following for all $i, j \in [k]$ and all $a \in \mathbf{dom}$:

$$\big(\ w_i = a \iff u_i = a \ \text{ or } \ v_i = a\ \big) \quad \text{and} \quad \big(\ w_i = w_j \iff u_i = u_j \ \text{ or } \ v_i = v_j\ \big).$$

We obtain φ' from φ (and ψ' from ψ) by replacing every $u_i \in \{u_1, \ldots, u_k\} \cap \text{free}(\varphi)$ (and $v_i \in \{v_1, \ldots, v_k\} \cap \text{free}(\psi)$) by w_i. Finally, we let $q = \{\ (w_1, \ldots, w_k) \ : \ \varphi' \wedge \psi'\ \}$, where we can assume that $\varphi' \wedge \psi'$ is (equivalent to) a conjunctive formula of the form $(*)$. Note that for every database D it holds that $q(D) = q_\varphi(D) \cap q_\psi(D)$.

To compute the number of result tuples in a UCQ $q = \bigcup_{i \in [d]} q_i(\overline{u}_i)$ we first define for every $I \subseteq [d]$ the CQ $q_I = \bigcap_{i \in I} q_i$. To take care of equivalent queries q_I and $q_{I'}$ we define

the equivalence relation $I \cong I' \iff q_I \equiv q_{I'}$ and let $\mathfrak{P}$ be the partition of $\{I \,:\, \emptyset \neq I \subseteq [d]\}$ into equivalence classes. For an $\mathcal{I} \in \mathfrak{P}$ we denote by $q_{\mathcal{I}}$ the common homomorphic core of all q_I, $I \in \mathcal{I}$, and define $a_{\mathcal{I}} := \sum_{I \in \mathcal{I}} (-1)^{|I|+1}$. By the inclusion-exclusion principle we get:

$$|q(D)| \quad = \quad \sum_{\emptyset \neq I \subseteq [d]} (-1)^{|I|+1} \cdot |q_I(D)| \quad = \quad \sum_{\mathcal{I} \in \mathfrak{P}} a_{\mathcal{I}} \cdot |q_{\mathcal{I}}(D)|. \tag{1}$$

If all $q_{\mathcal{I}}$ with non-zero coefficients $a_{\mathcal{I}}$ are q-hierarchical, then we can compute the result size of the UCQ as a linear combination of a constant number of q-hierarchical CQs. We will show in Theorem 4.5 that this approach is indeed optimal, justifying the following definition.

▶ **Definition 4.4.** A UCQ q is *exhaustively q-hierarchical* if $q_{\mathcal{I}}$ is q-hierarchical for every $\mathcal{I} \in \mathfrak{P}$ with $a_{\mathcal{I}} \neq 0$.

Being exhaustively q-hierarchical is a stronger requirement than being q-hierarchical, e.g., the UCQ $\{(x,y) : Sx \wedge Exy\} \cup \{(x,y) : Exy \wedge Ty\}$ is q-hierarchical, but not exhaustively q-hierarchical. The straightforward way of deciding whether a UCQ q is exhaustively q-hierarchical requires time $2^{poly(q)}$, and it is open whether this can be improved. The next theorem shows that the exhaustively q-hierarchical queries are precisely those UCQs that allow for efficient dynamic counting algorithms.

▶ **Theorem 4.5.**

(a) *There is a dynamic algorithm that receives an exhaustively q-hierarchical UCQ q and a σ-db D_0, computes in $t_p = 2^{poly(q)} \cdot O(\|D_0\|)$ preprocessing time a data structure that can be updated in time $t_u = 2^{poly(q)}$ and computes $|q(D)|$ in time $t_c = O(1)$.*

(b) *Let $\epsilon > 0$ and let q be a UCQ that is not exhaustively q-hierarchical. There is no dynamic algorithm with arbitrary preprocessing time and $t_u = O(n^{1-\varepsilon})$ update time that computes $|q(D)|$ in time $t_c = O(n^{1-\varepsilon})$, unless the OMv-conjecture or the OV-conjecture fails.*

Proof. Part (a) follows from the upper bound of Theorem 3.2 (ai) and the inclusion-exclusion argument (1). For proving part (b) let $\mathfrak{H} \subseteq \mathfrak{P}$ be the set of equivalence classes $\mathcal{I}$ such that $a_{\mathcal{I}} \neq 0$ and $q_{\mathcal{I}}$ is q-hierarchical, and let $\mathfrak{N} \subseteq \mathfrak{P}$ be the set of equivalence classes $\mathcal{I}$ such that $a_{\mathcal{I}} \neq 0$ and $q_{\mathcal{I}}$ is *not* q-hierarchical. By Definition 4.4 we have that $\mathfrak{N} \neq \emptyset$. Moreover, since for all distinct $\mathcal{I}, \mathcal{I}' \in \mathfrak{N}$ the queries $q_{\mathcal{I}}$ and $q_{\mathcal{I}'}$ are not homomorphically equivalent, we can choose a $\mathcal{J} \in \mathfrak{N}$, which is *minimal* in the sense that for every $\mathcal{I} \in \mathfrak{N} \setminus \{\mathcal{J}\}$ there is no homomorphism from $q_{\mathcal{I}}$ to $q_{\mathcal{J}}$.

Now suppose that D is a database from the class of databases that map homomorphically into $q_{\mathcal{J}}$ and let $h\colon D \to q_{\mathcal{J}}$ be a homomorphism. For every $\mathcal{I} \in \mathfrak{N} \setminus \{\mathcal{J}\}$ it holds that there is no homomorphism $h'\colon q_{\mathcal{I}} \to D$, since otherwise $h \circ h'$ would be a homomorphism from $q_{\mathcal{I}}$ to $q_{\mathcal{J}}$. Hence, $q_{\mathcal{I}}(D) = \emptyset$ for all $\mathcal{I} \in \mathfrak{N} \setminus \{\mathcal{J}\}$ and thus, by (1) we have

$$|q(D)| \quad = \quad a_{\mathcal{J}} \cdot |q_{\mathcal{J}}(D)| \quad + \quad \sum_{\mathcal{I} \in \mathfrak{H}} a_{\mathcal{I}} \cdot |q_{\mathcal{I}}(D)|.$$

Assume for contradiction that we can efficiently compute $|q(D)|$ with update time t_u and counting time t_c. By Theorem 3.2 (ai) we can maintain $|q_{\mathcal{I}}(D)|$ for each $\mathcal{I} \in \mathfrak{H}$ with update time $poly(q_{\mathcal{I}})$ and counting time $O(1)$. Thus, we can compute $|q_{\mathcal{J}}(D)| = \frac{1}{a_{\mathcal{J}}} \big(|q(D)| - \sum_{\mathcal{I} \in \mathfrak{H}} a_{\mathcal{I}} \cdot |q_{\mathcal{I}}(D)| \big)$ with update time $t_u + 2^{poly(q)}$ and counting time $t_c + 2^{poly(q)}$. Since $q_{\mathcal{J}}$ is a non-q-hierarchical homomorphic core, the lower bound for maintaining $|q(D)|$ follows from Theorem 3.2 (bii). This completes the proof of Theorem 4.5. ◀

5 CQs and UCQs with integrity constraints

In the presence of integrity constraints, the characterisation of tractable queries changes and depends on the query as well as on the set of constraints. When considering a scenario where databases are required to satisfy a set Σ of constraints, we allow to execute a given **update** command only if the resulting database still satisfies all constraints in Σ. When speaking of (σ, Σ)-*dbs* we mean σ-dbs D that satisfy all constraints in Σ. Two queries q and q' are Σ-*equivalent* (for short: $q \equiv_\Sigma q'$) if $q(D) = q'(D)$ for every (σ, Σ)-db D.

We first consider *small domain constraints*, i.e., constraints δ of the form $R[i] \subseteq C$ where $R \in \sigma$, $i \in \{1, \ldots, \text{ar}(R)\}$, and $C \subseteq \mathbf{dom}$ is a finite set. A σ-db D *satisfies* δ if $\pi_i(R^D) \subseteq C$.

For these constraints we are able to give a clear picture of the tractability landscape by reducing CQs and UCQs with small domain constraints to UCQs without integrity constraints and applying the characterisations for UCQs achieved in Section 4. We start with an example that illustrates how a query can be simplified in the presence of small domain constraints.

▶ **Example 5.1.** Consider the Boolean query $q_{S\text{-}E\text{-}T} := \{ () : \exists x \exists y (Sx \wedge Exy \wedge Ty) \}$, which is not q-hierarchical. By Theorem 3.2 it cannot be answered by a dynamic algorithm with sublinear update time and sublinear answer time, unless the OMv-conjecture fails. But in the presence of the small domain constraint $\delta_{sd} := S[1] \subseteq C$ for a set $C = \{a_1, \ldots, a_c\} \subseteq \mathbf{dom}$, the query $q_{S\text{-}E\text{-}T}$ is $\{\delta_{sd}\}$-equivalent to the q-hierarchical UCQ $q' := \bigcup_{a_i \in C} \{ () : \exists y (Sa_i \wedge Ea_iy \wedge Ty) \}$. Therefore, by Theorem 4.1, q' and hence $q_{S\text{-}E\text{-}T}$ can be answered with constant update time and constant answer time on all databases that satisfy δ_{sd}.

For handling the general case, assume we are given a set Σ of small domain constraints and an arbitrary k-ary CQ q of the form $(**)$ where φ is of the form $(*)$. We define a function $Dom_{q,\Sigma}$ that maps each $x \in \text{vars}(q)$ to a set $Dom_{q,\Sigma}(x) \subseteq \mathbf{dom}$ as follows. As an initialisation let $f(x) = \mathbf{dom}$ for each $x \in \text{vars}(q)$. Consider each constraint δ in Σ and let $S[i] \subseteq C$ be the form of δ. Consider each atom ψ_j of φ and let $Rv_1 \cdots v_r$ be the form of ψ_j. If $R = S$ and $v_i \in \mathbf{var}$, then let $f(v_i) := f(v_i) \cap C$. We let $Dom_{q,\Sigma}$ be the mapping f obtained at the end of this process and define the set of variables of q that are restricted by Σ by $rvars_\Sigma(q) := \{x \in \text{vars}(q) : Dom_{q,\Sigma}(x) \neq \mathbf{dom}\}$. Let $M_{q,\Sigma}$ be the set of all mappings $\alpha : V \to \mathbf{dom}$ with $V = rvars_\Sigma(q)$ and $\alpha(x) \in Dom_{q,\Sigma}(x)$ for each $x \in V$. Note that $M_{q,\Sigma}$ is finite; and it is empty if, and only if, $Dom_{q,\Sigma}(x) = \emptyset$ for some $x \in \text{vars}(q)$. For a mapping $\alpha : V \to \mathbf{dom}$ with $V \subseteq \mathbf{var}$ we let q_α be the k-ary CQ obtained from q as follows: for each $x \in V$, if present in q, the existential quantifier "$\exists x$" is omitted, and afterwards every occurrence of x in q is replaced with the constant $\alpha(x)$. Clearly, $q_\alpha(D) \subseteq q(D)$ for every σ-db D. With these notations, we obtain the following (the proof can be found in [6]).

▶ **Lemma 5.2.** *For a CQ q and a set Σ of small domain constraints, let $M := M_{q,\Sigma}$. If $M = \emptyset$, then $q(D) = \emptyset$ for every (σ, Σ)-db D. Otherwise, q is Σ-equivalent to the UCQ $q_\Sigma := \bigcup_{\alpha \in M} q_\alpha$.*

This reduction from a CQ q to a UCQ q_Σ directly translates to UCQs: if q is a union of the CQs $q_1, \ldots, q_d$, then we let $q_\Sigma := \bigcup_{i \in [d]} (q_i)_\Sigma$. Note that if the UCQ q is a homomorphic core, then so is q_Σ. Therefore, the following dichotomy theorem for UCQs under small domain constraints is a direct consequence of Lemma 5.2 and the Theorems 4.1, 4.2, and 4.5.

▶ **Theorem 5.3.** *Let q be a UCQ that is a homomorphic core and Σ a set of small domain constraints with $M_{q,\Sigma} \neq \emptyset$. Suppose that the OMv-conjecture and the OV-conjecture hold.*

(1a) *If q_Σ is t-hierarchical, then q can be tested on (σ, Σ)-dbs in constant time with linear preprocessing time and constant update time.*

(1b) *If q_Σ is not t-hierarchical, then on the class of (σ,Σ)-dbs testing in time $O(n^{1-\epsilon})$ is not possible with $O(n^{1-\epsilon})$ update time.*

(2a) *If q_Σ is q-hierarchical, then there is a data structure with linear preprocessing and constant update time that allows to enumerate $q(D)$ with constant delay on (σ,Σ)-dbs.*

(2b) *If q_Σ is not q-hierarchical and in addition self-join-free, then $q(D)$ cannot be enumerated with $O(n^{1-\epsilon})$ delay and $O(n^{1-\epsilon})$ update time on (σ,Σ)-dbs.*

(3a) *If q_Σ is exhaustively q-hierarchical, then there is data structure with linear preprocessing and constant update time that allows to compute $|q(D)|$ in constant time on (σ,Σ)-dbs.*

(3b) *If q_Σ is not exhaustively q-hierarchical, then computing $|q(D)|$ on (σ,Σ)-dbs in time $O(n^{1-\epsilon})$ is not possible with $O(n^{1-\epsilon})$ update time.*

Thus, the tractability of a UCQ q on (σ,Σ)-dbs only depends on the structure of the query q_Σ. Note that while the size of q_Σ might be $c^{O(q)}$, where c is the largest number of constants in a small domain, it can be checked in time $poly(q)$ whether q_Σ is (t- or q-)hierarchical.

Let us take a brief look at two other kinds of constraints: inclusion dependencies and functional dependencies, which both can also cause a hard query to become tractable. An *inclusion dependency* δ is of the form $R[i_1,\dots,i_m] \subseteq S[j_1,\dots,j_m]$ where $R,S \in \sigma$, $m \geqslant 1$, $i_1,\dots,i_m \in \{1,\dots,\mathrm{ar}(R)\}$, and $j_1,\dots,j_m \in \{1,\dots,\mathrm{ar}(S)\}$. A σ-db D *satisfies* δ if $\pi_{i_1,\dots,i_m}(R^D) \subseteq \pi_{j_1,\dots,j_m}(S^D)$. As an example, consider the query $q_{S\text{-}E\text{-}T}$ and the inclusion dependency $\delta_{ind} := E[2] \subseteq T[1]$. Obviously, $q_{S\text{-}E\text{-}T}$ is $\{\delta_{ind}\}$-equivalent to the q-hierarchical (and hence easy) CQ $q' := \{\ () \ :\ \exists x \exists y\ (\ Sx \wedge Exy\ )\ \}$. To turn this into a general principle, we say that an inclusion dependency δ of the form $R[i_1,\dots,i_m] \subseteq S[j_1,\dots,j_m]$ *can be applied* to a CQ q if q contains an atom ψ_1 of the form $Rv_1\cdots v_r$ and an atom ψ_2 of the form $Sw_1\cdots w_s$ such that

1. $(v_{i_1},\dots,v_{i_m}) = (w_{j_1},\dots,w_{j_m})$,
2. for all $j \in [s] \setminus \{j_1,\dots,j_m\}$ we have $w_j \in \mathbf{var}$, $w_j \notin \mathrm{free}(q)$, $\mathrm{atoms}(w_j) = \{\psi_2\}$, and
3. for all $j,j' \in [s] \setminus \{j_1,\dots,j_m\}$ with $j \neq j'$ we have $w_j \neq w_{j'}$;

and *applying* δ *to* q *at* (ψ_1,ψ_2) then yields the CQ q' which is obtained from q by omitting the atom ψ_2 and omitting the quantifiers $\exists z$ for all $z \in \mathrm{vars}(\psi_2) \setminus \{w_{j_1},\dots,w_{j_m}\}$. By this construction we have $\mathrm{vars}(q') = \mathrm{vars}(q) \setminus \{w_j \ :\ j \in [s] \setminus \{j_1,\dots,j_m\}\}$.

▶ **Claim 5.4.** *$q' \equiv_{\{\delta\}} q$, and if q is q-hierarchical, then so is q'.*

See [6] for a proof. From the claim it follows that we can simplify a query by iteratively applying inclusion dependencies to pairs of atoms of the query. In some cases, this transforms queries that are hard in general into Σ-equivalent queries that are q-hierarchical and hence easy for dynamic evaluation. E.g., an iterated application of $\delta_{ind} := E[2] \subseteq E[1]$ transforms the non-t-hierarchical query $\{\,(x,y) : \exists z_1 \exists z_2\ (\,Exy \wedge Eyz_1 \wedge Ez_1z_2\,)\,\}$ into the q-hierarchical query $\{\,(x,y) : Exy\,\}$. However, the limitations of this approach are documented by the query $q := \{\,() : \exists x\exists y\exists z\exists z'\,(\,Sx \wedge Exy \wedge Ty \wedge Rzz'\,)\,\}$, which is Σ-equivalent to the q-hierarchical query $q' := \{\,() : \exists z\exists z'\ Rzz'\,\}$, for $\Sigma := \{\,R[1,2] \subseteq E[1,2]\ ,\ R[1] \subseteq S[1]\ ,\ R[2] \subseteq T[1]\,\}$, but where q' cannot be obtained by iteratively applying dependencies of Σ to q.

Also, the presence of *functional dependencies* can cause a hard query to become tractable: Consider the functional dependency $\delta_{fd} := E[1 \to 2]$, which is satisfied by a database D iff for every $a \in \mathbf{dom}$ there is at most one $b \in \mathbf{dom}$ such that $(a,b) \in E^D$. On databases that satisfy δ_{fd}, the query $q_{S\text{-}E\text{-}T}$ can be evaluated with constant answer time and constant update time as follows: One can store for every b the number m_b of elements $(a,b) \in E^D$ such that $a \in S^D$, and, in addition, the number $m = \sum_{b \in T^D} m_b$, which is non-zero if and only if $q_{S\text{-}E\text{-}T}(D) = \texttt{yes}$. The functional dependency guarantees that every update affects at

most one number m_b and one summand of m. Using constant access data structures, the query result can therefore be maintained with constant update time.

The nature of this example is somewhat different compared to the approaches for small domain constraints or inclusion constraints described above: We can show that the query becomes tractable, but we are not aware of any $\{\delta_{fd}\}$-equivalent q-hierarchical CQ or UCQ that would explain its tractability via a reduction to the setting without integrity constraints. To exploit the full power of functional dependencies for improving dynamic query evaluation, it seems therefore necessary to come up with new algorithmic approaches that go beyond the techniques we have for (q- or t-)hierarchical queries.

References

1 Amir Abboud, Richard Ryan Williams, and Huacheng Yu. More applications of the polynomial method to algorithm design. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 218–230. SIAM, 2015. doi:10.1137/1.9781611973730.17.

2 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. URL: http://webdam.inria.fr/Alice/.

3 Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2007. doi:10.1007/978-3-540-74915-8_18.

4 Pablo Barceló, Georg Gottlob, and Andreas Pieris. Semantic acyclicity under constraints. In Tova Milo and Wang-Chiew Tan, editors, *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 343–354. ACM, 2016. doi:10.1145/2902251.2902302.

5 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 303–318. ACM, 2017. doi:10.1145/3034786.3034789.

6 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering ucqs under updates and in the presence of integrity constraints. *CoRR*, abs/1709.10039, 2017. arXiv:1709.10039.

7 Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison, editors, *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, pages 77–90. ACM, 1977. doi:10.1145/800105.803397.

8 Hubie Chen and Stefan Mengel. A trichotomy in the complexity of counting answers to conjunctive queries. In Marcelo Arenas and Martín Ugarte, editors, *18th International Conference on Database Theory, ICDT 2015, March 23-27, 2015, Brussels, Belgium*, volume 31 of *LIPIcs*, pages 110–126. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/LIPIcs.ICDT.2015.110.

9 Hubie Chen and Stefan Mengel. Counting answers to existential positive queries: A complexity classification. In Tova Milo and Wang-Chiew Tan, editors, *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 315–326. ACM, 2016. doi:10.1145/2902251.2902279.

10 Rada Chirkova and Jun Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012. doi:10.1561/1900000020.

11 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms.

12 Víctor Dalmau and Peter Jonsson. The complexity of counting homomorphisms seen from the other side. *Theor. Comput. Sci.*, 329(1-3):315–323, 2004. doi:10.1016/j.tcs.2004.08.008.

13 Arnaud Durand and Stefan Mengel. Structural tractability of counting of solutions to conjunctive queries. *Theory Comput. Syst.*, 57(4):1202–1249, 2015. doi:10.1007/s00224-014-9543-y.

14 Georg Gottlob, Stephanie Tien Lee, Gregory Valiant, and Paul Valiant. Size and treewidth bounds for conjunctive queries. *J. ACM*, 59(3):16:1–16:35, 2012. doi:10.1145/2220357.2220363.

15 Gianluigi Greco and Francesco Scarcello. Counting solutions to conjunctive queries: structural and hybrid tractability. In Richard Hull and Martin Grohe, editors, *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, pages 132–143. ACM, 2014. doi:10.1145/2594538.2594559.

16 Martin Grohe. The complexity of homomorphism and constraint satisfaction problems seen from the other side. *J. ACM*, 54(1):1:1–1:24, 2007. doi:10.1145/1206035.1206036.

17 Martin Grohe, Thomas Schwentick, and Luc Segoufin. When is the evaluation of conjunctive queries tractable? In Jeffrey Scott Vitter, Paul G. Spirakis, and Mihalis Yannakakis, editors, *Proceedings on 33rd Annual ACM Symposium on Theory of Computing, July 6-8, 2001, Heraklion, Crete, Greece*, pages 657–666. ACM, 2001. doi:10.1145/380752.380867.

18 Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, SIGMOD'93, Washington, D.C., USA, May 25–28, 1993*, pages 157–166. ACM, 1993. doi:10.1145/170036.170066.

19 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 21–30. ACM, 2015. doi:10.1145/2746539.2746609.

20 Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1259–1274. ACM, 2017. doi:10.1145/3035918.3064027.

21 Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. Computing join queries with functional dependencies. In Tova Milo and Wang-Chiew Tan, editors, *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 327–342. ACM, 2016. doi:10.1145/2902251.2902289.

22 Christoph Koch. Incremental query evaluation in a ring of databases. In Jan Paredaens and Dirk Van Gucht, editors, *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA*, pages 87–98. ACM, 2010. doi:10.1145/1807085.1807100.

23 Christoph Koch, Daniel Lupei, and Val Tannen. Incremental view maintenance for collection programming. In Tova Milo and Wang-Chiew Tan, editors, *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 75–90. ACM, 2016. doi:10.1145/2902251.2902286.

24 Dániel Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *J. ACM*, 60(6):42:1–42:51, 2013. doi:10.1145/2535926.

25 Bernard M. E. Moret and Henry D. Shapiro. *Algorithms from P to NP: Volume 1: Design & Efficiency*. Benjamin-Cummings, 1991.

26 Milos Nikolic, Mohammad Dashti, and Christoph Koch. How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 511–526. ACM, 2016. doi:10.1145/2882903.2915246.

27 Sushant Patnaik and Neil Immerman. Dyn-fo: A parallel, dynamic complexity class. *J. Comput. Syst. Sci.*, 55(2):199–209, 1997. doi:10.1006/jcss.1997.1520.

28 Thomas Schwentick and Thomas Zeume. Dynamic complexity: recent updates. *SIGLOG News*, 3(2):30–52, 2016. doi:10.1145/2948896.2948899.

29 Thomas Zeume and Thomas Schwentick. Dynamic conjunctive queries. In Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014.*, pages 38–49. OpenProceedings.org, 2014. doi:10.5441/002/icdt.2014.08.

A Proof of Lemma 3.5

Proof. Along Definition 3.3 it is straightforward to construct an algorithm which decides in time $poly(q)$ whether a given CQ q is t-hierarchical.

Let $q := q_\varphi(z_1, \ldots, z_k)$ be a given t-hierarchical CQ. Let A_0 be the set of all atoms ψ of q with $\text{vars}(\psi) \subseteq \text{free}(q)$, and let φ_0 be the quantifier-free conjunctive formula $\varphi_0 := \bigwedge_{\psi \in A_0} \psi$. For each $Z \subseteq \text{free}(q)$ let A_Z be the set of all atoms ψ of q such that $Z = \text{vars}(\psi) \cap \text{free}(q)$ and $\text{vars}(\psi) \not\supseteq Z$. Let $Z_1, \ldots, Z_n$ (for $n \geqslant 0$) be a list of all those $Z \subseteq \text{free}(q)$ with $A_Z \neq \emptyset$. For each $j \in [n]$ let $A_j := A_{Z_j}$ and let $Y_j := \big(\bigcup_{\psi \in A_j} \text{vars}(\psi)\big) \setminus Z_j$.

▶ **Claim A.1.** *$Y_j \cap Y_{j'} = \emptyset$ for all $j, j' \in [n]$ with $j \neq j'$.*

Proof. We know that $Z_j \neq Z_{j'}$. W.l.o.g. there is a $z \in Z_j$ with $z \notin Z_{j'}$.

For contradiction, assume that $Y_j \cap Y_{j'}$ contains some variable y. Then, $y \in \text{vars}(\psi)$ for some $\psi \in A_j$ and $y \in \text{vars}(\psi')$ for some $\psi' \in A_{j'}$. By definition of A_j we know that $\text{vars}(\psi) \cap \text{free}(q) = Z_j$, and hence $z \in \text{vars}(\psi)$. By definition of $A_{j'}$ we know that $\text{vars}(\psi') \cap \text{free}(q) = Z_{j'}$, and hence $z \notin \text{vars}(\psi')$. Hence, $\psi \in \text{atoms}(z)$ and $\psi' \notin \text{atoms}(z)$. Since $\psi \in \text{atoms}(y)$ and $\psi' \in \text{atoms}(y)$, we obtain that $\text{atoms}(z) \cap \text{atoms}(y) \neq \emptyset$ and $\text{atoms}(y) \not\subseteq \text{atoms}(z)$. But by assumption, q is t-hierarchical, and this contradicts condition (ii) of Definition 3.3. ◀

For each $j \in [n]$ consider the conjunctive formula $\varphi_j := \exists y_1^{(j)} \cdots \exists y_{\ell_j}^{(j)} \bigwedge_{\psi \in A_j} \psi$, where $\ell_j := |Y_j|$ and $(y_1^{(j)}, \ldots, y_{\ell_j}^{(j)})$ is a list of all variables in Y_j. Using Claim A.1, it is straightforward to see that $q' := \{ (z_1, \ldots, z_k) : \varphi_0 \wedge \bigwedge_{j \in [n]} \varphi_j \}$ is a generalised CQ that is equivalent to q. Furthermore, q' can be constructed in time $poly(q)$. To complete the proof of Lemma 3.5 we consider for each $j \in [n]$ the CQ $q_j := \{ \overline{z}^{(j)} : \varphi_j \}$, where $\overline{z}^{(j)}$ is a tuple of length $|Z_j|$ consisting of all the variables in Z_j.

▶ **Claim A.2.** *q_j is q-hierarchical, for each $j \in [n]$.*

Proof. First of all, note that q_j satisfies condition (ii) of Definition 3.1, since $\text{free}(q_j) = Z_j$, $\text{atoms}_{q_j}(z) = A_j$ for every $z \in Z_j$, and $\text{atoms}_{q_j}(y) \subseteq A_j$ for every $y \in Y_j = \text{vars}(q_j) \setminus \text{free}(q_j)$.

For contradiction, assume that q_j is not q-hierarchical. Then, q_j violates condition (i) of Definition 3.1. I.e., there are variables $x, x' \in Z_j \cup Y_j$ and atoms $\psi_1, \psi_2, \psi_3 \in A_j$ such that $\text{vars}(\psi_1) \cap \{x, x'\} = \{x\}$, $\text{vars}(\psi_2) \cap \{x, x'\} = \{x'\}$, and $\text{vars}(\psi_3) \cap \{x, x'\} = \{x, x'\}$. Since $\text{vars}(\psi) \cap \text{free}(q) = Z_j$ for all $\psi \in A_j$, we know that $x, x' \notin \text{free}(q)$. Therefore, $x, x' \in \text{vars}(q) \setminus \text{free}(q)$, and hence ψ_1, ψ_2, ψ_3 are atoms of q which witness that condition (i) of Definition 3.3 is violated. This contradicts the assumption that q is t-hierarchical. ◀

This completes the proof of Lemma 3.5. ◀

B Illustration of the effect of the EXCLUDE^j-Procedure used in the proof of Lemma 4.3

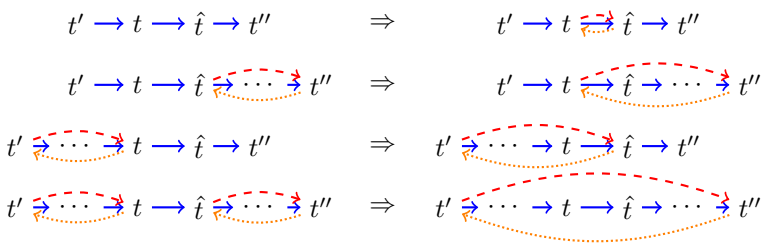


Figure 1 Modifications of the data structure caused by $\text{EXCLUDE}^j(t)$. In this illustration, a blue arrow from t to $\hat{t}$ means that $\hat{t} = \mathsf{next}^j(t)$; a dashed red arrow from t to $\hat{t}$ means that $\hat{t} = \mathsf{skip}^j[t]$, and a dotted red arrow from $\hat{t}$ to t means that $t = \mathsf{skipback}^j[\hat{t}]$.

Expressivity and Complexity of MongoDB Queries

Elena Botoeva
Faculty of Computer Science, Free University of Bozen-Bolzano, Italy
botoeva@inf.unibz.it

Diego Calvanese
Faculty of Computer Science, Free University of Bozen-Bolzano, Italy
calvanese@inf.unibz.it

Benjamin Cogrel
Faculty of Computer Science, Free University of Bozen-Bolzano, Italy
cogrel@inf.unibz.it

Guohui Xiao[1]
Faculty of Computer Science, Free University of Bozen-Bolzano, Italy
xiao@inf.unibz.it

Abstract

In this paper, we consider MongoDB, a widely adopted but not formally understood database system managing JSON documents and equipped with a powerful query mechanism, called the aggregation framework. We provide a clean formal abstraction of this query language, which we call MQuery. We study the expressivity of MQuery, showing the equivalence of its well-typed fragment with nested relational algebra. We further investigate the computational complexity of significant fragments of it, obtaining several (tight) bounds in combined complexity, which range from LogSpace to alternating exponential-time with a polynomial number of alternations.

2012 ACM Subject Classification Information systems → Semi-structured data, Theory of computation → Data modeling, Theory of computation → Database query languages (principles)

Keywords and phrases MongoDB, NoSQL, aggregation framework, expressivity

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.9

Related Version A full version of this paper with more details and selected proofs is available as a technical report [3].

Acknowledgements We thank Christoph Koch, Dan Suciu, Henrik Ingo, and Martin Rezk for helpful discussions. This research has been partially supported by the project "Ontology-based Data Access for NoSQL Databases" (OBDAM), funded through the 2016 call issued by the Research Committee of the Free University of Bozen-Bolzano.

1 Introduction

JavaScript Object Notation (JSON) is currently adopted extensively as the de-facto standard format for representing nested data. JSON organizes data as semi-structured tree-shaped documents, with a minimalistic set of node types, and as such is commonly considered a lightweight alternative to XML. JSON documents can also be seen as complex values [11, 1, 9, 7], in particular due to the presence of nested arrays. Consider, e.g., the document

[1] Corresponding author

Listing 1 A sample JSON document in the `bios` collection.

```
{ "_id": 4,
  "awards": [
    { "award": "Rosing Prize", "year": 1999, "by": "Norwegian Data Association" },
    { "award": "Turing Award", "year": 2001, "by": "ACM" },
    { "award": "IEEE John von Neumann Medal", "year": 2001, "by": "IEEE" } ],
  "birth": "1926-08-27",
  "contribs": [ "OOP", "Simula" ],
  "death": "2002-08-10",
  "name": { "first": "Kristen", "last": "Nygaard" } }
```

in Listing 1, containing personal information (such as name and birth-date) about Kristen Nygaard, and information about the awards he received, the latter stored inside an array.

Following its massive adoption by practitioners, recently JSON has also received attention in the database theory community. A powerful (Turing-complete, in its full generality) Datalog-like query language for JSON named JLogic is introduced in [12], where the expressive power and complexity of the full language and of significant fragments are studied. In [4], both JSON and its main schema language JSON Schema[2] are formalized, and their expressive power and the computational complexity of basic computational tasks, such as satisfiability and evaluation of expressions, are studied. Although some of the latter results apply to the simple *find* query language[3] of the widespread JSON-based document database system MongoDB, still little is known about the precise formal properties of the query languages for JSON with rich capabilities popular among practitioners, such as JSONiq [10] and SQL++ [16].

Differently from XML, where XQuery is the official standard query language, embraced also by the developer community, so far there is no standard query language for JSON. However, in terms of adoption, the *MongoDB aggregation framework*[4] is currently the most prominent language providing rich querying capabilities over collections of JSON documents, and hence has become the de-facto standard language for JSON. This language is modeled on the flexible notion of a data processing pipeline, where a query consists of multiple stages, each defining a transformation using a specific operator, applied to the set of documents produced by the previous stage. As such, the language is very expressive and rich in features, but it has been developed in an ad-hoc manner, resulting in some counter-intuitive behavior.

Here, we propose a first study on the formal foundations and computational properties of the MongoDB aggregation framework. Since JSON documents can be seen as complex values and are closely related to XML documents, we expect the aggregation framework to have many similarities with well-known query languages for complex values, such as monad algebra [5, 15], nested relational algebra (NRA) [19, 8] and Core XQuery [15].

Our first contribution is a formalization of the JSON data model and of the aggregation framework query language. We aim at achieving a good balance between the contrasting requirements of capturing all aspects of MongoDB, and of keeping the formalization sufficiently simple and streamlined so as to allow for a formal study of the language properties. To do so, we deliberately abstract away some low-level features of MongoDB, which appear to be motivated by implementation aspects and possibly by ad-hoc choices, and we make some simplifying assumptions, commonly considered in database theory. Specifically, we adopt set semantics (as opposed to bag or list semantics), and we abstract away from order within

[2] `http://json-schema.org/`
[3] `https://docs.mongodb.com/manual/crud/`
[4] `https://docs.mongodb.com/manual/core/aggregation-pipeline/`

documents. Our formal language, which we call *MQuery*, includes the ***m**atch*, ***u**nwind*, ***p**roject*, ***g**roup*, and ***l**ookup* operators, roughly corresponding to the NRA operators select, unnest, project, nest, and left join, respectively. In our investigation, we consider various fragments of MQuery, which we denote by $\mathcal{M}^{\alpha}$, where α consists of the initials of the stages allowed in the fragment. As a useful side-effect of our formalization effort, we point out different "features" exhibited by MongoDB's query language that are somewhat counter-intuitive, and that might need to be reconsidered by the MongoDB developers.

Our second contribution is a characterization of the expressive power of MQuery obtained by comparing it with NRA. Given the regular structure of nested relations, the comparison requires considering JSON documents that are suitably *well-typed*, for which we define a relational view, and restricting the attention to *well-typed* MQuery, given that an arbitrary MQuery might produce non well-typed documents. We devise translations in both directions between well-typed MQuery and NRA, showing that the two languages are equivalent in expressive power. We also consider the $\mathcal{M}^{\text{MUPG}}$ fragment, where we rule out the lookup operator, which allows for joining a given document collection with external ones. Actually, we establish that already $\mathcal{M}^{\text{MUPG}}$ is equivalent to NRA over a single relation, and hence is capable of expressing arbitrary joins (within one collection), contrary to what is believed in the community of MongoDB practitioners. Interestingly, all our translations are compact (i.e., polynomial), hence complexity results between MQuery and NRA carry over.

Finally, we carry out an investigation of the computational complexity of $\mathcal{M}^{\text{MUPGL}}$ and its fragments. In particular, we establish that what we consider the minimal fragment, which allows only for match, is LogSpace-complete (in combined complexity). Projection and grouping allow one to create exponentially large objects, but by representing intermediate results compactly as DAGs, one can still evaluate $\mathcal{M}^{\text{MPGL}}$ queries in PTime. The use of unwind alone causes loss of tractability in combined complexity, specifically it leads to NP-completeness, but remains LogSpace-complete in query complexity. Adding also project or lookup leads again to intractability even in query complexity, although $\mathcal{M}^{\text{MUPL}}$ stays NP-complete in combined complexity. In the presence of unwind, grouping provides another source of complexity, since it allows one to create doubly-exponentially large objects; indeed we show PSpace-hardness of $\mathcal{M}^{\text{MUG}}$. Finally, we establish that the full language and also the $\mathcal{M}^{\text{MUPG}}$ fragment are TA$[2^{n^{O(1)}}, n^{O(1)}]$-complete (i.e., complete for exponential time with a polynomial number of alternations under LogSpace reductions) in combined complexity. As a byproduct of this study, we also establish that the TA$[2^{n^{O(1)}}, n^{O(1)}]$ lower bound previously known for the combined complexity of Boolean query evaluation in NRA is actually tight (the best known upper bound was ExpSpace [15]).

2 Preliminaries

We recap the basics of nested relational algebra (NRA) [13, 8], mainly to fix the notation.

Let $\mathcal{A}$ be a countably infinite set of attribute names and relation schema names. A *relation schema* has the form $R(S)$, where $R \in \mathcal{A}$ is a relation schema name and S is a finite set of attributes, each of which is an atomic attribute (i.e., an attribute name in $\mathcal{A}$) or a schema of a sub-relation. A relation schema can also be obtained through an NRA operation (see below). We use the function att to retrieve the attributes from a relation schema name, i.e., $\mathsf{att}(R) = S$. Let Δ be the domain of all atomic attributes in $\mathcal{A}$. An *instance* $\mathcal{R}$ of a relation schema $R(S)$ is a finite set of tuples over $R(S)$. A *tuple* t *over* $R(S)$ is a finite set $\{a_1{:}v_1, \ldots, a_n{:}v_n\}$ such that if a_i is an atomic attribute, then $v_i \in \Delta$, and if a_i is a relation schema, then v_i is an instance of a_i. We may refer to relation schemas by their name only.

$$e ::= a \mid c \mid f \mid (f?\, e{:}\, e) \mid \mathsf{subrel}(t, \ldots, t) \qquad t ::= \{b{:}e, \ldots, b{:}e\}$$
$$f ::= \mathbf{true} \mid a = a \mid a = c \mid \neg f \mid f \wedge f \mid f \vee f$$

Figure 1 Syntax of expressions e used in extended projection. Here, $a \in \mathsf{att}(R)$, c is a constant, f a Boolean expression, b a fresh attribute name, t a tuple definition, and $\mathsf{subrel}(t_1, \ldots, t_n)$ a relation definition, which constructs a relation from the tuples $t_1, \ldots, t_n$ of the same schema.

VALUE ::= LITERAL | OBJECT | ARRAY
OBJECT ::= {{ LIST<KEY : VALUE> }}
ARRAY ::= [LIST<VALUE>]

LIST<T> ::= ε | LIST$^+$<T>
LIST$^+$<T> ::= T | T , LIST$^+$<T>

Figure 2 Syntax of JSON objects. We use double curly brackets to distinguish objects from sets.

A *filter* ψ over a set $A \subseteq \mathcal{A}$ is a Boolean formula constructed from atoms of the form $(a = v)$ or $(a = a')$, where $\{a, a'\} \subseteq A$, and v is an atomic value or a relation. Let R and R' be relation schemas. We use the following operators:

(1) *set union* $R \cup R'$ and *set difference* $R \setminus R'$, for $\mathsf{att}(R) = \mathsf{att}(R')$;
(2) *cross-product* $R \times R'$, resulting in a relation schema with attributes $\{\text{rel1}.a \mid a \in \mathsf{att}(R)\} \cup \{\text{rel2}.a \mid a \in \mathsf{att}(R')\}$;
(3) *selection* $\sigma_\psi(R)$, where ψ is a filter over $\mathsf{att}(R)$;
(4) *projection* $\pi_P(R)$, for $P \subseteq \mathsf{att}(R)$;
(5) *extended projection* $\pi_P(R)$, where P may also contain elements of the form b/e, for b a fresh attribute name, and e an expression constructed according to the grammar shown in Figure 1. Notice that such an expression is computable in AC^0 in data complexity;
(6) *nest* $\nu_{\{a_1,\ldots,a_n\}\to b}(R)$, resulting in a schema with attributes $(\mathsf{att}(R) \setminus \{a_1, \ldots, a_n\}) \cup \{b(a_1, \ldots, a_n)\}$; and
(7) *unnest* $\chi_a(R)$, resulting in a schema with attributes $(\mathsf{att}(R) \setminus \{a\}) \cup \mathsf{att}(a)$.

For more details on (5)–(7) (including the semantics of extended projection), we refer to [3]. Given an NRA query Q and a (relational) database $\mathcal{D}$, the result of evaluating Q over $\mathcal{D}$ is denoted by $ans_{\mathsf{ra}}(Q, \mathcal{D})$.

3 JSON Documents

In this section, we propose a formalization of the syntax and the semantics of JSON documents. With respect to MongoDB, we abstract away the order of key-value pairs within a document.

A MongoDB database stores collections of documents, where a collection corresponds to a table in a (nested) relational database, and a document to a row in a table. We define the syntax of documents. *Literals* are atomic values, such as strings, numbers, and Booleans. A *JSON object* is a finite set of key-value pairs, where a *key* is a string and a *value* can be a literal, an object, or an array of values, constructed inductively according to the grammar in Figure 2 (where the terminals are '{{', '}}', '[', ']', ':', and ','). We require that the set of key-value pairs constituting a JSON object does not contain the same key twice. A *(MongoDB) document* is a JSON object not nested within any other object, with a special key '`_id`', used to identify the document. Listing 1 shows a document with keys `_id`, `awards`, `birth`, etc. Given a collection name C, a *(MongoDB) collection for* C is a finite set F_C of documents, each identified by its value of `_id`, i.e., each value of `_id` is unique in F_C. Given a set $\mathbb{C}$ of collection names, a *MongoDB database instance* D *(over* $\mathbb{C}$*)* is a set of collections, one for each name $C \in \mathbb{C}$. We write $D.C$ to denote the collection for name C.

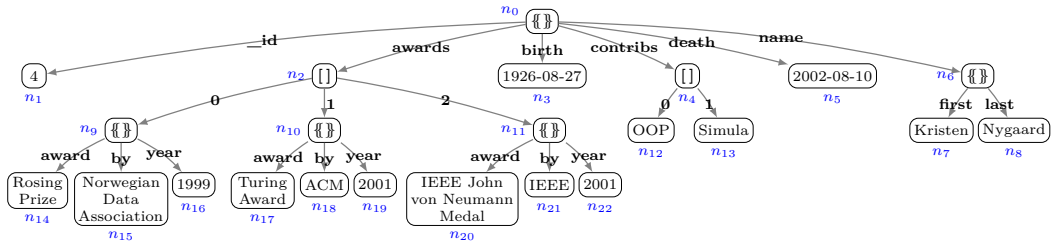


Figure 3 The tree t_{KN} corresponding to the JSON document in Listing 1.

We formalize JSON objects as finite *unordered, unranked, node-labeled, and edge-labeled trees* (see Figure 3 for the tree t_{KN} corresponding to the document in Listing 1, where we have additionally labeled nodes with n_i, to refer to them later). We assume three disjoint sets of labels: the sets K of *keys* and I of *indexes* (non-negative integers), used as edge-labels, and the set V of *literals*, containing the special elements **null**, **true**, and **false**, and used as node labels. A *tree* is a tuple $(N, E, L_{\mathsf{n}}, L_{\mathsf{e}})$, where N is a set of nodes, E is the edge relation, $L_{\mathsf{n}} : N \to V \cup \{\text{`}\{\!\{\}\!\}\text{'}, \text{`}[]\text{'}\}$ is a node labeling function, and $L_{\mathsf{e}} : E \to K \cup I$ is an edge labeling function, such that

- **(i)** (N, E) forms a tree,
- **(ii)** a node labeled by a literal must be a leaf,
- **(iii)** all outgoing edges of a node labeled by '$\{\!\{\}\!\}$' must be labeled by keys, and
- **(iv)** all outgoing edges of a node labeled by '$[]$' must be labeled by distinct indexes.

The *type* of a node x in a tree t, denoted $\mathsf{type}(x, t)$, is defined as $\mathsf{literal}$ if $L_{\mathsf{n}}(x) \in V$, object if $L_{\mathsf{n}}(x) = \text{`}\{\!\{\}\!\}\text{'}$, and $\mathsf{array}$ if $L_{\mathsf{n}}(x) = \text{`}[]\text{'}$. $\mathsf{root}(t)$ denotes the root of t. A *forest* is a set of trees.

We define inductively the *value represented by* a node x in a tree t, denoted $\mathsf{value}(x, t)$:

- **(i)** $\mathsf{value}(x, t) = L_{\mathsf{n}}(x)$, if x is a leaf in t;
- **(ii)** let $x_1, \ldots, x_m$, be all children of x with $L_{\mathsf{e}}(x, x_i) = k_i$. Then $\mathsf{value}(x, t)$ is $\{\!\{k_1{:}\mathsf{value}(x_1, t), \ldots, k_m{:}\mathsf{value}(x_m, t)\}\!\}$ if $\mathsf{type}(x, t) = \mathsf{object}$, and $[\mathsf{value}(x_1, t), \ldots, \mathsf{value}(x_m, t)]$, if $\mathsf{type}(x, t) = \mathsf{array}$.

The *JSON value represented by* t is then $\mathsf{value}(\mathsf{root}(t), t)$. Conversely, the *tree corresponding to a value* u, denoted $\mathsf{tree}(u)$, is defined as $(N, E, L_{\mathsf{n}}, L_{\mathsf{e}})$, where N is the set of all x_v such that v is an object, array, or literal value appearing in u, and for $x_v \in N$:

- **(i)** if v is a literal, then $L_{\mathsf{n}}(x_v) = v$ and x_v is a leaf;
- **(ii)** if $v = \{\!\{k_1{:}v_1, \ldots, k_m{:}v_m\}\!\}$ for $m \geq 0$, then $L_{\mathsf{n}}(x_v) = \text{`}\{\!\{\}\!\}\text{'}$, and x_v has m children $x_{v_1}, \ldots, x_{v_m}$ with $L_{\mathsf{e}}(x_v, x_{v_i}) = k_i$;
- **(iii)** if $v = [v_1, \ldots, v_m]$ for $m \geq 0$, then $L_{\mathsf{n}}(x_v) = \text{`}[]\text{'}$, and x_v has m children $x_{v_1}, \ldots, x_{v_m}$ with $L_{\mathsf{e}}(x_v, x_{v_i}) = i - 1$.

In the following, when convenient, we blur the distinction between JSON values and the corresponding trees.

4 The MQuery Language

MongoDB is equipped with an expressive query mechanism provided by the *aggregation framework* (we refer to [3] for its formal syntax, but we provide in App. A some examples to illustrate its main features). Our first contribution is a formalization of the core aspects of this query language, where we use set (as opposed to bag and list) semantics, and we deliberately abstract away some low-level features that either are not relevant for understanding the

$$\varphi ::= \textbf{true} \mid p = v \mid \exists p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \qquad P ::= p \mid p/d \mid p, P \mid p/d, P$$
$$d ::= v \mid p \mid [d, \ldots, d] \mid \beta \mid (\beta ?\, d : d) \qquad G ::= p/p \mid p/p, G$$
$$\beta ::= \textbf{true} \mid p = p \mid p = v \mid \exists p \mid \neg\beta \mid \beta \vee \beta \mid \beta \wedge \beta \qquad A ::= p/p \mid p/p, A$$
$$s ::= \mu_\varphi \mid \omega_p \mid \rho_P \mid \gamma_{G:A} \mid \lambda_p^{p=C.p} \qquad \text{MQuery} ::= C \triangleright s \triangleright \cdots \triangleright s$$

Figure 4 The MQuery language. Here, p denotes a path, v a value, C a collection name, φ a criterion, d a value definition, β a Boolean value definition, s a stage, P a list for project, G a list for grouping, and A a list for aggregation.

expressive power and computational properties of the language, or appear ad-hoc and possibly are remnants of experimental development. We call the resulting language *MQuery*.

An *MQuery* is a sequence of stages, also called a *pipeline*, applied to a collection name C, where each stage transforms a forest into another forest. The grammar of MQuery is given in Figure 4. In an MQuery, *paths*, which are (possibly empty) concatenations of keys, are used to access actual values in a tree, similarly to how attributes are used in relational algebra. We use ε to denote the empty path. For two paths p and p', we say that p' is a *(strict) prefix* of p, if $p = p'.p''$, for some (non-empty) path p''. MQuery allows for five types of *stages*[5]:

- ***m**atch* μ_φ, which selects trees according to criterion φ. Such criterion is a Boolean combination of atomic conditions $p = v$, expressing the equality of a path p to a value v, or $\exists p$, expressing the existence of a path p. E.g., for $\varphi_1 =$ (`_id`=4), $\varphi_2 =$ (awards.award="Turing Award"), and $\varphi_3 =$ (name = {{first: "Kristen"}}), μ_{φ_1} and μ_{φ_2} select t_{KN}, but μ_{φ_3} does not. (See App. A.1 for details.)
- ***u**nwind* ω_p, which flattens an array reached through a path p in the input tree, and outputs a tree for each element of the array. E.g., ω_{awards} applied to t_{KN} produces three trees, which coincide on all key-value pairs, except for the `awards` key, whose values are nested objects such as, e.g., {{award: "Turing Award", year: 2001, by: "ACM"}}. (See App. A.2.)
- ***p**roject* ρ_P, which modifies trees by projecting away paths, renaming paths, or introducing new paths. Here P is a sequence of elements of the form p or q/d, where p is a path to be kept, q is a new path whose value is defined by d, and among all such paths p and q, there is no pair p, p' where p is a prefix of p'. A *value definition* d can provide for q:
  - **(i)** a constant v,
  - **(ii)** the value reached through a path p (i.e., *renaming* path p to q),
  - **(iii)** a new array defined through its values,
  - **(iv)** the value of a Boolean expression β, or
  - **(v)** a value computed through a conditional expression $(\beta ?\, d_1 : d_2)$.

  E.g., $\rho_{\text{bool}/(\text{birth=death}),\, \text{cond}/((\exists\text{awards})?\, \text{contribs}:\, \texttt{\_id}),\, \text{newArray}/[0,1]}$ applied to t_{KN} produces {{bool: **false**, cond: ["OOP", "Simula"], newArray: [0,1]}}. (See App. A.3.)
- ***g**roup* $\gamma_{G:A}$, which groups trees according to a grouping condition G and aggregates values of interest according to A. Both G and A are (possibly empty) sequences of elements of the form p/p', where p' is a path in the input trees, and p a path in the output trees. Each different combination $\vec{v}$ of values in the input trees for the p's in G determines a group. For each such group there is a tree in the output with an `_id` whose value is constructed from $\vec{v}$ and the ps in G. The remaining keys in each output tree have as value an array constructed using the aggregation expression A. Consider, e.g.,

[5] We suggest readers unfamiliar with MongoDB to read the following paragraphs in parallel to the respective subsections in App. A, which contain additional examples and the actual syntax of MongoDB.

as input {{a: 1, b: "x"}}, {{a: 1, b: "y"}}, and {{a: 2, b: "z"}}. Then $\gamma_{\mathrm{c/a\,:\,bs/b}}$ produces the two groups {{_id: {{c: 1}}, bs: ["x","y"]}} and {{_id: {{c: 2}}, bs: ["z"]}}. (See App. A.4.)

- *lookup* $\lambda_p^{p_1=C.p_2}$, which joins input trees with trees in an external collection C, using a local path p_1 and a path p_2 in C to express the join condition, and stores the matching trees in an array under a path p. E.g., let C consist of {{_id: 1, a: 3}} and {{_id: 2, a: 4}}. Then $\lambda_{\mathrm{docs}}^{\mathtt{\_id}=C.\mathrm{a}}$ evaluated over t_{KN} adds to it docs: [{{_id: 2, a: 4}}]. (See App. A.5.)

Observe that the Boolean expressions β allowed in a project stage are more expressive than those in the criterion φ of a match stage, since in the former one can also compare the values of two paths, while in the latter one can only compare the value of a path to a constant value. We consider also various fragments $\mathcal{M}^{\alpha}$ of MQuery, where α consists of the initials of the allowed stages. E.g., $\mathcal{M}^{\mathrm{MUPGL}}$ denotes MQuery itself, while $\mathcal{M}^{\mathrm{MUPG}}$ disallows lookup.

To define the semantics of MQuery, we introduce some auxiliary notions.

First, we show how to interpret paths over trees. Specifically, a path p is interpreted as the set of nodes reachable via p from the root, where the indexes of intermediate arrays that might be encountered in the tree are skipped. Given a tree $t = (N, E, L_{\mathsf{n}}, L_{\mathsf{e}})$, we interpret a (possibly empty) path p, and its concatenation $p.i_1 \dots i_m$ with indexes $i_1, \dots, i_m$, respectively as the sets of nodes $[\![p]\!]^t$ and $[\![p.i_1 \dots i_m]\!]^t$, according to the following inductive definition (below, q is a path, $j_1, \dots, j_n$ are indexes, and k is a key): $[\![\varepsilon]\!]^t = \{\mathsf{root}(t)\}$, and

$$[\![q.j_1 \dots j_n]\!]^t = \{y \in N \mid \text{there is } x \in [\![q.j_1 \dots j_{n-1}]\!]^t \text{ s.t. } (x,y) \in E \text{ and } L_{\mathsf{e}}(x,y) = j_n\}$$
$$[\![q.k]\!]^t = \{y \in N \mid \text{there are } j_1, \dots, j_n,\ n \geq 0, \text{ and } x \in [\![q.j_1 \dots j_n]\!]^t \text{ s.t. } (x,y) \in E \text{ and } L_{\mathsf{e}}(x,y) = k\}$$

For example, referring to the tree t_{KN} in Figure 3, $[\![\varepsilon]\!]^{t_{\mathrm{KN}}} = \{n_0\}$, $[\![\mathtt{\_id}]\!]^{t_{\mathrm{KN}}} = \{n_1\}$, $[\![\mathrm{awards}]\!]^{t_{\mathrm{KN}}} = \{n_2\}$, $[\![\mathrm{awards.1}]\!]^{t_{\mathrm{KN}}} = \{n_{10}\}$, and $[\![\mathrm{awards.award}]\!]^{t_{\mathrm{KN}}} = \{n_{14}, n_{17}, n_{20}\}$. When $[\![p]\!]^t = \emptyset$, we say that the path p is *missing* in t.

Given a tree t and a path p, when $\mathsf{type}(x,t) = \mathsf{ty}$, for each $x \in [\![p]\!]^t$, where $\mathsf{ty} \in \{\mathsf{array}, \mathsf{literal}, \mathsf{object}\}$, we define the *type of* p *in* t, denoted $\mathsf{type}(p,t)$, to be ty. Also, when $\mathsf{type}(p,t) = \mathsf{array}$ and $\mathsf{type}(x,t) = \mathsf{ty}$ for each $x \in [\![p.i]\!]^t$ for $i \in I$, we write $\mathsf{type}(p[\,],t) = \mathsf{ty}$.

Second, we define when a tree t satisfies a criterion or a Boolean value definition φ, denoted $t \models \varphi$, as follows. It always holds that $t \models \mathbf{true}$, while:

$t \models (p = v)$, if there is x in $[\![p]\!]^t$ or in $[\![p.i]\!]^t$ for $i \in I$ s.t. $\mathsf{value}(x,t) = v$ holds
$t \models (\exists p)$, if $[\![p]\!]^t \neq \emptyset$
$t \models \neg\varphi$, if $t \not\models \varphi$
$t \models (p_1 = p_2)$, if there is a value v s.t. $t \models (p_1 = v) \wedge (p_2 = v)$, or $t \models \neg(\exists p_1) \wedge \neg(\exists p_2)$
$t \models \varphi_1 \wedge \varphi_2$, if $t \models \varphi_1$ and $t \models \varphi_2$
$t \models \varphi_1 \vee \varphi_2$, if $t \models \varphi_1$ or $t \models \varphi_2$

In this definition, we employ the classical semantics for "deep" equality of non-literal values, where we ignore duplicates and order in arrays, in line with set semantics. We also assume that $(v = \mathbf{null})$ holds iff v is **null**. Note that, the equality $(p = v)$ may hold both when v is the array reached by p and when v is an element inside this array. E.g., $t_{\mathrm{KN}} \models (\mathrm{contribs} = [\text{"OOP"}, \text{"Simula"}])$ and $t_{\mathrm{KN}} \models (\mathrm{contribs} = \text{"OOP"})$. Also note that the values of several paths inside an array can come from different array elements. E.g., $t_{\mathrm{KN}} \models (\mathrm{awards.award} = \text{"Rosing Prize"}) \wedge (\mathrm{awards.year} = 2001)$.

Next, we define the evaluation of a value definition d in a tree t, denoted by $\mathsf{eval}(d,t)$, as:

d, if $d \in V$; the value of $(t \models d)$, if d is a Boolean value definition;
$\mathsf{subtree}(t,d)$, if d is a path; $[\mathsf{eval}(d_1,t), \dots, \mathsf{eval}(d_m,t)]$, if $d = [d_1, \dots, d_m]$;
$\mathsf{eval}(d',t)$, if $d = (c?\, d_1 : d_2)$, where $d' = d_1$ when $t \models c$ and $d' = d_2$ otherwise.

Finally, we informally introduce some auxiliary operators over trees (for a formal definition, see App. B). Let t, t_1, t_2 be trees, F a forest, p a path, and N a set of nodes. Then:

- $\mathsf{subtree}(t,p)$ returns the subtree of t rooted at the single node in $[\![p]\!]^t$ when $|[\![p]\!]^t| = 1$. Instead, when $|[\![p]\!]^t| > 1$, it returns the array of all subtrees rooted at the nodes in

Table 1 The semantics of MQuery stages.

Match	$F \triangleright \mu_\varphi = \{t \mid t \in F \text{ and } t \models \varphi\}$
Unwind	$\omega_p(t) = \{\mathsf{replace}(t, \mathsf{subtree}(t,p), \mathsf{subtree}(t,p.i))\}_{[\![p.i]\!]^t \neq \emptyset,\, i \in I}$ if p is a first array, and $\omega_p(t) = \emptyset$ otherwise $F \triangleright \omega_p = \bigcup_{t \in F} \omega_p(t)$
Project	$\rho_p(t) =$ $\mathsf{subtree}(t, N_p)$, where N_p are all the nodes in t on a path from $\mathsf{root}(t)$ to a leaf via some $x \in [\![p]\!]^t$ $\rho_{q/d}(t) = \mathsf{attach}(q, \mathsf{eval}(d,t))$, unless d is a path and $t \not\models \exists d$, in which case $\rho_{q/d}(t)$ returns the empty tree $F \triangleright \rho_P = \{ \bigoplus_{p \in P} \rho_p(t) \ \oplus \ \bigoplus_{(q/d) \in P} \rho_{q/d}(t) \mid t \in F \}$
Group	$F \triangleright \gamma_{g_1/y_1,..,g_n/y_n\,:\,a_1/b_1,..,a_m/b_m} =$ $\Big\{ \mathsf{attach}(\texttt{\_id}, \mathbf{null}) \oplus \bigoplus_{i=1}^m \mathsf{attach}\big(a_i, \mathsf{array}(F \triangleright \mu_{\varphi \wedge \exists b_i}, b_i)\big) \mid \varphi = \bigwedge_{j=1}^n (\neg \exists y_j),\ (F \triangleright \mu_\varphi) \neq \emptyset \Big\} \cup$ $\Big\{ \bigoplus_{j \in J} \mathsf{attach}(\texttt{\_id}.g_j, t_j) \oplus \bigoplus_{i=1}^m \mathsf{attach}\big(a_i, \mathsf{array}(F \triangleright \mu_{\psi \wedge \exists b_i}, b_i)\big) \mid J \in 2^{\{1,..,n\}} \setminus \emptyset,$ $t_j \in \mathsf{forest}(F, y_j) \text{ for } j \in J,\ \psi = \bigwedge_{j \in J}((y_j = t_j) \wedge \exists y_j) \wedge \bigwedge_{j \notin J} (\neg \exists y_j),\ (F \triangleright \mu_\psi) \neq \emptyset \Big\}$
Lookup	$\lambda_p^{p_1=C.p_2}[F'](t) = t \oplus \mathsf{attach}(p, \mathsf{array}(F' \triangleright \mu_\varphi, \varepsilon))$, for $\varphi = (p_2 = \mathsf{subtree}(t, p_1))$ if $t \models \exists p_1$, and $\varphi = \neg \exists p_2$ otherwise $F \triangleright \lambda_p^{p_1=C.p_2}[F'] = \{\lambda_p^{p_1=C.p_2}[F'](t) \mid t \in F\}$

$[\![p]\!]^t$, and when $[\![p]\!]^t = \emptyset$, it returns **null**. E.g., $\mathsf{subtree}(t_{\mathrm{KN}}, \mathrm{name})$ returns {{first: "Kristen", last: "Nygaard"}}, and $\mathsf{subtree}(t_{\mathrm{KN}}, \mathrm{awards.year})$ returns [1999,2001,2001].

- $\mathsf{subtree}(t, N)$ returns the subtree (i.e., a subgraph) of t induced by the set N of nodes.
- $\mathsf{attach}(p, t)$ constructs a new tree by attaching t (via its root) to the end of the path p. E.g., $\mathsf{attach}(\mathrm{info.name}, \{\{\mathrm{first: "Kristen"}\}\})$ returns {{info: {{name: {{first: "Kristen"}}}}}}.
- $\mathsf{replace}(t, t_1, t_2)$ constructs a new tree by replacing in t its subtree t_1 by a new tree t_2.
- $t_1 \oplus t_2$ constructs a new tree resulting from merging t_1 and t_2 by identifying nodes reachable via identical paths. E.g., {{name: {{first: "Kristen"}}}} $\oplus$ {{name: {{last: "Nygaard"}}}} returns {{name: {{first: "Kristen", last: "Nygaard"}}}}.
- $\mathsf{array}(F, p)$ constructs a new tree that is the array of all $\mathsf{subtree}(t, p)$ for $t \in F$, while $\mathsf{forest}(F, p)$ keeps all $\mathsf{subtree}(t, p)$ in a set.

Now, we are ready to define the semantics of the MQuery stages: specifically, given a forest F and a stage s, we define the forest $F \triangleright s$ (for a lookup stage, we also require an additional forest F' as parameter), as shown in Table 1. We observe that, for all operators except *group*, each tree in the input can be processed independently of the other trees, and gives rise to zero, one, or more trees in the output. Below, we provide some explanations:

Match. We just observe that *match* might produce an empty output.

Unwind. We say that a path p is a *first array* in t if $\mathsf{type}(p, t) = \mathsf{array}$ and $\mathsf{type}(p', t) \neq \mathsf{array}$, for each strict prefix p' of p. When p is a first array in t with value different from [ ], then $\omega_p(t)$ contains one tree for each element in such an array, obtained by replacing in t the array by the element. In all other cases (i.e., when p is a first array in t and its value is [ ], when p is missing in t, when $\mathsf{type}(p, t) \neq \mathsf{array}$, or when $\mathsf{type}(p, t) = \mathsf{array}$ but p is not a first array in t), we have that $\omega_p(t)$ is empty.

Project. ρ_P produces exactly one output tree from each input tree, obtained by applying to the input tree each of the elements in P, independently of the other elements. Note that, when $q/p \in P$ and p is missing in the input tree, then also q is missing in the output tree. Instead, for $q/[p] \in P$ with p missing, q is present in the output with value [**null**].

Group. In $\gamma_{G\,:\,A}$, when G is empty, i.e., $n = 0$, φ is the empty conjunction and hence true, so all input trees are grouped in one output tree where the value of `_id` is **null**. Instead, when $G = g_1/y_1, \ldots, g_n/y_n$ with $n \geq 1$, then the set of input trees is partitioned into "groups", where each group corresponds to a (possibly empty) subset Y of $\{y_1, \ldots, y_n\}$, so that the trees in the group agree not only on the respective values t_j reached through

all the paths $y_j \in Y$, but also on the non-existence of paths not in Y. Each group $Y = \{y_{j_1}, \ldots, y_{j_k}\}$ gives rise to one output tree. In the case where $k > 0$ and all g_{j_i}s are keys, in the output tree the value of `_id` for that group is $\{\!\{g_{j_1}{:}t_{j_1}, \ldots, g_{j_k}{:}t_{j_k}\}\!\}$, and in the case where $k = 0$ (i.e., $Y = \emptyset$) the value of `_id` is **null**. Moreover, for each pair a_i/b_i in A, the values of b_i of all trees in a group are collected in an array, and such an array is inserted in the output tree for that group as the value of a_i.

Lookup. $\lambda_p^{p_1=C.p_2}[F']$ produces exactly one output tree from each input tree. Each such tree coincides with the input tree, except for one additional array containing all the trees of F' for which the value of p_2 coincides with the value of p_1 in the input tree.

We clarify what we mean by "employing set semantics". For every stage s and forest F, $F \triangleright s$ is a set of trees, i.e., contains no duplicates. Duplicates are detected by comparing trees using deep equality, where comparison of arrays ignores the element indexes. However trees might contain arrays with duplicates. Also, array indexes are sometimes important for merging trees correctly when computing the result of a project stage (see Example 24 in App. A.3).

The semantics of an MQuery is obtained by composing (via $\triangleright$) the answers of its stages.

▶ **Definition 1.** Let $\boldsymbol{q} = C \triangleright s_1 \triangleright \cdots \triangleright s_n$ be an MQuery. The *result of evaluating* $\boldsymbol{q}$ *over* a MongoDB instance D, denoted $ans_{\mathsf{mo}}(\boldsymbol{q}, D)$, is defined as F_n, where $F_0 = D.C$, and for $i \in \{1, \ldots, n\}$, $F_i = (F_{i-1} \triangleright s_i)$ if s_i is not a lookup stage, and $F_i = (F_{i-1} \triangleright s_i[D.C'])$ if s_i is a lookup stage referring to an external collection name C'.

Counter-intuitive Choices in the Semantics of MongoDB

We conclude this section by discussing some choices in the semantics of MongoDB that we consider counter-intuitive, and that could be considered as an inconsistency in the behavior of operators. Therefore, in MQuery, we have chosen a cleaner, more uniform semantics.

"Entering an array" when comparing value and path. In MongoDB, the satisfaction relation $t \models (p = v)$ behaves differently in *match* and in *project* when the type of p in t is array. In *match*, equality holds when v is

(1) exactly the array value of p, or

(2) an element inside the array value of p,

while *project* checks only condition *(1)*. In MQuery, we take a uniform approach, in which $t \models (p = v)$ in *project* behaves as in *match*.

Null and missing values. In some cases, MongoDB does not distinguish

(a) when the value of a path p is **null**, i.e., $[\![p]\!]^t = \{x\}$ and $\mathsf{value}(x, t) = \textbf{null}$, from

(b) when p is missing in t.

In particular, in *match* both *(a)* and *(b)* imply that $t \models (p = \textbf{null})$. Instead, in *project*, only *(a)* implies it. Similarly, in *group*, when grouping by one path (e.g., $\gamma_{g/p\,:\,A}$), MongoDB puts the trees satisfying *(a)* and *(b)* into the same group (having `_id` $= \{\!\{g : \textbf{null}\}\!\}$). Instead, when grouping with multiple paths (e.g., $\gamma_{g_1/p_1, g_2/p_2, \ldots\,:\,A}$), the trees with all p_i missing are put into a separate group (having `_id` $= \{\!\{\}\!\}$). In MQuery, instead, we systematically distinguish the cases *(a)* and *(b)*.

Comparison of values. In MongoDB, equality of non-literal values is determined by comparing their binary representation[6]. Hence, two objects with the same key-value pairs but in different orders, will not be considered the same, which might result in missed answers. In MQuery, we employ the classical semantics for "deep" equality of non-literal values.

[6] `https://docs.mongodb.org/manual/reference/bson-types/#comparison-sort-order`

5 Expressivity of MQuery

In this section we characterize the expressivity of MQuery in terms of nested relational algebra (NRA), and we do so by developing translations between the two languages.

5.1 Nested Relational View of MongoDB

We start by defining a nested relational view of MongoDB instances. In the case of a MongoDB instance with an irregular structure, there is no natural way to define such a relational view. This happens either when the type of a path in a tree is not defined, or when a path has different types in two trees in the instance. Therefore, in order to define a schema for the relational view, which is also independent of the actual MongoDB instances, we impose on them some form of regularity. We start by introducing the notion of *type* of a tree, which is analogous to complex object types [15], and similar to JSON schema [17].

▶ **Definition 2.** Consider JSON values constructed according to the following grammar:

TYPE ::= literal | {{LIST<KEY:TYPE>}} | [TYPE]

Given such a JSON value d, we call the tree $\mathsf{tree}(d)$ a *type*. We say that a tree t is *of type* τ if for every path p we have that $t \models \exists p$ implies

(i) $\tau \models \exists p$,

(ii) $\mathsf{type}(p, t) = \mathsf{type}(p, \tau)$, and

(iii) $\mathsf{type}(p[\,], t) = \mathsf{type}(p[\,], \tau)$.

A forest F is *of type* τ if all trees in F are of type τ. A forest (resp., tree) is *well-typed* if it is of some type.

We now associate to each type τ a relation schema $\mathsf{rschema}(\tau)$ in which, intuitively, attributes correspond to paths, and each nested relation corresponds to an array in τ. In the following definition, given paths p and q, we say that $p.q$ is a *simple extension* of p if there is no strict prefix q' of q such that $\mathsf{type}(p.q', \tau) = \mathsf{array}$.

▶ **Definition 3.** For a type τ, the *relation schema* $\mathsf{rschema}(\tau)$, is defined as $R_\tau(\mathsf{ratt}_\tau(\varepsilon))$, where, for a path p in τ, $\mathsf{ratt}_\tau(p)$ is the set of simple extensions p' of p such that p' is an *atomic attribute* if $\mathsf{type}(p', \tau) = \mathsf{literal}$, and p' is a *sub-relation* if $\mathsf{type}(p', \tau) = \mathsf{array}$. In the latter case, p' has attributes $\{p'.\texttt{lit}\}$ if $\mathsf{type}(p'[\,], \tau) = \mathsf{literal}$, and $\mathsf{ratt}_\tau(p')$ otherwise.

Observe that the names of sub-relations and of atomic attributes in $\mathsf{rschema}(\tau)$ are given by paths from the root in τ, and therefore are unique.

Next, we define the relational view of a well-typed forest. In this view, to capture the semantics of the missing paths, we introduce the new constant **missing**.

▶ **Definition 4.** The *relational view* of a well-typed forest F, denoted $\mathsf{rel}(F)$, is defined as $\{\mathsf{rtuple}_\tau(R_\tau, \varepsilon, t) \mid t \in F\}$, where τ is the type of F. For a relation name R in $\mathsf{rschema}(\tau)$ and a path p, $\mathsf{rtuple}_\tau(R, p, t)$ is the R-tuple $\{p.q : \mathsf{rval}(p.q, t)\}_{p.q \in \mathsf{ratt}_\tau(p)}$, where

$$\mathsf{rval}(p.q, t) = \begin{cases} \textbf{missing}, & \text{if } [\![q]\!]^t = \emptyset; \\ \mathsf{value}(\mathsf{subtree}(t, q)), & \text{if } p.q \text{ is atomic}; \\ \left\{(p.q.\texttt{lit} : \mathsf{value}(\mathsf{subtree}(t, q.i))) \mid [\![q.i]\!]^t \neq \emptyset, \text{ for } i \in I\right\}, & \text{if } \mathsf{att}_\tau(p.q) = \{p.q.\texttt{lit}\}; \\ \left\{\mathsf{rtuple}_\tau\,(p.q, p.q, \mathsf{subtree}(t, q.i)) \mid [\![q.i]\!]^t \neq \emptyset, \text{ for } i \in I\right\}, & \text{otherwise.} \end{cases}$$

_id	awards		birth	contribs	name.first	name.last
	awards.award	awards.year		contribs.lit		
4	Rosing Prize	1999	1926-08-27	OOP	Kristen	Nygaard
	Turing Award	2001		Simula		
	IEEE John von Neumann Medal	2001				

Figure 5 Relational view of the document about Kristen Nygaard.

▶ **Example 5.** Consider the type $\tau_{\texttt{bios}}$ for `bios`:

```
{ "_id": "literal",    "awards": [ { "award": "literal", "year": "literal" } ],
  "birth": "literal",  "contribs": [ "literal" ],
  "name": { "first": "literal", "last": "literal" } }
```

Then, $\mathsf{rschema}(\tau_{\texttt{bios}})$ is defined as $\texttt{bios}\big(\texttt{\_id}, \text{awards(awards.award, awards.year)}, \text{birth}, \text{contribs(contribs.}\texttt{lit}\text{)}, \text{name.first}, \text{name.last}\big)$. Moreover, for the tree t in Figure 3, the relational view $\mathsf{rel}(\{t\})$ is illustrated in Figure 5. ◀

To define the relational view of MongoDB instances, we introduce the notion of *(MongoDB) type constraints*, which are given by a set $\mathcal{S}$ of pairs (C, τ), one for each collection name C, where τ is a type. We say that a database D *satisfies* the constraints $\mathcal{S}$ if $D.C$ is of type τ, for each $(C, \tau) \in \mathcal{S}$. For a given $\mathcal{S}$, for each $(C, \tau) \in \mathcal{S}$, we refer to τ by τ_C. Moreover, we assume that in $\mathsf{rschema}(\tau_C)$, the relation name R_{τ_C} is actually C.

▶ **Definition 6.** Let $\mathcal{S}$ be a set of type constraints, and D a MongoDB instance satisfying $\mathcal{S}$. The *relational view* $\mathsf{rdb}_{\mathcal{S}}(D)$ of D with respect to $\mathcal{S}$ is the instance $\{\mathsf{rel}(D.C) \mid (C, \tau) \in \mathcal{S}\}$.

Finally, we define equivalence between MQueries and NRA queries. To this purpose, we also define equivalence between two kinds of answers: well-typed forests and nested relations.

▶ **Definition 7.** A well-typed forest F is *equivalent* to a nested relation $\mathcal{R}$, denoted $F \simeq \mathcal{R}$, if $\mathsf{rel}(F) = \mathcal{R}$. An MQuery $\boldsymbol{q}$ is *equivalent to* an NRA query Q w.r.t. type constraints $\mathcal{S}$, denoted $\boldsymbol{q} \equiv_{\mathcal{S}} Q$, if $ans_{\mathsf{mo}}(\boldsymbol{q}, D) \simeq ans_{\mathsf{ra}}(Q, \mathsf{rdb}_{\mathcal{S}}(D))$, for each MongoDB instance D satisfying $\mathcal{S}$.

Notice that the above definition of equivalence between well-typed forests and nested relations appears to be asymmetric, since it would in principle allow for nested relations that are not equivalent to any well-typed forest. We remark, however, that the MongoDB view of a nested relation always exists, is well-typed, and can be defined in a straightforward way. Therefore, we can consider both translations (from NRA to MQuery, and vice-versa), as defined on well-typed forests and their relational views.

5.2 From NRA to MQuery

We now show that $\mathcal{M}^{\text{MUPGL}}$ captures NRA, while $\mathcal{M}^{\text{MUPG}}$ captures NRA over a single collection.

In our translation from NRA to MQuery, we have to deal with the fact that an NRA query in general has a *tree* structure where the leaves are relation names, while an MQuery contains *one sequence* of stages. So, we first show how to "linearize" tree-shaped NRA expressions into a MongoDB pipeline. More precisely, we show that it is possible to combine two $\mathcal{M}^{\text{MUPG}}$ sequences $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$ of stages into a single $\mathcal{M}^{\text{MUPG}}$ sequence $\mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$, so that the results of $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$ can be accessed from the result of $\mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$ for further processing. We define $\mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$ as $\mathsf{dup} \triangleright \mathsf{subq}_1(\boldsymbol{q}_1) \triangleright \mathsf{subq}_2(\boldsymbol{q}_2)$.

The idea of dup is to create for each tree t of the input forest two trees differentiated by an ad-hoc key-value pair $\texttt{actRel}: j$ and storing the original tree as $\texttt{rel}j: t$, for $j \in \{1, 2\}$. More precisely, we want to obtain for each forest F that $F \triangleright \mathsf{dup} = F_1 \cup F_2$, where

Table 2 Subquery $\mathsf{subq}_j(s)$ for stage s, where we have detailed only the short forms for project and group stages. We use $e_{[p\to q]}$ to denote the expression e in which every occurrence of the path p is replaced by the path q, and use norm to abbreviate $\rho_{\text{actRel},\,\{\text{rel}i/((\text{actRel}=i)?\,\text{rel}i:\,\text{dummy})\}_{i=1,2}}$.

s	$\mathsf{subq}_j(s)$	s	$\mathsf{subq}_j(s)$
μ_φ	$\mu_{(\text{actRel}=3-j)\vee\varphi_{[p\to\text{rel}j.p]}}$	$\gamma_{g/y:a/b}$	$\gamma_{g/\text{rel}j.y,\,\text{actRel}:a/\text{rel}j.b,\,\text{rel}(3-j)}\triangleright$
ω_p	$\rho_{\text{actRel},\,\text{rel}(3-j),\,\text{rel}j/((\text{actRel}=j)?\,\text{rel}j:\,\{\!\{p\text{:}[0]\}\!\})}$ $\triangleright\,\omega_{\text{rel}j.p}\triangleright\,\mathsf{norm}$		$\rho_{\text{actRel}/\texttt{\_id}.\text{actRel},\,\text{rel}j.\texttt{\_id}.g/\texttt{\_id}.g,\,\text{rel}j.a/a,\,\text{rel}(3-j)}$ $\triangleright\rho_{\text{actRel},\,\text{rel}j,\,\text{rel}(3-j)/((\text{actRel}=3-j)?\,\text{rel}(3-j):\,[0])}$ $\triangleright\,\omega_{\text{rel}(3-j)}\triangleright\,\mathsf{norm}$
$\rho_{p,\,q/d}$	$\rho_{\text{actRel},\,\text{rel}(3-j),\,\text{rel}j.p,}$ $\text{rel}j.q/((\text{actRel}=j)?\,d_{[q'\to\text{rel}j.q']}:\,\text{dummy})$		

$F_1 = \{\!\{\text{actRel: 1, rel1: } t\}\!\}_{t\in F}$ and $F_2 = \{\!\{\text{actRel: 2, rel2: } t\}\!\}_{t\in F}$. This is achieved by setting $\mathsf{dup} = \rho_{\text{origDoc}/\varepsilon,\,\text{actRel}/[1,2]}\triangleright\omega_{\text{actRel}}\triangleright\rho_{\text{actRel},\,\{\text{rel}j/((\text{actRel}=j)?\,\text{origDoc: dummy})\}_{j=1,2}}$, where `dummy` is a path that does not exist in any collection.

The idea of $\mathsf{subq}_j(\boldsymbol{q}_j)$ is to execute $\boldsymbol{q}_j$ so that it affects the trees from F_j, but not from F_{3-j}, and to obtain that $(F_1\cup F_2)\triangleright\mathsf{subq}_1(\boldsymbol{q}_1)\triangleright\mathsf{subq}_2(\boldsymbol{q}_2)$ evaluates to the forest $\{\!\{\text{actRel: 1, rel1: } t\}\!\}_{t\in(F\triangleright\boldsymbol{q}_1)}\cup\{\!\{\text{actRel: 2, rel2: } t\}\!\}_{t\in(F\triangleright\boldsymbol{q}_2)}$. Before describing subq_j formally, we provide the intuition in an example.

▶ **Example 8.** Consider the sequences of stages $\boldsymbol{q}_1 = \mu_{a=1}\triangleright\rho_{a,\,b}$ and $\boldsymbol{q}_2 = \mu_{c=\text{"x"}}\triangleright\rho_{c,\,d}$, and the forest $F = \{t_1, t_2, t_3, t_4\}$, for $t_1 = \{\!\{$a:1, b:6, d:8$\}\!\}$, $t_2 = \{\!\{$a:1, b:7, c:"x", d:9$\}\!\}$, $t_3 = \{\!\{$a:2, b:6, c:"x", d:7$\}\!\}$, and $t_4 = \{\!\{$a:3, b:8, c:"y", d:6$\}\!\}$.

Denote by t_i^{pq} the tree resulting from t_i by applying $\rho_{p,\,q}$ to it. Then $F\triangleright\boldsymbol{q}_1$ evaluates to $\{t_1^{ab}, t_2^{ab}\}$, and $F\triangleright\boldsymbol{q}_2$ to $\{t_2^{cd}, t_3^{cd}\}$. Thus, $(F_1\cup F_2)\triangleright\mathsf{subq}_1(\boldsymbol{q}_1)\triangleright\mathsf{subq}_2(\boldsymbol{q}_2)$ should be $\{\!\{\text{actRel: 1, rel1: } t_1^{ab}\}\!\}$, $\{\!\{\text{actRel: 1, rel1: } t_2^{ab}\}\!\}$, $\{\!\{\text{actRel: 2, rel2: } t_2^{cd}\}\!\}$, $\{\!\{\text{actRel: 2, rel2: } t_3^{cd}\}\!\}\}$. We achieve this by setting $\mathsf{subq}_1(\boldsymbol{q}_1) = \mu_{(\text{actRel}=2)\vee(\text{rel1}.a=1)}\triangleright\rho_{\text{rel2},\,\text{actRel},\,\text{rel1}.a,\,\text{rel1}.b}$ and $\mathsf{subq}_2(\boldsymbol{q}_2) = \mu_{(\text{actRel}=1)\vee(\text{rel2}.c=\text{"x"})}\triangleright\rho_{\text{rel1},\,\text{actRel},\,\text{rel2}.c,\,\text{rel2}.d}$. Here, in the case of $\mathsf{subq}_1(\boldsymbol{q}_1)$, by transforming the criterion $(a=1)$ into $(\text{actRel}=2)\vee(\text{rel1}.a=1)$ we make sure to preserve the trees in F_2, and we check the condition on the correct path `rel1.a` in the trees in F_1. Similarly, the project stage $\rho_{a,b}$ became $\rho_{\text{rel2},\,\text{actRel},\,\text{rel1}.a,\,\text{rel1}.b}$ in order to preserve the t_is in F_2 stored under `rel2`, to preserve the auxiliary key `actRel`, and to project the correct paths `rel1.a` and `rel1.b` in the trees from F_1. ◀

Formally, when $\boldsymbol{q}_j = s_1\triangleright\cdots\triangleright s_n$ then $\mathsf{subq}_j(\boldsymbol{q}_j)$ is defined as $\mathsf{subq}_j(s_1)\triangleright\cdots\triangleright\mathsf{subq}_j(s_n)$, for $j\in\{1,2\}$, where subq_j for single stages is defined in Table 2. Recall that the idea of $\mathsf{subq}_j(s)$ is to affect only the trees in F_j, hence:

- $\mathsf{subq}_j(\mu_\varphi)$ selects all trees in F_{3-j} (those that satisfy $(\text{actRel}=3-j)$), while from F_j it selects the trees that satisfy φ, where all original paths p are replaced by `rel`$j.p$.
- The unwind stage ω_p cannot be implemented simply by $\omega_{\text{rel}j.p}$, since all trees in F_{3-j} would be lost (they do not contain the path `rel`$j.p$). Therefore we first create a temporary non-empty array ($[0]$) under the path `rel`$j.p$ in the trees from F_{3-j}, unwind the path `rel`$j.p$, and then in norm normalize the trees by making sure that the trees with $(\text{actRel}=i)$ contain only `rel`i but not `rel`$(3-i)$.
- The encoding of the project stage $\rho_{p,\,q/d}$ makes sure that `rel`$(3-j)$ and `actRel` are not lost, and that the path `rel`$j.q$ is not created in the trees from F_{3-j} (guaranteed by the conditional expression for q/d).
- The encoding of the group stage $\gamma_{g/y:a/b}$ adds `actRel` to the grouping condition and aggregates `rel`$(3-j)$ so as to group all trees from F_{3-j} in one tree, and then renames appropriately the paths `_id.actRel`, `_id.g`, and `a`. It is concluded similarly to $\mathsf{subq}_j(\omega_p)$ in order to flatten the array `rel`$(3-j)$ where all trees from F_{3-j} have been aggregated.

Table 3 Translation from NRA to $\mathcal{M}^{\text{MUPG}}$. We extend the function att from schema names to NRA queries such that $\mathsf{att}(Q)$ is the attribute set of the schema implied by an NRA query Q.

Q	$\mathsf{nra2mq}(Q)$
C	$\rho_{\mathsf{att}(C)}$
$\sigma_\psi(Q_1)$	$\mathsf{nra2mq}(Q_1) \triangleright \rho_{\mathsf{att}(Q_1),\,\mathrm{cond}/\psi} \triangleright \mu_{\mathrm{cond}=\mathbf{true}} \triangleright \rho_{\mathsf{att}(Q_1)}$
$\pi_S(Q_1)$	$\mathsf{nra2mq}(Q_1) \triangleright \rho_S$
$\nu_{S\to b}(Q_1)$	$\mathsf{nra2mq}(Q_1) \triangleright \rho_{(\mathsf{att}(Q_1)\setminus S),\,\{b.p/p \mid p\in S\}} \triangleright \gamma_{(\mathsf{att}(Q_1)\setminus S)\,:\,b} \triangleright \rho_{b,\,\{p/\texttt{\_id}.p \mid p\in\mathsf{att}(Q_1)\setminus S\}}$
$\chi_a(Q_1)$	$\mathsf{nra2mq}(Q_1) \triangleright \omega_a$
$Q_1 \times Q_2$	$\mathsf{pipeline}(\mathsf{nra2mq}(Q_1), \mathsf{nra2mq}(Q_2)) \triangleright \gamma_{:\,\mathrm{rel1},\,\mathrm{rel2}} \triangleright \omega_{\mathrm{rel1}} \triangleright \omega_{\mathrm{rel2}}$
$Q_1 \cup Q_2$	$\mathsf{pipeline}(\mathsf{nra2mq}(Q_1), \mathsf{nra2mq}(Q_2)) \triangleright \rho_{\{\mathrm{p}i/((\mathrm{actRel}=1)?\,\mathrm{rel1}.\mathrm{p}i:\,\mathrm{rel2}.\mathrm{p}i)\}_{i=1}^{n}} \triangleright \gamma_{\mathrm{p1},\ldots,\mathrm{p}n\,:} \triangleright \rho_{\{\mathrm{p}i/\texttt{\_id}.\mathrm{p}i\}_{i=1}^{n}}$
$Q_1 \setminus Q_2$	$\mathsf{pipeline}(\mathsf{nra2mq}(Q_1), \mathsf{nra2mq}(Q_2)) \triangleright \rho_{\mathrm{rel2},\,\{\mathrm{p}i/((\mathrm{actRel}=1)?\,\mathrm{rel1}.\mathrm{p}i:\,\mathrm{rel2}.\mathrm{p}i)\}_{i=1}^{n}} \triangleright \gamma_{\mathrm{p1},\ldots,\mathrm{p}n\,:\,\mathrm{rel2}} \triangleright \mu_{\mathrm{rel2}=[]} \triangleright \rho_{\{\mathrm{p}i/\texttt{\_id}.\mathrm{p}i\}_{i=1}^{n}}$

Now, having defined $\mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$, we are ready to show how to translate NRA to MQuery. We start with a singleton set $\mathcal{S} = \{(C, \tau_C)\}$ of type constraints for a collection name C, and consider an NRA query Q over the relation name C (with schema $\mathsf{rschema}(\tau_C)$). The translation of Q is the $\mathcal{M}^{\text{MUPG}}$ query $C \triangleright \mathsf{nra2mq}(Q)$, where $\mathsf{nra2mq}(Q)$ is defined inductively in Table 3. To encode select, the filter is translated as a Boolean value definition, except that atoms of the form $\neg(a = \mathbf{missing})$ become $\exists a$. The translation of $Q_1 \times Q_2$ first groups all input trees in one tree, where the answer trees t_i to Q_i are aggregated in arrays $\texttt{rel}i$, $i \in \{1, 2\}$, and then unwinds these two arrays, thus producing all possible pairs (t_1, t_2). The translations of $Q_1 \cup Q_2$ and $Q_1 \setminus Q_2$, where we assume that $\mathsf{att}(Q_i) = \{\mathrm{p1}, \ldots, \mathrm{p}n\}$, first create fresh paths $\texttt{p}i$ in each tree to be used in the grouping condition. Then, in the case of union it only remains to rename the paths $\texttt{\_id.p}i$ back to $\texttt{p}i$, while in the case of difference, we also select only those "tuples" that were not present in the answer to Q_2.

▶ **Example 9.** Consider the forest F from Example 8 stored under a collection name C, and the type $\tau_C = \{\!\{\text{a: literal, b: literal, c: literal, d: literal}\}\!\}$. Then $\mathsf{rschema}(\tau_C)$ is defined as C(a,b,c,d), and $\mathsf{rel}(F)$ is the relation $\{$(a: 1, b: 6, c: **missing**, d: 8), (a: 1, b: 7, c: "x", d: 9), (a: 2, b: 6, c: "x", d: 7), (a: 3, b: 8, c: "y", d: 6)$\}$. Let Q be the NRA query $\sigma_{\mathrm{rel1}.b=\mathrm{rel2}.d}\big(Q_1 \times Q_2\big)$, where $Q_1 = \pi_{a,b}(\sigma_{a=1}(C))$ and $Q_2 = \pi_{c,d}(\sigma_{c=\text{"x"}}(C))$. Then Q evaluated over $\mathsf{rel}(F)$ returns $\{$(rel1.a: 1, rel1.b: 7, rel2.c: "x", rel2.d: 7)$\}$.

Now, $\mathsf{nra2mq}(Q_j) = \rho_{a,\,b,\,c,\,d} \triangleright \boldsymbol{q}_j$, for $j = 1, 2$, where $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$ are as in Example 8. Since $\rho_{a,\,b,\,c,\,d} \triangleright \boldsymbol{q}_j$ and $\boldsymbol{q}_j$ are equivalent (return the same answers over all forests), we have that $F \triangleright \mathsf{pipeline}(\mathsf{nra2mq}(Q_1), \mathsf{nra2mq}(Q_2)) = F \triangleright \mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$. Denote by F' the result of $F \triangleright \mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$, see Example 8. Then

- $F'' = F' \triangleright \gamma_{:\,\mathrm{rel1},\,\mathrm{rel2}}$ is the forest $\{\{\!\{\text{rel1: } [t_1^{ab}, t_2^{ab}], \text{ rel2: } [t_2^{cd}, t_3^{cd}]\}\!\}\}$.
- $F''' = F'' \triangleright \omega_{\mathrm{rel1}} \triangleright \omega_{\mathrm{rel2}}$ is the forest $\{\{\!\{\text{rel1: } t_1^{ab}, \text{ rel2: } t_2^{cd}\}\!\}, \{\!\{\text{rel1: } t_2^{ab}, \text{ rel2: } t_2^{cd}\}\!\}, \{\!\{\text{rel1: } t_1^{ab}, \text{ rel2: } t_3^{cd}\}\!\}, \{\!\{\text{rel1: } t_2^{ab}, \text{ rel2: } t_3^{cd}\}\!\}\}$.
- Finally, $F''' \triangleright \rho_{\mathrm{cond}/(\mathrm{rel1}.b=\mathrm{rel2}.d)} \triangleright \mu_{\mathrm{cond}=\mathbf{true}} \triangleright \rho_{\mathrm{rel1}.a,\,\mathrm{rel1}.b,\,\mathrm{rel2}.c,\,\mathrm{rel2}.d}$ is the forest $\{\{\!\{\text{rel1: } t_2^{ab}, \text{ rel2: } t_3^{cd}\}\!\}\}$, or equivalently $\{\{\!\{\text{rel1: } \{\!\{\text{a: 1, b: 7}\}\!\}, \text{ rel2: } \{\!\{\text{c: "x", d: 7}\}\!\}\}\!\}\}$. ◀

▶ **Theorem 10.** *Let Q be a NRA query over C. Then $C \triangleright \mathsf{nra2mq}(Q) \equiv_{\mathcal{S}} Q$.*

Next, we consider NRA queries across several collections, and show how to translate them to $\mathcal{M}^{\text{MUPGL}}$. Let $\mathcal{S}$ be a set of type constraints for collection names $C_1, \ldots, C_n$, with $n \geq 2$, Q an NRA query over $C_1, \ldots, C_n$, and C_1 the collection over which we evaluate the generated MQuery. The translation of Q is the $\mathcal{M}^{\text{MUPGL}}$ query $C_1 \triangleright \mathsf{bring}(C_2, \ldots, C_n) \triangleright \mathsf{nra2mq}^\star(Q)$, where intuitively (1) the phase $\mathsf{bring}(C_2, \ldots, C_n)$ "brings in" the trees from the collections $C_2, \ldots, C_n$, and (2) the function $\mathsf{nra2mq}^\star(Q)$, adapted from $\mathsf{nra2mq}(Q)$, simulates the NRA

operators in Q. More precisely, we want that if $F_1, \ldots, F_n$ are collections for $C_1, \ldots, C_n$, the result of $F_1 \triangleright \mathsf{bring}(C_2, \ldots, C_n)[F_2, \ldots, F_n]$ is the forest $\bigcup_{i=1}^{n} \{\{\!\{\text{actColl}\colon i,\ \text{coll}i\colon t\}\!\}\}_{t \in F_i}$. This is done by setting $\mathsf{bring}(C_2, \ldots, C_n)$ as

$$\gamma_{:\,\text{coll1}/\varepsilon} \triangleright \lambda^{\text{dummy}=C_2.\text{dummy}}_{\text{coll2}} \triangleright \cdots \triangleright \lambda^{\text{dummy}=C_n.\text{dummy}}_{\text{coll}n} \triangleright \rho_{\text{coll1},..,\text{coll}n,\,\text{actColl}/[1..n]} \triangleright \omega_{\text{actColl}} \triangleright$$
$$\rho_{\text{actColl},\,\{\text{coll}i/((\text{actColl}=i)?\,\text{coll}i\colon[0])\}_{i=1}^{n}} \triangleright \omega_{\text{coll1}} \triangleright \cdots \triangleright \omega_{\text{coll}n} \triangleright \rho_{\text{actColl},\,\{\text{coll}i/((\text{actColl}=i)?\,\text{coll}i\colon\text{dummy})\}_{i=1}^{n}}$$

Moreover, we define the function $\mathsf{nra2mq}^{\star}(Q)$ that differs from $\mathsf{nra2mq}(Q)$ in the translation of the collection names as $\mathsf{nra2mq}^{\star}(C_i) = \mu_{\text{actColl}=i} \triangleright \rho_{\{p/\text{coll}i.p \,\mid\, p \in \mathsf{att}(C_i)\}}$.

▶ **Theorem 11.** *Let Q be an NRA query over $C_1, \ldots, C_n$, and $\boldsymbol{q} = C_1 \triangleright \mathsf{bring}(C_2, \ldots, C_n) \triangleright \mathsf{nra2mq}^{\star}(Q)$. Then $\boldsymbol{q} \equiv_{\mathcal{S}} Q$. Moreover, the size of $\boldsymbol{q}$ is polynomial in the size of Q.*

Thus, we obtain that $\mathcal{M}^{\text{MUPGL}}$ captures full NRA, and that $\mathcal{M}^{\text{MUPG}}$ captures NRA over a single collection. We observe that the above translation serves the purpose of understanding the expressive power of MQuery, but is likely to produce queries that MongoDB will not be able to efficiently execute in practice, even on relatively small database instances. We also note that the translation from NRA to MQuery works even if we allow for database instances D such that $D.C$ is not strictly of type τ_C, but may also contain other paths not in τ_C.

5.3 From MQuery to NRA

In this section, we aim at defining a translation from MQuery to NRA, and for this we want to exploit the structure, i.e., the stages of MQueries. Hence, we define a translation $\mathsf{mq2nra}(s)$ from stages s to NRA expressions such that, for an MQuery $C \triangleright s_1 \triangleright \cdots \triangleright s_n$, the corresponding NRA query is defined as $C \circ \mathsf{mq2nra}(s_1) \circ \cdots \circ \mathsf{mq2nra}(s_n)$[7], where we identify the collection name C with the corresponding relation schema in the relational view. However, such a translation might not always be possible, since MQuery is capable of producing non well-typed forests, for which the relational view is not defined. This capability is due to value definitions in a project operator: already a query as simple as $\rho_{\text{a}/(\texttt{\_id}=1?\,[0,1]\colon\texttt{"s"})}$ produces from the well-typed forest $\{\{\!\{\texttt{\_id}\colon 1\}\!\},\ \{\!\{\texttt{\_id}\colon 2\}\!\}\}$ a non well-typed one: $\{\{\!\{\texttt{\_id}\colon 1,\ \text{a}\colon[0,1]\}\!\},\ \{\!\{\texttt{\_id}\colon 2,\ \text{a}\colon\texttt{"s"}\}\!\}\}$. Therefore, in order to derive such a translation $\mathsf{mq2nra}(s)$, we restrict our attention to MQueries with stages preserving well-typedness.

▶ **Definition 12.** Given a type τ (and a type τ'), a stage s is *well-typed* for τ (and τ'), if for each forest F of type τ (and each forest F' of type τ'), $F \triangleright s$ (resp., $F \triangleright s[F']$ when s is a lookup stage) is a well-typed forest.

We observe that the match, unwind, group and lookup stages are always well-typed, and, given such a stage s and input types τ, τ', we can compute the output type τ_o of s:

- **(i)** match does not change the input type, i.e., $\tau_o = \tau$,
- **(ii)** for unwind and group stages s, τ_o is obtained by evaluating s over $\{\tau\}$, i.e., $\{\tau_o\} = \{\tau\} \triangleright s$, and
- **(iii)** similarly, the output type for a lookup stage is the single tree in $(\{\tau\} \triangleright \lambda^{p_1=C.p_2}_{p}[\{\tau'\}])$.

As for a project stage $s = \rho_P$ and an input type τ, we can check whether s is well-typed for τ, and if yes, we can compute the output type τ_o of s, as follows. For each $p/d \in P$, we compute the type τ_d of d with respect to τ; if all τ_d are defined, then s is well-typed and τ_o is the type where $\mathsf{subtree}(\tau_o, p)$ coincides with τ_d for each $p/d \in P$, and that agrees with τ on all $p \in P$; otherwise s is not well-typed. The *type* τ_d of a value definition d with respect to a type τ is defined inductively as follows: $\tau_v = \tau'$ for a value v, if v is of type τ', and

[7] We follow the convention that $(f \circ g)(x) = g(f(x))$.

undefined otherwise; $\tau_\beta = \mathsf{literal}$ for a Boolean value definition β; $\tau_p = \mathsf{subtree}(\tau, p)$, for a path p; $\tau_{[d_1,\ldots,d_n]} = [\tau_{d_1}]$ if $\tau_{d_1} = \cdots = \tau_{d_n}$, and undefined otherwise; $\tau_{(c?\, d_1 :\, d_2)}$ is τ_{d_1} if c is valid, τ_{d_2} if c is unsatisfiable, τ_{d_1} if c is satisfiable and not valid and $\tau_{d_1} = \tau_{d_2}$, and undefined otherwise.

Then, given a set $\mathcal{S}$ of type constraints and an MQuery $\boldsymbol{q} = C \triangleright s_1 \triangleright \cdots \triangleright s_n$, we can check whether each stage in $\boldsymbol{q}$ is well-typed for its input type determined by $\boldsymbol{q}$ and $\mathcal{S}$. To do so, we take the input type for s_1 to be τ_0, where $(C, \tau_0) \in \mathcal{S}$, and we compute sequentially the input type for each stage s_i, as long as this is possible, i.e., all stages preceding it are well-typed.

The translation $\mathsf{mq2nra}(s)$, for well-typed stages s, is quite natural, although it requires some attention to properly capture the semantics of MQuery. It is reported in [3].

▶ **Theorem 13.** *Let $\mathcal{S}$ be a set of type constraints, $\boldsymbol{q}$ an MQuery $C \triangleright s_1 \triangleright \cdots \triangleright s_m$ in which each stage is well-typed for its input type, and $Q = C \circ \mathsf{mq2nra}(s_1) \circ \cdots \circ \mathsf{mq2nra}(s_m)$. Then $\boldsymbol{q} \equiv_{\mathcal{S}} Q$, moreover, the size of Q is polynomial in the size of $\boldsymbol{q}$ and $\mathcal{S}$.*

A natural question is when an MQuery can be translated to NRA even if it contains non well-typed stages. E.g., in the example above, this can happen when the path `a` is projected away in the subsequent stages without being actually used. We leave this for future work.

6 Complexity of MQuery

In this section we report results on the complexity of different fragments of MQuery. Specifically, we are concerned with the combined and query complexity of the *Boolean query evaluation* problem (i.e., the problem of checking non-emptiness of query answers).

We first establish that $\mathcal{M}^{\text{MUPGL}}$ and $\mathcal{M}^{\text{MUPG}}$ are complete for exponential time with a polynomial number of alternations under LogSpace reductions [6, 14][8]. That is, they have the same complexity as monad algebra with atomic equality and negation [15], which however is strictly less expressive than NRA. As a corollary, we obtain a tight bound for NRA.

▶ **Theorem 14.** *$\mathcal{M}^{\text{MUPG}}$ and $\mathcal{M}^{\text{MUPGL}}$ are $\text{TA}[2^{n^{O(1)}}, n^{O(1)}]$-complete in combined complexity, and in AC^0 in data complexity.*

▶ **Corollary 15.** *NRA is $\text{TA}[2^{n^{O(1)}}, n^{O(1)}]$-complete in combined complexity.*

Next, we study some of the less expressive fragments of MQuery. We consider match to be an essential operator, and we start with the minimal fragment $\mathcal{M}^{\text{M}}$, for which we show that query answering is tractable and very efficient.

▶ **Theorem 16.** *$\mathcal{M}^{\text{M}}$ is LogSpace-complete in combined complexity.*

The project and group operators allow one to create exponentially large values by duplicating the existing ones. For instance, the result of $\{\{\!\{a{:}1\}\!\}\} \triangleright s_1 \triangleright \cdots \triangleright s_n$, for $s_1 = \cdots = s_n = \rho_{a.\ell/a,\, a.r/a}$ consists of a full binary tree of depth n. Nevertheless, without the unwind operator it is still possible to maintain tractability.

▶ **Theorem 17.** *$\mathcal{M}^{\text{MP}}$ is PTime-hard in query complexity and $\mathcal{M}^{\text{MPGL}}$ is in PTime in combined complexity.*

[8] We observe that $\text{TA}[2^{n^{O(1)}}, n^{O(1)}]$ lies between NExpTime and ExpSpace, hence is provably intractable.

We can identify the unwind operator as one of the sources of complexity, as it allows one to multiply the number of trees each time it is used in the pipeline. Indeed, adding the unwind operator alone causes already loss of tractability, provided the input tree contains multiple arrays (hence in combined complexity).

▶ **Theorem 18.** *$\mathcal{M}^{\text{MU}}$ is* LogSpace*-complete in query complexity and* NP*-complete in combined complexity.*

Adding project and lookup does not increase the combined complexity, but does increase the query complexity, since they allow for creating multiple arrays from a fixed input tree.

▶ **Theorem 19.** *$\mathcal{M}^{\text{MUP}}$ and $\mathcal{M}^{\text{MUL}}$ are* NP*-hard in query complexity, and $\mathcal{M}^{\text{MUPL}}$ is in* NP *in combined complexity.*

In the presence of unwind, group provides another source of complexity, since in $\mathcal{M}^{\text{MUG}}$ we can generate doubly exponentially large trees, analogously to monad algebra [15]. Let $t_0 = \{\!\{\texttt{\_id}\colon \{\!\{\text{x}\colon 0\}\!\}\}\!\}$ and $t_1 = \{\!\{\texttt{\_id}\colon \{\!\{\text{x}\colon 1\}\!\}\}\!\}$. The result of applying the $\mathcal{M}^{\text{MUG}}$ query $s_1 \triangleright \cdots \triangleright s_n$, where $s_i = \gamma_{\texttt{:}\,x/\texttt{\_id}.x} \triangleright \gamma_{x.l/x,\, x.r/x\texttt{:}} \triangleright \omega_{\texttt{\_id}.x.l} \triangleright \omega_{\texttt{\_id}.x.r}$, to $\{t_0, t_1\}$ is a forest containing 2^{2^n} trees, each encoding one 2^n-bit value. Below we show that already $\mathcal{M}^{\text{MUG}}$ queries are PSpace-hard.

▶ **Theorem 20.** *$\mathcal{M}^{\text{MUG}}$ is* PSpace*-hard in query complexity.*

7 Conclusions and Future Work

We have carried out a first formal investigation on the foundations and computational properties of the MongoDB aggregation framework, currently the most widely adopted expressive query language for JSON. We proposed a clean abstraction for its five main operators, which we called MQuery. Our formalization focuses on set semantics and, similarly to [12], ignores ordering; bag and list semantics are left for future work. MQuery also "polishes" some counter-intuitive aspects in the syntax and semantics of the actual aggregation framework, which are inherited from its ad-hoc development. We believe that these last changes, which are independent of our simplifying assumptions, make the framework more uniform, and we consequently encourage the designers of MongoDB to adopt them.

We have studied the expressivity of MQuery, establishing the equivalence between its well-typed fragment and NRA, by developing compact translations in both directions. This shows that, despite its design driven by practical requirements, the aggregation framework relies on solid foundations, and hence is worth attention from the DB theory community. We hope that our study will also clarify the apparent confusion among practitioners about its capabilities to perform joins, in particular in the absence of lookup. Moreover, we analyzed the computational complexity of significant fragments of MQuery, obtaining several (tight) bounds. As a byproduct, we obtained also a tight bound for NRA.

With version v3.4, MongoDB has been extended with a *graph-lookup* stage in a pipeline, allowing for a recursive search on a collection, and it is of interest to understand how this affects formal and computational properties. We also propose to investigate the properties of MQuery when the well-typedness restrictions are lifted, and to compare it to JLogic [12], which is likewise able to handle flexible types. We are currently working on applying the results presented here, to provide high-level access to MongoDB data sources by relying on the standard ontology-based data access (OBDA) paradigm [18]. For this, we build on the translation from NRA to MQuery presented in Section 5.2 [2].

References

1 Serge Abiteboul and Catriel Beeri. The power of languages for the manipulation of complex values. *Very Large Database Journal*, 4(4):727–794, 1995. URL: `http://www.vldb.org/journal/VLDBJ4/P727.pdf`.

2 Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Martin Rezk, and Guohui Xiao. OBDA beyond relational DBs: A study for MongoDB. In *Proc. of the 29th Int. Workshop on Description Logics (DL)*, volume 1577 of *CEUR Workshop Proceedings,* `http://ceur-ws.org/`, 2016.

3 Elena Botoeva, Diego Calvanese, Benjamin Cogrel, and Guohui Xiao. Expressivity and complexity of MongoDB (Extended version). CoRR Technical Report arXiv:1603.09291, arXiv.org e-Print archive, 2017. Available at `http://arxiv.org/abs/1603.09291`.

4 Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoc. JSON: data model, query languages and schema specification. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 123–135. ACM, 2017. `doi:10.1145/3034786.3056120`.

5 Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1):3–48, 1995. `doi:10.1016/0304-3975(95)00024-Q`.

6 Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981. `doi:10.1145/322234.322243`.

7 Evgeny Dantsin and Andrei Voronkov. Complexity of query answering in logic databases with complex values. In Sergei I. Adian and Anil Nerode, editors, *Logical Foundations of Computer Science, 4th International Symposium, LFCS'97, Yaroslavl, Russia, July 6-12, 1997, Proceedings*, volume 1234 of *Lecture Notes in Computer Science*, pages 56–66. Springer, 1997. `doi:10.1007/3-540-63045-7_7`.

8 Jan Van den Bussche. Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. *Theor. Comput. Sci.*, 254(1-2):363–377, 2001. `doi:10.1016/S0304-3975(99)00301-1`.

9 Jan Van den Bussche and Jan Paredaens. The expressive power of complex values in object-based data models. *Inf. Comput.*, 120(2):220–236, 1995. `doi:10.1006/inco.1995.1110`.

10 Daniela Florescu and Ghislain Fourny. Jsoniq: The history of a query language. *IEEE Internet Computing*, 17(5):86–90, 2013. `doi:10.1109/MIC.2013.97`.

11 Stéphane Grumbach and Victor Vianu. Tractable query languages for complex object databases. In Daniel J. Rosenkrantz, editor, *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 29-31, 1991, Denver, Colorado, USA*, pages 315–327. ACM Press, 1991. `doi:10.1145/113413.113442`.

12 Jan Hidders, Jan Paredaens, and Jan Van den Bussche. J-logic: Logical foundations for JSON querying. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 137–149. ACM, 2017. `doi:10.1145/3034786.3056106`.

13 Gerhard Jaeschke and Hans-Jörg Schek. Remarks on the algebra of non first normal form relations. In Jeffrey D. Ullman and Alfred V. Aho, editors, *Proceedings of the ACM Symposium on Principles of Database Systems, March 29-31, 1982, Los Angeles, California, USA*, pages 124–138. ACM, 1982. `doi:10.1145/588111.588133`.

14 David S. Johnson. A catalog of complexity classes. In *Handbook of Theoretical Computer Science*, volume A, chapter 2, pages 67–161. Elsevier Science Publishers, 1990.

15 Christoph Koch. On the complexity of nonrecursive XQuery and functional query languages on complex values. *ACM Trans. on Database Systems*, 31(4):1215–1256, 2006. doi:10.1145/1189769.1189771.

16 Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. The SQL++ semi-structured data model and query language: A capabilities survey of SQL-on-Hadoop, NoSQL and NewSQL databases. CoRR Technical Report arXiv:1405.3631, arXiv.org e-Print archive, 2017. Available at http://arxiv.org/abs/1405.3631.

17 Felipe Pezoa, Juan L. Reutter, Fernando Suárez, Martín Ugarte, and Domagoj Vrgoc. Foundations of JSON schema. In Jacqueline Bourdeau, Jim Hendler, Roger Nkambou, Ian Horrocks, and Ben Y. Zhao, editors, *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*, pages 263–273. ACM, 2016. doi:10.1145/2872427.2883029.

18 Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. *J. Data Semantics*, 10:133–173, 2008. doi:10.1007/978-3-540-77688-8_5.

19 Stan J. Thomas and Patrick C. Fischer. Nested relational structures. *Advances in Computing Research*, 3:269–307, 1986.

A Examples and Syntax of the MongoDB Aggregation Framework

The MongoDB *aggregation framework* provides a powerful querying mechanism, in which a query consists of a pipeline of *stages*, each transforming a forest into a new forest. We formalized a core part of this query language consisting of five stages as *MQuery*. In the examples below, we provide all queries both as MQueries and in the actual MongoDB syntax. We assume to have a second document in the `bios` collection as follows:

```
{ "_id": 6,
  "awards": [
    { "award": "Award for the Advancement of Free Software", "year": 2001, "by": "FSF" },
    { "award": "NLUUG Award", "year": 2003, "by": "NLUUG" } ],
  "birth": "1956-01-31",
  "contribs": [ "Python" ],
  "name": { "first": "Guido", "last": "van Rossum" } }
```

A.1 Match

The match operator takes as input a criterion, a Boolean condition on the trees, and returns the trees that satisfy that condition.

▶ **Example 21.** The following MQuery selects trees where the value of the path name.first is Kristen, and there exists an awards path:

$$\texttt{bios} \triangleright \mu_{\text{name.first}=\text{“Kristen”} \wedge \exists \text{awards}}$$

where the corresponding MongoDB query is:

```
db.bios.aggregate([
    {$match: {"name.first": {$eq: "Kristen"},
              "awards": {$exists: true} }} ])
```

This query returns the document about Kristen Nygaard. ◀

▶ **Example 22.** Consider the following query consisting of a match stage with two conditions on keys inside the `awards` array:

$\texttt{bios} \triangleright \mu_{\text{awards.year}=1999 \wedge \text{awards.award}=\text{“Turing Award”}}$

The corresponding MongoDB query is:

```
db.bios.aggregate([
    {$match: {"awards.year": {$eq: 1999},
              "awards.award": {$eq: "Turing Award"} }} ])
```

The query returns all persons that have received an award in 1999, and the Turing award in a possibly different year. Observe that it does not impose that one array element must satisfy all the conditions. This query retrieves the document about Kristen Nygaard because he received an award (the Rosing Prize) in 1999 in addition to the Turing Award (in 2001). ◀

A.2 Unwind

The unwind operator creates a new document for every element in an array.

▶ **Example 23.** The following MQuery unwinds path `awards`:

$\texttt{bios} \triangleright \omega_{\text{awards}}$

The corresponding MongoDB query is:

```
db.bios.aggregate([
    {$unwind: "$awards"} ])
```

When applied to the document about Guido van Rossum, it outputs 2 documents:

```
{ "_id": 6,
  "awards": { "award": "Award for the Advancement of Free Software", "year": 2001,
    "by": "FSF" },
  "birth": "1956-01-31",
  "contribs": [ "Python" ],
  "name": { "first": "Guido", "last": "van Rossum" } },
{ "_id": 6,
  "awards": { "award": "NLUUG Award", "year": 2003, "by": "NLUUG" },
  "birth": "1956-01-31",
  "contribs": [ "Python" ],
  "name": { "first": "Guido", "last": "van Rossum" } }
```

However, unwinding path `birth` in the same document gives the empty result, since the value of this path is not an array. ◀

A.3 Project

The project stage is similar to the extended projection from relational algebra.

▶ **Example 24.** The following query preserves the paths starting with `_id`, `name`, `awards.award` and `awards.year`:

$\texttt{bios} \triangleright \rho_{\texttt{\_id}, \text{name}, \text{awards.award}, \text{awards.year}}$

The corresponding MongoDB query is:

```
db.bios.aggregate([
   {$project: { "name": true, "awards.award": true, "awards.year": true}} ])
```

The document about Kristen Nygaard is then transformed into the document:

```
{ "_id": 4,
  "name": { "first": "Kristen", "last": "Nygaard" },
  "awards": [ { "award": "Rosing Prize", "year": 1999 },
              { "award": "Turing Award", "year": 2001 },
              { "award": "IEEE John von Neumann Medal", "year": 2001 } ] }
```

Observe that `_id` is preserved by MongoDB by default. In our formalization, though, the behaviour of project is the same for all paths. Note also that the information by whom the awards were given is lost as the path `awards.by` was not passed as a parameter. ◀

▶ **Example 25.** Project allows for renaming paths. The following query renames `name.first` to `firstName`, `awards.award` and `awards.year` to `awardsName` and `awardsYear`, respectively, and a non-existing path `abc` to `invisible`:

$$\texttt{bios} \triangleright \rho_{\texttt{\_id},\, \text{firstName/name.first},\, \text{awardsName/awards.award},\, \text{awardsYear/awards.year},\, \text{invisible/abc}}$$

The corresponding MongoDB query is:

```
db.bios.aggregate([
   {$project: {"firstName": "$name.first",
               "awardsName": "$awards.award", "awardsYear": "$awards.year",
               "invisible": "$abc" }} ])
```

It produces from the document about Kristen Nygaard:

```
{ "_id": 4,
  "firstName": "Kristen",
  "awardsName": [ "Rosing Prize", "Turing Award", "IEEE John von Neumann Medal" ],
  "awardsYear": [ 1999, 2001, 2001 ] }
```

Note that in the resulting document `awardsName` and `awardsYear` are two separate arrays unlike in the previous example, where keeping `awards.award` and `awards.year` without renaming them does not create two arrays. Also note that since there is no path `abc` in the input document, the result does not contain `invisible` key. ◀

▶ **Example 26.** Project also allows for creating new values, either fresh or from the existing ones. The following query introduces new keys `occupation` with value "Computer Scientist", `fields` whose value is array consisting of the name, birth date and contributions, `sameFirstAndLastNames` whose value is the Boolean value of a comparison, and `condValue` whose value is calculated based on a condition:

$$\texttt{bios} \triangleright \rho_{\substack{\texttt{\_id},\, \text{occupation/“Computer Scientist”},\, \text{fields/[name, birth, contribs]},\\ \text{sameFirstAndLastNames/(name.first=name.last)},\, \text{condValue/((\_id=4)? contribs: name)}}}$$

The corresponding MongoDB query is:

```
db.bios.aggregate([
    {$project: { "occupation": {$literal: "Computer Scientist"},
                 "fields": ["$name", "$birth", "$contribs"],
                 "sameFirstAndLastNames": {$eq: ["$name.first", "$name.last"]},
                 "condValue": {$cond: {
                     if: {$eq: ["$_id", 4]}, then: "$contribs", else: "$name" }} ])
```

It produces from the documents in the `bios` collection:

```
{ "_id": 4,                          "occupation": "Computer Scientist",
  "fields": [ {"first": "Kristen", "last": "Nygaard"}, "1926-08-27",
    [ "OOP", "Simula" ] ],
  "sameFirstAndLastNames": false,  "condValue": ["OOP", "Simula" ],
{ "_id": 6,                          "occupation": "Computer Scientist",
  "fields": [ {"first": "Guido", "last": "van Rossum"}, "1956-01-31", [ "Python" ] ],
  "sameFirstAndLastNames": false,  "condValue": {"first": "Guido",
    "last": "van Rossum"} }
```

Note that this project stage is a non-well-typed one. First, the array `fields` is not a well-typed array. Second, the types of `condValue` in the two resulting trees do not coincide. This demonstrates that project is a very powerful stage and can produce from a well-typed input forest a non-well-typed one. ◀

A.4 Group

The group stage allows to combine different trees into one. More specifically, the set of input trees is partitioned into several according to a grouping condition, and for each group a single output tree is produced. The documents are grouped together according to the grouping condition, and only the values of paths specified in the aggregation expression are included in the output, combined into an array for each group. Notice that, in the MongoDB syntax, the grouping condition G is specified through the key-value pair `_id` : G.

Let us consider an additional collection, called `awards`, focusing on the award information:

```
{ "_id": 1, "person_id": 4, "name": "Rosing Prize", "in": 1999 },
{ "_id": 2, "person_id": 4, "name": "Turing Award", "in": 2001 },
{ "_id": 3, "person_id": 4, "name": "IEEE John von Neumann Medal", "in": 2001 },
{ "_id": 4, "person_id": 6, "name": "Award for the Advancement of Free Software",
                                                              "in": 2001 },
{ "_id": 5, "person_id": 6, "name": "NLUUG Award", "in": 2003 }
```

▶ **Example 27.** The following query returns for each year the identifiers of scientists that received an award in that year:

$$\texttt{awards} \triangleright \gamma_{\text{year/in}\,:\,\text{scientists/person\_id}}$$

The corresponding MongoDB query is:

```
db.awards.aggregate([
    {$group: { "_id": {"year": "$in"}, "scientists": {$addToSet: "$person_id"} }} ])
```

Running this query over the `awards` collection produces the following output:

```
{ "_id": { "year": 2001 }, "scientists": [4, 6] },
{ "_id": { "year": 1999 }, "scientists": [4] },
{ "_id": { "year": 2003 }, "scientists": [6] }
```

▶ **Example 28.** The following group stage has no aggregation condition, so all input documents are aggregated into one. It returns the names of all the scientists in the `bios` collection:

$$\texttt{bios} \triangleright \gamma_{:\,\text{names/name}}$$

The corresponding MongoDB query is:

```
db.bios.aggregate([
    {$group: { "_id": null, "names": {$addToSet: "$name"} }} ])
```

Running this query over the `bios` collection produces the following output:

```
{ "_id": null,
  "names": [ { "first": "Kristen", "last": "Nygaard" },
             { "first": "Guido", "last": "van Rossum" } ] }
```

▶ **Example 29.** The following query groups persons according to their date of death:

$$\texttt{bios} \triangleright \gamma_{\text{death}\,:\,\text{names/name}}$$

The corresponding MongoDB query is:

```
db.bios.aggregate([
    {$group: { "_id": "$death", "names": {$addToSet: "$name"} }} ])
```

When executing over the `bios` collection, it produces the following output:

```
{ "_id": "2002-08-10",
  "names" : [ { "first": "Kristen", "last": "Nygaard" } ] },
{ "_id": null,
  "names": [ { "first": "Guido", "last": "van Rossum" } ] }
```

Since the `death` path is not present in the document about Guido van Rossum, the latter is grouped in the document where `_id` is **null**. ◀

A.5 Lookup

The lookup stage joins input documents with documents in an external collection, using a local path and a path in the external collection to express the join condition, and stores the matching external documents in an array.

▶ **Example 30.** For each document in the `bios` collection, the following query collects information about the awards received by the scientist from the `awards` collection and stores it in the `awards_info` array:

$$\texttt{bios} \triangleright \lambda^{\text{\_id=awards.person\_id}}_{\text{awards\_info}}$$

The corresponding MongoDB query is:

```
db.bios.aggregate([
 {$lookup: {
     from: "awards", localField: "_id", foreignField: "person_id", as: "awards_info" }}
])
```

Executing this query over the `bios` collection produces the following result:

```
{ "_id": 4,
  "awards": [
    { "award": "Rosing Prize", "year": 1999, "by": "Norwegian Data Association" },
    { "award": "Turing Award", "year": 2001, "by": "ACM" },
    { "award": "IEEE John von Neumann Medal", "year": 2001, "by": "IEEE" } ],
  "birth": "1926-08-27",
  "contribs": [ "OOP", "Simula" ],
  "death": "2002-08-10",
  "name": { "first": "Kristen", "last": "Nygaard" },
  "awards_info": [
    { "_id": 1, "person_id": 4, "name": "Rosing Prize", "in": 1999 },
    { "_id": 2, "person_id": 4, "name": "Turing Award", "in": 2001 },
    { "_id": 3, "person_id": 4, "name": "IEEE John von Neumann Medal", "in": 2001 } ] },
{ "_id": 6,
  "awards": [
    { "award": "Award for the Advancement of Free Software", "year": 2001, "by": "FSF" },
    { "award": "NLUUG Award", "year": 2003, "by": "NLUUG" } ],
  "birth": "1956-01-31",
  "contribs": [ "Python" ],
  "name": { "first": "Guido", "last": "van Rossum" },
  "awards_info": [
    { "_id": 4, "person_id": 6, "name": "Award for the Advancement of Free Software",
                                                          "in": 2001 },
    { "_id": 5, "person_id": 6, "name": "NLUUG Award", "in": 2003 } ] }
```

B Details on the Semantics of tree operations in MQuery

In the following, let $t = (N, E, L_{\mathsf{n}}, L_{\mathsf{e}})$ be a tree. Below, when we mention reachability, we mean reachability along the edge relation.

subtree: *The subtree of t rooted at x and induced by M*, for $x \in M$ and $M \subseteq N$, denoted $\mathsf{subtree}(t, x, M)$, is defined as $(N', E|_{N'\times N'}, L_{\mathsf{n}}|_{N'}, L_{\mathsf{e}}|_{E'})$ where N' is the subset of nodes in M reachable from x by traversing only nodes in M. Note that $\mathsf{subtree}(t, x, M)$ might be the empty tree, e.g., when M is a set of nodes disconnected from x. We write $\mathsf{subtree}(t, M)$ as abbreviation for $\mathsf{subtree}(t, \mathsf{root}(t), M)$.

For a path p with $|[\![p]\!]^t| = 1$, *the subtree* $\mathsf{subtree}(t, p)$ *of t hanging from p* is defined as $\mathsf{subtree}(t, r_p, N')$ where $\{r_p\} = [\![p]\!]^t$, and N' are the nodes reachable from r_p via E. For a path p with $|[\![p]\!]^t| = 0$, $\mathsf{subtree}(t, p)$ is defined as $\mathsf{tree}(\mathbf{null})$.

attach: The *tree* $\mathsf{attach}(k_1 \dots k_n, t)$ *constructed by inserting the path $k_1 \dots k_n$ on top of the tree t*, for $n \geq 1$, is defined as $(N', E', L'_{\mathsf{n}}, L'_{\mathsf{e}})$, where

- **(i)** $N' = N \cup \{x_0, x_1, \dots, x_{n-1}\}$, for fresh $x_0, \dots, x_{n-1}$,
- **(ii)** $E' = E \cup \{(x_0, x_1), (x_1, x_2), \dots, (x_{n-1}, \mathsf{root}(t))\}$,
- **(iii)** $L'_{\mathsf{n}} = L_{\mathsf{n}} \cup \{(x_0, \text{`}\{\!\{\}\!\}\text{'}), \dots, (x_{n-1}, \text{`}\{\!\{\}\!\}\text{'})\}$, and
- **(iv)** $L'_{\mathsf{e}} = L_{\mathsf{e}} \cup \{((x_0, x_1), k_1), \dots, ((x_{n-2}, x_{n-1}), k_{n-1}), ((x_{n-1}, \mathsf{root}(t)), k_n)\}$.

intersection: Let $t_j = (N^j, E^j, L^j_{\mathsf{n}}, L^j_{\mathsf{e}})$, $j = 1, 2$, be trees. The function $t_1 \cap t_2$ returns the set of pairs of nodes $(x_n, y_n) \in N^1 \times N^2$ reachable along identical paths in t_1 and t_2, that is, such that there exist $(x_0, x_1), \ldots, (x_{n-1}, x_n)$ in E^1, for $x_0 = \mathsf{root}(t_1)$, and $(y_0, y_1), \ldots, (y_{n-1}, y_n)$ in E^2, for $y_0 = \mathsf{root}(t_2)$, with $L^1_{\mathsf{n}}(x_i) = L^2_{\mathsf{n}}(y_i)$ and $L^1_{\mathsf{e}}(x_{i-1}, x_i) = L^2_{\mathsf{e}}(y_{i-1}, y_i)$, for $1 \leq i \leq n$.

merge: Let $t_j = (N^j, E^j, L^j_{\mathsf{n}}, L^j_{\mathsf{e}})$, $j = 1, 2$, be trees such that $N^1 \cap N^2 = \emptyset$, and for each path p leading to a leaf in t_2, i.e., $t_2 \models (p = v)$ for some literal value v, we have that $t_1 \not\models \exists p$ and the other way around. Then the *tree* $t_1 \oplus t_2$ *resulting from merging* t_1 *and* t_2 is defined as $(N, E, L_{\mathsf{n}}, L_{\mathsf{e}})$, where

- **(i)** $N = N^1 \cup N^{2'}$, for $N^{2'} = N^2 \setminus \{x_2 \mid (x_1, x_2) \in t_1 \cap t_2\}$,
- **(ii)** $E = E^1 \cup (E^2 \cap (N^{2'} \times N^{2'})) \cup ((t_1 \cap t_2) \circ E^2)$,
- **(iii)** $L_{\mathsf{n}} = L^1_{\mathsf{n}} \cup L^2_{\mathsf{n}}|_{N^{2'}}$, and
- **(iv)** $L_{\mathsf{e}} = L^1_{\mathsf{e}} \cup L^2_{\mathsf{e}}|_{N^{2'} \times N^{2'}} \cup \{((x_1, y_2), \ell) \mid L^2_{\mathsf{e}}(y_1, y_2) = \ell, (x_1, y_1) \in t_1 \cap t_2\}$

replace: Let $t = (N, E, L_{\mathsf{n}}, L_{\mathsf{e}})$ and $t_j = (N^j, E^j, L^j_{\mathsf{n}}, L^j_{\mathsf{e}})$, $j = 1, 2$, be trees such that t_1 is a subtree of t with $\mathsf{root}(t_1) \neq \mathsf{root}(t)$ and N^2 is disjoint from N. Further, let x be the parent of $\mathsf{root}(t_1)$ in t, i.e., $(x, \mathsf{root}(t_1)) \in E$, with $L_{\mathsf{n}}(x, \mathsf{root}(t_1)) = \ell$. Then the tree $\mathsf{replace}(t, t_1, t_2)$ resulting from replacing t_1 by t_2 in t is defined as $(N', E', L'_{\mathsf{n}}, L'_{\mathsf{e}})$, where

- **(i)** $N' = N \setminus N^1 \cup N^2$,
- **(ii)** $E' = E \cap (N' \times N') \cup E^2 \cup \{(x, \mathsf{root}(t_2))\}$,
- **(iii)** $L'_{\mathsf{n}} = L_{\mathsf{n}} \setminus L^1_{\mathsf{n}} \cup L^2_{\mathsf{n}}$, and
- **(iv)** $L'_{\mathsf{e}} = L_{\mathsf{e}}|_{E'} \cup L^2_{\mathsf{e}} \cup \{((x, \mathsf{root}(t_2)), \ell)\}$.

array: Let $\{t_1, \ldots, t_n\}$, $n \geq 0$, be a forest and p a path. The operator $\mathsf{array}(\{t_1, \ldots, t_n\}, p)$ creates the tree encoding the array of the values of the path p in the trees $t_1, \ldots, t_n$. Let $t^p_j = \mathsf{subtree}(t_j, p)$ with $(N^j, E^j, L^j_{\mathsf{n}}, L^j_{\mathsf{e}})$ where all N^j are mutually disjoint, $t_{j_1} \neq t_{j_2}$ for $j_1 \neq j_2$, and $r_j = \mathsf{root}(t^p_j)$, where $1 \leq j \leq m \leq n$ (without loss of generality, we may assume that $t_1, \ldots, t_n$ are ordered accordingly). Then, $\mathsf{array}(\{t_1, \ldots, t_n\}, p)$ is the tree $(N, E, L_{\mathsf{n}}, L_{\mathsf{e}})$ where

- **(i)** $N = \left(\bigcup_{j=1}^n N^j\right) \cup \{v_0\}$,
- **(ii)** $E = \left(\bigcup_{j=1}^n E^j\right) \cup \{(v_0, r_1), \ldots, (v_0, r_n)\}$,
- **(iii)** $L_{\mathsf{n}} = \left(\bigcup_{j=1}^n L^j_{\mathsf{n}}\right) \cup \{(v_0, \text{`[]'})\}$, and
- **(iv)** $L_{\mathsf{e}} = \left(\bigcup_{j=1}^n L^j_{\mathsf{e}}\right) \cup \{((v_0, r_1), 0), \ldots, ((v_0, r_n), n-1)\}$.

We also define $\mathsf{subtree}(t, p)$ for paths p such that $|[\![p]\!]^t| > 1$. In this case it returns the tree encoding the array of all subtrees hanging from p. Formally, $\mathsf{subtree}(t, p) = \mathsf{array}(\{t_1, \ldots, t_n\}, \varepsilon)$, where $\{r_1, \ldots, r_n\} = [\![p]\!]^t$, N_j the set of nodes reachable from r_j via E, and $t_j = \mathsf{subtree}(t, r_j, N_j)$. We observe that the definition of the array operator is recursive as it uses the generalized $\mathsf{subtree}$ operator.

On the Expressive Power of Query Languages for Matrices

Robert Brijder
Hasselt University, Hasselt, Belgium

Floris Geerts
University of Antwerp, Antwerp, Belgium

Jan Van den Bussche
Hasselt University, Hasselt, Belgium

Timmy Weerwag
Hasselt University, Hasselt, Belgium

Abstract

We investigate the expressive power of MATLANG, a formal language for matrix manipulation based on common matrix operations and linear algebra. The language can be extended with the operation inv of inverting a matrix. In MATLANG + inv we can compute the transitive closure of directed graphs, whereas we show that this is not possible without inversion. Indeed we show that the basic language can be simulated in the relational algebra with arithmetic operations, grouping, and summation. We also consider an operation eigen for diagonalizing a matrix, which is defined so that different eigenvectors returned for a same eigenvalue are orthogonal. We show that inv can be expressed in MATLANG + eigen. We put forward the open question whether there are boolean queries about matrices, or generic queries about graphs, expressible in MATLANG + eigen but not in MATLANG + inv. The evaluation problem for MATLANG + eigen is shown to be complete for the complexity class $\exists\mathbf{R}$.

2012 ACM Subject Classification Theory of computation → Database theory, Theory of computation → Database query languages (principles), Computing methodologies → Linear algebra algorithms

Keywords and phrases matrix query languages, relational algebra with aggregates, query evaluation problem, graph queries

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.10

Acknowledgements We thank Bart Kuijpers for telling us about the complexity class $\exists\mathbf{R}$. We thank Lauri Hella and Wied Pakusa for helpful discussions, and Christoph Berkholz and Anuj Dawar for their help with the proof of Proposition 10. R.B. is a postdoctoral fellow of the Research Foundation – Flanders (FWO).

1 Introduction

Data scientists often use matrices to represent their data, as opposed to using the relational data model. These matrices are then manipulated in programming languages such as R or MATLAB. These languages have common operations on matrices built-in, notably matrix multiplication; matrix transposition; elementwise operations on the entries of matrices; solving nonsingular systems of linear equations (matrix inversion); and diagonalization (eigenvalues and eigenvectors). Such programming languages trace back to the APL language [20].

Providing database support for matrices and multidimensional arrays has been a long-standing research topic [31], originally geared towards applications in scientific data management, and more recently motivated by machine learning over big data [5, 36, 8, 29].

Database theory and finite model theory provide a rich picture of the expressive power of query languages [1, 24]. In this paper we would like to bring matrix languages into this picture. There is a lot of current interest in languages that combine matrix operations with relational query languages or logics, both in database systems [19] and in finite model theory [10, 11, 18]. In the present study, however, we focus on matrices alone. Indeed, given their popularity, we believe the expressive power of matrix sublanguages also deserves to be understood in its own right.

The contents of this paper can be introduced as follows. We begin the paper by defining the language MATLANG as an analog for matrices of the relational algebra for relations. This language is based on five elementary operations reflecting basic matrix operations available in R, namely, the one-vector; turning a vector in a diagonal matrix; matrix multiplication; matrix transposition; and pointwise function application. We give examples showing that this basic language is capable of expressing common matrix manipulations. For example, the Google matrix of any directed graph G can be computed in MATLANG, starting from the adjacency matrix of G.

Well-typedness and well-definedness notions of MATLANG expressions are captured via a simple data model for matrices. In analogy to the relational model, a schema consists of a number of matrix names, and an instance assigns matrices to the names. Recall that in a relational schema, a relation name is typed by a set of attribute symbols. In our case, a matrix name is typed by a pair $\alpha \times \beta$, where α and β are size symbols that indicate, in a generic manner, the number of rows and columns of the matrix.

In Section 3 we show that our language can be simulated in the relational algebra with aggregates [23, 28], using a standard representation of matrices as relations. The only aggregate function that is needed is summation. In fact, MATLANG is already subsumed by aggregate logic with only three nonnumerical variables. Conversely, MATLANG can express all queries from graph databases (binary relational structures) to binary relations that can be expressed in first-order logic with three variables. In contrast, the four-variable query asking if the graph contains a four-clique, is not expressible.

In Section 4 we extend MATLANG with an operation for inverting a matrix, and we show that the extended language is strictly more expressive. Indeed, the transitive closure of binary relations becomes expressible. The possibility of reducing transitive closure to matrix inversion has been pointed out by several researchers [26, 9, 33]. We show that the restricted setting of MATLANG suffices for this reduction to work. That transitive closure is not expressible without inversion, follows from the locality of relational algebra with aggregates [28].

Another prominent operation of linear algebra, with many applications in data mining and graph analysis [16, 27], is to return eigenvectors and eigenvalues. There are various ways to define this operator formally. In Section 5 we define the operation eigen to return a basis of eigenvectors, in which eigenvectors for a same eigenvalue are orthogonal. We show that the resulting language MATLANG + eigen can express inversion. The argument is well known from linear algebra, but our result shows that it can be carried out in MATLANG, once more attesting that we have defined an adequate matrix language. It is natural to conjecture that MATLANG + eigen is actually strictly more powerful than MATLANG + inv in expressing, say, boolean queries about matrices. Proving this is an interesting open problem.

Finally, in Section 6 we look into the evaluation problem for MATLANG+eigen expressions. In practice, matrix computations are performed using techniques from numerical mathematics [14]. It remains of foundational interest, however, to know whether the evaluation of expressions is effectively computable. We need to define this problem with some care, since we work with arbitrary complex numbers. Even if the inputs are, say, 0-1 matrices, the outputs of the eigen operation can be complex numbers. Moreover, until now we have allowed arbitrary pointwise functions, which we should restrict somehow if we want to discuss computability. Our approach is to restrict pointwise functions to be semi-algebraic, i.e., definable over the real numbers. We will observe that the input-output relation of an expression e, applied to input matrices of given dimensions, is definable in the existential theory of the real numbers, by a formula of size polynomial in the size of e and the given dimensions. This places natural decision versions of the evaluation problem for MATLANG + eigen in the complexity class $\exists\mathbf{R}$ (combined complexity). We show moreover that there exists a fixed expression (data complexity) for which the evaluation problem is $\exists\mathbf{R}$-complete, even restricted to input matrices with integer entries. It also follows that equivalence of expressions, over inputs of given dimensions, is decidable.

2 MATLANG

We assume a sufficient supply of *matrix variables*, which serve to indicate the inputs to expressions in MATLANG. Variables can also be introduced in let-constructs inside expressions. The syntax of MATLANG expressions is defined by the grammar:

$e ::= M$		(matrix variable)
	$\mathsf{let}\ M = e_1\ \mathsf{in}\ e_2$	(local binding)
|	e^*	(conjugate transpose)
|	$\mathbf{1}(e)$	(one-vector)
|	$\mathsf{diag}(e)$	(diagonalization of a vector)
|	$e_1 \cdot e_2$	(matrix multiplication)
|	$\mathsf{apply}[f](e_1, \ldots, e_n)$	(pointwise application, $f \in \Omega$)

In the last rule, f is the name of a function $f : \mathbf{C}^n \to \mathbf{C}$, where $\mathbf{C}$ denotes the complex numbers. Formally, the syntax of MATLANG is parameterized by a repertoire Ω of such functions, but for simplicity we will not reflect this in the notation.

▶ **Example 1.** Let $c \in \mathbf{C}$ be a constant; we also use c as a name for the constant function $c : \mathbf{C} \to \mathbf{C} : z \mapsto c$. Then

$$\mathsf{let}\ N = \mathbf{1}(M)^*\ \mathsf{in}\ \ \mathsf{apply}[c](\mathbf{1}(N))$$

is an example of an expression. At this point, this is a purely syntactical example; we will see its semantics shortly. The expression is actually equivalent to $\mathsf{apply}[c](\mathbf{1}(\mathbf{1}(M)^*))$. The let-construct is useful to give names to intermediate results, but is not essential for now. It will become essential later, when we enrich MATLANG with the eigen operation. ◀

In defining the semantics of the language, we begin by defining the basic matrix operations. Following practical matrix sublanguages such as R or MATLAB, we will work throughout with matrices over the complex numbers. However, a real-number version of the language could be defined as well.

$$\begin{pmatrix} 0 & 1+i \\ 2 & 3-i \\ 4+4i & 5 \end{pmatrix}^* = \begin{pmatrix} 0 & 2 & 4-4i \\ 1-i & 3+i & 5 \end{pmatrix} \qquad \mathbf{1}\begin{pmatrix} 2 & 3 & 4 \\ 4 & 5 & 6 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \cdot \begin{pmatrix} 6 & 5 & 4 & 3 \\ 2 & 1 & 0 & -1 \end{pmatrix} = \begin{pmatrix} 10 & 7 & 4 & 1 \\ 26 & 19 & 12 & 5 \\ 42 & 31 & 20 & 9 \end{pmatrix} \qquad \mathsf{diag}\begin{pmatrix} 6 \\ 7 \end{pmatrix} = \begin{pmatrix} 6 & 0 \\ 0 & 7 \end{pmatrix}$$

$$\mathsf{apply}[\dot{-}]\left(\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}\right) = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

Figure 1 Basic matrix operations of MATLANG. The matrix multiplication example is taken from Axler's book [3].

Transpose: If A is a matrix then A^* is its conjugate transpose. So, if A is an $m \times n$ matrix then A^* is an $n \times m$ matrix and the entry $A^*_{i,j}$ is the complex conjugate of the entry $A_{j,i}$.

One-vector: If A is an $m \times n$ matrix then $\mathbf{1}(A)$ is the $m \times 1$ column vector consisting of all ones.

Diag: If v is an $m \times 1$ column vector then $\mathsf{diag}(v)$ is the $m \times m$ diagonal square matrix with v on the diagonal and zero everywhere else.

Matrix multiplication: If A is an $m \times n$ matrix and B is an $n \times p$ matrix then the well known matrix multiplication AB is defined to be the $m \times p$ matrix where $(AB)_{i,j} = \sum_{k=1}^{n} A_{i,k}B_{k,j}$. In MATLANG we explicitly denote this as $A \cdot B$.

Pointwise application: If $A^{(1)}, \ldots, A^{(n)}$ are matrices of the same dimensions $m \times p$, then $\mathsf{apply}[f](A^{(1)}, \ldots, A^{(n)})$ is the $m \times p$ matrix C where $C_{i,j} = f(A^{(1)}_{i,j}, \ldots, A^{(n)}_{i,j})$.

▶ **Example 2.** The operations are illustrated in Figure 1. In the pointwise application example, we use the function $\dot{-}$ defined by $x \dot{-} y = x - y$ if x and y are both real numbers and $x \geq y$, and $x \dot{-} y = 0$ otherwise.

2.1 Formal semantics

The formal semantics of expressions is defined in a straightforward manner, as shown in Figure 2. An *instance* I is a function, defined on a nonempty finite set $\mathrm{var}(I)$ of matrix variables, that assigns a matrix to each element of $\mathrm{var}(I)$. Figure 2 provides the rules that allow to derive that an expression e, on an instance I, successfully evaluates to a matrix A. We denote this success by $e(I) = A$. The reason why an evaluation may not succeed can be found in the rules that have a condition attached to them. The rule for variables fails when an instance simply does not provide a value for some input variable. The rules for diag, apply, and matrix multiplication have conditions on the dimensions of matrices, that need to be satisfied for the operations to be well-defined.

▶ **Example 3** (Scalars)**.** The expression from Example 1, regardless of the matrix assigned to M, evaluates to the 1×1 matrix whose single entry equals c. We introduce the shorthand c for this constant expression. Obviously, in practice, scalars would be built in the language and would not be computed in such a roundabout manner. In this paper, however, we are interested in expressiveness, so we start from a minimal language and then see what is already expressible in this language.

$$\frac{M \in \mathrm{var}(I)}{M(I) = I(M)} \qquad \frac{e_1(I) = A \qquad e_2(I[M := A]) = B}{(\mathsf{let}\ M = e_1\ \mathsf{in}\ e_2)(I) = B} \qquad \frac{e(I) = A}{e^*(I) = A^*} \qquad \frac{e(I) = A}{\mathbf{1}(e)(I) = \mathbf{1}(A)}$$

$$\frac{e(I) = A \qquad A \text{ is a column vector}}{\mathsf{diag}(e)(I) = \mathsf{diag}(A)}$$

$$\frac{e_1(I) = A \qquad e_2(I) = B \qquad \text{number of columns of } A \text{ equals the number of rows of } B}{e_1 \cdot e_2(I) = A \cdot B}$$

$$\frac{\forall k = 1, \ldots, n : (e_k(I) = A_k) \qquad \text{all } A_k \text{ have the same dimensions}}{\mathsf{apply}[f](e_1, \ldots, e_n)(I) = \mathsf{apply}[f](A_1, \ldots, A_n)}$$

Figure 2 Big-step operational semantics of MATLANG. The notation $I[M := A]$ denotes the instance that is equal to I, except that M is mapped to the matrix A.

▶ **Example 4** (Scalar multiplication)**.** Let A be any matrix and let C be a 1×1 matrix; let c be the value of C's single entry. Viewing C as a scalar, we define the operation $C \odot A$ as multiplying every entry of A by c. We can express $C \odot A$ as

$$\mathsf{let}\ M = \mathbf{1}(A) \cdot C \cdot \mathbf{1}(A^*)^*\ \mathsf{in}\ \mathsf{apply}[\times](M, A).$$

If A is an $m \times n$ matrix, we compute in variable M the $m \times n$ matrix where every entry equals c. Then pointwise multiplication is used to do the scalar multiplication.

▶ **Example 5** (Google matrix)**.** Let A be the adjacency matrix of a directed graph (modeling the Web graph) on n nodes numbered $1, \ldots, n$. Let $0 < d < 1$ be a fixed "damping factor". Let k_i denote the outdegree of node i. For simplicity, we assume k_i is nonzero for every i. Then the Google matrix [7, 6] of A is the $n \times n$ matrix G defined by

$$G_{i,j} = d\frac{A_{ij}}{k_i} + \frac{1-d}{n}.$$

The calculation of G from A can be expressed in MATLANG as follows:

$$\begin{array}{l}\mathsf{let}\ J = \mathbf{1}(A) \cdot \mathbf{1}(A)^*\ \mathsf{in}\\ \mathsf{let}\ K = A \cdot J\ \mathsf{in}\\ \mathsf{let}\ B = \mathsf{apply}[/](A, K)\ \mathsf{in}\\ \mathsf{let}\ N = \mathbf{1}(A)^* \cdot \mathbf{1}(A)\ \mathsf{in}\\ \mathsf{apply}[+](d \odot B, (1-d) \odot \mathsf{apply}[1/x](N) \odot J)\end{array}$$

In variable J we compute the $n \times n$ matrix where every entry equals one. In K we compute the $n \times n$ matrix where all entries in the ith row equal k_i. In N we compute the 1×1 matrix containing the value n. The pointwise functions applied are addition, division, and reciprocal. We use the shorthand for constants (d and $1 - d$) from Example 3, and the shorthand $\odot$ for scalar multiplication from Example 4.

▶ **Example 6** (Minimum of a vector). Let $v = (v_1, \ldots, v_n)^*$ be a column vector of real numbers; we would like to extract the minimum from v. This can be done as follows:

let $V = v \cdot \mathbf{1}(v)^*$ in
let $C = \mathsf{apply}[\leq](V, V^*) \cdot \mathbf{1}(v)$ in
let $N = \mathbf{1}(v)^* \cdot \mathbf{1}(v)$ in
let $S = \mathsf{apply}[=](C, \mathbf{1}(v) \cdot N)$ in
let $M = \mathsf{apply}[1/x](S^* \cdot \mathbf{1}(v))$ in
$M \cdot v^* \cdot S$

The pointwise functions applied are $\leq$, which returns 1 on (x, y) if $x \leq y$ and 0 otherwise; $=$, defined analogously; and the reciprocal function. In variable V we compute a square matrix holding n copies of v. Then in variable C we compute the $n \times 1$ column vector where C_i counts the number of v_j such that $v_i \leq v_j$. If $C_i = n$ then v_i equals the minimum. Variable N computes the scalar n and column vector S is a selector where $S_i = 1$ if v_i equals the minimum, and $S_i = 0$ otherwise. Since the minimum may appear multiple times in v, we compute in M the inverse of the multiplicity. Finally we sum the different occurrences of the minimum in v and divide by the multiplicity.

2.2 Types and schemas

We have already remarked that, due to conditions on the dimensions of matrices, MATLANG expressions are not well-defined on all instances. For example, if I is an instance where $I(M)$ is a 3×4 matrix and $I(N)$ is a 2×4 matrix, then the expression $M \cdot N$ is not defined on I. The expression $M \cdot N^*$, however, is well-defined on I. We now introduce a notion of schema, which assigns types to matrix names, so that expressions can be type-checked against schemas.

Our types need to be able to guarantee equalities between numbers of rows or numbers of columns, so that apply and matrix multiplication can be typechecked. Our types also need to be able to recognize vectors, so that diag can be typechecked.

Formally, we assume a sufficient supply of *size symbols*, which we will denote by the letters α, β, γ. A size symbol represents the number of rows or columns of a matrix. Together with an explicit 1, we can indicate arbitrary matrices as $\alpha \times \beta$, square matrices as $\alpha \times \alpha$, column vectors as $\alpha \times 1$, row vectors as $1 \times \alpha$, and scalars as 1×1. Formally, a *size term* is either a size symbol or an explicit 1. A *type* is then an expression of the form $s_1 \times s_2$ where s_1 and s_2 are size terms. Finally, a *schema* $\mathcal{S}$ is a function, defined on a nonempty finite set $\mathrm{var}(\mathcal{S})$ of matrix variables, that assigns a type to each element of $\mathrm{var}(\mathcal{S})$.

The typechecking of expressions is now shown in Figure 3. The figure provides the rules that allow to infer an output type τ for an expression e over a schema $\mathcal{S}$. To indicate that a type can be successfully inferred, we use the notation $\mathcal{S} \vdash e : \tau$. When we cannot infer a type, we say e is not well-typed over $\mathcal{S}$. For example, when $\mathcal{S}(M) = \alpha \times \beta$ and $\mathcal{S}(N) = \gamma \times \beta$, then the expression $M \cdot N$ is not well-typed over $\mathcal{S}$. The expression $M \cdot N^*$, however, is well-typed with output type $\alpha \times \gamma$.

To establish the soundness of the type system, we need a notion of conformance of an instance to a schema.

Formally, a *size assignment* σ is a function from size symbols to positive natural numbers. We extend σ to any size term by setting $\sigma(1) = 1$. Now, let $\mathcal{S}$ be a schema and I an instance with $\mathrm{var}(I) = \mathrm{var}(\mathcal{S})$. We say that I is an instance of $\mathcal{S}$ if there is a size assignment σ such that for all $M \in \mathrm{var}(\mathcal{S})$, if $\mathcal{S}(M) = s_1 \times s_2$, then $I(M)$ is a $\sigma(s_1) \times \sigma(s_2)$ matrix. In that case we also say that I *conforms* to $\mathcal{S}$ by the size assignment σ.

$$\frac{M \in \mathrm{var}(\mathcal{S})}{\mathcal{S} \vdash M : \mathcal{S}(M)} \qquad \frac{\mathcal{S} \vdash e_1 : \tau_1 \qquad \mathcal{S}[M := \tau_1] \vdash e_2 : \tau_2}{\mathcal{S} \vdash \mathsf{let}\ M = e_1\ \mathsf{in}\ e_2 : \tau_2} \qquad \frac{\mathcal{S} \vdash e : s_1 \times s_2}{\mathcal{S} \vdash e^* : s_2 \times s_1}$$

$$\frac{\mathcal{S} \vdash e : s_1 \times s_2}{\mathcal{S} \vdash \mathbf{1}(e) : s_1 \times 1} \qquad \frac{\mathcal{S} \vdash e : s \times 1}{\mathcal{S} \vdash \mathsf{diag}(e) : s \times s} \qquad \frac{\mathcal{S} \vdash e_1 : s_1 \times s_2 \qquad \mathcal{S} \vdash e_2 : s_2 \times s_3}{\mathcal{S} \vdash e_1 \cdot e_2 : s_1 \times s_3}$$

$$\frac{n > 0 \qquad f : \mathbf{C}^n \to \mathbf{C} \qquad \forall k = 1, \ldots, n : (\mathcal{S} \vdash e_k : \tau)}{\mathcal{S} \vdash \mathsf{apply}[f](e_1, \ldots, e_n) : \tau}$$

Figure 3 Typechecking MATLANG. The notation $\mathcal{S}[M := \tau]$ denotes the schema that is equal to $\mathcal{S}$, except that M is mapped to the type τ.

We now obtain the following obvious but desirable property.

▶ **Proposition 7** (Safety). *If $\mathcal{S} \vdash e : s_1 \times s_2$, then for every instance I conforming to $\mathcal{S}$, by size assignment σ, the matrix $e(I)$ is well-defined and has dimensions $\sigma(s_1) \times \sigma(s_2)$.*

3 Expressive power of MATLANG

It is natural to represent an $m \times n$ matrix A by a ternary relation

$$Rel_2(A) := \{(i, j, A_{i,j}) \mid i \in \{1, \ldots, m\},\ j \in \{1, \ldots, n\}\}.$$

In the special case where A is an $m \times 1$ matrix (column vector), A can also be represented by a binary relation $Rel_1(A) := \{(i, A_{i,1}) \mid i \in \{1, \ldots, m\}\}$. Similarly, a $1 \times n$ matrix (row vector) A can be represented by $Rel_1(A) := \{(j, A_{1,j}) \mid j \in \{1, \ldots, n\}\}$. Finally, a 1×1 matrix (scalar) A can be represented by the unary singleton relation $Rel_0(A) := \{(A_{1,1})\}$.

Note that in MATLANG, we perform calculations on matrix entries, but not on row or column indices. This fits well to the relational model with aggregates as formalized by Libkin [28]. In this model, the columns of relations are typed as "base", indicated by **b**, or "numerical", indicated by **n**. In the relational representations of matrices presented above, the last column is of type **n** and the other columns (if any) are of type **b**. In particular, in our setting, numerical columns hold complex numbers.

Given this representation of matrices by relations, MATLANG can be simulated in the relational algebra with aggregates. Actually, the only aggregate operation we need is summation. We will not reproduce the formal definition of the relational algebra with summation [28], but note the following salient points:

- Expressions are built up from relation names using the classical operations union, set difference, cartesian product ($\times$), selection (σ), and projection (π), plus two new operations: *function application* and *summation*.
- For selection, we only use equality and nonequality comparisons on base columns. No selection on numerical columns will be needed in our setting.
- For any function $f : \mathbf{C}^n \to \mathbf{C}$, the operation $\mathsf{apply}[f; i_1, \ldots, i_n]$ can be applied to any relation r having columns $i_1, \ldots, i_n$, which must be numerical. The result is the relation $\{(t, f(t(i_1), \ldots, t(i_n))) \mid t \in r\}$, appending a numerical column to r. We allow $n = 0$, in which case f is a constant.

- The operation $\mathsf{sum}[i; i_1, \ldots, i_n]$ can be applied to any relation r having columns i, i_1, …, i_n, where column i must be numerical. In our setting we only need the operation in cases where columns i_1, …, i_n are base columns. The result of the operation is the relation

$$\{(t(i_1), \ldots, t(i_n), \sum_{t' \in \mathsf{group}[i_1,\ldots,i_n](r,t)} t'(i)) \mid t \in r\},$$

where

$$\mathsf{group}[i_1, \ldots, i_n](r, t) = \{t' \in r \mid t'(i_1) = t(i_1) \wedge \cdots \wedge t'(i_n) = t(i_n)\}.$$

Again, n can be zero, in which case the result is a singleton.

3.1 From MATLANG to relational algebra with summation

To state the translation formally, we assume a supply of *relation variables*, which, for convenience, we can take to be the same as the matrix variables. A *relation type* is a tuple of **b**'s and **n**'s. A *relational schema* $\mathcal{S}$ is a function, defined on a nonempty finite set $\mathrm{var}(\mathcal{S})$ of relation variables, that assigns a relation type to each element of $\mathrm{var}(\mathcal{S})$.

One can define well-typedness for expressions in the relation algebra with summation, and define the output type. We omit this definition here, as it follows a well-known methodology [37] and is analogous to what we have already done for MATLANG in Section 2.2.

To define relational instances, we assume a countably infinite universe **dom** of abstract atomic data elements. It is convenient to assume that the natural numbers are contained in **dom**. We stress that this assumption is not essential but simplifies the presentation. Alternatively, we would have to work with explicit embeddings from the natural numbers into **dom**.

Let τ be a relation type. A *tuple of type* τ is a tuple $(t(1), \ldots, t(n))$ of the same arity as τ, such that $t(i) \in \mathbf{dom}$ when $\tau(i) = \mathbf{b}$, and $t(i)$ is a complex number when $\tau(i) = \mathbf{n}$. A *relation of type* τ is a finite set of tuples of type τ. An *instance* of a relational schema $\mathcal{S}$ is a function I defined on $\mathrm{var}(\mathcal{S})$ so that $I(R)$ is a relation of type $\mathcal{S}(R)$ for every $R \in \mathrm{var}(\mathcal{S})$.

We must connect the matrix data model to the relational data model. Let $\tau = s_1 \times s_2$ be a matrix type. Let us call τ a *general type* if s_1 and s_2 are both size symbols; a *vector type* if s_1 is a size symbol and s_2 is 1, or vice versa; and the *scalar type* if τ is 1×1. To every matrix type τ we associate a relation type

$$Rel(\tau) := \begin{cases} (\mathbf{b}, \mathbf{b}, \mathbf{n}) & \text{if } \tau \text{ is general;} \\ (\mathbf{b}, \mathbf{n}) & \text{if } \tau \text{ is a vector type;} \\ (\mathbf{n}) & \text{if } \tau \text{ is scalar.} \end{cases}$$

Then to every matrix schema $\mathcal{S}$ we associate the relational schema $Rel(\mathcal{S})$ where $Rel(\mathcal{S})(M) = Rel(\mathcal{S}(M))$ for every $M \in \mathrm{var}(\mathcal{S})$. For each instance I of $\mathcal{S}$, we define the instance $Rel(I)$ over $Rel(\mathcal{S})$ by

$$Rel(I)(M) = \begin{cases} Rel_2(I(M)) & \text{if } \mathcal{S}(M) \text{ is a general type;} \\ Rel_1(I(M)) & \text{if } \mathcal{S}(M) \text{ is a vector type;} \\ Rel_0(I(M)) & \text{if } \mathcal{S}(M) \text{ is the scalar type.} \end{cases}$$

Here we use the relational representations Rel_2, Rel_1 and Rel_0 of matrices introduced in the beginning of Section 3.

▶ **Theorem 8.** *Let $\mathcal{S}$ be a matrix schema, and let e be a* MATLANG *expression that is well-typed over $\mathcal{S}$ with output type τ. Let $\ell = 2$, 1, or 0, depending on whether τ is general, a vector type, or scalar, respectively.*

1. *There exists an expression $Rel(e)$ in the relational algebra with summation, that is well-typed over $Rel(\mathcal{S})$ with output type $Rel(\tau)$, such that for every instance I of $\mathcal{S}$, we have $Rel_\ell(e(I)) = Rel(e)(Rel(I))$.*
2. *The expression $Rel(e)$ uses neither set difference, nor selection conditions on numerical columns.*
3. *The only functions used in $Rel(e)$ are those used in pointwise applications in e; complex conjugation; multiplication of two numbers; and the constant functions 0 and 1.*

Proof. We only give a few representative examples.

- If M is of type $\alpha \times \beta$ then $Rel(M^*)$ is $\mathsf{apply}[\overline{z}; 3]\, \pi_{2,1,3}(M)$, where $\overline{z}$ is the complex conjugate. If M is of type $\alpha \times 1$, however, $Rel(M^*)$ is $\mathsf{apply}[\overline{z}; 2](M)$.
- If M is of type $1 \times \alpha$ then $Rel(\mathbf{1}(M))$ is $\pi_3(\mathsf{apply}[1; 2](M))$. Here, 1 stands for the constant 1 function.
- If M is of type $\alpha \times 1$ then $Rel(\mathsf{diag}(M))$ is

$$\sigma_{\$1=\$2}(\pi_1(M) \times M) \cup \mathsf{apply}[0; \,]\, \sigma_{\$1 \neq \$2}(\pi_1(M) \times \pi_1(M)).$$

- If M is of type $\alpha \times \beta$ and N is of type $\beta \times \gamma$, then $Rel(M \cdot N)$ is

$$\mathsf{sum}[7; 1, 5]\ \mathsf{apply}[\times; 3, 6]\, \sigma_{\$2=\$4}(M \times N).$$

  If, however, M is of type $\alpha \times 1$ and N is of type 1×1, then $Rel(M \cdot N)$ is

$$\pi_{1,4}\ \mathsf{apply}[\times; 2, 3](M \times N).$$

  We use pointwise multiplication.
- If M and N are of type $1 \times \beta$ then $Rel(\mathsf{apply}[f](M, N))$ is $\pi_{1,5}\ \mathsf{apply}[f; 2, 4]\, \sigma_{\$1=\$3}(M \times N)$.

We may ignore the let-construct as it does not add expressive power. ◀

▶ Remark. The different treatment of general types, vector types, and scalar types is necessary because in our version of the relational algebra, selections can only compare base columns for equality; in particular we can not select for the value 1.

▶ Remark. We can sharpen the above theorem a bit if we work in the relational calculus with aggregates. Every MATLANG expression can already be expressed by a formula in the relational calculus with summation that uses only three distinct base variables (variables ranging over values in base columns).

3.2 Expressing graph queries

So far we have looked at expressing matrix queries in terms of relational queries. It is also natural to express relational queries as matrix queries. This works best for binary relations, or graphs, which we can represent by their adjacency matrices.

Formally, define a *graph schema* to be a relational schema where every relation variable is assigned the type $(\mathbf{b}, \mathbf{b})$ of arity two. We define a *graph instance* as an instance I of a graph schema, where the active domain of I equals $\{1, \ldots, n\}$ for some positive natural number n. The assumption that the active domain always equals an initial segment of the natural numbers is convenient for forming the bridge to matrices. This assumption, however, is not essential for our results to hold. Indeed, the logics we consider do not have any built-in

predicates on base variables, besides equality. Hence, they view the active domain elements as abstract data values.

To every graph schema $\mathcal{S}$ we associate a matrix schema $Mat(\mathcal{S})$, where $Mat(\mathcal{S})(R) = \alpha \times \alpha$ for every $R \in \text{var}(\mathcal{S})$, for a fixed size symbol α. So, all matrices are square matrices of the same dimension. Let I be a graph instance of $\mathcal{S}$, with active domain $\{1, \ldots, n\}$. We will denote the $n \times n$ adjacency matrix of a binary relation r over $\{1, \ldots, n\}$ by $Adj_I(r)$. Now any such instance I is represented by the matrix instance $Mat(I)$ over $Mat(\mathcal{S})$, where $Mat(I)(R) = Adj_I(I(R))$ for every $R \in \text{var}(\mathcal{S})$.

A *graph query* over a graph schema $\mathcal{S}$ is a function that maps each graph instance I of $\mathcal{S}$ to a binary relation on the active domain of I. We say that a MATLANG expression e *expresses* the graph query q if e is well-typed over $Mat(\mathcal{S})$ with output type $\alpha \times \alpha$, and for every graph instance I of $\mathcal{S}$, we have $Adj_I(q(I)) = e(Mat(I))$.

We can now give a partial converse to Theorem 8. We assume active-domain semantics for first-order logic [1]. Please note that the following result deals only with pure first-order logic, without aggregates or numerical columns.

▶ **Theorem 9.** *Every graph query expressible in* FO^3 *(first-order logic with equality, using at most three distinct variables) is expressible in* MATLANG. *The only functions needed in pointwise applications are boolean functions on* $\{0, 1\}$*, and testing if a number is positive.*

We can complement the above theorem by showing that the quintessential first-order query requiring four variables is not expressible.

▶ **Proposition 10.** *The graph query over a single binary relation R that maps I to $I(R)$ if $I(R)$ contains a four-clique, and to the empty relation otherwise, is not expressible in* MATLANG.

4 Matrix inversion

Matrix inversion (solving nonsingular systems of linear equations) is an ubiquitous operation in data analysis. We can extend MATLANG with matrix inversion as follows. Let $\mathcal{S}$ be a schema and e be an expression that is well-typed over $\mathcal{S}$, with output type of the form $\alpha \times \alpha$. Then the expression e^{-1} is also well-typed over $\mathcal{S}$, with the same output type $\alpha \times \alpha$. The semantics is defined as follows. For an instance I, if $e(I)$ is an invertible matrix, then $e^{-1}(I)$ is defined to be the inverse of $e(I)$; otherwise, it is defined to be the zero square matrix of the same dimensions as $e(I)$. The extension of MATLANG with inversion is denoted by MATLANG + inv.

▶ **Example 11** (PageRank)**.** Recall Example 5 where we computed the Google matrix of A. In the process we already showed how to compute the $n \times n$ matrix B defined by $B_{i,j} = A_{i,j}/k_i$, and the scalar N holding the value n. So, in the following expression, we assume we already have B and N. Let I be the $n \times n$ identity matrix, and let $\mathbf{1}$ denote the $n \times 1$ column vector consisting of all ones. The PageRank vector v of A can be computed as follows [12]:

$$v = \frac{1-d}{n}(I - dB)^{-1}\mathbf{1}.$$

This calculation is readily expressed in MATLANG + inv as

$$(1-d) \odot \mathsf{apply}[1/x](N) \odot \mathsf{apply}[-](\mathsf{diag}(\mathbf{1}(A)), d \odot B)^{-1} \cdot \mathbf{1}(A).$$

▶ **Example 12** (Transitive closure). We next show that the reflexive-transitive closure of a binary relation is expressible in MATLANG + inv. Let A be the adjacency matrix of a binary relation r on $\{1,\ldots,n\}$. Let I be the $n \times n$ identity matrix, expressible as $\mathsf{diag}(\mathbf{1}(A))$. From earlier examples we know how to compute the scalar 1×1 matrix N holding the value n. The matrix $B = \frac{1}{n+1}A$ has 1-norm strictly less than 1, so $S = \sum_{k=0}^{\infty} B^k$ converges, and is equal to $(I - B)^{-1}$ [14, Lemma 2.3.3]. Now (i, j) belongs to the reflexive-transitive closure of r if and only if $S_{i,j}$ is nonzero. Thus, we can express the reflexive-transitive closure of r as

$$\mathsf{apply}[\neq 0]\big(\mathsf{apply}[-](\mathsf{diag}(\mathbf{1}(A)), \mathsf{apply}[1/(x+1)](N) \odot A)^{-1}\big),$$

where $x \neq 0$ is 1 if $x \neq 0$ and 0 otherwise. We can obtain the transitive closure by multiplying the above expression with A. ◀

By Theorem 8, any graph query expressible in MATLANG is expressible in the relational algebra with aggregates. It is known [17, 28] that such queries are local. The transitive-closure query from Example 12, however, is not local. We thus conclude:

▶ **Theorem 13.** MATLANG + inv *is strictly more powerful than* MATLANG *in expressing graph queries.*

Once we have the transitive closure, we can do many other things such as checking bipartiteness of undirected graphs, checking connectivity, checking cyclicity. MATLANG is expressive enough to reduce these queries to the transitive-closure query, as shown in the following example for bipartiteness. The same approach via FO^3 can be used for connectedness or cyclicity.

▶ **Example 14** (Bipartiteness). To check bipartiteness of an undirected graph, given as a symmetric binary relation R without self-loops, we first compute the transitive closure T of the composition of R with itself. Then the FO^3 condition $\neg\exists x\exists y(R(x, y) \wedge T(y, x))$ expresses that R is bipartite (no odd cycles). The result now follows from Theorem 9.

▶ **Example 15** (Number of connected components). Using transitive closure we can also easily compute the number of connected components of a binary relation R on $\{1,\ldots,n\}$, given as an adjacency matrix. We start from the union of R and its converse. This union, denoted by S, is expressible by Theorem 9. We then compute the reflexive-transitive closure C of S. Now the number of connected components of R equals $\sum_{i=1}^{n} 1/k_i$, where k_i is the degree of node i in C. This sum is simply expressible as $\mathbf{1}(C)^* \cdot \mathsf{apply}[1/x](C \cdot \mathbf{1}(C))$.

5 Eigenvalues

Another workhorse in data analysis is diagonalizing a matrix, i.e., finding a basis of eigenvectors. Formally, we define the operation eigen as follows. Let A be an $n \times n$ matrix. Recall that A is called diagonalizable if there exists a basis of $\mathbf{C}^n$ consisting of eigenvectors of A. In that case, there also exists such a basis where eigenvectors corresponding to a same eigenvalue are orthogonal. Accordingly, we define $\mathsf{eigen}(A)$ to return an $n \times n$ matrix, the columns of which form a basis of $\mathbf{C}^n$ consisting of eigenvectors of A, where eigenvectors corresponding to a same eigenvalue are orthogonal. If A is not diagonalizable, we define $\mathsf{eigen}(A)$ to be the $n \times n$ zero matrix.

Note that eigen is nondeterministic; in principle there are infinitely many possible results. This models the situation in practice where numerical packages such as R or MATLAB return approximations to the eigenvalues and a set of corresponding eigenvectors. Eigenvectors,

however, are not unique. Hence, some care must be taken in extending MATLANG with the eigen operator. Syntactically, as for inversion, whenever e is a well-typed expression with a square output type, we now also allow the expression $\mathsf{eigen}(e)$, with the same output type. Semantically, however, the rules of Figure 2 must be adapted so that they do not infer statements of the form $e(I) = B$, but rather of the form $B \in e(I)$, i.e., B is a possible result of $e(I)$. The let-construct now becomes crucial; it allows us to assign a possible result of eigen to a new variable, and work with that intermediate result consistently.

In this and the next section, we assume notions from linear algebra. An excellent introduction to the subject has been given by Axler [3].

▶ Remark (Eigenvalues). We can easily recover the eigenvalues from the eigenvectors, using inversion. Indeed, if A is diagonalizable and $B \in \mathsf{eigen}(A)$, then $\Lambda = B^{-1}AB$ is a diagonal matrix with all eigenvalues of A on the diagonal, so that the ith eigenvector in B corresponds to the eigenvalue in the ith column of Λ. This is the well-known eigendecomposition. However, the same can also be accomplished without using inversion. Indeed, suppose $B = (v_1, \ldots, v_n)$, and let λ_i be the eigenvalue to which v_i corresponds. Then $AB = (\lambda_1 v_1, \ldots, \lambda_n v_n)$. Each eigenvector is nonzero, so we can divide away the entries from B in AB (setting division by zero to zero). We thus obtain a matrix where the ith column consists of zeros or λ_i, with at least one occurrence of λ_i. By counting multiplicities, dividing them out, and finally summing, we obtain $\lambda_1, \ldots, \lambda_n$ in a column vector. We can apply a final diag to get it back into diagonal form. The MATLANG expression for doing all this uses similar tricks as those shown in Examples 5 and 6. ◀

The above remark suggests a shorthand in MATLANG + eigen where we return both B and Λ together:

$$\mathsf{let}\ (B, \Lambda) = \mathsf{eigen}(A)\ \mathsf{in}\ \ldots$$

This models how the eigen operation works in the languages R and MATLAB. We agree that Λ, like B, is the zero matrix if A is not diagonalizable.

▶ **Example 16** (Rank of a matrix)**.** Since the rank of a diagonalizable matrix equals the number of nonzero entries in its diagonal form, we can express the rank of a diagonalizable matrix A as follows:

$$\mathsf{let}\ (B, \Lambda) = \mathsf{eigen}(A)\ \mathsf{in}\ \mathbf{1}(A)^* \cdot \mathsf{apply}[\neq 0](\Lambda) \cdot \mathbf{1}(A).$$

▶ **Example 17** (Graph partitioning)**.** A well-known heuristic for partitioning an undirected graph without self-loops is based on an eigenvector corresponding to the second-smallest eigenvalue of the Laplacian matrix [27]. The Laplacian L can be derived from the adjacency matrix A as $\mathsf{let}\ D = \mathsf{diag}(A \cdot \mathbf{1}(A))\ \mathsf{in}\ \mathsf{apply}[-](D, A)$. (Here D is the degree matrix.) Now let $(B, \Lambda) \in \mathsf{eigen}(L)$. In an analogous way to Example 6, we can compute a matrix E, obtained from Λ by replacing the occurrences of the second-smallest eigenvalue by 1 and all other entries by 0. Then the eigenvectors corresponding to this eigenvalue can be isolated from B (and the other eigenvectors zeroed out) by multiplying $B \cdot E$. ◀

It turns out that MATLANG + inv is subsumed by MATLANG + eigen.

▶ **Theorem 18.** *Matrix inversion is expressible in* MATLANG + eigen*.*

A very interesting open problem is the following: *Are there graph queries expressible deterministically in* MATLANG + eigen*, but not in* MATLANG + inv*?* This is an interesting question for further research. The answer may depend on the functions that can be used in pointwise applications.

▶ Remark (Determinacy). The stipulation *deterministically* in the above open question is important. Ideally, we use the nondeterministic eigen operation only as an intermediate construct. It is an aid to achieve a powerful computation, but the final expression should have only a single possible output on every input. The expression of Example 16 is deterministic in this sense, as is the expression for inversion underlying the proof of Theorem 18.

6 The evaluation problem

The evaluation problem asks, given an input instance I and an expression e, to compute the result $e(I)$. There are some issues with this naive formulation, however. Indeed, in our theory we have been working with arbitrary complex numbers. How do we even represent the input? Notably, the eigen operation on a matrix with only rational entries may produce irrational entries. In fact, the eigenvalues of an adjacency matrix (even of a tree) need not even be definable in radicals [13]. Practical systems, of course, apply techniques from numerical mathematics to compute rational approximations. But it is still theoretically interesting to consider the exact evaluation problem.

Our approach is to represent the output symbolically, following the idea of constraint query languages [21, 25]. Specifically, we can define the input-output relation of an expression, for given dimensions of the input matrices, by an existential first-order logic formula over the reals. Such formulas are built from real variables, integer constants, addition, multiplication, equality, inequality ($<$), disjunction, conjunction, and existential quantification.

▶ **Example 19.** Consider the expression $\mathsf{eigen}(M)$ over the schema consisting of a single matrix variable M. Any instance I where $I(M)$ is an $n \times n$ matrix A can be represented by a tuple of $2 \times n \times n$ real numbers. Indeed, let $a_{i,j} = \Re A_{i,j}$ (the real part of a complex number), and let $b_{i,j} = \Im A_{i,j}$ (the imaginary part). Then $I(M)$ can be represented by the tuple $(a_{1,1}, b_{1,1}, a_{1,2}, b_{1,2}, \ldots, a_{n,n}, b_{n,n})$. Similarly, any $B \in \mathsf{eigen}(A)$ can be represented by a similar tuple. We introduce the variables $x_{M,i,j,\Re}$, $x_{M,i,j,\Im}$, $y_{i,j,\Re}$, and $y_{i,j,\Im}$, for $i, j \in \{1, \ldots, n\}$, where the x-variables describe an arbitrary input matrix and the y-variables describe an arbitrary possible output matrix. Denoting the input matrix by $[\bar{x}]$ and the output matrix by $[\bar{y}]$, we can now write an existential formula expressing that $[\bar{y}]$ is a possible result of eigen applied to $[\bar{x}]$:

- To express that $[\bar{y}]$ is a basis, we write that there exists a nonzero matrix $[\bar{z}]$ such that $[\bar{y}] \cdot [\bar{z}]$ is the identity matrix. It is straightforward to express this condition by a formula.
- To express, for each column vector v of $[\bar{y}]$, that v is an eigenvector of $[\bar{x}]$, we write that there exists λ such that $[\bar{x}] \cdot v = \lambda[\bar{x}]$.
- The final and most difficult condition to express is that distinct eigenvectors v and w that correspond to a same eigenvalue are orthogonal. We cannot write $\exists\lambda([\bar{x}] \cdot v = \lambda v \wedge [\bar{x}] \cdot w = \lambda w) \rightarrow v^* \cdot w = 0$, as this is not a proper existential formula. (Note though that the conjugate transpose of v is readily expressed.) Instead, we avoid an explicit quantifier and replace the antecedent by the conjunction, over all positions i, of $v_i \neq 0 \neq w_i \rightarrow ([\bar{x}] \cdot v)_i / v_i = ([\bar{x}] \cdot w)_i / w_i$.
- A final detail is that we should also be able to express that $[\bar{x}]$ is not diagonalizable, for in that case we need to define $[\bar{y}]$ to be the zero matrix. Nondiagonalizability is equivalent to the existence of a Jordan form with at least one 1 on the superdiagonal. We can express this as follows. We postulate the existence of an invertible matrix $[\bar{z}]$ such that the product $[\bar{z}] \cdot [\bar{x}] \cdot [\bar{z}]^{-1}$ has all entries zero, except those on the diagonal and the superdiagonal. The entries on the superdiagonal can only by 0 or 1, with at least one 1. Moreover, if an entry i, j on the superdiagonal is nonzero, the entries i, i and j, j must be equal. ◀

The approach taken in the above example leads to the following general result. The operations of MATLANG are handled using similar ideas as illustrated above for the eigen operation, and are actually easier. The let-construct, and the composition of subexpressions into larger expression, are handled by existential quantification.

▶ **Theorem 20.** *An input-sized expression consists of a schema $\mathcal{S}$, an expression e in* MATLANG + eigen *that is well-typed over $\mathcal{S}$ with output type $t_1 \times t_2$, and a size assignment σ defined on the size symbols occurring in $\mathcal{S}$. There exists a polynomial-time computable translation that maps any input-sized expression as above to an existential first-order formula ψ over the vocabulary of the reals, expanded with symbols for the functions used in pointwise applications in e, such that*

1. *Formula ψ has the following free variables:*
   - *For every $M \in \mathrm{var}(\mathcal{S})$, let $\mathcal{S}(M) = s_1 \times s_2$. Then ψ has the free variables $x_{M,i,j,\Re}$ and $x_{M,i,j,\Im}$, for $i = 1, \ldots, \sigma(s_1)$ and $j = 1, \ldots, \sigma(s_2)$.*
   - *In addition, ψ has the free variables $y_{i,j,\Re}$ and $y_{i,j,\Im}$, for $i = 1, \ldots, \sigma(t_1)$ and $j = 1, \ldots, \sigma(t_2)$.*

   The set of these free variables is denoted by $\mathrm{FV}(\mathcal{S}, e, \sigma)$.
2. *Any assignment ρ of real numbers to these variables specifies, through the x-variables, an instance I conforming to $\mathcal{S}$ by σ, and through the y-variables, a $\sigma(t_1) \times \sigma(t_2)$ matrix B.*
3. *Formula ψ is true over the reals under such an assignment ρ, if and only if $B \in e(I)$.*

The existential theory of the reals is decidable; actually, the full first-order theory of the reals is decidable [2, 4]. But, specifically the class of problems that can be reduced in polynomial time to the existential theory of the reals forms a complexity class on its own, known as $\exists\mathbf{R}$ [34, 35]. The above theorem implies that the *partial evaluation problem for* MATLANG + eigen belongs to this complexity class. We define this problem as follows. The idea is that an arbitrary specification, expressed as an existential formula χ over the reals, can be imposed on the input-output relation of an input-sized expression.

▶ **Definition 21.** The *partial evaluation problem* is a decision problem that takes as input:

- an input-sized expression $(\mathcal{S}, e, \sigma)$, where all functions used in pointwise applications are explicitly defined using existential formulas over the reals;
- an existential formula χ with free variables in $\mathrm{FV}(\mathcal{S}, e, \sigma)$ (see Theorem 20).

The problem asks if there exists an instance I conforming to $\mathcal{S}$ by σ and a matrix $B \in e(I)$ such that (I, B) satisfies χ.

For example, χ may completely specify the matrices in I by giving the values of the entries as rational numbers, and may express that the output matrix has at least one nonzero entry.

An input $(\mathcal{S}, e, \sigma, \chi)$ is a yes-instance to the partial evaluation problem precisely when the existential sentence $\exists \mathrm{FV}(\mathcal{S}, e, \sigma)(\psi \wedge \chi)$ is true in the reals, where ψ is the formula obtained by Theorem 20. Hence we can conclude:

▶ **Corollary 22.** *The partial evaluation problem for* MATLANG + eigen *belongs to $\exists\mathbf{R}$.*

Since the full theory of the reals is decidable, our theorem implies many other decidability results. We give just two examples.

▶ **Corollary 23.** *The equivalence problem for input-sized expressions is decidable. This problem takes as input two input-sized expressions $(\mathcal{S}, e_1, \sigma)$ and $(\mathcal{S}, e_2, \sigma)$ (with the same $\mathcal{S}$ and σ) and asks if for all instances I conforming to $\mathcal{S}$ by σ, we have $B \in e_1(I) \Leftrightarrow B \in e_2(I)$.*

Note that the equivalence problem for MATLANG expressions on arbitrary instances (size not fixed) is undecidable by Theorem 9, since equivalence of FO^3 formulas over binary relational vocabularies is undecidable [15].

▶ **Corollary 24.** *The determinacy problem for input-sized expressions is decidable. This problem takes as input an input-sized expression $(\mathcal{S}, e, \sigma)$ and asks if for every instance I conforming to $\mathcal{S}$ by σ, there exists at most one $B \in e(I)$.*

Corollary 22 gives an $\exists\mathbf{R}$ upper bound on the combined complexity of query evaluation [38]. Our final result is a matching lower bound, already for data complexity alone.

▶ **Theorem 25.** *There exists a fixed schema $\mathcal{S}$ and a fixed expression e in* MATLANG + eigen*, well-typed over $\mathcal{S}$, such that the following problem is hard for $\exists\mathbf{R}$: Given an integer instance I over $\mathcal{S}$, decide whether the zero matrix is a possible result of $e(I)$. The pointwise applications in e use only simple functions definable by quantifier-free formulas over the reals.*

▶ Remark (Complexity of deterministic expressions). Our proof of Theorem 25 relies on the nondeterminism of the eigen operation. Coming back to our remark on determinacy at the end of the previous section, it is an interesting question for further research to understand not only the expressive power but also the complexity of the evaluation problem for *deterministic* MATLANG + eigen expressions.

7 Conclusion

There is a commendable trend in contemporary database research to leverage, and considerably extend, techniques from database query processing and optimization, to support large-scale linear algebra computations. In principle, data scientists could then work directly in SQL or related languages. Still, some users will prefer to continue using the matrix sublanguages they are more familiar with. Supporting these languages is also important so that existing code need not be rewritten.

From the perspective of database theory, it then becomes relevant to understand the expressive power of these languages as well as possible. In this paper we have proposed a framework for viewing matrix manipulation from the point of view of expressive power of database query languages. Moreover, our results formally confirm that the basic set of matrix operations offered by systems in practice, formalized here in the language MATLANG + inv + eigen, really is adequate for expressing a range of linear algebra techniques and procedures.

In the paper we have already mentioned some intriguing questions for further research. Deep inexpressibility results have been developed for logics with rank operators [30]. Although these results are mainly concerned with finite fields, they might still provide valuable insight in our open questions. Also, we have not covered all standard constructs from linear algebra. For instance, it may be worthwhile to extend our framework with the operation of putting matrices in upper triangular form, with the Gram-Schmidt procedure (which is now partly hidden in the eigen operation), and with the singular value decomposition.

Finally, we note that various authors have proposed to go beyond matrices, introducing data models and algebra for tensors or multidimensional arrays [31, 22, 32]. When moving to more and more powerful and complicated languages, however, it becomes less clear at what point we should simply move all the way to full SQL, or extensions of SQL with recursion.

References

1 S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2 D.S. Arnon. Geometric reasoning with logic and algebra. *Artificial Intelligence*, 37:37–60, 1988.
3 S. Axler. *Linear Algebra Done Right*. Springer, third edition, 2015.
4 S. Basu, R. Pollack, and M.-F. Roy. *Algorithms in Real Algebraic Geometry*. Springer, second edition, 2008.
5 M. Boehm, M.W. Dusenberry, D. Eriksson, A.V. Evfimievski, F.M. Manshadi, N. Pansare, B. Reinwald, F.R. Reiss, P. Sen, A.C. Surve, and S. Tatikonda. SystemML: Declarative machine learning on Spark. *Proceedings of the VLDB Endowment*, 9(13):1425–1436, 2016.
6 A. Bonato. *A Course on the Web Graph*, volume 89 of *Graduate Studies in Mathematics*. American Mathematical Society, 2008.
7 S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30:107–117, 1998.
8 L. Chen, A. Kumar, J. Naughton, and J.M. Patel. Towards linear algebra over normalized data. *Proceedings of the VLDB Endowment*, 10(11):1214–1225, 2017.
9 S. Datta, R. Kulkarni, A. Mukherjee, T. Schwentick, and T. Zeume. Reachability is in DynFO. In M.M. Halldórsson, K. Iwama, N. Kobayashi, and B. Speckmann, editors, *Proceedings 42nd International Colloquium on Automata, Languages and Programming, Part II*, volume 9135 of *Lecture Notes in Computer Science*, pages 159–170. Springer, 2015.
10 A. Dawar. On the descriptive complexity of linear algebra. In W. Hodges and R. de Queiroz, editors, *Logic, Language, Information and Computation, Proceedings 15th WoLLIC*, volume 5110 of *Lecture Notes in Computer Science*, pages 17–25. Springer, 2008.
11 A. Dawar, M. Grohe, B. Holm, and B. Laubner. Logics with rank operators. In *Proceedings 24th Annual IEEE Symposium on Logic in Computer Science*, pages 113–122, 2009.
12 G.M. Del Corso, A. Gulli, and F. Romani. Fast PageRank computation via a sparse linear system. *Internet Mathematics*, 2(3):251–273, 2005.
13 C.D. Godsil. Some graphs with characteristic polynomials which are not solvable by radicals. *Journal of Graph Theory*, 6:211–214, 1982.
14 G.H. Golub and C.F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, fourth edition, 2013.
15 E. Grädel, E. Rosen, and M. Otto. Undecidability results on two-variable logics. *Archive of Mathematical Logic*, 38:313–354, 1999.
16 D.J. Hand, H. Mannila, and P. Smyth. *Principles of Data Mining*. MIT Press, 2001.
17 Lauri Hella, Leonid Libkin, Juha Nurmonen, and Limsoon Wong. Logics with aggregate operators. *J. ACM*, 48(4):880–907, 2001. doi:10.1145/502090.502100.
18 B. Holm. *Descriptive Complexity of Linear Algebra*. PhD thesis, University of Cambridge, 2010.
19 D. Hutchison, B. Howe, and D. Suciu. LaraDB: A minimalist kernel for linear and relational algebra computation. In F.N. Afrati and J. Sroka, editors, *Proceedings 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, pages 2:1–2:10, 2017.
20 K.E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., 1962.
21 Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. Constraint query languages. *J. Comput. Syst. Sci.*, 51(1):26–52, 1995. doi:10.1006/jcss.1995.1051.
22 M. Kim. *TensorDB and Tensor-Relational Model for Efficient Tensor-Relational Operations*. PhD thesis, Arizona State University, 2014.
23 A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *jacm*, 29(3):699–717, 1982.
24 Ph.G. Kolaitis. On the expressive power of logics on finite models. In *Finite Model Theory and Its Applications*, chapter 2. Springer, 2007.

25 G. Kuper, L. Libkin, and J. Paredaens, editors. *Constraint Databases*. Springer, 2000.

26 B. Laubner. *The Structure of Graphs and New Logics for the Characterization of Polynomial Time*. PhD thesis, Humboldt-Universität zu Berlin, 2010.

27 J. Leskovec, A. Rajaraman, and J.D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, second edition, 2014.

28 Leonid Libkin. Expressive power of SQL. *Theor. Comput. Sci.*, 296(3):379–404, 2003. doi:10.1016/S0304-3975(02)00736-3.

29 H.Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. In-database factorized learning. In J.L. Reutter and D. Srivastava, editors, *Proceedings 11th Alberto Mendelzon International Workshop on Foundations of Data Management*, volume 1912 of *CEUR Workshop Proceedings*, 2017.

30 W. Pakusa. *Linear Equation Systems and the Search for a Logical Characterisation of Polynomial Time*. PhD thesis, RWTH Aachen, 2015.

31 F. Rusu and Y. Cheng. A survey on array storage, query languages, and systems. arXiv:1302.0103, 2013.

32 T. Sato. Embedding Tarskian semantics in vector spaces. arXiv:1703.03193, 2017.

33 T. Sato. A linear algebra approach to datalog evaluation. *Theory and Practice of Logic Programming*, 17(3):244–265, 2017.

34 M. Schaefer. Complexity of some geometric and topological problems. In D. Eppstein and E.R. Gansner, editors, *Graph Drawing*, volume 5849 of *Lecture Notes in Computer Science*, pages 334–344. Springer, 2009.

35 M. Schaefer and D. Štefankovič. Fixed points, Nash equilibria, and the existential theory of the reals. *Theory of Computing Systems*, 60(2):172–193, 2017.

36 M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *Proceedings 2016 International Conference on Management of Data*, pages 3–18. ACM, 2016.

37 J. Van den Bussche, D. Van Gucht, and S. Vansummeren. A crash course in database queries. In *Proceedings 26th ACM Symposium on Principles of Database Systems*, pages 143–154. ACM Press, 2007.

38 M. Vardi. The complexity of relational query languages. In *Proceedings 14th ACM Symposium on the Theory of Computing*, pages 137–146, 1982.

Enumeration Complexity of Conjunctive Queries with Functional Dependencies

Nofar Carmeli
Technion, Haifa, Israel
snofca@cs.technion.ac.il

Markus Kröll
TU Wien, Vienna, Austria
kroell@dbai.tuwien.ac.at

Abstract

We study the complexity of enumerating the answers of Conjunctive Queries (CQs) in the presence of Functional Dependencies (FDs). Our focus is on the ability to list output tuples with a constant delay in between, following a linear-time preprocessing. A known dichotomy classifies the acyclic self-join-free CQs into those that admit such enumeration, and those that do not. However, this classification no longer holds in the common case where the database exhibits dependencies among attributes. That is, some queries that are classified as hard are in fact tractable if dependencies are accounted for. We establish a generalization of the dichotomy to accommodate FDs; hence, our classification determines which combination of a CQ and a set of FDs admits constant-delay enumeration with a linear-time preprocessing.

In addition, we generalize a hardness result for cyclic CQs to accommodate a common type of FDs. Further conclusions of our development include a dichotomy for enumeration with linear delay, and a dichotomy for CQs with disequalities. Finally, we show that all our results apply to the known class of "cardinality dependencies" that generalize FDs (e.g., by stating an upper bound on the number of genres per movies, or friends per person).

2012 ACM Subject Classification Mathematics of computing → Enumeration, Theory of computation → Database query languages (principles)

Keywords and phrases Enumeration, Complexity, CQs

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.11

Related Version A full version of the paper is available at [8], `https://arxiv.org/abs/1712.07880`.

Funding This work was supported by the German-Israeli Foundation for Scientific Research and Development (GIF), Grant no. I-2436-407.6/2016 and by the Austrian Science Fund (FWF): W1255-N23, P25207-N23, P25518-N23.

1 Introduction

When evaluating a non-boolean Conjunctive Query (CQ) over a database, the number of results can be huge. Since this number may be larger than the size of the database itself, we need to use specific measures of enumeration complexity to describe the hardness of such a problem. In this perspective, the best we can hope for is to constantly output results, in such a way that the delay between them is unaffected by the size of the database instance. For this to be possible, we need to allow a precomputation phase before printing the first result, as linear time preprocessing is necessary to read the input instance.

A known dichotomy determines when the answers to self-join-free acyclic CQs can be enumerated with constant delay after linear time preprocessing [3]. This class of enumeration problems, denoted by $\mathsf{DelayC_{lin}}$, can be regarded as the most efficient class of nontrivial enumeration problems and therefore current work on query enumeration has focused on this class [10, 15, 5]. Bagan et al.[3] show that a subclass of acyclic queries, called *free-connex*, are exactly those that are enumerable in $\mathsf{DelayC_{lin}}$, under the common assumption that boolean matrix multiplication cannot be solved in quadratic time. An acyclic query is called free-connex if the query remains acyclic when treating the head of the query as an additional atom. This and all other results in this paper hold under the RAM model [16].

The above mentioned dichotomy only holds when applied to databases with no additional assumptions, but oftentimes this is not the case. In practice, there is usually a connection between different attributes, and *Functional Dependencies* (FDs) and *Cardinality Dependencies* (CDs) are widely used to model situations where some attributes imply others. As the following example shows, these constraints also have an immediate effect on the complexity of enumerating answers for queries over such a schema.

▶ **Example 1.** For a list of actors and the production companies they work with, we have the query: $Q(actor, production) \leftarrow \text{Cast}(movie, actor), \text{Release}(movie, production)$. At first glance, it appears as though this query is not in $\mathsf{DelayC_{lin}}$, as it is acyclic but not free-connex. Nevertheless, if we take the fact that a movie has only one production company into account, we have the FD Release : $movie \rightarrow production$, and the enumeration problem becomes easy: we only need to iterate over all tuples of Cast and replace the *movie* value with the single *production* value that the relation Release assigns to it. This can be done in linear time by first sorting (in linear time [12]) both relations according to *movie*. ◀

Example 1 shows that the dichotomy by Bagan et al. [3] does not hold in the presence of FDs. In fact, we believe that dependencies between attributes are so common in real life, that ignoring them in such dichotomies can lead to missing a significant portion of the tractable cases. Therefore, to get a realistic picture of the enumeration complexity of CQs, we have to take dependencies into account. The goal of this work is to generalize the dichotomy to fully accommodate FDs.

Towards this goal, we introduce an extension of a query Q according to the FDs. The extension is called the FD-extended query, and denoted Q^+. In this extension, each atom, as well as the head of the query, contains all variables that can be implied by its variables according to some FD. This way, instead of classifying every combination of CQ and FDs directly, we encode the dependencies within the extended query, and use the classification of Q^+ to gain insight regarding Q. This approach draws inspiration from the proof of a dichotomy in the complexity of *deletion propagation*, in the presence of FDs [13]. However, the problem and consequently the proof techniques are fundamentally different.

The FD-extension is defined in such a way that if Q is satisfied by an assignment, then the same assignment also satisfies the extension Q^+, as the underlying instance is bound by the FDs. In fact, we can show that enumerating the solutions of Q under FDs can be reduced to enumerating the solutions of Q^+. Therefore, tractability of Q^+ ensures that Q can be efficiently solved as well. By using the positive result in the known dichotomy, Q^+ is tractable w.r.t enumeration if it is free-connex. Moreover, it can be shown that the structural restrictions of acyclicity and free-connex are closed under taking FD-extensions. Hence, the class of all queries Q such that Q^+ is free-connex is an extension of the class of free-connex queries, and this extension is in fact proper. We denote the classes of queries Q such that Q^+ is acyclic or free-connex as FD-acyclic respectively FD-free-connex.

To reach a dichotomy, we now need to answer the following question: Is it possible that Q can be enumerated efficiently even if Q^+ is not free-connex? To show that an enumeration problem is not within a given class, enumeration complexity has few tools to offer. One such tool is a notion of completeness for enumeration problems [9]. However, this notion focuses on problems with a complexity corresponding to higher classes of the polynomial hierarchy. So in order to deal with this problem, Bagan et al. [3] reduced the matrix multiplication problem to enumerating the answers to any query that is acyclic but not free-connex. This reduction fails, however, when dependencies are imposed on the data, as the constructed database instance does not necessarily satisfy the underlying dependencies.

As it turns out, however, the structure of the FD-extended query Q^+ allows us to extend this reduction to our setting. By carefully expanding the reduced instance such that on the one hand, the dependencies hold and on the other hand, the reduction can still be performed within linear time, we establish a dichotomy. That is, we show that the tractability of enumerating the answers of a self-join-free query Q in the presence of FDs is exactly characterized by the structure of Q^+: Given an FD-acyclic query Q, we can enumerate the answers to Q within the class $\mathsf{DelayC_{lin}}$ iff Q is FD-free-connex.

The resulting extended dichotomy, as well as the original one, brings insight to the case of acyclic queries. Concerning unrestricted CQs, providing even a first solution of a query in linear time is impossible in general. This is due to the fact that the parameterized complexity of answering boolean CQs, taking the query size as the parameter, is $\mathsf{W}[1]$-hard [14]. This does not imply, however, that there are no cyclic queries with the corresponding enumeration problems in $\mathsf{DelayC_{lin}}$. The fact that no such queries exist requires an additional proof, which was presented by Brault-Baron [6]. This result holds under a generalization of the triangle finding problem, which is considered not to be solvable within linear time [17]. As before, this proof does no longer apply in the presence of FDs. Moreover, it is possible for Q to be cyclic and Q^+ acyclic. In fact, Q^+ may even be free-connex, and therefore tractable in $\mathsf{DelayC_{lin}}$. We show that, under the same assumptions used by Brault-Baron [6], the evaluation problem for a self-join-free CQ in the presence of unary FDs where Q^+ is cyclic cannot be solved in linear time. As linear time preprocessing is not enough to achieve the first result, a consequence is that enumeration within $\mathsf{DelayC_{lin}}$ is impossible in that case. This covers all types of CQs and shows a full dichotomy, at least for the case of unary FDs.

The results we present here are not limited to FDs. CDs (Cardinality Dependencies) [7, 2] are a generalization of FDs, denoted $(R_i : A \to B, c)$. Here, the right-hand side does not have to be unique for every assignment to the left-hand side, but there can be at most c different values to the variables of B for every value of the variables of A. FDs are in fact a special case of CDs where $c = 1$. Constraints of that form appear naturally in many applications. For example: a movie has only a handful of directors and there are at most 200 countries. We show that all results described in this paper also apply to CDs. Moreover, we show how our results can be easily used to yield additional results, such as a dichotomy for CQs with disequalities, and a dichotomy to evaluate CQs with linear delay.

Contributions. Our main contributions are as follows.

- We extend the class of queries that can be evaluated in $\mathsf{DelayC_{lin}}$ by incorporating the FDs. This extension is the class of FD-free-connex CQs.
- We establish a dichotomy for the enumeration complexity of self-join-free FD-acyclic CQs. Consequently, we get a dichotomy for self-join-free acyclic CQs under FDs.
- We show a lower bound for FD-cyclic CQs. In particular, we get a dichotomy for all self-join-free CQs in the presence of unary FDs.
- We extend our results to CDs.

This work is organized as follows: In Section 2 we provide definitions and results that we will use. Section 3 introduces FD-extended queries and establishes the equivalence between a query and its FD-extension. The generalized version of the dichotomy is shown in Section 4. In Section 5, a lower bound for cyclic queries under unary FDs is shown, and Section 6 shows that all results from the previous sections extend to CDs. Concluding remarks are given in Section 7. All missing proof details can be found in the full version of this article [8].

2 Preliminaries

In this section we provide preliminary definitions as well as state results that we will use throughout this paper.

Schemas and Functional Dependencies. A *schema* $\mathcal{S}$ is a pair $(\mathcal{R}, \Delta)$ where $\mathcal{R}$ is a finite set $\{R_1, \ldots, R_n\}$ of *relational symbols* and Δ is a set of *Functional Dependencies* (FDs). We denote the *arity* of a relational symbol R_i as $\text{arity}(R_i)$. An FD $\delta \in \Delta$ has the form $R_i \colon A \to B$, where $R_i \in \mathcal{R}$ and A, B are non-empty with $A, B \subseteq \{1, \ldots, \text{arity}(R_i)\}$.

Let *dom* be a finite set of constants. A database I over schema $\mathcal{S}$ is called an *instance* of $\mathcal{S}$, and it consists of a finite relation $R_i^I \subseteq dom^{\text{arity}(R_i)}$ for every relational symbol $R_i \in \mathcal{R}$, such that all FDs in Δ are *satisfied*. An FD $\delta = R_i \colon A \to B$ is said to be satisfied if, for all tuples $u, v \in R_i^I$ that are equal on the indices of A, u and v are equal on the indices of B. Here we assume that all FDs are of the form $R_i \colon A \to b$, where $b \in \{1, \ldots, \text{arity}(R_i)\}$, as we can replace an FD of the form $R_i \colon A \to B$ where $|B| > 1$ by the set of FDs $\{R_i \colon A \to b \mid b \in B\}$. If $|A| = 1$, we say that δ is a *unary* FD.

Conjunctive Queries. Let *var* be a set of variables disjoint from *dom*. A *Conjunctive Query* (CQ) over a schema $\mathcal{S} = (\mathcal{R}, \Delta)$ is an expression of the form $Q(\vec{x}) \leftarrow R_1(\vec{v}_1), \ldots, R_m(\vec{v}_m)$, where $R_1, \ldots, R_m$ are relational symbols of $\mathcal{R}$, the tuples $\vec{x}, \vec{v}_1, \ldots, \vec{v}_m$ hold variables, and every variable in $\vec{x}$ appears in at least one of $\vec{v}_1, \ldots, \vec{v}_m$. We often denote this query as $Q(\vec{x})$ or even Q. Define the variables of Q as $var(Q) = \bigcup_{i=1}^{m} \vec{v}_i$, and define the *free variables* of Q as $\text{free}(Q) = \vec{x}$. We call $Q(\vec{x})$ the head of Q, and the atomic formulas $R_i(\vec{v}_i)$ are called *atoms*. We further use $\text{atoms}(Q)$ to denote the set of atoms of Q. A CQ is said to contain *self-joins* if some relation symbol appears in more than one atom.

For the *evaluation* $Q(I)$ of a CQ Q with free variables $\vec{x}$ over a database I, we define $Q(I)$ to be the set of all *mappings* $\mu|_{\vec{x}}$ such that μ is a homomorphism from $R_1(\vec{v}_1), \ldots, R_m(\vec{v}_m)$ into I, where $\mu|_{\vec{x}}$ denotes the restriction (or projection) of μ to the variables $\vec{x}$. The problem $\textsc{Decide}_{\Delta}\langle Q \rangle$ is, given a database instance I, determining whether such a mapping exists.

Given a query Q over a schema $\mathcal{S} = (\mathcal{R}, \Delta)$, we often identify an FD $\delta \in \Delta$ as a mapping between variables. That is, if δ has the form $R_i : A \to b$ for $A = \{a_1, \ldots, a_{|A|}\}$, we sometimes denote it by $R_i \colon \{\vec{v}_i[a_1], \ldots, \vec{v}_i[a_{|A|}]\} \to \vec{v}_i[b]$, where $\vec{u}[k]$ is the k-th variable of $\vec{u}$. To distinguish between these two representations, we usually denote subsets of integers by $A, B, C, \ldots$, integers by $a, b, c, \ldots$, and variables by letters from the end of the alphabet.

Hypergraphs. A *hypergraph* $\mathcal{H} = (V, E)$ is a pair consisting of a set V of *vertices*, and a set E of non-empty subsets of V called *hyperedges* (sometimes *edges*). A *join tree* of a hypergraph $\mathcal{H} = (V, E)$ is a tree T where the nodes are the hyperedges of $\mathcal{H}$, and the *running intersection* property holds, namely: for all $u \in V$ the set $\{e \in E \mid u \in e\}$ forms a connected subtree in T. A hypergraph $\mathcal{H}$ is said to be *acyclic* if there exists a join tree for $\mathcal{H}$. Two vertices in a hypergraph are said to be *neighbors* if they appear in the same edge. A *clique* of

a hypergraph is a set of vertices, which are pairwise neighbors in $\mathcal{H}$. A hypergraph $\mathcal{H}$ is said to be *conformal* if every clique of $\mathcal{H}$ is contained in some edge of $\mathcal{H}$. A *chordless cycle* of $\mathcal{H}$ is a tuple $(x_1, \ldots, x_n)$ such that the set of neighboring pairs of variables of $\{x_1, \ldots, x_n\}$ is exactly $\{\{x_i, x_{i+1}\} \mid 1 \leq i \leq n-1\} \cup \{\{x_n, x_1\}\}$. It is well known (see [4]) that a hypergraph is acyclic iff it is conformal and contains no chordless cycles.

A *pseudo-minor* of a hypergraph $\mathcal{H} = (V, E)$ is a hypergraph obtained from $\mathcal{H}$ by a finite series of the following operations: (1) *vertex removal*: removing a vertex from V and from all edges in E that contain it. (2) *edge removal*: removing an edge e from E provided that some other $e' \in E$ contains it. (3) *edge contraction*: replacing all occurrences of a vertex v (within every edge) with a vertex u, provided that u and v are neighbors.

Classes of CQs. To a CQ Q we associate a hypergraph $\mathcal{H}(Q) = (V, E)$ where the vertices V are the variables of Q and every hyperedge E is a set of variables occurring in a single atom of Q, that is $E = \{\{v_1, \ldots, v_n\}\} \mid R_i(v_1, \ldots, v_n) \in \text{atoms}(Q)\}$. With a slight abuse of notation, we also identify atoms of Q with edges of $\mathcal{H}(Q)$. A CQ Q is said to be *acyclic* if $\mathcal{H}(Q)$ is acyclic, and it is said to be *free-connex* if both Q and $(V, E \cup \{\text{free}(Q)\})$ are acyclic.

A *head-path* for a CQ Q is a sequence of variables $(x, z_1, \ldots, z_k, y)$ with $k \geq 1$, such that: (1) $\{x, y\} \subseteq \text{free}(Q)$ (2) $\{z_1, \ldots, z_k\} \subseteq V \setminus \text{free}(Q)$ (3) It is a *chordless path* in $\mathcal{H}(Q)$, that is, two succeeding variables appear together in some atom, and no two non-succeeding variables appear together in an atom. Bagan et al. [3] showed that an acyclic CQ has a head-path iff it is not free-connex.

Enumeration Complexity. Given a finite alphabet Σ and binary relation $R \subseteq \Sigma^* \times \Sigma^*$, we denote by $\textsc{Enum}\langle R \rangle$ the *enumeration problem* of given an instance $x \in \Sigma^*$, to output all $y \in \Sigma^*$ such that $(x, y) \in R$. In this paper we adopt the *Random Access Machine* (RAM) model (see [16]). Previous results in the field assume different variations of the RAM model. Here we assume that the length of memory registers is linear in the size of value registers, that is, the accessible memory is polynomial. For a class $\mathcal{C}$ of enumeration problems, we say that $\textsc{Enum}\langle R \rangle \in \mathcal{C}$, if there is a RAM that – on input $x \in \Sigma^*$– outputs all $y \in \Sigma^*$ with $(x, y) \in R$ without repetition such that the first output is computed in time $p(|x|)$ and the delay between any two consecutive outputs after the first is $d(|x|)$, where:

- For $\textsc{Enum}\langle R \rangle \in \mathsf{DelayC_{lin}}$, we have $p(|x|) \in O(|x|)$ and $d(|x|) \in O(1)$.
- For $\textsc{Enum}\langle R \rangle \in \mathsf{DelayLin}$, we have $p(|x|), d(|x|) \in O(|x|)$.

Let $\textsc{Enum}\langle R_1 \rangle$ and $\textsc{Enum}\langle R_2 \rangle$ be enumeration problems. We say that there is an *exact reduction* from $\textsc{Enum}\langle R_1 \rangle$ to $\textsc{Enum}\langle R_2 \rangle$, written as $\textsc{Enum}\langle R_1 \rangle \leq_e \textsc{Enum}\langle R_2 \rangle$, if there are mappings σ and τ such that for every $x \in \Sigma^*$ the mapping $\sigma(x)$ is computable in $O(|x|)$, for every $y \in \Sigma^*$ with $(\sigma(x), y) \in R_2$, $\tau(y)$ is computable in constant time and $\{\tau(y) \mid y \in \Sigma^* \text{ with } (\sigma(x), y) \in R_2\} = \{y' \in \Sigma^* \mid (x, y') \in R_1\}$ in multiset notation. Intuitively, σ is used to map instances of $\textsc{Enum}\langle R_1 \rangle$ to instances of $\textsc{Enum}\langle R_2 \rangle$, and τ is used to map solutions to $\textsc{Enum}\langle R_2 \rangle$ to solutions of $\textsc{Enum}\langle R_1 \rangle$. An enumeration class $\mathcal{C}$ is said to be *closed under exact reduction* if for every $\textsc{Enum}\langle R_1 \rangle$ and $\textsc{Enum}\langle R_2 \rangle$ such that $\textsc{Enum}\langle R_1 \rangle \leq_e \textsc{Enum}\langle R_2 \rangle$ and $\textsc{Enum}\langle R_2 \rangle \in \mathcal{C}$, we have $\textsc{Enum}\langle R_1 \rangle \in \mathcal{C}$. Bagan et al. [3] proved that $\mathsf{DelayC_{lin}}$ is closed under exact reduction. The same proof holds for any meaningful enumeration complexity class that guarantees generating all unique answers with at least linear preprocessing time and at least constant delay between answers.

Enumerating Answers to CQs. For a CQ Q over a schema $\mathcal{S} = (\mathcal{R}, \Delta)$, we denote by $\textsc{Enum}_\Delta\langle Q \rangle$ the enumeration problem $\textsc{Enum}\langle R \rangle$, where R is the binary relation between

instances I over $\mathcal{S}$ and sets of mappings $Q(I)$. We consider the size of the query as well as the size of the schema to be fixed. Bagan et al. [3] showed that a self-join-free acyclic CQ is in $\mathsf{DelayC_{lin}}$ iff it is free-connex:

▶ **Theorem 2** ([3]). *Let Q be an acyclic CQ without self-joins over a schema $\mathcal{S} = (\mathcal{R}, \emptyset)$.*
1. *If Q is free-connex, then $\textsc{Enum}_\emptyset\langle Q\rangle \in \mathsf{DelayC_{lin}}$.*
2. *If Q is not free-connex, then $\textsc{Enum}_\emptyset\langle Q\rangle \notin \mathsf{DelayC_{lin}}$, assuming the product of two $n \times n$ boolean matrices cannot be computed in time $O(n^2)$.*

3 FD-Extended CQs

In this section, we formally define the extended query Q^+. We then discuss the relationship between Q and Q^+: their equivalence w.r.t. enumeration and the possible structural differences between them. As a result, we obtain that if Q^+ is in a class of queries that allows for tractable enumeration, then Q is tractable as well.

We first define Q^+. The *extension* of an atom $R(\vec{v})$ according to an FD $S\colon A \to b$ where $S(\vec{u}) \in \text{atoms}(Q)$ is possible if $\vec{u}[A] \subseteq \vec{v}$ but $\vec{u}[b] \notin \vec{v}$. In that case, $\vec{u}[b]$ is added to the variables of R. The *FD-extension* of a query is defined by iteratively extending all atoms as well as the head according to every possible dependency in the schema, until a fixpoint is reached. The schema extends accordingly: the arities of the relations increase as their corresponding atoms extend, and dummy variables are added to adjust to that change in case of self-joins. The FDs apply in every relation that contains all relevant variables.

▶ **Definition 3.** [(FD-Extended Query)] Let $Q(\vec{w}) \leftarrow R_1(\vec{v_1}), \ldots, R_m(\vec{v_m})$ be a CQ over a schema $\mathcal{S} = (\mathcal{R}, \Delta)$. We define two types of extension steps:
- The extension of an atom $R_i(\vec{v_i})$ according to an FD $R_j\colon A \to b$.
  Prerequisites: $\vec{v_j}[A] \subseteq \vec{v_i}$ and $\vec{v_j}[b] \notin \vec{v_i}$.
  Effect: The arity of R_i increases by one, and $R_i(\vec{v_i})$ is replaced by $R_i(\vec{v_i}, \vec{v_j}[b])$. In addition, every $R_k(\vec{v_k})$ such that $R_k{=}R_i$ and $k \neq i$ is replaced with $R_k(\vec{v_k}, t_k)$, where t_k is a fresh variable.
- The extension of the head $Q(\vec{w})$ according to an FD $R_j\colon A \to b$.
  Prerequisites: $\vec{v_j}[A] \subseteq \vec{w}$ and $\vec{v_j}[b] \notin \vec{w}$.
  Effect: The head is replaced by $Q(\vec{w}, \vec{v_j}[b])$.

The *FD-extension* of Q is the query $Q^+(\vec{y}) \leftarrow R_1^+(\vec{u_m}), \ldots, R_m^+(\vec{u_m})$, obtained by performing all possible extension steps on Q according to FDs of Δ until a fixpoint is reached. The extension is defined over the schema $\mathcal{S}^+ = (\mathcal{R}^+, \Delta_{Q^+})$, where $\mathcal{R}^+$ is $\mathcal{R}$ with the extended arities, and $\Delta_{Q^+} = \{R_i^+\colon C \to d \mid \exists (R_j\colon A \to b) \in \Delta \text{ s.t. } \vec{u_i}[C] = \vec{v_j}[A] \text{ and } \vec{u_i}[d] = \vec{v_j}[b]\}$.

Given a query, its FD-extension is unique up to a permutation of the added variables, and renaming of the new variables. As the order of the variables and the naming make no difference w.r.t. enumeration, we can treat the FD-extension as unique.

▶ **Example 4.** Consider a schema with $\Delta = \{R_1\colon 1 \to 2, R_3\colon 2,3 \to 1\}$, and the query $Q(x) \leftarrow R_1(x,y), R_2(x,z), R_2(u,z), R_3(w,y,z)$. As the FDs are $x \to y$ and $yz \to w$, the FD-extension is $Q^+(x,y) \leftarrow R_1^+(x,y), R_2^+(x,z,y,w), R_2^+(u,z,t_1,t_2), R_3^+(w,y,z)$. We first apply $x \to y$ on the head, and then $x \to y$ and consequently $yz \to w$ on $R_2(x,z)$. These two FDs now appear in the schema also for R_2, and the FDs of the extended schema are $\Delta_{Q^+} = \{R_1^+\colon 1 \to 2, R_2^+\colon 1 \to 3, R_2^+\colon 3,2 \to 4, R_3^+\colon 2,3 \to 1\}$. ◀

We later show that the enumeration complexity of a CQ Q over a schema with FDs only depends on the structure of Q^+, which is implicitly given by Q. Therefore, we introduce the notions of acyclic and free-connex queries for FD-extensions:

▶ **Definition 5.** Let Q be a CQ over a schema $\mathcal{S} = (\mathcal{R}, \Delta)$, and let Q^+ be its FD-extension.
- We say that Q is *FD-acyclic*, if Q^+ is acyclic.
- We say that Q is *FD-free-connex*, if Q^+ is free-connex.
- We say that Q is *FD-cyclic*, if Q^+ is cyclic.

The following proposition shows that the classes of acyclic queries and free-connex queries are both closed under constructing FD-extensions.

▶ **Proposition 6.** *Let Q be a CQ over a schema $\mathcal{S} = (\mathcal{R}, \Delta)$.*
- *If the query Q is acyclic, then it is FD-acyclic.*
- *If the query Q is free-connex, then it is FD-free-connex.*

Example 1 shows that the converse of the proposition above does not hold. This means that, by Theorem 2, there are queries Q such that we can enumerate the answers to Q^+ in $\mathsf{DelayC_{lin}}$, but we cannot enumerate the answers to Q with the same complexity, if we do not assume the FDs. The following lemma shows that enumerating the answers of Q (when relying on the FDs) is in fact equally hard as enumerating the answers of Q^+.

▶ **Theorem 7.** *Let Q be a CQ over a schema $\mathcal{S} = (\mathcal{R}, \Delta)$, and let Q^+ be its FD-extended query. Then $\textsc{Enum}_{\Delta}\langle Q\rangle \leq_e \textsc{Enum}_{\Delta_{Q^+}}\langle Q^+\rangle$ and $\textsc{Enum}_{\Delta_{Q^+}}\langle Q^+\rangle \leq_e \textsc{Enum}_{\Delta}\langle Q\rangle$.*

Proof Sketch. We first sketch the reduction $\textsc{Enum}_{\Delta}\langle Q\rangle \leq_e \textsc{Enum}_{\Delta_{Q^+}}\langle Q^+\rangle$. Given an instance I for the problem $\textsc{Enum}_{\Delta}\langle Q\rangle$, we set $\sigma(I) = I^+$ as described next. We start by removing tuples that interfere with the extended dependencies. For every dependency $R_j\colon X \to y$ and every atom $R_k(\vec{v_k})$ that contains the variables $X \cup \{y\}$, we only keep tuples of R_k^I that agree with some tuple of R_j^I over the values of $X \cup \{y\}$. Next, we follow the extension of the schema, and in each step we extend some R_i^I to $R_i^{I'}$ according to some FD $R_j\colon X \to y$. For each tuple $t \in R_i^I$, if there is no tuple $s \in R_j^I$ that agrees with t over the values of X, then we remove t altogether. Otherwise, we copy t to $R_i^{I'}$ and assign y with the same value that s assigns it. Given an answer $\mu \in Q^+(\sigma(I))$, we set $\tau(\mu)$ to be the projection of μ to free(Q). To show that $\textsc{Enum}_{\Delta_{Q^+}}\langle Q^+\rangle \leq_e \textsc{Enum}_{\Delta}\langle Q\rangle$, we describe the construction of an instance $\sigma(I^+)$ by "reversing" the extension steps. If an atom was extended, we simply remove the added attribute. If the head was extended using some $R_j\colon X \to y$, then for each tuple in $R_j^{I^{i+1}}$ that assigns y and X with the values y_0 and $\vec{x}_0$ respectively, we add the value y_0 to a lookup table with pointer $(X, \vec{x}_0, y)$. For every $\mu \in Q(\sigma(I^+))$, $\tau(\mu)$ is defined as μ extended by the values from the lookup table. ◀

The direction $\textsc{Enum}_{\Delta}\langle Q\rangle \leq_e \textsc{Enum}_{\Delta_{Q^+}}\langle Q^+\rangle$ of Theorem 7 proves that FD-extensions can be used to expand tractable enumeration classes, as the following corollary states.

▶ **Corollary 8.** *Let $\mathcal{C}$ be an enumeration class that is closed under exact reduction. Let Q be a CQ and let Q^+ be its FD-extended query. If $\textsc{Enum}_{\Delta_{Q^+}}\langle Q^+\rangle \in \mathcal{C}$, then $\textsc{Enum}_{\Delta}\langle Q\rangle \in \mathcal{C}$.*

Since free-connex queries are in $\mathsf{DelayC_{lin}}$ and $\mathsf{DelayC_{lin}}$ is closed under exact reduction, if Q is an FD-free-connex query, then the corresponding enumeration problem is in $\mathsf{DelayC_{lin}}$. This follows from Theorem 2 and the fact that $\textsc{Enum}_{\Delta_{Q^+}}\langle Q^+\rangle \leq_e \textsc{Enum}_{\emptyset}\langle Q^+\rangle$.

▶ **Corollary 9.** *Let Q be a CQ over a schema $\mathcal{S} = (\mathcal{R}, \Delta)$. If Q is FD-free-connex, then $\textsc{Enum}_{\Delta}\langle Q\rangle \in \mathsf{DelayC_{lin}}$.*

We can now revisit Example 1. The query $Q(x, y) \leftarrow R_1(z, x), R_2(z, y)$ is not free-connex. Therefore, disregarding the FDs, according to Theorem 2 it is not in $\mathsf{DelayC_{lin}}$. However, given $R_2\colon z \to y$, the FD-extended query is $Q^+(x, y) \leftarrow R_1^+(z, y, x), R_2^+(z, y)$. As it is free-connex, enumerating Q^+ is in $\mathsf{DelayC_{lin}}$ by Corollary 9.

4 A Dichotomy for Acyclic CQs

In this section, we characterize which self-join-free FD-acyclic queries are in $\mathsf{DelayC_{lin}}$. We use the notion of FD-extended queries defined in the previous section to establish a dichotomy stating that enumerating the answers to an FD-acyclic query is in $\mathsf{DelayC_{lin}}$ iff the query is FD-free-connex. We will prove the following theorem:

▶ **Theorem 10.** *Let Q be an FD-acyclic CQ without self-joins over a schema $\mathcal{S} = (\mathcal{R}, \Delta)$.*

- *If Q is FD-free-connex, then $\textsc{Enum}_{\Delta}\langle Q\rangle \in \mathsf{DelayC_{lin}}$.*
- *If Q is not FD-free-connex, then $\textsc{Enum}_{\Delta}\langle Q\rangle \notin \mathsf{DelayC_{lin}}$, assuming that the product of two $n \times n$ boolean matrices cannot be computed in time $O(n^2)$.*

The positive case for the dichotomy was described in Corollary 9. Note that the restriction of considering only self-joins-free queries is required only for the negative side. This assumption is standard [3, 6, 13], as it allows to assign different atoms with different relations independently. The hardness result described here builds on that of Bagan et al. [3] for databases that are assumed not to have FDs, and it relies on the hardness of *the boolean matrix multiplication problem*. This problem is defined as the enumeration $\textsc{Enum}_{\emptyset}\langle\Pi\rangle$ of the query $\Pi(x, y) \leftarrow A(x, z), B(z, y)$ over the schema $(\{A, B\}, \emptyset)$ where $A, B \subseteq \{1, \ldots, n\}^2$. It is strongly conjectured that this problem is not computable in $O(n^2)$ time and currently, the best known algorithms require $O(n^\omega)$ time for some $2.37 < \omega < 2.38$ [11, 1].

The original proof describes an exact reduction $\textsc{Enum}_{\emptyset}\langle\Pi\rangle \leq_e \textsc{Enum}_{\emptyset}\langle Q\rangle$. Since Q is acyclic but not free-connex, it contains a head-path $(x, z_1, \ldots, z_k, y)$. Given an instance of the matrix multiplication problem, an instance of $\textsc{Enum}_{\emptyset}\langle Q\rangle$ is constructed, where the variables x,y and $z_1, \ldots, z_k$ of the head-path respectively encode the variables x, y and z of Π, while all other variables of Q are assigned constants. This way, A is encoded by an atom containing x and z_1, and B is encoded by an atom containing z_k and y. Atoms containing some z_i and z_{i+1} only propagate the value of z. Since x and y are in $free(Q)$, but z_i are not, the answers to Q correspond to those of Π. As no atom of Q contains both x and y, the instance can be constructed in linear time. Constant delay enumeration for Q after linear time preprocessing would result in the computation of the answers of Π in $O(n^2)$ time.

FDs restrict the relations that can be assigned to atoms. This means that the reduction cannot be freely performed on databases with FDs, and the proof no longer holds. The following example illustrates where the reduction fails in the presence of FDs.

▶ **Example 11.** The CQ from Example 1 has the form $Q(x, y) \leftarrow R_1(z, x), R_2(z, y)$ with the single FD $\Delta = \{R_2 \colon z \to y\}$. In the previous section, we show that it is in $\mathsf{DelayC_{lin}}$, so the reduction should fail. Indeed, it would assign R_2 with the same relation as B of the matrix multiplication problem, but this may have two tuples with the same z value and different y values. Therefore, the construction does not yield a valid instance of $\textsc{Enum}_{\Delta}\langle Q\rangle$. ◀

We now give a detailed sketch of a modification of this construction that shows that $\textsc{Enum}_{\emptyset}\langle\Pi\rangle \leq_e \textsc{Enum}_{\Delta_{Q^+}}\langle Q^+\rangle$. Any violations of the FDs are fixed by carefully picking more variables other than those of the head-path to take the roles of x,y and z of the matrix multiplication problem. This is done by introducing the sets V_x,V_y and V_z which are subsets of $var(Q)$. We say that a variable β *plays the role* of α, if $\beta \in V_\alpha$.

To clarify the explanation of the reduction, we start by describing a restricted case, where all FDs are unary. The basic idea in the case of general FDs will remain the same, but it will require a more involved construction of the sets V_α.

4.1 Unary Functional Dependencies

For the unary case, we define the sets V_x, V_y and V_z to be the sets of variables that iteratively imply x, y and some z_i respectively. That is, for $\alpha \in \{x, y, z_1, \ldots, z_k\}$ we first set $V_\alpha := \{\alpha\}$, and then apply $V_\alpha := V_\alpha \cup \{\gamma \in var(Q) \mid \gamma \to \beta \in \Delta_{Q^+} \wedge \beta \in V_\alpha\}$ until a fixpoint is reached. We then define $V_z := V_{z_1} \cup \cdots \cup V_{z_k}$.

The Reduction. Let $I = (A^I, B^I)$ be an instance of $\text{Enum}_\emptyset\langle\Pi\rangle$. In order to define $\sigma(I)$, we describe how to construct the relation R^I for every atom $R(\vec{v}) \in \text{atoms}(Q^+)$. If $var(R) \cap V_y = \emptyset$, then every tuple $(a, c) \in A^I$ is copied to a tuple in R^I. Variables in V_x get the value a, variables in V_z get the value c, and variables that play no role are assigned a constant $\perp$. That is, we define $R^{\sigma(I)} = \{(f(v_1, a, c), \ldots, f(v_k, a, c)) \mid (a, c) \in A^I\}$, where:

$$f(v_i, a, c) = \begin{cases} a & \text{if } v_i \in V_x \setminus V_z, \\ c & \text{if } v_i \in V_z \setminus V_x, \\ (a, c) & \text{if } v_i \in V_x \cap V_z, \\ \perp & \text{otherwise.} \end{cases}$$

Otherwise, $var(R) \cap V_y \neq \emptyset$, and we show that $var(R) \cap V_x = \emptyset$. In this case we define the relation similarly with B^I. Given a tuple $(c, b) \in B^I$, the variables of V_y get the value b, and those of V_z are assigned with c.

▶ **Example 12.** Consider the FD-extended query $Q^+(x, y, v) \leftarrow R(u, x, z), S(v, y, z)$ with $\Delta_{Q^+} = \{R\colon u \to x, R\colon u \to z, S\colon y \to v\}$. Using the head-path (x, z, y), the reduction will set $V_x = \{x, u\}$, $V_y = \{y\}$, $V_z = \{z, u\}$. Given an instance of the matrix multiplication problem with relations A and B, every tuple $(a, c) \in A$ will result in a tuple $((a, c), a, c) \in R$, and every tuple $(c, b) \in B$ will result in a tuple $(\perp, b, c) \in S$. ◀

We now outline the correctness of this reduction:

Well-defined reduction: For an atom R, either we have $var(R) \cap V_y = \emptyset$ or $var(R) \cap V_x = \emptyset$. That is, no atom contains variables from both V_x and V_y. Due to the definition of Q^+, this atom would otherwise also contain both x and y. However, they cannot appear in the same relation according to the definition of a head-path. The reduction is therefore well defined, and it can be constructed in linear time via copy and projection.

Preserving FDs: The construction ensures that if an FD $\gamma \to \alpha$ exists, then γ has all the roles of α. Therefore, either α has no role and corresponds to the constant $\perp$, or every value that appears in α also appears in γ. In any case, all FDs are preserved.

1-1 mapping of answers: If a variable of V_z would appear in the head of Q^+, then by the definition of Q^+, some z_i will be in the head as well. This cannot happen according to the definition of a head-path. Therefore, the head only encodes the x and y values of the matrix multiplication problem, so two different solutions to $\text{Enum}_{\Delta_{Q^+}}\langle Q^+\rangle$ must differ in either x or y, and correspond to different solutions of $\text{Enum}_\emptyset\langle\Pi\rangle$. For the other direction, the head necessarily contains the variables x and y. Therefore, two different solutions to $\text{Enum}_\emptyset\langle\Pi\rangle$ also correspond to different solutions of $\text{Enum}_{\Delta_{Q^+}}\langle Q^+\rangle$.

4.2 General Functional Dependencies

Next we show how to lift the idea of this reduction to the case of general FDs. In the case of unary FDs, we ensure that the construction does not violate a given FD $\gamma \to \alpha$, by simply encoding the values of α to γ. In the general case, when allowing more than one variable on the left-hand side of an FD $\gamma_1, \ldots, \gamma_k \to \alpha$, we must be careful when choosing the variables

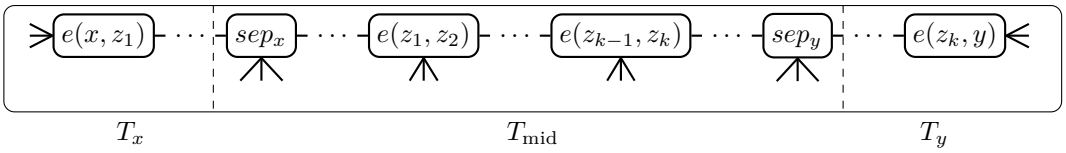


Figure 1 Join tree T of $\mathcal{H}(Q^+)$ for head-paths of length greater than 3. The subtrees T_x, T_y and T_{mid} are disjoint, and are separated by the nodes sep_x and sep_y.

γ_j to which we copy the values of α. Otherwise, as the following example shows, we will not be able to construct the instance in linear time.

▶ **Example 13.** Consider the query $Q(x, y) \leftarrow R_1(x, z, t_1), R_2(z, y, t_1, t_2)$ over a schema with the FD $R_2 \colon t_1 t_2 \to y$. Note that $Q = Q^+$ is acyclic but not free-connex, and that (x, z, y) is a head-path in $\mathcal{H}(Q^+)$. To repeat the idea shown in the unary case and ensure that the FDs still hold, the variable on the right-hand side of every FD is encoded to the variables on the left-hand side. If we encode y to t_1, then R_1 would contain the encodings of x, y and z. This means that its size will not be linear in that of the matrix multiplication instance, and we cannot hope for linear time construction. On the other hand, if we choose to encode y only to t_2, the reduction works. ◀

In the following central lemma, we describe a way of carefully picking the variables to which we assign roles, such that all FDs hold and yet the instance can be constructed in linear time. The idea is that we consider the join-tree of Q^+ and define V_x and V_y to hold variables that appear only in disjoint parts of this tree. This ensures that no atom contains variables of each. The property of a join-tree is used to guarantee that V_x and V_y are inclusive enough to correct all FD violations.

▶ **Lemma 14.** *Let Q be a CQ with no self-joins over a schema $\mathcal{S} = (\mathcal{R}, \Delta)$, such that Q^+ is acyclic but not free-connex. Denote a head-path of Q^+ by $(x, z_1, \ldots, z_k, y)$. Then there exist sets of variables V_x, V_y, V_z such that:*

1. *$x \in V_x$, $y \in V_y$, $\{z_1, \ldots z_k\} \subseteq V_z$.*
2. *For all $U \to v \in \Delta_{Q^+}$ such that $v \in V_\alpha$ with $\alpha \in \{x, y, z\}$, we have $U \cap V_\alpha \neq \emptyset$.*
3. *For every $R \in \text{atoms}(Q^+)$, we have $var(R) \cap V_y = \emptyset$ or $var(R) \cap V_x = \emptyset$.*
4. *$V_z \cap \text{free}(Q^+) = \emptyset$*

Proof Sketch. We first define a partition of the atoms of Q into three sets: T_x, T_y and T_{mid}, where T_{mid} may be empty. Let T be a join tree of $\mathcal{H}(Q^+)$, and denote the hyperedges on the head-path by $e(x, z_1), \ldots, e(z_k, y)$. Note that, by definition, each hyperedge of the head-path is a vertex of T. By the running intersection property of T, we can conclude that there is a simple path P from $e(x, z_1)$ to $e(z_k, y)$ in T, such that $e(z_1, z_2), \ldots, e(z_{k-1}, z_k)$ lie on that path in the order induced by the head-path. Let sep_x be the first node on the path P that does not contain x. This exists because $e(z_k, y)$ does not contain x, as the head-path is chordless. Similarly, let sep_y be the last node on P that does not contain y. Let T_x be the set of nodes v in T such that the unique path from v to $e(x, z_1)$ does not go through sep_x. Similarly, let T_y be the set of nodes w in T such that the unique path from w to $e(z_k, y)$ does not go through sep_y. Next set $T_{\text{mid}} = V(T) \setminus (T_x \cup T_y)$. Note that the nodes of T are exactly $T_x \cup T_{\text{mid}} \cup T_y$, and we can show that this union is disjoint (see Figure 1). Also note that $e(x, z_1) \in T_x$ and $e(z_k, y) \in T_y$, but T_{mid} may be empty if the head-path is of length three. Therefore, we established a partition of the atoms to two or three sets.

Next we define the sets of variables V_x, V_y and V_z. To do so, for $w \in var(Q)$, denote $\mathsf{Implies}(w) = \{u \in var(Q) \mid u \in U \text{ with } U \to w \in \Delta_{Q^+}\}$. Intuitively, $\mathsf{Implies}(w)$ is the set of all variables on the left-hand side of FDs that have w on the right-hand side. We now define V_x to contain x, and recursively to contain variables that imply those of V_x, but we do not take variables that appear outside of T_x. V_y is defined symmetrically. V_z is defined to contain $z_1, \ldots, z_k$, and recursively contain variables that imply those of V_z, but now we do not take variables that appear in the head of the query.

More formally, we recursively define:

- $\boldsymbol{V_x}$: Base $V_x := \{x\}$; Rule $V_x := V_x \cup \{t \in \mathsf{Implies}(w) \mid w \in V_x\} \setminus var(T_y \cup T_{\text{mid}})$
- $\boldsymbol{V_y}$: Base $V_y := \{y\}$; Rule $V_y := V_y \cup \{t \in \mathsf{Implies}(w) \mid w \in V_y\} \setminus var(T_x \cup T_{\text{mid}})$
- $\boldsymbol{V_z}$: Base $V_z := \{z_1, \ldots z_k\}$; Rule $V_z := V_z \cup \{t \in \mathsf{Implies}(w) \mid w \in V_z\} \setminus \text{free}(Q^+)$

We now prove that V_x, V_y and V_z meet the requirements of the lemma.

1. The first claim is immediate from the definition of the sets.
2. We first show the claim for $\alpha = x$. Let $\delta = U \to v \in \Delta_{Q^+}$, and let $e(U, v)$ be an atom containing all variables of δ. As $v \in V_x$, we know that $e(U, v) \notin T_y \cup T_{\text{mid}}$, therefore $e(U, v) \in T_x$. Assume by contradiction that $U \cap V_x = \emptyset$. Let $u \in U$. By definition of V_x, this means that $u \in var(e_u)$ for some $e_u \in T_y \cup T_{\text{mid}}$. As T_x, T_y and T_{mid} are disjoint, we have that $e_u \notin T_x$, which means that the path between e_u and $e(x, z_1)$ goes through sep_x. This means that the path from e_u to $e(U, v)$ goes through sep_x too, otherwise the concatenation of this path with the path from $e(U, v)$ to $e(x, z_1)$ would result in a path from e_u to $e(x, z_1)$ not going through sep_x. By the running intersection property, $u \in var(sep_x)$. Since this is true for all for all $u \in U$, it follows that $v \in var(sep_x)$ by definition of Q^+, contradicting the fact that $v \in V_x$. The case $\alpha = y$ is symmetric.
   Now for the case where $\alpha = z$. If $U \cap V_z = \emptyset$, then $U \subseteq \text{free}(Q^+)$, and by the definition of Q^+, $z_i \in \text{free}(Q^+)$, which is a contradiction to the fact that $v \in var(Q) \setminus \text{free}(Q^+)$.
3. Let $R \in \text{atoms}(Q^+)$. If $R \in T_x$, then by definition of V_y we have that $var(R) \cap V_y = \emptyset$. Otherwise, $R \in T_y \cup T_{\text{mid}}$, and similarly $var(R) \cap V_x = \emptyset$.
4. By definition of V_z, it does not contain any variables of $\text{free}(Q^+)$. ◀

With the sets V_x, V_y, V_z at hand, we can now perform the reduction between the two problems for general FDs. The reduction is based on the case of unary FDs, but with the sets defined according to Lemma 14. Requirements 1 and 4 on the sets guarantee a one-to-one mapping between the results of the two problems, requirement 2 guarantees that all FDs are preserved, and requirement 3 guarantees linear time construction.

▶ **Lemma 15.** *Let Q be a CQ with no self-joins over a schema $\mathcal{S} = (\mathcal{R}, \Delta)$. If Q^+ is acyclic and not free-connex, then* $\textsc{Enum}_{\emptyset}\langle\Pi\rangle \leq_e \textsc{Enum}_{\Delta_{Q^+}}\langle Q^+\rangle$.

This lemma, along with Theorem 7, establishes the hardness result in Theorem 10. This result does not contradict the dichotomy given in Theorem 2: If for a given query Q we have that Q^+ is acyclic but not free-connex, then Q cannot be free-connex by Proposition 6.

Note that Theorem 10, just like the dichotomy presented by Bagan et al. [3], also applies for CQs with disequalities. The extension for such a query is performed as before, ignoring the disequalities. The equivalence described in Theorem 7 still holds, and the proof remains intrinsically the same. The proof of the hardness result presented here also remains similar, with the sole difference that during the construction we take a different and disjoint domain for each variable. This guarantees that all possible disequalities are preserved.

5 Cyclic CQs

In the previous section, we established a classification of FD-acyclic CQs, but we did not consider FD-cyclic queries. A known result states that, under certain assumptions, self-join-free cyclic queries are not in $\mathsf{DelayC_{lin}}$ [6]. In this section, we therefore explore how FD-extensions can be used to obtain some insight on the implications of this result in the presence of FDs. We show that (under the same assumptions) self-join-free FD-cyclic queries that contain only unary FDs cannot be evaluated in linear time. For schemas containing only unary FDs, this extends the dichotomy presented in the previous section to all CQs, and also proves a dichotomy for the queries that can be enumerated in linear delay. We will prove the following theorem:

▶ **Theorem 16.** *Let Q be a CQ with no self-joins over a schema $\mathcal{S} = (\mathcal{R}, \Delta)$, where Δ only contains unary FDs. If Q is FD-cyclic, then* $\textsc{Decide}_{\Delta}\langle Q\rangle$ *cannot be solved in linear time, assuming that the* $\textsc{Tetra}(k)$ *problem cannot be solved in linear time for any k.*

As before, the initial hardness proof for cyclic queries no longer holds in the presence of FDs, and we modify the reduction to fix any violations of the FDs. We start by describing the assumption used to obtain the conditional lower bounds. We define $\textsc{Tetra}(k)$ to be the hypergraph with the vertices $\{1, \ldots, k\}$ and the edges $\{\{1, \ldots, k\} \setminus \{i\} \mid i \in \{1, \ldots, k\}\}$. Let $\mathcal{H}$ be a hypergraph. With a slight abuse of notation, we also denote by $\textsc{Tetra}(k)$ the decision problem of whether $\mathcal{H}$ contains a subhypergraph isomorphic to $\textsc{Tetra}(k)$. Note that $\textsc{Tetra}(3)$ is the problem of deciding whether a graph contains a triangle, which is strongly believed to be not solvable within time linear in the size of the graph [17]. The generalization of this assumption is that the $\textsc{Tetra}(k)$ problem cannot be solved in time linear in the size of the graph for any k. This is a stronger assumption than we used in Section 4, as the $\textsc{Tetra}(3)$ can be reduced to the matrix multiplication problem [17]. We will show that if Q^+ is cyclic and only unary FDs are present, the problem $\textsc{Tetra}(k)$ for some k can be reduced to $\textsc{Decide}_{\Delta_{Q^+}}\langle Q^+\rangle$.

▶ **Definition 17.** Let $\mathcal{H}$ be a cyclic hypergraph. We denote by $\mathsf{Tet_{pm}}(\mathcal{H})$ the pseudo-minors of $\mathcal{H}$ isomorphic to $\textsc{Tetra}(k)$ for some k, which are obtained in one of the following ways:

1. Vertex removal steps followed by all possible edge removals.
2. Vertex and edge removal steps that lead to a chordless cycle, followed by edge contraction and edge removal steps that result in a $\textsc{Tetra}(3)$.

Given a query Q, we define $\mathsf{Tet_{pm}}(Q) = \mathsf{Tet_{pm}}(\mathcal{H}(Q))$.

Brault-Baron [6] showed that if $\mathcal{H}$ is cyclic, then $\mathsf{Tet_{pm}}(\mathcal{H}) \neq \emptyset$. This proof is provided in the full version of this paper. For the reduction we will present next, we first need to show that for an FD-cyclic query Q, no pseudo-minor in $\mathsf{Tet_{pm}}(Q^+)$ contains all variables of any FD $X \to y$. Here, we assume that Δ only contains non-trivial FDs, meaning $y \notin X$.

▶ **Lemma 18.** *Let Q be an FD-cyclic CQ with no self-joins over a schema $\mathcal{S} = (\mathcal{R}, \Delta)$. For every $\mathcal{H}_{pm} = (V, E) \in \mathsf{Tet_{pm}}(Q^+)$ and non-trivial $X \to y \in \Delta_{Q^+}$, we have $X \cup \{y\} \not\subseteq V$.*

Proof Sketch. Assume by contradiction that the variables of the FD $\delta = X \to y$ are all part of the pseudo-minor $\mathcal{H}_{pm}$. Note that the variables $X \cup \{y\}$ must appear in a common edge that corresponds to the atom that defines δ. We distinguish between two cases. If $\mathcal{H}_{pm}$ is obtained only by vertex removal and edge removal steps, then by the definition of $\textsc{Tetra}(k)$ it also contains an edge e with $X \subseteq e$ and $y \notin e$. However, this contradicts the fact that Q^+ is an FD-extension, as every edge containing X must also contain y. The other case is

that $\mathcal{H}_{pm}$ is a $\textsc{Tetra}(3)$ obtained by edge contraction steps performed on a cycle C. Then $X \cup \{y\}$ is contained in a single edge in C, as none of the vertices $X \cup \{y\}$ have been deleted. Thus, we have that $|X| = 1$ and we can denote $X = \{x\}$. As C is a cycle, it contains an edge e with $x \in e$ and $y \notin e$, which contradicts the fact that Q^+ is an FD-extension. ◀

We are now ready to establish the reduction. Given a pseudo-minor of $\mathsf{Tet_{pm}}(Q^+)$ isomorphic to some $\textsc{Tetra}(k)$, we can reduce the problem of checking whether a hypergraph contains a subhypergraph isomorphic to $\textsc{Tetra}(k)$ to finding a boolean answer to Q^+.

▶ **Lemma 19.** *Let Q be an FD-cyclic CQ with no self-joins over a schema $\mathcal{S} = (\mathcal{R}, \Delta)$, where Δ only contains unary FDs. Let $\mathcal{H}_{pm} \in \mathsf{Tet_{pm}}(Q^+)$ be a pseudo-minor of $\mathcal{H}(Q^+)$ isomorphic to $\textsc{Tetra}(k)$. Then, $\textsc{Tetra}(k) \leq_m \textsc{Decide}_{\Delta_{Q^+}}\langle Q^+\rangle$, and this reduction can be computed in linear time.*

Proof Sketch. Given an input hypergraph $\mathcal{G}$ for the $\textsc{Tetra}(k)$ problem, we define an instance I of $\textsc{Decide}_{\Delta_{Q^+}}\langle Q^+\rangle$. We consider a sequence $\mathcal{H}(Q^+) = \mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_t = \mathcal{H}_{pm}$ of pseudo-minors, each one obtained by performing one operation over the previous one. We define the instance I inductively, by first generating relations that correspond to the edges of $\mathcal{H}_{pm}$, and then "reversing" the operations. For every edge e of $\mathcal{H}_{pm}$, we define a relation R_e^t that contains all edges of $\mathcal{G}$ that have the same size as e. We then construct the relations R_e^i of $\mathcal{H}_i$ given the relations R_e^{i+1} of $\mathcal{H}_{i+1}$. We make the following case distinction: If an edge e was removed as some e' contains it, then the relation R_e is added as a projection of $R_{e'}$. If $\mathcal{H}_{i+1}$ is obtained from $\mathcal{H}_i$ by an edge contraction in which a vertex v is replaced by u, then the values corresponding to u in every tuple are copied to the index of v. If a vertex v is removed, then it is assigned with a constant value, and then the following steps are performed on every tuple to correct any FD violations. First, the values of all variables implied by v are concatenated to its value, and then the new value of v is concatenated to all variables implying it. Since Q^+ is an FD-extension, and since only unary FDs are present, we can conclude that whenever a vertex is removed, if x implies y, then y is present in every edge containing x. This fact guarantees that the FD-correction steps can be performed. This construction defines relations that correspond to $\mathcal{H}(Q^+)$, which form I in such a way that $\mathcal{G}$ has a subhypergraph isomorphic to $\mathcal{H}_{pm}$ iff $Q^+(I) \neq \emptyset$. Compliance to any FDs included in $\mathcal{H}_i$ is shown by induction on the sequence, and the induction base holds trivially due to Lemma 18. ◀

Theorem 16 is an immediate consequence of Lemma 19. As in the previous section, by taking a disjoint domain for every variable in the proof of Lemma 19, Theorem 16 also holds for CQs with disequalities. In terms of enumeration complexity, Theorem 16 means that any enumeration algorithm for the answers of such a query cannot output a first solution (or decide that there is none) within linear time, and we get the following corollary.

▶ **Corollary 20.** *Let Q be a CQ with no self-joins over a schema $\mathcal{S} = (\mathcal{R}, \Delta)$, where Δ only contains unary FDs. If Q is FD-cyclic, then $\textsc{Enum}_\Delta\langle Q\rangle \notin \mathsf{DelayC_{lin}}$, assuming that the $\textsc{Tetra}(k)$ problem cannot be solved in linear time for any k.*

Less restrictive than constant delay enumeration, the class DelayLin consists of enumeration problems that can be solved with a linear delay between solutions. A lower bound for this class can be achieved similarly to Corollary 20. Regarding tractability, as acyclic CQs are in DelayLin [3], we conclude from Corollary 8 that FD-acyclic CQs are in this class as well. Thus, we obtain a dichotomy stating that CQs are in DelayLin iff they are FD-acyclic.

▶ **Theorem 21.** *Let Q be a CQ with no self-joins over a schema $\mathcal{S} = (\mathcal{R}, \Delta)$, where Δ only contains unary FDs.*

- *If Q is FD-acyclic, then* $\text{ENUM}_{\Delta}\langle Q\rangle \in$ DelayLin.
- *Otherwise (if Q is FD-cyclic),* $\text{ENUM}_{\Delta}\langle Q\rangle \notin$ DelayLin, *assuming that the* $\text{TETRA}(k)$ *problem cannot be solved in linear time for any k.*

We conclude this section with a short discussion about the extension of our results to general FDs. The following example shows that the proof for Theorem 16 that was provided here cannot be lifted to general FDs. Exploring this extension is left for future work.

▶ **Example 22.** Consider the query $Q() \leftarrow R_1(x,y,u), R_2(x,w,z), R_3(y,v,z), R_4(u,v,w)$, over a schema with all possible two-to-one FDs in the relations R_1, R_2 and R_3. That is, $\Delta = \{xy \to u, yu \to x, ux \to y, zy \to v, yv \to z, vz \to y, xz \to w, zw \to x, wx \to z\}$. Note that $Q^+ = Q$. The hypergraph $\mathcal{H}(Q^+)$ is cyclic, yet it is unclear whether Q can be solved in linear time, and whether $\text{TETRA}(3)$ can be reduced to answering Q^+. Using Lemma 18, $\mathcal{H}(Q^+)$ has triangle pseudo-minors that do not contain all variables of any FD. Consider for example the one obtained by removing all vertices other than x, y, z. A construction similar to that of Lemma 19 would assign u with the values of x and y, assign v with the values of y and z, and assign w with the values of x and z. This results in the edge $\{u, v, w\}$ containing all three values of any possible triangle, meaning that this edge cannot be constructed in linear time. Other choices of triangle pseudo-minors lead to similar encoding problems. ◀

6 Cardinality Dependencies

In this last section, we show that the results of this paper also apply to CQs over schemas with cardinality dependencies. *Cardinality Dependencies* (CDs) [2, 7] are a generalization of FDs, where the left-hand side does not uniquely determine the right-hand side, but rather provides a bound on the number of distinct values it can have. Formally, Δ is the set of *CDs* of a schema $\mathcal{S} = (\mathcal{R}, \Delta)$. Every $\delta \in \Delta$ has the form $(R_i \colon A \to B, c)$, where $R_i \colon A \to B$ is an FD and c is a positive integer. A CD δ is *satisfied* by an instance I over $\mathcal{S}$, if every set of tuples $S \subseteq (R_i)^I$ that agrees on the indices of A, but no pair of them agrees on all indices of B, contains at most c tuples. It follows from the definition that δ is an FD if $c = 1$.

Denote by Δ^{FD} the FDs obtained from a set of CDs Δ by setting all c values to one. Given a query Q over $\mathcal{S} = (\mathcal{R}, \Delta)$, we define the *CD-extended query* Q^+ of Q to be the FD-extended query of Q over $\mathcal{S} = (\mathcal{R}, \Delta^{\text{FD}})$. The schema $\mathcal{S}^+$ is defined with the original c values, and the CDs are $\Delta_{Q^+} = \{(R_i^+ \colon A \to b, c) \mid \exists (R_j \colon A \to B, c) \in \Delta, b \in B, A \cup \{b\} \subseteq var(R_i^+)\}$. Note that FD-extensions are indeed a special case of CD-extensions.

The hardness results extend to CDs because FDs are a special case of CDs. Since every instance that preserves the FDs Δ^{FD} also preserves the CDs Δ, we can conclude that $\text{ENUM}_{\Delta^{\text{FD}}}\langle Q\rangle \leq_e \text{ENUM}_{\Delta}\langle Q\rangle$. When only FDs are present we can apply Theorem 7, and get $\text{ENUM}_{\Delta^{\text{FD}}_{Q^+}}\langle Q^+\rangle \leq_e \text{ENUM}_{\Delta^{\text{FD}}}\langle Q\rangle$. Combining the two we get the following lemma.

▶ **Lemma 23.** *Let Q be a CQ over a schema $\mathcal{S} = (\mathcal{R}, \Delta)$, where Δ is a set of CDs, and let Q^+ be the corresponding CD-extension. Then* $\text{ENUM}_{\Delta^{FD}_{Q^+}}\langle Q^+\rangle \leq_e \text{ENUM}_{\Delta}\langle Q\rangle$.

Lemma 23 implies that all negative results presented in this paper hold for CDs. In order to extend the positive results, we need to show that the CD-extension is at least as hard as the original query w.r.t. enumeration. We use a slight relaxation of exact reductions: For $\text{ENUM}\langle R_1\rangle \leq_{e'} \text{ENUM}\langle R_2\rangle$, instead of a bijection between the sets of outputs, one output of $\text{ENUM}\langle R_1\rangle$ corresponds to at most a constant number of outputs of $\text{ENUM}\langle R_2\rangle$.

▶ **Lemma 24.** *Let Q be a CQ over a schema $\mathcal{S} = (\mathcal{R}, \Delta)$, where Δ is a set of CDs, and let Q^+ be the corresponding CD-extension. Then* $\textsc{Enum}_{\Delta}\langle Q\rangle \leq_{e'} \textsc{Enum}_{\Delta_{Q^+}}\langle Q^+\rangle$.

Proof Sketch. When dealing with FDs, we assume that the right-hand side has only one variable, as we can use such FDs to describe all possible ones. With CDs this no longer holds. Nonetheless, every instance of the schema $\mathcal{S} = (\mathcal{R}, \Delta)$ is also an instance of $\mathcal{S}^1 = (\mathcal{R}, \Delta^1)$, where $\Delta^1 = \{(R_i\colon A \to b, c) \mid (R_i\colon A \to B, c) \in \Delta, b \in B\}$. Therefore, we can conclude that $\textsc{Enum}_{\Delta}\langle Q\rangle \leq_e \textsc{Enum}_{\Delta^1}\langle Q\rangle$.

We now show that $\textsc{Enum}_{\Delta^1}\langle Q\rangle \leq_{e'} \textsc{Enum}_{\Delta^+}\langle Q^+\rangle$. The proof remains the same as in Theorem 7, except now, for each tuple extended from R_i^I to $R_i^{I^+}$ we can have at most c new tuples. Since this process is only done a constant number of times, the construction still only requires linear time, and the rest of the proof holds. Note that now one solution of $\textsc{Enum}_{\Delta^+}\langle Q^+\rangle$ may correspond to several solutions of $\textsc{Enum}_{\Delta^1}\langle Q\rangle$, as some variables were possibly added to the head. However, as the possible values of the added head variables are bounded by CDs, the number of solutions of Q^+ that correspond to one solution of Q is bounded by a constant. ◀

$\mathsf{DelayC_{lin}}$ is closed under this type of reduction. To avoid printing duplicates, we store the printed results. This requires a polynomial amount of memory, where the power of the polynomial is $|\text{free}(Q)|$. Defining the classes of *CD-acyclic* and *CD-free-connex* queries similarly to the case with FDs, we can use Lemma 23 and Lemma 24 with Theorem 10 to generalize the dichotomy presented in Section 4 to accommodate CDs.

▶ **Theorem 25.** *Let Q be a CD-acyclic CQ with no self-joins over a schema $\mathcal{S} = (\mathcal{R}, \Delta)$, where Δ is a set of CDs.*

- *If Q is CD-free-connex, then* $\textsc{Enum}_{\Delta}\langle Q\rangle \in \mathsf{DelayC_{lin}}$.
- *If Q is not CD-free-connex, then* $\textsc{Enum}_{\Delta}\langle Q\rangle \notin \mathsf{DelayC_{lin}}$, *assuming that the product of two $n \times n$ boolean matrices cannot be computed in time $O(n^2)$.*

Similarly, we conclude the hardness of self-join-free CD-cyclic CQs over schemas that contain only unary CDs, of the form $(A \to B, c)$ with $|A| = 1$. Combining Lemma 23 with Theorem 16, we have that such queries cannot be evaluated in linear time, assuming that the $\textsc{Tetra}(k)$ problem cannot be solved in linear time for any k.

7 Concluding Remarks

Previous hardness results regarding the enumeration complexity of CQs no longer hold in the presence of dependencies. In this paper, we have shown that some of the queries which where previously classified as hard are in fact tractable in the presence of FDs, and that the others remain intractable. We have classified the enumeration complexity of self-join-free CQs according to their FD-extension. Under previously used complexity assumptions: a query is in $\mathsf{DelayC_{lin}}$ if its extension is free-connex, it is not in $\mathsf{DelayC_{lin}}$ if its extension is acyclic but not free-connex, and it is not even decidable in linear time if the schema has only unary FDs and its extension is cyclic. In addition to our results on constant delay enumeration of CQs with FDs, the tools provided here have immediate implications in other settings, such as for CQs with disequalities, schemas with CDs, and other enumeration classes such as $\mathsf{DelayLin}$.

This work opens up quite a few directions for future work. Our proof for the hardness of FD-cyclic CQs assumes that all FDs are unary. The question of whether this result holds for general FDs, along with the classification of Example 22, remains open. This result, as well as the original one given by Brault-Baron [6] assumes the hardness of the $\textsc{Tetra}(k)$

problem for every k. It will be interesting to see whether we can get the same result based on a weaker assumption. Another possible direction involves CDs. To show that enumerating CD-free-connex CQs can be done in $\mathsf{DelayC_{lin}}$, we require polynomial space to store all printed results. It is unclear whether there exists a solution that requires less space. Finally, we wish to explore how the tools provided here can be used to extend other known results on query enumeration, such as a dichotomy for enumerating CQs [6] with negation, to accommodate FDs.

References

1 Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on*, pages 434–443. IEEE, 2014.

2 Myrto Arapinis, Diego Figueira, and Marco Gaboardi. Sensitivity of counting queries. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, Rome, Italy*, pages 120:1–120:13, 2016.

3 Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *International Workshop on Computer Science Logic*, pages 208–222. Springer, 2007.

4 Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30(3):479–513, 1983. doi:10.1145/2402.322389.

5 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 303–318, 2017.

6 Johann Brault-Baron. *De la pertinence de l'énumération: complexité en logiques propositionnelle et du premier ordre*. PhD thesis, Université de Caen, 2013.

7 Yang Cao, Wenfei Fan, Tianyu Wo, and Wenyuan Yu. Bounded conjunctive queries. *PVLDB*, 7(12):1231–1242, 2014.

8 Nofar Carmeli and Markus Kröll. Enumeration complexity of conjunctive queries with functional dependencies. *CoRR*, abs/1712.07880, 2017. arXiv:1712.07880.

9 Nadia Creignou, Markus Kröll, Reinhard Pichler, Sebastian Skritek, and Heribert Vollmer. On the complexity of hard enumeration problems. In *Language and Automata Theory and Applications - 11th International Conference, LATA 2017, Umeå, Sweden*, pages 183–195, 2017.

10 Arnaud Durand, Nicole Schweikardt, and Luc Segoufin. Enumerating answers to first-order queries over databases of low degree. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '14, pages 121–131, 2014.

11 François Le Gall. Powers of tensors and fast matrix multiplication. In Katsusuke Nabeshima, Kosaku Nagasaka, Franz Winkler, and Ágnes Szántó, editors, *International Symposium on Symbolic and Algebraic Computation, ISSAC '14, Kobe, Japan, July 23-25, 2014*, pages 296–303. ACM, 2014. doi:10.1145/2608628.2608664.

12 Etienne Grandjean. Sorting, linear time and the satisfiability problem. *Ann. Math. Artif. Intell.*, 16:183–236, 1996. doi:10.1007/BF02127798.

13 Benny Kimelfeld. A dichotomy in the complexity of deletion propagation with functional dependencies. In Michael Benedikt, Markus Krötzsch, and Maurizio Lenzerini, editors, *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 191–202. ACM, 2012. doi:10.1145/2213556.2213584.

14 Christos H. Papadimitriou and Mihalis Yannakakis. On the complexity of database queries. *J. Comput. Syst. Sci.*, 58(3):407–427, 1999.

15 Luc Segoufin and Alexandre Vigny. Constant delay enumeration for FO queries over databases with local bounded expansion. In *20th International Conference on Database Theory, ICDT 2017, Venice, Italy*, pages 20:1–20:16, 2017.

16 Yann Strozecki. *Enumeration complexity and matroid decomposition*. PhD thesis, Université Paris Diderot – Paris 7, 2010. Available at `http://www.prism.uvsq.fr/~ystr/these_strozecki`.

17 Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, Las Vegas, Nevada, USA*, pages 645–654, 2010.

Preserving Constraints with the Stable Chase

David Carral
Center for Advancing Electronics Dresden (cfaed), TU Dresden, Germany
david.carral@tu-dresden.de

Markus Krötzsch
Center for Advancing Electronics Dresden (cfaed), TU Dresden, Germany
markus.kroetzsch@tu-dresden.de

Maximilian Marx
Center for Advancing Electronics Dresden (cfaed), TU Dresden, Germany
maximilian.marx@tu-dresden.de

Ana Ozaki
Center for Advancing Electronics Dresden (cfaed), TU Dresden, Germany
ana.ozaki@tu-dresden.de

Sebastian Rudolph
Computational Logic Group, TU Dresden, Germany
sebastian.rudolph@tu-dresden.de

Abstract

Conjunctive query answering over databases with constraints – also known as (tuple-generating) dependencies – is considered a central database task. To this end, several versions of a construction called *chase* have been described. Given a set Σ of dependencies, it is interesting to ask which constraints not contained in Σ that are initially satisfied in a given database instance are preserved when computing a chase over Σ. Such constraints are an example for the more general class of *incidental constraints*, which when added to Σ as new dependencies do not affect certain answers and might even speed up query answering. After formally introducing incidental constraints, we show that deciding incidentality is undecidable for tuple-generating dependencies, even in cases for which query entailment is decidable. For dependency sets with a finite universal model, the core chase can be used to decide incidentality. For the infinite case, we propose the *stable chase*, which generalises the core chase, and study its relation to incidental constraints.

2012 ACM Subject Classification Theory of computation → Database constraints theory, Theory of computation → Logic and databases

Keywords and phrases Incidental constraints, Tuple-generating dependencies, Infinite core chase, Universal Model, BCQ entailment

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.12

Acknowledgements This work is partly supported by the German Research Foundation (DFG) in CRC 912 (HAEC) and in Emmy Noether grant KR 4381/1-1.

1 Introduction

The *chase* [7, 14, 23] is an essential family of algorithms used to solve entailment questions in databases in the presence of constraints, such as computing certain answers to queries in data integration scenarios. Given a database instance $\mathcal{I}$ and a set of dependencies Σ, chase procedures compute an instance that extends $\mathcal{I}$ and satisfies all constraints in Σ, and that is *universal* in the sense that it admits a homomorphism into any other model of $\mathcal{I}$ and Σ. In

particular, such a universal model can be used for query answering, as it entails exactly the certain answers to conjunctive queries over $\mathcal{I}$ and Σ.

Now $\mathcal{I}$ might satisfy constraints that are not part of Σ, and it is a relevant question to ask whether or not they are *preserved* by the chase, i.e., whether they still hold in the computed universal model. This can be viewed as an extension of integrity checks to the virtual, possibly infinite views that are defined by a set of dependencies. Moreover, constraints that are preserved in this sense can safely be assumed to hold, and hence be used in algorithms. For instance, query rewriting algorithms can benefit from additional constraints [22, 25].

For the case of Datalog rules (full dependencies) Σ, constraint preservation is a known problem in databases [1, 28], which is typically further generalised by asking if some set of constraints Γ is implied by Σ given arbitrary input instances $\mathcal{I}$ that merely satisfy certain input constraints Γ'. Constraint preservation then is the special case where $\Gamma = \Gamma'$. Traditionally, one is asking which constraints Γ are satisfied in the (unique, finite) least model of Σ, but there have also been works that consider all (first-order) models [29].

Unfortunately, however, these simple notions of constraint preservation (or implication) are no longer meaningful if we consider more general theories Σ that may contain tuple-generating dependencies. Which constraints are preserved then becomes highly sensitive to the details of the chase, since a constraint might be preserved in some universal models of $\mathcal{I}$ and Σ but not in others. It is often possible to preserve a constraint even if it is not logically entailed by $\mathcal{I}$ and Σ. How can we find out if any universal model preserves a particular constraint, and how can we possibly compute such a model? The answer is not obvious, especially in the general case where universal models are necessarily infinite.

To tackle this problem, we propose the notion of *incidental* constraints to capture the intuitive idea of a constraint being "preservable" (possibly with some effort). Concretely, a constraint ρ is incidental for $\mathcal{I}$ and Σ if adding ρ to Σ does not lead to any additional answer to conjunctive queries over $\mathcal{I}$ (and thereby to many other positive queries). Constraints that do not hold in $\mathcal{I}$ may therefore be incidental, too.

We only require conjunctive query equivalence rather than semantic equivalence, since the main use of the chase is positive query answering. Incidentality therefore is not the same as logical entailment. For example, any constraint whose premise is not entailed (as a Boolean conjunctive query, BCQ) is incidental, and is satisfied by all universal models, yet may not be a entailed. Dependencies can be incidental even if violated in some universal models:

▶ **Example 1.** Consider the dependency $\rho = R(x, y) \to \exists z.R(y, z)$ and an instance $\mathcal{I}$ with a single relation $R(n_0, n_1)$ where n_0 and n_1 are nulls. Then $\mathcal{I}$ and ρ has a universal model that is an infinite R-chain, starting at $R(n_0, n_1)$. The dependency $\rho' = R(y, z) \to \exists x.R(x, y)$ is not satisfied in this model, but is incidental for $\mathcal{I}$ and ρ. Indeed, $\mathcal{I}$ and $\{\rho, \rho'\}$ has a universal model that is a two-way infinite R-chain, which entails the same queries as the one-way infinite chain, but is not a universal model of $\mathcal{I}$ and ρ.

We study incidentality and the related problem of recognising incidental constraints. This problems turns out to be hard: it is on the second level of the arithmetic hierarchy, and remains undecidable even in cases where conjunctive query answering is decidable. We give a complete (and computable) characterisation for theories that admit a finite model. Even for cases where a finitary computation procedure is impossible, we seek a deeper understanding of models that preserve incidental constraints. This leads us to develop a new notion of chase, which we use to establish the existence of core models that characterise both BCQ answers and the entailed incidental relationships. In summary, our main contributions are as follows:

- We formalise a new notion of constraint preservation based on incidental dependencies.
- We show that incidentality is not recursively enumerable in general, and remains undecidable even in restricted cases.
- We show that the *core chase* [14] can be used to decide incidentality for cases where a finite universal model exists.
- We develop the *stable chase* as a generalisation of the core chase to the infinite case.
- We show that the stable chase produces a core that can be used both for query answering and for characterising full incidental dependencies.

Finally, we combine our results to establish the existence of a model that entails the same queries as a universal model and that satisfies exactly the tuple-generating depenencies that are incidental. This model can no longer be universal, but it is a core.

2 Preliminaries

We consider countably infinite, disjoint sets of *constants* Δ_c and of *nulls* Δ_n. A *schema* $\mathcal{S}$ is a finite set of relation symbols, where $\mathsf{ar}(R)$ is the *arity* of $R \in \mathcal{S}$. An *instance* $\mathcal{I}$ over $\mathcal{S}$ assigns to each relation symbol $R \in \mathcal{S}$ a (possibly infinite) $\mathsf{ar}(R)$-ary relation $R^{\mathcal{I}}$ over $\Delta_c \cup \Delta_n$. We often refrain from mentioning the schema $\mathcal{S}$ of an instance $\mathcal{I}$ explicitly, assuming that such a signature has been fixed. The *active domain* of $\mathcal{I}$, denoted by $\Delta^{\mathcal{I}}$, is the set of all *domain elements* that occur in relations of $\mathcal{I}$. We write $\boldsymbol{a}$ for a tuple $\langle a_1, \dots, a_n \rangle$ of domain elements.

Morphisms. Let $\mathcal{I}$ and $\mathcal{J}$ be instances over a schema $\mathcal{S}$. A *homomorphism* $h : \mathcal{I} \to \mathcal{J}$ is a function from $\Delta^{\mathcal{I}}$ to $\Delta^{\mathcal{J}}$ such that (i) $h(c) = c$ for all $c \in \Delta^{\mathcal{I}} \cap \Delta_c$, and (ii) $\boldsymbol{a} \in R^{\mathcal{I}}$ implies $h(\boldsymbol{a}) \in R^{\mathcal{J}}$ for all $R \in \mathcal{S}$ and $\boldsymbol{a} = \langle a_1, \dots, a_n \rangle \in (\Delta^{\mathcal{I}})^{\mathsf{ar}(R)}$, where $h(\boldsymbol{a})$ is short for tuple $\langle h(a_1), \dots, h(a_n) \rangle$. It is *strong* if (ii) is strengthened to require $\boldsymbol{a} \in R^{\mathcal{I}}$ if and only if $h(\boldsymbol{a}) \in R^{\mathcal{J}}$.[1] An *embedding* is an injective strong homomorphism, and an *isomorphism* is a bijective strong homomorphism (i.e., a surjective embedding). An *endomorphism* of $\mathcal{I}$ is a homomorphism $h : \mathcal{I} \to \mathcal{I}$.

Dependencies and Queries. We use a countably infinite set Δ_v of *variables*, disjoint from $\Delta_c \cup \Delta_n$. A *term* is an element $t \in \Delta_v \cup \Delta_c$. We use letters x, y, z, u, v, w and expressions such as $\boldsymbol{x}$ for tuples $\langle x_1, \dots, x_\ell \rangle$ of the corresponding elements. We treat such tuples as sets when order is not relevant. An *atom* is a formula $R(\boldsymbol{t})$ with $R \in \mathcal{S}$ and $|\boldsymbol{t}| = \mathsf{ar}(R)$. First-order formulae are defined as usual. We write $\varphi[\boldsymbol{x}]$ to emphasise that the free variables in φ are a subset of $\boldsymbol{x}$. A *tuple generating dependency* (*TGD*) is a formula of the form

$$\forall \boldsymbol{x}, \boldsymbol{z}.(\varphi[\boldsymbol{x}, \boldsymbol{z}] \to \exists \boldsymbol{y}.\psi[\boldsymbol{x}, \boldsymbol{y}]) \tag{1}$$

where the *body* φ and the *head* ψ are conjunctions of atoms, and ψ contains at least one conjunct. TGDs never contain free variables, hence we usually omit the universal quantifiers. A TGD is *full* if it does not contain existentially quantified variables. A *Boolean conjunctive query* (*BCQ*), or simply a *query*, is a formula of the form $\exists \boldsymbol{y}.\varphi[\boldsymbol{y}]$ with φ a conjunction of atoms. We allow TGDs with empty bodies to assert facts (possibly with existentials), and we often omit $\to$ in this case. We generally assume that Σ denotes a *finite* set of TGDs.

A conjunction of atoms φ (resp. a BCQ $q = \exists \boldsymbol{y}.\varphi[\boldsymbol{y}]$) gives rise to a finite instance $\mathcal{I}_\varphi$ (resp. $\mathcal{I}_q$), obtained by treating φ as a set of relational tuples using a fresh null n_x in place

[1] Strong homomorphisms were called *full* by Deutsch et al. [14].

of each variable x. Conversely, every finite instance $\mathcal{I}$ induces a conjunction $\varphi_{\mathcal{I}}$ that has an atom for every relational tuple, using fresh variables x_n in place of nulls n. The BCQ $q_{\mathcal{I}}$ then is the existential closure of $\varphi_{\mathcal{I}}$. Note that TGDs can encode a given finite instance $\mathcal{I}$ using a dependency $\to q_{\mathcal{I}}$. This is why we will generally state our results for sets Σ of TGDs without mentioning an additional instance.

Universal Models and Cores. Instances naturally correspond to first-order interpretations. We let $\models$ denote first-order modelhood and entailment. Note that, for an instance $\mathcal{I}$ and a finite instance $\mathcal{J}$, we have $\mathcal{I} \models q_{\mathcal{J}}$ iff there exists a homomorphism $h : \mathcal{J} \to \mathcal{I}$. The set of all BCQs modelled (entailed) by an interpretation $\mathcal{I}$ or a set of TGDs Σ is denoted with $\mathsf{BCQ}(\mathcal{I})$ and $\mathsf{BCQ}(\Sigma)$, respectively. A model $\mathcal{J} \models \Sigma$ is *universal* if, for every model $\mathcal{I} \models \Sigma$, there is a homomorphism $h : \mathcal{J} \to \mathcal{I}$. In this case, $\mathsf{BCQ}(\mathcal{J}) = \mathsf{BCQ}(\Sigma)$, i.e., $\mathcal{J}$ and Σ are *BCQ-equivalent* [14]. Two instances $\mathcal{I}$ and $\mathcal{J}$ are *BCQ-equivalent* if $\mathsf{BCQ}(\mathcal{I}) = \mathsf{BCQ}(\mathcal{J})$.

▶ **Definition 2.** An instance $\mathcal{I}$ is a *core* if every endomorphism of $\mathcal{I}$ is an embedding. A core $\mathcal{I}$ is called *a core of* $\mathcal{J}$ if there is an endomorphism h of $\mathcal{J}$ such that $\mathcal{I}$ is the restriction of $\mathcal{J}$ to the image of h.

Definition 2 corresponds to Bauslaugh's property **IN** [5], and has also been used, e.g., in studies of constraint satisfaction [8]. Bauslaugh favours a stronger definition based on isomorphisms instead of endomorphisms (property **ISN**), but this forces cores to be unique up to isomorphism, which is too restrictive for our needs. There are several further definitions of cores, all of which differ only on infinite instances [5, 6]. For finite instances, Definition 2 agrees with the one in [14] and a unique core (up to isomorphism) always exists, whereas (for Definition 2) infinite instances may have no core or several cores (see examples in Section 4).

Applying Rules. A TGD ρ as in (1) is *applicable* to an instance $\mathcal{I}$ if there is a homomorphism $h : \mathcal{I}_\varphi \to \mathcal{I}$. We then extend h to $\mathcal{I}_\psi$ by defining, for all variables $y \in \boldsymbol{y}$ that are existentially quantified, $h(n_y) = n_{y,\rho,h}$ to be a null that is specific for y, ρ, and h, where we assume that all nulls of the form $n_{y,\rho,h}$ exist and are mutually distinct. Let $\rho(\mathcal{I})$ denote the union of $\mathcal{I}$ with all sets of the form $h(\mathcal{I}_\psi)$ for some extended homomorphism $h : \mathcal{I}_\varphi \to \mathcal{I}$. For a set Σ of TGDs, we set $\Sigma(\mathcal{I}) = \bigcup_{\rho \in \Sigma} \rho(\mathcal{I})$.

3 Incidental Dependencies

It is intuitive to ask whether a dependency ρ that holds for a finite instance $\mathcal{I}$ is "preserved" by a given set Σ of TGDs. We formalise this as follows, where we omit $\mathcal{I}$ since it can be captured by a TGD in Σ:

▶ **Definition 3.** A TGD ρ is *incidental* for a set Σ of TGDs if $\mathsf{BCQ}(\Sigma) = \mathsf{BCQ}(\Sigma \cup \{\rho\})$. The set of all incidental TGDs of Σ is denoted $\mathsf{ICDT}(\Sigma)$.

Clearly, $\Sigma \subseteq \mathsf{ICDT}(\Sigma)$. Indeed, every TGD that is logically entailed is also incidental. However, the converse is not true, as illustrated in Example 1 (where the instance $\mathcal{I}$ can be expressed by a TGD $\to \exists x, y.R(x, y)$). In particular, incidental TGDs are not automatically "preserved" in an arbitrary chase procedure, hence we avoid this terminology, though it was used previously, e.g., related to constraint preservation under non-recursive full TGDs [28].

Note that our notion of incidental TGDs is not specific to BCQs. Indeed, BCQ-equivalent sets of TGDs are also equivalent with respect to many other types of negation-free queries, such as Datalog queries and its numerous fragments, including (unions of) conjunctive regular

path queries [18, 11], monadic [12] and linear [19] Datalog queries, (nested) monadically defined queries [27, 9] and many more. Queries with negation, however, are not preserved: in Example 1, $\Sigma \models \exists x.\forall y.\neg R(y,x)$ whereas $\Sigma \cup \{\rho\} \not\models \exists x.\forall y.\neg R(y,x)$.

A noteworthy property is that a TGD is incidental exactly if it does not lead to a newly entailed BCQ in a single derivation step. To state this formally, we use $\rho(p)$ to abbreviate the BCQ $q_{\rho(\mathcal{I}_p)}$ for any BCQ p and TGD ρ.

▶ **Theorem 4.** *For a TGD ρ and a set Σ of TGDs, $\rho \in \mathsf{ICDT}(\Sigma)$ iff $\rho(q) \in \mathsf{BCQ}(\Sigma)$ for every $q \in \mathsf{BCQ}(\Sigma)$.*

Proof. ($\Rightarrow$) For the contrapositive, assume that $q \in \mathsf{BCQ}(\Sigma)$ and $\rho(q) \notin \mathsf{BCQ}(\Sigma)$. Then ρ is applicable to $\mathcal{I}_q$. Let $\mathcal{J}$ be any model of Σ. Since $\mathcal{J} \models q$, there is a homomorphism $\mathcal{I}_q \to \mathcal{J}$, hence ρ is applicable to $\mathcal{J}$. Therefore, any instance $\mathcal{J}' \supseteq \mathcal{J}$ that satisfies ρ entails $\rho(q)$. Since $\mathcal{J}$ was arbitrary, we find that $\Sigma \cup \{\rho\} \models \rho(q)$. Hence ρ is not incidental for Σ.

($\Leftarrow$) Assume that $\rho(q) \in \mathsf{BCQ}(\Sigma)$ for all $q \in \mathsf{BCQ}(\Sigma)$. Suppose for a contradiction that $\Sigma \cup \{\rho\} \models q$ for some $q \notin \mathsf{BCQ}(\Sigma)$. Then there is a finite derivation $\mathcal{I} = \rho_k(\rho_{k-1}(\dots\rho_0(\mathcal{I}_\emptyset)\dots))$ with $\mathcal{I}_\emptyset$ the empty instance and rules $\rho_i \in \Sigma \cup \{\rho\}$, such that $\mathcal{I} \models q$. Let q w.l.o.g. be such that k is minimal. Then $\Sigma \models q_\mathcal{J}$ for $\mathcal{J} = \rho_{k-1}(\dots\rho_0(\mathcal{I}_\emptyset)\dots)$, and we have $\rho_k = \rho$. By the assumption on ρ, also $\Sigma \models \rho(q_\mathcal{J})$. Therefore $\Sigma \models q_\mathcal{I}$, since $\Sigma \models q_\mathcal{I} = q_{\rho(\mathcal{J})}$ is isomorphic to $\rho(q_\mathcal{J})$. Hence, since $\mathcal{I} \models q$ implies $q_\mathcal{I} \models q$, we get the desired contradiction $\Sigma \models q$. ◀

An important insight from the preceding theorem is that incidentality for some set Σ can be established solely on $\mathsf{BCQ}(\Sigma)$.

▶ **Lemma 5.** *For every two BCQ-equivalent sets Σ, Σ' of TGDs, $\mathsf{ICDT}(\Sigma) = \mathsf{ICDT}(\Sigma')$.*

Proof. Let $\rho \in \mathsf{ICDT}(\Sigma)$ be an incidental TGD for Σ. Then, by Theorem 4, $\rho(q) \in \mathsf{BCQ}(\Sigma)$ for every $q \in \mathsf{BCQ}(\Sigma)$. Due to BCQ equivalence, this means $\rho(q) \in \mathsf{BCQ}(\Sigma')$ for every $q \in \mathsf{BCQ}(\Sigma')$, which, by the other direction of Theorem 4, implies that $\rho \in \mathsf{ICDT}(\Sigma')$. The converse follows by symmetry. ◀

One of the uses of this result is to show the following theorem, which asserts that individual incidental TGDs are also jointly incidental, i.e., do not entail any additional BCQs together.

▶ **Theorem 6.** *For every set Σ of TGDs, $\mathsf{BCQ}(\Sigma) = \mathsf{BCQ}(\mathsf{ICDT}(\Sigma))$.*

Proof. Let $q \in \mathsf{BCQ}(\mathsf{ICDT}(\Sigma))$ be a BCQ. Then, by compactness, there is a finite subset $\Gamma = \{\gamma_1, \dots, \gamma_k\} \subseteq \mathsf{ICDT}(\Sigma)$ such that $q \in \mathsf{BCQ}(\Sigma \cup \Gamma)$. But then $\mathsf{BCQ}(\Sigma) = \mathsf{BCQ}(\Sigma \cup \{\gamma_1\}) = \mathsf{BCQ}(\Sigma \cup \{\gamma_1, \gamma_2\}) = \dots = \mathsf{BCQ}(\Sigma \cup \Gamma)$: Since γ_1 is incidental for Σ, we have $\mathsf{BCQ}(\Sigma) = \mathsf{BCQ}(\Sigma \cup \{\gamma_1\})$. By Lemma 5, γ_2 is incidental for $\Sigma \cup \{\gamma_1\}$, i.e., $\mathsf{BCQ}(\Sigma \cup \{\gamma_1\}) = \mathsf{BCQ}(\Sigma \cup \{\gamma_1, \gamma_2\})$. Further applications of Lemma 5 show that γ_k is incidental for $\Sigma \cup \{\gamma_1, \dots, \gamma_{k-1}\}$, yielding the above equality. This shows $q \in \mathsf{BCQ}(\Sigma)$. Hence, $\mathsf{BCQ}(\mathsf{ICDT}(\Sigma)) \subseteq \mathsf{BCQ}(\Sigma)$, and by monotonicity, we also have $\mathsf{BCQ}(\Sigma) \subseteq \mathsf{BCQ}(\mathsf{ICDT}(\Sigma))$. ◀

▶ **Definition 7.** Incidental is the following decision problem. Given a set Σ of TGDs and a TGD ρ, is ρ incidental for Σ?

Since BCQ entailment checking over a set of TGDs is undecidable, it is not surprising that the same is true for Incidental. However, the problem is actually on the second level of the arithmetic hierarchy [26], i.e., strictly harder than query answering, and neither incidental dependencies nor non-incidental dependencies can be recursively enumerated (are in RE):

▶ **Theorem 8.** INCIDENTAL *is* Π^0_2*-complete, and in particular neither in* RE *nor in* coRE.

Proof. For membership note that we can characterise incidentality by quantifying over (finite) derivations (or proofs) in some theory. Indeed, a TGD ρ is incidental for Σ if: for all derivations that show $\Sigma \cup \{\rho\} \models q$ for some BCQ q, there is a derivation that shows $\Sigma \models q$. Using Gödel numbers for representing derivations, this condition can be expressed as a $\forall\exists$-sentence in first-order arithmetic.

We show hardness by many-one reduction from the *universal halting problem*, which is as follows: given a (deterministic) Turing machine $\mathcal{M}$, does $\mathcal{M}$ halt on all inputs? Universal halting is known to be complete for Π^0_2 (see [26, Theorem VIII], and apply Post's Theorem).

For the reduction, we construct for a given TM $\mathcal{M}$ a set $\Sigma_\mathcal{M}$ of TGDs and a full TGD ρ such that ρ is incidental for Σ iff $\mathcal{M}$ halts universally. The rules of $\Sigma_\mathcal{M}$ consist of three parts: Σ_1 ensures that each model contains representations of all possible inputs; Σ_2 simulates $\mathcal{M}$ on a particular input; Σ_3 marks elements of an accepting TM simulation with a specific unary relation halted. The rule ρ then asserts that initial elements in TM simulations are always marked by halted, which is incidental if all runs have indeed terminated. The detailed constructions in each case are given in the appendix. ◀

There are many known classes of TGD sets for which query answering becomes decidable, such as acyclic TGDs or guarded TGDs [15, 16, 2, 3, 21, 13], and INCIDENTAL is indeed in coRE in this case.

▶ **Theorem 9.** *Let $\mathcal{C}$ be a class of sets of TGDs over which BCQ entailment is decidable. There is an algorithm that, given $\Sigma \in \mathcal{C}$, enumerates all TGDs ρ such that $\rho \notin \mathsf{ICDT}(\Sigma)$.*

Proof. Let ρ be an arbitrary TGD. Then ρ is non-incidental iff there is some BCQ q such that either $\Sigma \models q$ but $\Sigma \cup \{\rho\} \not\models q$, or $\Sigma \not\models q$ but $\Sigma \cup \{\rho\} \models q$. Due to monotonicity of TGDs, only the second case can occur. Now, enumerating all q such that $\Sigma \not\models q$ and checking $\Sigma \cup \{\rho\} \models q$ yields a semi-decision procedure for non-incidentality. Using a suitable diagonalisation, we can enumerate all $\rho \notin \mathsf{ICDT}(\Sigma)$. ◀

By Theorem 9, establishing non-incidentality of a given rule ρ is in RE, even if $\Sigma \cup \{\rho\} \notin \mathcal{C}$. On the other hand, INCIDENTAL in general remains undecidable even if BCQ-entailment is decidable, and even when asking only for the incidentality of one fixed full dependency.

▶ **Theorem 10.** *There is a class $\mathcal{C}$ of sets of TGDs for which BCQ answering is decidable, and a full dependency ρ for which $\Sigma \cup \{\rho\} \in \mathcal{C}$ for all $\Sigma \in \mathcal{C}$, such that the following problem is undecidable: given some $\Sigma \in \mathcal{C}$, is ρ incidental for Σ?*

Proof. We show undecidability by reducing the halting problem of deterministic Turing machines when started on the empty tape. Consider a Turing machine $\mathcal{M} = \langle Q, \Gamma, \delta, q_s, q_e \rangle$ as in the proof of Theorem 8, which w.l.o.g. does not return to its initial state q_s in any run. We consider predicate symbols as used in the proof of Theorem 8, and define the set $\tau(\mathcal{M})$ of TGDs to contain the rules Σ_2 as in this proof, together with the additional rules (facts):

$$\to \exists v, w.\mathsf{head}_{q_s}(v) \wedge \mathsf{symbol}_\square(v) \wedge \mathsf{right}(v, w) \wedge \mathsf{symbol}_\square(w) \wedge \mathsf{end}(w) \tag{2}$$

$$\to \exists v.\mathsf{right}(v, v) \wedge \mathsf{right}^+(v, v) \wedge \mathsf{next}(v, v) \wedge \mathsf{end}(v) \wedge \bigwedge_{q \in Q \setminus \{q_s\}} \mathsf{head}_q(v) \wedge \bigwedge_{\sigma \in \Gamma} \mathsf{symbol}_\sigma(v) \tag{3}$$

Here, (2) encodes the initial configuration of $\mathcal{M}$ on the empty tape, which is the start of a Turing machine simulation as effected by Σ_2; and (3) creates an element that stands in

all possible relations not involving head_{q_s}. Let $\rho = \mathsf{head}_{q_e}(x) \to \mathsf{halted}(x)$, and let $\mathcal{C}$ be the class of all TGD sets of the form $\tau(\mathcal{M})$ or $\tau(\mathcal{M}) \cup \{\rho\}$.

BCQ answering over TGD sets of $\mathcal{C}$ is decidable. Indeed, any BCQ that does not contain head_{q_s} is trivially entailed by any $\Sigma \in \mathcal{C}$, due to (3). On the other hand, if a connected component in a BCQ contains head_{q_s}, then it describes a property of a finite initial segment of the simulation of a TM, which can be checked effectively.

For a Turing machine $\mathcal{M}$, the full TGD ρ is incidental for $\tau(\mathcal{M})$ iff $\mathcal{M}$ does *not* halt on the empty input. Indeed, if $\mathcal{M}$ does not halt, then the only occurrence of head_{q_e} in a universal model of $\tau(\mathcal{M})$ is in the element created due to (3), and ρ is already satisfied by this element. Conversely, if $\mathcal{M}$ halts, then head_{q_e} occurs for an element that is connected to the starting sequence a created due to (2). Hence, there is a BCQ of the form $q = \exists \boldsymbol{x}.\mathsf{head}_{q_s}(x_0) \wedge p_1(x_0, x_1) \wedge \ldots \wedge p_n(x_{n-1}, x_n) \wedge \mathsf{halting}(x_n)$ with $p_i \in \{\mathsf{right}, \mathsf{next}\}$, such that $\tau(\mathcal{M}) \not\models q$ and $\tau(\mathcal{M}) \cup \{\rho\} \models q$. ◀

The previous result is particularly interesting since it only considers situations where query answering is decidable, both for the TGDs with and without the candidate dependency ρ. In spite of this general result, concrete classes of TGD sets with decidable BCQ entailment may allow us to decide Incidental, as discussed in the next section.

4 Cores and Incidentals

In this section we relate incidental dependencies with the notion of a core of an instance. Theorem 11 shows that if a set of TGDs has a finite universal model $\mathcal{I}$ then all incidental dependencies follow from the core of $\mathcal{I}$. It then follows from Theorem 11 that if the core chase [14] (also, see Definition 19) terminates then Incidental is decidable. In the following, let $\mathsf{core}(\mathcal{I})$ denote the core of a finite instance $\mathcal{I}$.

▶ **Theorem 11.** *Let Σ be a set of TGDs with a finite universal model $\mathcal{I}$ and let ρ be a TGD. Then, $\rho \in \mathsf{ICDT}(\Sigma)$ iff $\mathsf{core}(\mathcal{I}) \models \rho$.*

Proof. ($\Rightarrow$) Consider $\rho = \varphi[\boldsymbol{x}, \boldsymbol{z}] \to \exists \boldsymbol{y}.\psi[\boldsymbol{x}, \boldsymbol{y}]$ with $\rho \in \mathsf{ICDT}(\Sigma)$. Let $h : \mathcal{I}_\varphi \to \mathsf{core}(\mathcal{I})$ be some homomorphism, and assume that it is extended to $\mathcal{I}_\psi$ using new nulls to map to as defined before. Then set $\mathcal{J} := \mathsf{core}(\mathcal{I}) \cup h(\mathcal{I}_\psi)$, i.e. the core with the consequence of ρ under h added (possibly by adding new elements). $\mathcal{J}$ is finite since $\mathsf{core}(\mathcal{I})$ is, and clearly $\rho(\mathsf{core}(\mathcal{I})) \models q_\mathcal{J}$. Therefore $\Sigma \cup \{\rho\} \models q_\mathcal{J}$, and hence $\Sigma \models q_\mathcal{J}$ by incidentality. So $\mathsf{core}(\mathcal{I}) \models q_\mathcal{J}$ since $\mathsf{core}(\mathcal{I})$ is a universal model, and we obtain a homomorphism $g : \mathcal{J} \to \mathsf{core}(\mathcal{I})$. But then the restriction of g to elements of $\Delta^{\mathsf{core}(\mathcal{I})}$ is an endomorphism, and therefore an embedding since $\mathsf{core}(\mathcal{I})$ is a core. Every embedding on a finite core is an isomorphism [20, 5], so g has an inverse $g^- : \mathsf{core}(\mathcal{I}) \to \mathsf{core}(\mathcal{I})$. For $\mathcal{K} = g(h(\mathcal{I}_\psi \cup \mathcal{I}_\psi))$ we have $\mathcal{K} \subseteq \mathsf{core}(\mathcal{I})$ and hence $g^-(\mathcal{K}) \subseteq \mathsf{core}(\mathcal{I})$. Since $g^-(g(h(\mathcal{I}_\psi))) = h(\mathcal{I}_\psi)$, we can find a homomorphism h' such that $h'(\mathcal{I}_\varphi) = h(\mathcal{I}_\varphi)$ and $h'(\mathcal{I}_\psi) \subseteq g^-(\mathcal{K}) \subseteq \mathsf{core}(\mathcal{I})$ (h' may differ from h in the choice of null values for existentially quantified variables). This shows that ρ is satisfied by $\mathsf{core}(\mathcal{I})$ for the particular match h. Since h was arbitrary, we obtain $\mathsf{core}(\mathcal{I}) \models \rho$.

($\Leftarrow$) This follows by direct application of the definitions. ◀

Given this connection between finite cores and incidental dependencies, one may ask whether it extends to cases where the set of TGDs does not admit a finite universal model. Unfortunately, this it not the case: Example 1 shows a case where an incidental dependency does not hold in a universal model that is in fact a core (the one-way infinite chain).

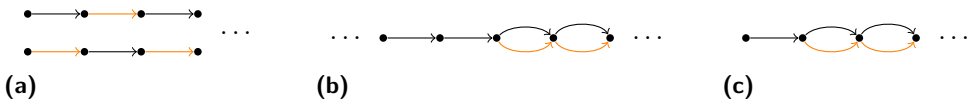


Figure 1 Universal models that have (a) two non-isomorphic cores, (b) no core, and (c) a core that is not a model, where R is black and S is orange (grey).

This discrepancy between incidentals and cores goes together with a general loss of good properties of the core on infinite models. Finite instances (i) always have a core, which is (ii) unique up to isomorphism [17, 20], and (iii) the core of a finite universal model of a set of TGDs is also a universal model [14]. Examples 12, 13, and 14 show that we no longer have any of these properties when dealing with infinite universal models.

▶ **Example 12.** Let Σ consist of the following TGDs:

$$\exists x, y.R(x, y) \qquad \exists x, y.S(z, w) \qquad R(x, y) \to \exists z.S(y, z) \qquad S(x, y) \to \exists z.R(y, z)$$

Figure 1a illustrates a universal model $\mathcal{I}$ of Σ. The upper and the lower chain of relations each by itself is a core of $\mathcal{I}$, but the chains are not isomorphic, so property (ii) does not hold.

▶ **Example 13.** Let Σ consist of the following three TGDs:

$$\exists x, y.R(x, y) \wedge S(x, y) \qquad R(x, y) \wedge S(x, y) \to \exists z.R(y, z) \wedge S(y, z) \qquad R(y, z) \to \exists x.R(x, y)$$

Figure 1b illustrates a universal model of Σ, which is not a core, since there are non-embedding endomorphisms that map parts of the single chain into the double chain. In fact, one can see that this instance does not have a core.

▶ **Example 14.** Let Σ consist of the following TGDs:

$$\exists x, y.R(x, y) \wedge S(x, y) \qquad R(x, y) \wedge S(x, y) \to \exists z.R(y, z) \wedge S(y, z) \qquad S(y, z) \to \exists x.R(x, y)$$

Figure 1c shows a universal model $\mathcal{I}$ of Σ. It is not a core, since there is a non-embedding endomorphism that maps each element to its right neighbour. This results in an instance that is isomorphic to $\mathcal{I}$ with the left-most node and its R-relation removed, which is a core of $\mathcal{I}$ but not a model for the third rule in Σ.

Nevertheless, cores can be relevant in finding instances that satisfy incidental dependencies. To this end, we consider a particularly well-behaved type of core that can be obtained as a limit of a growing sequence of finite cores.

▶ **Definition 15** (Core Cover)**.** An instance $\mathcal{J}$ has a *core cover* if there are finite subinstances $\mathcal{J}_0 \subseteq \mathcal{J}_1 \subseteq \mathcal{J}_2 \subseteq \ldots$ with $\mathcal{J} = \bigcup_{i \geq 0} \mathcal{J}_i$ such that, for all $\mathcal{J}_i$, every homomorphism $h : \mathcal{J}_i \to \mathcal{J}$ is an embedding.

▶ **Theorem 16.** *If an instance has a core cover then it is a core.*

Proof. Consider an instance $\mathcal{J}$ with core cover $(\mathcal{J}_i)_{i \geq 0}$, and an endomorphism $h : \mathcal{J} \to \mathcal{J}$. By Definition 15, the restriction $h_i : \mathcal{J}_i \to \mathcal{J}$ is an embedding for all $i \geq 0$. With $\mathcal{J} = \bigcup_{i \geq 0} \mathcal{J}_i$ and $\mathcal{J}_i \subseteq \mathcal{J}_{i+1}$ it follows that h is an embedding, otherwise, since injectivity and being strong both are finitary conditions, there would be a non-embedding $h_i : \mathcal{J}_i \to \mathcal{J}$. ◀

We remark that the condition that $\mathcal{J}_i \subseteq \mathcal{J}_{i+1}$ is needed for Theorem 16 to hold. Figure 2 illustrates an instance that satisfies the remaining conditions of Definition 15 for a set of disjoint instances $(\mathcal{J}_i)_{i \geq 0}$, but which is not a core.

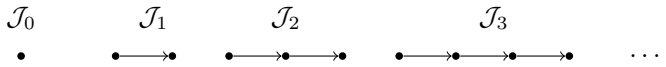


Figure 2 An instance that satisfies most conditions of Definition 15 but is not a core.

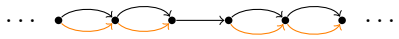


Figure 3 A core without a core cover, using two relations R (black) and S (orange/grey).

▶ **Example 17.** Having a core cover is a sufficient but not a necessary condition for an instance to be a core. Figure 3 illustrates an instance $\mathcal{I}$ that is a core. Indeed, any endomorphism must preserve the adjacency in this two-way infinite chain. But since one pair of elements is not S-related, only this very same pair can be mapped to this position in the chain, so the only endomorphism is the identity mapping.

However, $\mathcal{I}$ has no core cover, since any finite subset of $\mathcal{I}$ that contains the pair without the S connection can be mapped by a non-strong endomorphism into a sufficiently long fully R-S-connected segment of $\mathcal{I}$.

The next theorem shows that cores with core covers can characterise the set of full incidental dependencies for a set of TGDs.

▶ **Theorem 18.** *Let Σ be a set of TGDs and let $\mathcal{I}$ be an instance. Assume that $\mathsf{BCQ}(\mathcal{I}) = \mathsf{BCQ}(\Sigma)$ and $\mathcal{I}$ has a core cover. Then, $\rho \in \mathsf{ICDT}(\Sigma)$ iff $\mathcal{I} \models \rho$, for any full dependency ρ.*

Proof. ($\Rightarrow$) Let $(\mathcal{I}_i)_{i\geq 0}$ be a core cover for $\mathcal{I}$, and consider a full dependency $\rho : \varphi \to \psi$ that is incidental for Σ. If $\mathcal{I} \models \varphi$ for some homomorphism $h : \mathcal{I}_\varphi \to \mathcal{I}$, then there is $\mathcal{I}_i$ such that h can be considered as a homomorphism $\mathcal{I}_\varphi \to \mathcal{I}_i$. Let $\mathcal{J} := \mathcal{I}_i \cup h(\mathcal{I}_\psi)$, where we note that h does not introduce new nulls since ρ is full. Similar to the proof of Theorem 11, we find that $\Sigma \cup \{\rho\} \models q_\mathcal{J}$. Therefore $\mathcal{I} \models q_\mathcal{J}$ as ρ is incidental, and there is a corresponding homomorphism $g : \mathcal{J} \to \mathcal{I}$. Since $\Delta^\mathcal{J} = \Delta^{\mathcal{I}_i}$, g is a homomorphism $g : \mathcal{I}_i \to \mathcal{I}$, and therefore an embedding (Definition 15). This shows that $h(\mathcal{I}_\varphi \cup \mathcal{I}_\psi) \subseteq \mathcal{I}_i$ as required. Since h and $\mathcal{I}_i$ was arbitrary, we conclude that $\mathcal{I} \models \rho$.

($\Leftarrow$) This follows by direct application of the definitions. ◀

Given Theorem 18 and the observation that a core cover is closely related to a bottom-up construction of a core, one naturally wonders if a chase-like procedure could be used to obtain a suitable model. The prime candidate is the *core chase* of Deutsch et al. [14]:

▶ **Definition 19.** The *core chase sequence* for a set Σ of TGDs is a sequence $\mathcal{I}_0, \mathcal{I}_1, \ldots$ of instances, where $\mathcal{I}_0$ is the empty instance, and, for each $i > 0$, $\mathcal{I}_i$ is the core of $\Sigma(\mathcal{I}_{i-1})$. A finite core chase sequence $\mathcal{I}_0, \ldots, \mathcal{I}_\ell$ is *terminating* if $\mathcal{I}_\ell \models \Sigma$, and in this case, $\mathcal{I}_\ell$ is called the *core chase*.

Intuitively, the procedure defined by Deutsch et al. consists on applying the rules and taking the core of the resulting instance in each step. Deutsch et al. do not define the core chase for cases where Σ require infinite models, and indeed the limit of infinite core chase sequences is not defined here. While this issue can be repaired by using a more sophisticated definition, the deeper problem is that the result of applying the rules and then taking the core in each step may not be a core. This can be seen, e.g., from the TGDs in Example 12, on which an infinite core chase would simply produce the universal model shown in Figure 1b, which is not a core.

5 The Stable Chase

In the following section, we show that all sets of TGDs admit a BCQ-equivalent model that is a core and that characterises full incidental dependencies. To this end, we introduce the stable chase, a novel variant of the chase. Our approach can be viewed as a generalisation of the core chase where core computation is performed by looking for non-embedding homomorphisms of an instance into any future instance along a chase sequence. If such a homomorphism is found, all instances in the current chase sequence are rewritten as follows:

▶ **Definition 20.** Consider a homomorphism $h : \mathcal{I} \to \mathcal{J}$ on finite instances over $\mathcal{S}$, and let $\prec$ be a strict total order on $\Delta^{\mathcal{I}}$. The *h-rewriting* of an instance $\mathcal{K}$ is obtained as follows:
1. For all $R \in \mathcal{S}$ and $\boldsymbol{a} \in (\Delta^{\mathcal{K}} \cap \Delta^{\mathcal{I}})^{\mathsf{ar}(R)}$, with $h(\boldsymbol{a}) \in R^{\mathcal{J}}$, insert $\boldsymbol{a} \in R^{\mathcal{K}}$.
2. Replace all $a \in \Delta^{\mathcal{K}} \cap \Delta^{\mathcal{I}}$ by the $\prec$-least element $b \in \Delta^{\mathcal{K}} \cap \Delta^{\mathcal{I}}$ for which $h(a) = h(b)$.

The h-rewriting of a sequence of instances is the sequence of h-rewritings of its members.

▶ **Example 21.** Let $\mathcal{I}_{i,j}$ be the instance occurring in the i-th row, j-th column of Figure 4. Moreover, let $h : \mathcal{I}_{3,2} \to \mathcal{I}_{3,3}$ be the homomorphism that maps n_i to n_{i+1} for every $-1 \leq i \leq 2$. Then, $\mathcal{I}_{4,2}$ is the h-rewriting of $\mathcal{I}_{3,2}$, and (the sequence) $\mathcal{I}_{4,1}, \mathcal{I}_{4,2}, \mathcal{I}_{4,3}$ is the h-rewriting of (the sequence) $\mathcal{I}_{3,1}, \mathcal{I}_{3,2}, \mathcal{I}_{3,3}$.

We proceed with the definition of a *stabilising chase sequence* for a set of TGDs, which is a chase sequence that evolves in the sense that also previously derived instances may be modified at a later stage. The limit of this construction will yield a chase sequence from which we can obtain the potentially infinite core we are looking for.

▶ **Definition 22** (Stabilising Chase Sequence)**.** A *stabilising chase sequence* for a set Σ of TGDs is a series $\boldsymbol{\mathcal{Q}} = \boldsymbol{\mathcal{Q}}_0, \boldsymbol{\mathcal{Q}}_1, \ldots$ of chase sequences. Each $\boldsymbol{\mathcal{Q}}_k = \boldsymbol{\mathcal{Q}}_{k,0} \cdots \boldsymbol{\mathcal{Q}}_{k,\ell(k)}$ is a finite chase sequence of length $\ell(k) + 1$ consisting of instances $\boldsymbol{\mathcal{Q}}_{k,i}$, such that the following hold:
1. $\boldsymbol{\mathcal{Q}}_0$ is the singleton sequence containing the empty instance;
2. for all $k \geq 0$, either
   - **(2.a)** $\boldsymbol{\mathcal{Q}}_{k+1} = \boldsymbol{\mathcal{Q}}_{k,0}, \cdots, \boldsymbol{\mathcal{Q}}_{k,\ell(k)}, \Sigma(\boldsymbol{\mathcal{Q}}_{k,\ell(k)})$ is $\boldsymbol{\mathcal{Q}}_k$ extended by $\Sigma(\boldsymbol{\mathcal{Q}}_{k,\ell(k)})$, or
   - **(2.b)** $\boldsymbol{\mathcal{Q}}_{k+1}$ is the h-rewriting of $\boldsymbol{\mathcal{Q}}_k$ for some homomorphism $h : \boldsymbol{\mathcal{Q}}_{k,i} \to \boldsymbol{\mathcal{Q}}_{k,j}$ with $0 \leq i \leq j$ that is not an embedding,

where we require that the order $\prec$ from Definition 20 is an extension of the (partial) order in which new nulls are introduced, and that all possible rewritings will eventually be applied: if there is a homomorphism $h : \boldsymbol{\mathcal{Q}}_{k,i} \to \boldsymbol{\mathcal{Q}}_{k,j}$ as in (2.b), then there is $k' > k$ such that h is an embedding from the sub-structure of $\boldsymbol{\mathcal{Q}}_{k',i}$ on which h is defined to $\boldsymbol{\mathcal{Q}}_{k,j}$.

Our requirement on $\prec$ ensures that in cases where two elements are merged by a homomorphism in step (2.b), we will always pick one as a representative that has the longest history in the chase sequence. This ensures monotone growth of the domain within a sequence.

While we define the stabilising chase sequence $\boldsymbol{\mathcal{Q}} = \boldsymbol{\mathcal{Q}}_0, \boldsymbol{\mathcal{Q}}_1, \ldots$ to be infinite, it may happen that neither new derivations nor core constructions are possible at some stage. The process can still continue with step (2.a), appending copies of the last instance of the chase sequence, even if they contain no new derivations. Finite termination of the chase is therefore captured in the sequence becoming constant at some point.

▶ **Example 23.** Figure 4 illustrates a stabilising chase sequence $\boldsymbol{\mathcal{Q}}$ for the set of TGDs from Example 13. $\boldsymbol{\mathcal{Q}}_4$ is the h-rewriting of $\boldsymbol{\mathcal{Q}}_3$ for the non-strong homomorphism h denoted with dotted arrows in the figure.

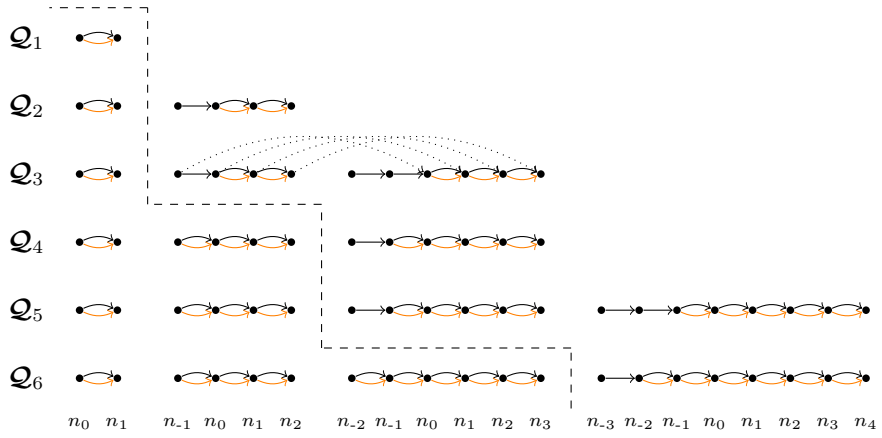


Figure 4 Stabilising chase sequence $\mathcal{Q}$ of Example 13 without the initial sequence $\mathcal{Q}_0$; relations R and S are denoted in black and orange (grey), respectively; domain elements are named below each column of instances.

▶ **Example 24.** A stabilising chase sequence might not be unique. For the set Σ of TGDs from Example 12, parallel chase steps as in (2.a) of Definition 22 yield instances that contain finite initial segments $R(a_0, a_1), S(a_1, a_2), \ldots$ and $S(b_0, b_1), R(b_1, b_2), \ldots$ of parallel chains as in Figure 1a. Non-embedding homomorphisms collapse the lower chain into (a longer future version of) the upper, or vice versa. In each case, the chase produces initial segments of a single infinite chain, which might begin with R or S depending on the chosen homomorphism.

For a particular stabilising chase sequence, however, the instances occurring in the i-th positions of the sequence will eventually stabilise to a unique structure.

▶ **Definition 25.** An instance $\mathcal{I}$ is *stable for position* i in a stabilising chase sequence $\mathcal{Q}$ if there is $k \geq 0$ such that $\mathcal{I} = \mathcal{Q}_{k',i}$ for all $k' \geq k$.

▶ **Lemma 26.** *There is a unique stable instance for every stabilising chase sequence* $\mathcal{Q}$ *and position* $i \geq 0$. *This stable instance is a core.*

Proof. There are three ways in which the finite structure $\mathcal{Q}_{\ell,i}$ may evolve for some $\ell \geq 0$: (1) $\mathcal{Q}_{\ell,i} = \mathcal{Q}_{\ell+1,i}$; (2) $\Delta^{\mathcal{Q}_{\ell,i}} \supset \Delta^{\mathcal{Q}_{\ell+1,i}}$; or (3) $R^{\mathcal{Q}_{\ell,i}} \subset R^{\mathcal{Q}_{\ell+1,i}}$ for some relational symbol R. The (not mutually exclusive) cases (2) and (3) can only occur for a finite number of times. For (2), it is clear that the finite domain cannot decrease in size infinitely often. Moreover, domain elements are only ever renamed if two of them are merged by a homomorphism during rewriting. The finite bound for (3) follows since there can only be at most finitely many relations over a finite domain. Therefore, there is some k for which $\mathcal{Q}_{k,i}$ is stable.

Stable instances are cores: otherwise they would admit a non-embedding endomorphism, which would eventually be used in step (2.b) of Definition 22, contradicting stability. ◀

We may therefore denote the stable instance for position i in $\mathcal{Q}$ by $\mathsf{st}(\mathcal{Q}, i)$, and use the sequence of stable instances to define an infinite structure:

▶ **Definition 27** (Stable Chase)**.** If $\mathcal{Q}$ is a stabilising chase sequence for some set Σ of TGDs, then $(\mathsf{st}(\mathcal{Q}, i))_{i\geq 0}$ is a *stable chase sequence* for Σ, and $\bigcup_{i\geq 0} \mathsf{st}(\mathcal{Q}, i)$ is a *stable chase* for Σ.

▶ **Example 28.** In Figure 4, all instances below the dashed line are stable in the stabilising chase sequence $\mathcal{Q}$ from Example 23 (using the TGDs Σ from Example 13). The corresponding

stable chase sequence is $(\boldsymbol{\mathcal{S}}_i)_{i\geq 0}$ where, for every $i \geq 0$, $\boldsymbol{\mathcal{S}}_i$ is a chain of length $2i$ of elements connected by R and S. This sequence $\boldsymbol{\mathcal{S}}$ is unique up to isomorphism in this case. The stable chase for Σ then is a two-way infinite chain of elements sequentially connected by R and S.

6 Properties of the Stable Chase

We start by showing that every set of TGDs admits a stable chase. We then show that the stable chase algorithm yields a model of Σ (Theorem 30) that is BCQ-equivalent to Σ (Theorem 31), and that is a core (Theorem 32). We show that it satisfies all full incidental dependencies (Theorem 33) and that it coincides with the result of the core chase in finite cases (Theorem 34). Nevertheless, we observe that the stable chase is neither unique nor a universal model. Finally, we show the existence of another BCQ-equivalent model that is a core and entails all incidental dependencies.

▶ **Theorem 29.** *Every set of TGDs has a stable chase sequence.*

Proof. We show that every set of TGDs admits a stabilising chase sequence $\boldsymbol{\mathcal{Q}}$. Indeed, let $\boldsymbol{\mathcal{Q}} = \boldsymbol{\mathcal{Q}}_0, \boldsymbol{\mathcal{Q}}_1, \ldots$ be a stabilising chase sequence constructed as follows:

1. Set $\boldsymbol{\mathcal{Q}}_0$ as the singleton sequence containing the empty instance.
2. For every $k \geq 0$: If every homomorphism $h : \boldsymbol{\mathcal{Q}}_{k,i} \to \boldsymbol{\mathcal{Q}}_{k,j}$ for every $0 \leq i \leq j$ is an embedding, then let $\boldsymbol{\mathcal{Q}}_{k+1} = \boldsymbol{\mathcal{Q}}_{k,0} \cdots \boldsymbol{\mathcal{Q}}_{k,\ell(k)} \Sigma(\boldsymbol{\mathcal{Q}}_{k,\ell(k)})$. Otherwise, $\boldsymbol{\mathcal{Q}}_{k+1}$ is the h-rewriting of $\boldsymbol{\mathcal{Q}}_k$ with h some (arbitrarily chosen) non-embedding homomorphism from some instance of $\boldsymbol{\mathcal{Q}}_k$ to another.

It is clear that the resulting series $\boldsymbol{\mathcal{Q}}$ satisfies 1 and 2 from Definition 22.

It remains to verify the fairness condition on the application of step (2.b). Consider some $k \geq 0$, some $0 \leq i \leq j$, and some non-embedding homomorphism $h : \boldsymbol{\mathcal{Q}}_{k,i} \to \boldsymbol{\mathcal{Q}}_{k,j}$. Then, let $\boldsymbol{\mathcal{Q}}_{k'}$ be the sequence in $\boldsymbol{\mathcal{Q}}$ with the same length as $\boldsymbol{\mathcal{Q}}_k$ such that k' is maximal (note that, $k' \geq k$). By item (2) and Definition 22, every homomorphism from $\boldsymbol{\mathcal{Q}}_{k'',i}$ to $\boldsymbol{\mathcal{Q}}_{k'',j}$ with $k'' \geq k'$ is an embedding. Moreover, we can show via induction that there is a homomorphism $h' : \boldsymbol{\mathcal{Q}}_{k,j} \to \boldsymbol{\mathcal{Q}}_{k'',j}$ for every $k'' \geq k'$. Note that, given some $k'' \geq k$, the existence of a non-embedding homomorphism $h'' : \boldsymbol{\mathcal{Q}}_{k'',i} \to \boldsymbol{\mathcal{Q}}_{k,j}$ would imply the existence of another homomorphism from $\boldsymbol{\mathcal{Q}}_{k'',i}$ to $\boldsymbol{\mathcal{Q}}_{k'',j}$ which is not an embedding either (namely, $h' \circ h''$). Hence, for every $k'' \geq k'$, every homomorphism $h'' : \boldsymbol{\mathcal{Q}}_{k'',i} \to \boldsymbol{\mathcal{Q}}_{k,j}$ is an embedding. ◀

▶ **Theorem 30.** *If $\mathcal{C}$ is a stable chase for Σ, then $\mathcal{C} \models \Sigma$.*

Proof. Let $\boldsymbol{\mathcal{Q}}$ be the stabilising chase sequence from which $\mathcal{C}$ was extracted. Consider any rule $\varphi \to \exists \boldsymbol{y}.\psi \in \Sigma$ that is applicable to $\mathcal{C}$ based on some homomorphism $h : \mathcal{I}_\varphi \to \mathcal{C}$. Since $\mathcal{I}_\varphi$ is finite, there is $i \geq 0$ such that h restricts to a homomorphism $\mathcal{I}_\varphi \to \mathsf{st}(\boldsymbol{\mathcal{Q}}, i)$. Let k be the least number such that $\boldsymbol{\mathcal{Q}}_{k,i} = \mathsf{st}(\boldsymbol{\mathcal{Q}}, i)$. By Definition 22, we find that $\boldsymbol{\mathcal{Q}}_{k,i} \subseteq \boldsymbol{\mathcal{Q}}_{k,j}$ for all $i \leq j \leq \ell(k)$. Moreover, there is $k' > k$ with $\ell(k') = \ell(k) + 1$ and $\boldsymbol{\mathcal{Q}}_{k',\ell(k)+1} = \Sigma(\boldsymbol{\mathcal{Q}}_{k'-1,\ell(k)})$ (step 2.a). Since $\mathsf{st}(\boldsymbol{\mathcal{Q}}, i) = \boldsymbol{\mathcal{Q}}_{k,i} = \boldsymbol{\mathcal{Q}}_{k'-1,i} \subseteq \boldsymbol{\mathcal{Q}}_{k'-1,\ell(k)}$, rule $\varphi \to \exists \boldsymbol{y}.\psi$ is applicable to $\boldsymbol{\mathcal{Q}}_{k'-1,\ell(k)}$ under h. Therefore, $\boldsymbol{\mathcal{Q}}_{k',\ell(k)+1}$ contains the result of this rule application, and by Definition 20 this remains true (possibly for some renaming of new nulls) in $\mathsf{st}(\boldsymbol{\mathcal{Q}}, \ell(k) + 1)$ and hence in $\mathcal{C}$. ◀

▶ **Theorem 31.** *If $\mathcal{C}$ is a stable chase for Σ, then $\mathcal{C}$ and Σ are BCQ-equivalent.*

Proof. By Theorem 30 and the definition of BCQ entailment, $\Sigma \models q$ implies $\mathcal{C} \models q$ for all BCQs q.

For the converse, let $\boldsymbol{\mathcal{Q}}$ be some stabilising chase sequence for $\mathcal{C}$. We show that $\mathcal{C} \models q$ implies $\Sigma \models q$ for every BCQ q. By compactness, it suffices to show that $\mathsf{st}(\boldsymbol{\mathcal{Q}}, i) \models q$ implies $\Sigma \models q$ for all for any $i \geq 0$. We show the stronger claim $\boldsymbol{\mathcal{Q}}_{k,i} \models q$ implies $\Sigma \models q$ for all $i \geq 0$ by induction on k.

For the base case, $\boldsymbol{\mathcal{Q}}_{0,0}$ is the empty instance, and the claim is immediate. Now assume the claim holds for all instances of $\boldsymbol{\mathcal{Q}}_k$. Definition 22 has two ways for constructing $\boldsymbol{\mathcal{Q}}_{k+1}$:

(2.a) Then the only new instance is $\Sigma(\boldsymbol{\mathcal{Q}}_{k,\ell(k)})$. Since the claim holds for $\mathcal{I} = \boldsymbol{\mathcal{Q}}_{k,\ell(k)}$, we find that $\Sigma \models q_{\mathcal{I}}$ for the corresponding BCQ $q_{\mathcal{I}}$. Therefore, any rule application that is possible on $\mathcal{I}$ is possible (up to isomorphism) in any model of Σ, and hence $\Sigma \models q_{\Sigma(\mathcal{I})}$, which entails the claim.

(2.b) Let $h : \boldsymbol{\mathcal{Q}}_{k,i} \to \boldsymbol{\mathcal{Q}}_{k,j}$ be the homomorphism used for the rewriting. Then h restricts to a homomorphism $\boldsymbol{\mathcal{Q}}_{k+1,i} \to \boldsymbol{\mathcal{Q}}_{k,j}$. By Definition 22, we find that $\boldsymbol{\mathcal{Q}}_{k+1,i'} \subseteq \Sigma(\boldsymbol{\mathcal{Q}}_{k+1,i'-1})$ for all $i' > i$. Therefore, $\boldsymbol{\mathcal{Q}}_{k+1,\ell(k)} \subseteq \Sigma^{\ell(k)-i}(\boldsymbol{\mathcal{Q}}_{k+1,i})$ (‡). It suffices to consider BCQs q that are entailed by $\boldsymbol{\mathcal{Q}}_{k+1,\ell(k)}$ (where $\ell(k) = \ell(k+1)$), since they subsume all BCQ entailment in any instance of $\boldsymbol{\mathcal{Q}}_{k+1}$. By (‡), $\boldsymbol{\mathcal{Q}}_{k+1,\ell(k)} \models q$ implies $\boldsymbol{\mathcal{Q}}_{k+1,i}, \Sigma \models q$. Using the homomorphism $h : \boldsymbol{\mathcal{Q}}_{k+1,i} \to \boldsymbol{\mathcal{Q}}_{k,j}$, we get $\boldsymbol{\mathcal{Q}}_{k,j}, \Sigma \models q$ and hence $\Sigma \models q$.

◀

▶ **Theorem 32.** *If $\mathcal{C}$ is a stable chase for Σ, then $\mathcal{C}$ is a core.*

Proof. By Definition 27, there is some rewritten chase sequence $\boldsymbol{\mathcal{S}} = \boldsymbol{\mathcal{S}}_0, \boldsymbol{\mathcal{S}}_1, \ldots$ with $\mathcal{C} = \bigcup_{i \geq 0} \boldsymbol{\mathcal{S}}_i$. Moreover, for every $i \geq j \geq 0$ and every homomorphism $h : \boldsymbol{\mathcal{S}}_i \to \boldsymbol{\mathcal{S}}_j$, h is an embedding. Since every element of $\boldsymbol{\mathcal{S}}$ is finite, every homomorphism h mapping such element to $\mathcal{C}$ is also an embedding. Since $\boldsymbol{\mathcal{S}}_{i-1} \subseteq \boldsymbol{\mathcal{S}}_i$ for every $i \geq 1$, we conclude that $\boldsymbol{\mathcal{S}}$ is a core cover for $\mathcal{C}$. Therefore, we can apply Theorem 16 to show that the theorem follows. ◀

The previous observation that the stable chase sequence yields a core cover, together with the BCQ-equivalence of stable chase and Σ (Theorem 31), lets us apply Theorem 18 to conclude that the stable chase does indeed characterise the full incidental dependencies:

▶ **Theorem 33.** *Every stable chase of Σ entails exactly those full dependencies that are incidental for Σ.*

As one would expect in the light of Theorem 8, the stable chase does not constitute a semi-decision procedure for incidentality or non-incidentality. On the one hand, the stable chase may not terminate, on the other hand we cannot even decide if a given finite instance in a stabilising chase sequence is already stable.

The core chase can be viewed as a special case of the stable chase procedure, since it can be obtained by prioritising step (2.b) in Definition 22, while applying it only to the last instance in a chase sequence $\boldsymbol{\mathcal{Q}}_k$ (this forces the homomorphism that is used to be an endomorphism). For finite models, this does not change the outcome, and indeed the stable chase coincides with the core chase whenever the latter is defined:

▶ **Theorem 34.** *If a set Σ of TGDs has a finite universal model, then the stable chase over Σ is equal to the result of the core chase, up to isomorphism.*

Proof. Deutsch et al. showed that the core chase yields a finite universal model in this case [14, Theorem 7]. Let $\mathcal{U}$ be this model, and let $\mathcal{C}$ be a stable chase of Σ. Since $\mathcal{U}$ is universal, there is a homomorphism $h : \mathcal{C} \to \mathcal{U}$, since $\mathcal{C}$ is a model (Theorem 30). Moreover, since $\mathcal{U}$ is finite, $\mathcal{U} \models q_{\mathcal{U}}$, and since $\mathcal{U}$ and $\mathcal{C}$ are BCQ-equivalent (Theorem 31), there is a homomorphism $h' : \mathcal{U} \to \mathcal{C}$. Therefore, the function $h \circ h'$ is an endomorphism over $\mathcal{C}$ with a

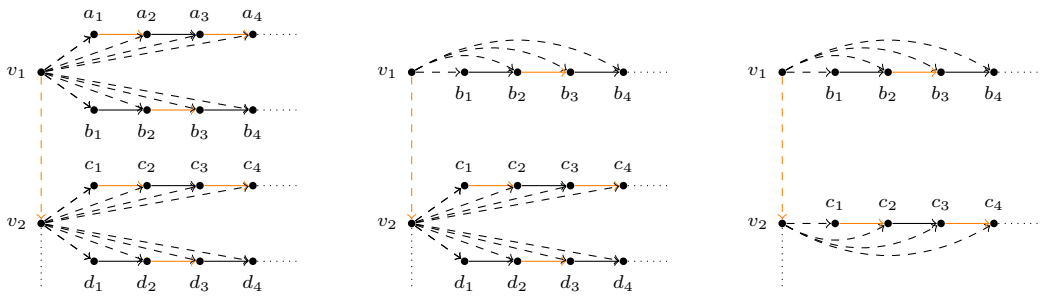


Figure 5 Instances $\mathcal{I}$ (left), $\mathcal{J}$ (middle), and $\mathcal{K}$ (right). Roles U, V, R, and S are represented with dashed and black, dashed and orange (possibly grey), black, and orange (possibly grey) arrows, respectively; dotted edges indicate the continuation of a sequence of elements up to some length.

finite range. Since $\mathcal{C}$ is a core (Theorem 32), every endomorphism (including $h \circ h'$) must be injective and hence, $\mathcal{C}$ must be finite. Since $\mathcal{C}$ is finite, BCQ-equivalent to $\mathcal{U}$, and a core, we conclude that $\mathcal{C}$ is equal to $\mathcal{U}$ up to isomorphism. ◀

We continue with some limitations of the stable chase: it may not yield a universal model, it may admit uncountably many non-isomorphic results, and it cannot be used to characterise non-full incidental TGDs. As already pointed out in Section 4, there are sets of TGDs that only admit universal models which are not cores (e.g., see the set of TGDs from Example 14). Hence, since the stable chase is guaranteed to yield a core (Theorem 32), it may not always produce a universal model. To illustrate the other limitations, consider the following example.

▶ **Example 35.** Consider a set Σ of TGDs containing the following dependencies.

$$\exists x, y.V(x, y) \qquad U(x, y) \wedge R(y, z) \to U(x, z)$$
$$V(x, y) \to \exists z.V(y, z) \qquad U(x, y) \wedge S(y, z) \to U(x, z)$$
$$V(x, y) \to \exists z, w.U(x, z) \wedge R(z, w) \qquad R(x, y) \to \exists z.S(y, z)$$
$$V(x, y) \to \exists z, w.U(x, z) \wedge S(z, w) \qquad S(x, y) \to \exists z.R(y, z)$$

Moreover, let $\mathcal{I}$, $\mathcal{J}$, and $\mathcal{K}$ be the instances depicted in Figure 5.

By iteratively applying the chase step (2.a) from Definition 22 during the computation of some stabilising chase sequence $\boldsymbol{\mathcal{Q}}$ of Σ, we can produce an instance such as $\mathcal{I}$ containing an arbitrarily long V chain, and two alternating R and S chains linked to every element v_i of such V-chain. Applying step (2.b) from Definition 22, we can, for each pair of chains in $\mathcal{I}$ linked to the same v_i, collapse the lower chain into the upper, or vice versa. In each case, the chase will produce initial segments of a single alternating infinite chain, which might begin with R or S. Applying such h-rewritings, we can produce instances such as $\mathcal{J}$ and $\mathcal{K}$.

The h-rewritings discussed above are somehow similar to the rewriting discussed in Example 24. However, in the current example, we have an infinite number of rewritings to consider–one for each element v_i in the infinite V chain. Taking into account all of these choices, we can generate uncountably many different stable chase sequences which can in turn be used to define uncountably many non-isomorphic stable chases.

Finally, note that there are (non-full) incidental TGDs for Σ, such as $V(x, y) \to \exists z.V(z, x)$, which are not entailed by any stable chase of Σ.

As highlighted by the previous example, instances resulting from the stable chase sequence may not be used to characterise non-full TGDs. Nevertheless, we can show that, for a set of

TGDs, there is an instance that satisfies all incidentals. While this result shows the existence of a suitable structure, it does not offer a constructive way of approximating it, since it relies on the (infinite) set of all incidentals to be given.

▶ **Theorem 36.** *Given a set Σ of TGDs, there is an instance $\mathcal{I}$ such that:*
1. *$\mathcal{I}$ is a core;*
2. *$\mathcal{I} \models \Sigma$;*
3. *$\mathsf{BCQ}(\mathcal{I}) = \mathsf{BCQ}(\Sigma)$; and*
4. *$\rho \in \mathsf{ICDT}(\Sigma)$ iff $\mathcal{I} \models \rho$, for any TGD ρ.*

Proof. To show this theorem we sketch how one can adapt the stable chase so that it can deal with infinite sets of TGDs. In this case infinite instances may occur in a stabilising chase of $\mathsf{ICDT}(\Sigma)$ and hence, the stable chase is not well-defined. To avoid this, we slightly modify Definition 22: In (2.a), instead of setting $\boldsymbol{\mathcal{Q}}_{k+1}$ as the extension of $\boldsymbol{\mathcal{Q}}_k$ with $\mathsf{ICDT}(\Sigma)(\boldsymbol{\mathcal{Q}}_{k,\ell(k)})$, we define this sequence as the extension of $\boldsymbol{\mathcal{Q}}_k$ with $\rho(\boldsymbol{\mathcal{Q}}_{k,\ell(k)})$ for some $\rho \in \mathsf{ICDT}(\Sigma)$. Moreover, we must also ensure fairness of the application of the rules in $\mathsf{ICDT}(\Sigma)$; i.e., each rule in $\mathsf{ICDT}(\Sigma)$ must be applied after the computation of a finite amount of sequences. With this modified version of the stable chase, one can show that it maintains its main properties. Then, by Theorem 29 there is some stable chase $\mathcal{I}$ of $\mathsf{ICDT}(\Sigma)$ which, by Theorem 32, $\mathcal{I}$ is a core; by Theorem 30, $\mathcal{I} \models \mathsf{ICDT}(\Sigma)$ and hence, $\mathcal{I}$ entails all subsets of $\mathsf{ICDT}(\Sigma)$, including Σ; and by Theorem 31, $\mathsf{BCQ}(\mathcal{I}) = \mathsf{BCQ}(\Sigma)$. Also, if $\rho \in \mathsf{ICDT}(\Sigma)$, then $\mathcal{I} \models \rho$ since $\mathcal{I} \models \mathsf{ICDT}(\Sigma)$. Conversely, if $\mathcal{I} \models \rho$ then $\rho(q) \in \mathsf{BCQ}(\Sigma)$ for every $q \in \mathsf{BCQ}(\Sigma)$. Therefore, by Theorem 4, $\rho \in \mathsf{ICDT}(\Sigma)$. ◀

7 Conclusion

To the best of our knowledge, this is the first study on constraint implication in the presence of arbitrary theories of tuple-generating dependencies. This idea is embodied in our new notion of incidental dependencies, which correspond to constraints that can be safely assumed to hold when checking BCQ entailment, despite not being a consequence of the given TGD set. Even for a single, fixed instance, finding incidental dependencies remains a challenging problem which is highly undecidable.

Our work reveals close connections between incidental dependencies and cores. If a finite universal model exists, its unique core perfectly characterises the incidentals. The correspondence breaks down if models become infinite, but we can still find cases where cores characterise at least all full incidental dependencies. However, one then has to be content with cores that are BCQ-equivalent to the universal models, but that are not universal themselves. To obtain such cores, we presented the stable chase as a generalisation of the core chase that can be used to build infinite models, and which is interesting in its own right.

On the theoretical level, several questions remain for future work: Is there a construction alike the stable chase which produces a BCQ-equivalent model which is indicative of *all* incidental TGDs (not just the full ones), without knowing all incidentals beforehand? What are the computational characteristics of Incidental for restricted classes of TGDs (such as guarded [4], sticky [10], etc.)? Obviously, all classes that warrant a finite universal model (such as diverse versions of acyclic TGDs [16, 24, 21, 13] and full TGDs) guarantee decidability, but the exact complexity of checking incidentality of individual TGDs would still be of interest. Further questions arise when considering equality-generating dependencies in addition to TGDs. Finally, it is of great importance to understand how known incidentals can be exploited toward more efficient practical query answering, as already suggested in some previous works [22, 25].

References

1 Serge Abiteboul and Richard Hull. Data functions, datalog and negation. In *Proc. SIGMOD Int. Conf. on Management of Data (SIGMOD'88)*, pages 143–153. ACM, 1988.

2 Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, and Eric Salvat. Extending decidable cases for rules with existential variables. In Craig Boutilier, editor, *Proc. 21st Int. Joint Conf. on Artificial Intelligence (IJCAI'09)*, pages 677–682. IJCAI, 2009.

3 Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, and Eric Salvat. On rules with existential variables: Walking the decidability line. *Artificial Intelligence*, 175(9–10):1620–1654, 2011.

4 Jean-François Baget, Marie-Laure Mugnier, Sebastian Rudolph, and Michaël Thomazo. Walking the complexity lines for generalized guarded existential rules. In Toby Walsh, editor, *Proc. 22nd Int. Joint Conf. on Artificial Intelligence (IJCAI'11)*, pages 712–717. AAAI Press/IJCAI, 2011.

5 Bruce L. Bauslaugh. Core-like properties of infinite graphs and structures. *Discrete Mathematics*, 138(1):101–111, 1995.

6 Bruce L. Bauslaugh. Cores and compactness of infinite directed graphs. *J. of Comb. Theory Ser. B*, 68(2), 1996.

7 Catriel Beeri and Moshe Y. Vardi. A proof procedure for data dependencies. *J. ACM*, 31(4):718–741, 1984.

8 Manuel Bodirsky. The core of a countably categorical structure. In Volker Diekert and Bruno Durand, editors, *Proc. 22nd Annual Symposium on Theoretical Aspects of Computer Science (STACS'05)*, volume 3404 of *Lecture Notes in Computer Science*, pages 110–120. Springer, 2005.

9 Pierre Bourhis, Markus Krötzsch, and Sebastian Rudolph. Reasonable highly expressive query languages. In Qiang Yang and Michael Wooldridge, editors, *Proc. 24th Int. Joint Conf. on Artificial Intelligence (IJCAI'15)*, pages 2826–2832. AAAI Press, 2015.

10 Andrea Calì, Georg Gottlob, and Andreas Pieris. Towards more expressive ontology languages: The query answering problem. *J. of Artif. Intell.*, 193:87–128, 2012.

11 Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Reasoning on regular path queries. *SIGMOD Record*, 32(4):83–92, 2003.

12 Stavros S. Cosmadakis, Haim Gaifman, Paris C. Kanellakis, and Moshe Y. Vardi. Decidable optimization problems for database logic programs (preliminary report). In Janos Simon, editor, *Proc. 20th Annual ACM Symposium on Theory of Computing (STOC'88)*, pages 477–490. ACM, 1988.

13 Bernardo Cuenca Grau, Ian Horrocks, Markus Krötzsch, Clemens Kupke, Despoina Magka, Boris Motik, and Zhe Wang. Acyclicity notions for existential rules and their application to query answering in ontologies. *J. of Artificial Intelligence Research*, 47:741–808, 2013.

14 Alin Deutsch, Alan Nash, and Jeffrey B. Remmel. The chase revisited. In Maurizio Lenzerini and Domenico Lembo, editors, *Proc. 27th Symposium on Principles of Database Systems (PODS'08)*, pages 149–158. ACM, 2008.

15 Alin Deutsch and Val Tannen. Reformulation of XML queries and constraints. In Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors, *Proc. 9th Int. Conf. on Database Theory (ICDT'03)*, volume 2572 of *LNCS*, pages 225–241. Springer, 2003.

16 Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

17 Ronald Fagin, Phokion G. Kolaitis, and Lucian Popa. Data exchange: Getting to the core. *ACM Trans. Database Syst.*, 30(1):174–210, 2005.

18 Daniela Florescu, Alon Levy, and Dan Suciu. Query containment for conjunctive queries with regular expressions. In Alberto O. Mendelzon and Jan Paredaens, editors, *Proc. 17th Symposium on Principles of Database Systems (PODS'98)*, pages 139–148. ACM, 1998.

19 Georg Gottlob and Christos H. Papadimitriou. On the complexity of single-rule datalog queries. *Inf. Comput.*, 183(1):104–122, 2003.

20 Pavol Hell and Jaroslav Nešetřil. The core of a graph. *Discrete Mathematics*, 109:117–126, 1992.

21 Markus Krötzsch and Sebastian Rudolph. Extending decidable existential rules by joining acyclicity and guardedness. In Toby Walsh, editor, *Proc. 22nd Int. Joint Conf. on Artificial Intelligence (IJCAI'11)*, pages 963–968. AAAI Press/IJCAI, 2011.

22 Markus Krötzsch and Veronika Thost. Ontologies for knowledge graphs: Breaking the rules. In Yolanda Gil, Elena Simperl, Paul Groth, Freddy Lecue, Markus Krötzsch, Alasdair Gray, Marta Sabou, Fabian Flöck, and Hideaki Takeda, editors, *Proc. 15th Int. Semantic Web Conf. (ISWC'16)*, volume 9981 of *LNCS*, pages 376–392. Springer, 2016.

23 David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. Testing implications of data dependencies. *ACM Transactions on Database Systems*, 4:455–469, 1979.

24 Bruno Marnette. Generalized schema-mappings: from termination to tractability. In Jan Paredaens and Jianwen Su, editors, *Proc. 28th Symposium on Principles of Database Systems (PODS'09)*, pages 13–22. ACM, 2009.

25 Mariano Rodriguez-Muro, Roman Kontchakov, and Michael Zakharyaschev. Ontology-based data access: Ontop of databases. In Harith Alani, Lalana Kagal, Achille Fokoue, Paul T. Groth, Chris Biemann, Josiane Xavier Parreira, Lora Aroyo, Natasha F. Noy, Chris Welty, and Krzysztof Janowicz, editors, *Proc. 12th Int. Semantic Web Conf. (ISWC'13)*, volume 8218 of *Lecture Notes in Computer Science*, pages 558–573. Springer, 2013.

26 Hartley Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. MIT Press, paperback edition edition, 1987.

27 Sebastian Rudolph and Markus Krötzsch. Flag & check: Data access with monadically defined queries. In Richard Hull and Wenfei Fan, editors, *Proc. 32nd Symposium on Principles of Database Systems (PODS'13)*, pages 151–162. ACM, 2013.

28 Yehoshua Sagiv. Optimizing datalog programs. In Moshe Y. Vardi, editor, *Proc. Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'87)*, pages 349–362. ACM, 1987.

29 Ke Wang and Li-Yan Yuan. Preservation of integrity constraints in definite DATALOG programs. *Inf. Process. Lett.*, 44(4):185–193, 1992.

8 Appendix: Proof of Theorem 8

▶ **Theorem 8.** INCIDENTAL *is* Π^0_2*-complete, and in particular neither in* RE *nor in* coRE.

Proof. For membership note that we can characterise incidentiality by quantifying over (finite) derivations (or proofs) in some theory. Indeed, a TGD ρ is incidental for Σ if: for all derivations that show $\Sigma \cup \{\rho\} \models q$ for some BCQ q, there is a derivation that shows $\Sigma \models q$. Using Gödel numbers for representing derivations, this condition can be expressed as a $\forall\exists$-sentence in first-order arithmetic.

We show hardness by many-one reduction from the *universal halting problem*, which is as follows: given a (deterministic) Turing machine $\mathcal{M}$, does $\mathcal{M}$ halt on all inputs? Universal halting is known to be complete for Π^0_2 (see [26, Theorem VIII], and apply Post's Theorem).

For the reduction, we construct for a given TM $\mathcal{M}$ a set $\Sigma_\mathcal{M}$ of TGDs and a full TGD ρ such that ρ is incidental for Σ iff $\mathcal{M}$ halts universally. The rules of $\Sigma_\mathcal{M}$ consist of three parts: Σ_1 ensures that each model contains representations of all possible inputs; Σ_2 simulates $\mathcal{M}$ on a particular input; Σ_3 marks elements of an accepting TM simulation with a specific unary relation halted. The rule ρ then asserts that initial elements in TM simulations are always marked by halted, which is incidental if all runs have indeed terminated.

In detail, let Q be the set of states, Γ be the tape alphabet with blank symbol $\square$, and $\delta : Q \times \Gamma \to Q \times \Gamma \times \{r, l\}$ the (total, deterministic) transition function of $\mathcal{M}$. Let $q_s, q_e \in Q$ be the starting and halting state, respectively. We assume without loss of generality that $\mathcal{M}$ never returns to q_s during a run. Our encoding uses the following unary relation symbols:

- symbol_σ: marks tape positions with symbol $\sigma \in \Gamma$
- head_q: marks tape position of head with TM in state $q \in Q$
- end: marks last (explicitly represented) tape position
- halted: used to mark halting configurations

as well as binary relations g, f (used to generate inputs), right, right^+ (right tape neighbour and its transitive closure), and next (tape cell in next configuration).

Now Σ_1 contains the followig rules

$$\rightarrow \exists y.\mathsf{symbol}_\square(y) \tag{4}$$

$$\mathsf{symbol}_\sigma(x) \rightarrow \exists y.g(x,y) \wedge \mathsf{symbol}_{\sigma'}(y) \tag{5}$$

$$\mathsf{symbol}_\sigma(x) \rightarrow \exists y.f(x,y) \wedge \mathsf{symbol}_\sigma(y) \wedge \mathsf{head}_{q_s}(y) \tag{6}$$

$$\mathsf{symbol}_\sigma(x) \wedge g(x,y) \wedge f(y,z) \rightarrow \exists v.f(x,v) \wedge \mathsf{right}(z,v) \tag{7}$$

$$\mathsf{symbol}_\square(x) \wedge f(x,y) \rightarrow \mathsf{end}(y) \tag{8}$$

each instantiated for all $\sigma, \sigma' \in \Gamma$. Models of Σ_1 projectively contain an infinite g-tree with root labelled $\mathsf{symbol}_\square$ (4) and other nodes labeled by symbols symbol_σ (5). Each node f-relates to the start of an initial sequence (6), which continues as a parallel copy of the finite path up until the root of the tree (7). The last cell of each initial tape is marked with end (8).

The set Σ_2 that simulates $\mathcal{M}$ is defined as follows. The following rules generate an infinite grid of TM tapes, with each tape one cell longer than the previous:

$$\mathsf{end}(x) \rightarrow \exists v, w.\mathsf{next}(x,v) \wedge \mathsf{right}(v,w) \wedge \mathsf{end}(w) \wedge \mathsf{symbol}_\square(w) \tag{9}$$

$$\mathsf{right}(x,y) \wedge \mathsf{next}(y,z) \rightarrow \exists v.\mathsf{next}(x,v) \wedge \mathsf{right}(v,z) \tag{10}$$

For every transition $\delta(q,\sigma) = \langle q', \sigma', r\rangle$, Σ_2 contains a rule:

$$\begin{aligned}&\mathsf{head}_q(x) \wedge \mathsf{symbol}_\sigma(x) \wedge \mathsf{right}(x,y) \wedge \mathsf{next}(x,x') \wedge \mathsf{next}(y,y') \\ &\qquad \rightarrow \mathsf{symbol}_{\sigma'}(x') \wedge \mathsf{head}_{q'}(y')\end{aligned} \tag{11}$$

and for every transition $\delta(q,\sigma) = \langle q', \sigma', l\rangle$ the rule

$$\begin{aligned}&\mathsf{head}_q(x) \wedge \mathsf{symbol}_\sigma(x) \wedge \mathsf{right}(y,x) \wedge \mathsf{next}(x,x') \wedge \mathsf{next}(y,y') \\ &\qquad \rightarrow \mathsf{symbol}_{\sigma'}(x') \wedge \mathsf{head}_{q'}(y').\end{aligned} \tag{12}$$

Finally, Σ_2 also contains the following rules for all $\sigma \in \Gamma$ and $q \in Q$:

$$\mathsf{right}(x,y) \rightarrow \mathsf{right}^+(x,y) \tag{13}$$

$$\mathsf{right}(x,y) \wedge \mathsf{right}^+(y,z) \rightarrow \mathsf{right}^+(x,z) \tag{14}$$

$$\mathsf{symbol}_\sigma(x) \wedge \mathsf{right}^+(x,y) \wedge \mathsf{head}_q(y) \wedge \mathsf{next}(x,x') \rightarrow \mathsf{symbol}_\sigma(x') \tag{15}$$

$$\mathsf{symbol}_\sigma(x) \wedge \mathsf{right}^+(y,x) \wedge \mathsf{head}_q(y) \wedge \mathsf{next}(x,x') \rightarrow \mathsf{symbol}_\sigma(x') \tag{16}$$

Rules (13) and (14) define right^+ to be (a superset of) the transitive closure of right, used by rules (15) and (16) to preserve tape contents at positions different from the head position. It is not hard to see that rules (9)–(16) create a simulatation of $\mathcal{M}$ for each starting configuration.

Finally, Σ_3 contains the rules

$$\mathsf{head}_{q_e}(x) \to \mathsf{halting}(x) \tag{17}$$

$$\mathsf{right}(x, y) \wedge \mathsf{halting}(y) \to \mathsf{halting}(x) \tag{18}$$

$$\mathsf{next}(x, y) \wedge \mathsf{halting}(y) \to \mathsf{halting}(x) \tag{19}$$

which propagate halting back to the first position of the first tape if the TM ever reaches q_e.

We claim that ρ is incidental for $\Sigma_{\mathcal{M}} = \Sigma_1 \cup \Sigma_2 \cup \Sigma_3$ iff $\mathcal{M}$ is universally halting. Indeed, if $\mathcal{M}$ is universally halting, then any universal model of $\Sigma_{\mathcal{M}}$ will have halting propagated back to the first cell of the first tape in each TM simulation, so that the rule is already satisfied in this model.

Conversely, if $\mathcal{M}$ is not universally halting, then there is an input $w = w_1 \cdots w_n$ on which it does not halt. Any universal model of $\Sigma_{\mathcal{M}}$ contains an initial tape for w, with the first position not marked by halting. The BCQ $\exists \boldsymbol{x}.\mathsf{halting}(x_1) \wedge \mathsf{head}_{q_s}(x_1) \wedge \mathsf{symbol}_{w_1}(x_1) \wedge \mathsf{right}(x_1, x_2) \wedge \ldots \wedge \mathsf{symbol}_{w_n}(x_n) \wedge \mathsf{right}(x_n, x_{n+1}) \wedge \mathsf{end}(x_n)$ is not entailed by $\Sigma_{\mathcal{M}}$ but by $\Sigma_{\mathcal{M}} \cup \{\rho\}$. ◀

Fast Sketch-based Recovery of Correlation Outliers

Graham Cormode
Department of Computer Science, University of Warwick, Coventry, UK
g.cormode@warwick.ac.uk

Jacques Dark
Department of Computer Science, University of Warwick, Coventry, UK
j.dark@warwick.ac.uk

Abstract

Many data sources can be interpreted as time-series, and a key problem is to identify which pairs out of a large collection of signals are highly correlated. We expect that there will be few, large, interesting correlations, while most signal pairs do not have any strong correlation. We abstract this as the problem of identifying the highly correlated pairs in a collection of n mostly pairwise uncorrelated random variables, where observations of the variables arrives as a stream. Dimensionality reduction can remove dependence on the number of observations, but further techniques are required to tame the quadratic (in n) cost of a search through all possible pairs.

We develop a new algorithm for rapidly finding large correlations based on sketch techniques with an added twist: we quickly generate sketches of random combinations of signals, and use these in concert with ideas from coding theory to decode the identity of correlated pairs. We prove correctness and compare performance and effectiveness with the best LSH (locality sensitive hashing) based approach.

2012 ACM Subject Classification Theory of computation → Sketching and sampling, Mathematics of computing → Probabilistic algorithms

Keywords and phrases correlation, sketching, streaming, dimensionality reduction

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.13

Funding The work of GC is supported by European Research Council grant ERC-2014-CoG 647557 and a Royal Society Wolfson Research Merit Award; JD is supported by a Microsoft Research PhD Scholarship (MRL 2014-038).

Acknowledgements We thank Milan Vojnovic for several discussions about this work.

1 Introduction

One of the most basic tasks in data analysis is to identify correlations between data sources, modeled as random variables. Discovered correlations are used to remove unnecessary features, to build predictive models, and to identify unexpected behaviors and dependencies. In this paper, we consider the most common measure of correlation: the Pearson product-moment correlation coefficient, which describes the linear relationship between a pair of random variables. This measure is simple to state and interpret: it is computed as the (sample) covariance of the two variables, divided by the product of the corresponding standard deviations. It ranges from -1 (strong negative correlation) through 0 (no correlation) to $+1$ (strong positive correlation). Hence, we are typically interested only in attribute pairs with correlation close to 1 in (absolute) magnitude.

For large numbers of variables, it can quickly become infeasible to compute the correlations of all of the quadratically many pairs. However, our observation is that most correlations are uninteresting: for many kinds of data, we expect that most pairs of variables would *not* display any (strong) correlation. For example, if we consider the activity profiles of users of a large web service, then we do not expect many pairs to be strongly correlated (there may be weak correlations due to similar time-of-day and day-of-week behavior)—any strong correlation between a pair would be unusual, indicating potentially nefarious activity worthy of further investigation. We model this by assuming that the number of correlated pairs is asymptotically smaller than the quadratically many possible pairs.

With this in mind, we can ask the following questions: given a stream of observation data, can we identify all correlation outliers (unusually large correlation coefficients, defined by being greater in magnitude than some parameter ϕ) with query time cost sub-quadratic in the number of variables and sub-linear in the number of observations?

This can be accomplished using a combination of a Fast Johnson-Lindenstrauss Transform (FJLT, to compress the rows of the input matrix) and Locality Sensitive Hashing (LSH, to efficiently find the outlier pairs). However, for small ϕ, the query time of this strategy looks like $n^{2-\Theta(\phi)}$, even as we shrink all the non-outlier correlations down to 0. Valiant [19] showed how to improve this to $n^{2-\Theta(\sqrt{\phi})}$, but we would like to remove the dependence on ϕ from the exponent of n.

This paper describes an algorithm which takes sketches of the rows and uses fast matrix multiplication to quickly transform them into an approximation of a sketch of the correlation matrix. We then remove the 1's along the diagonal, and use a heavy hitters recovery technique to pull out the outliers.

We then provide analysis for this algorithm, showing that for constant Frobenius norm[1] of the non-outlier non-diagonal correlations, this query process can be performed in time[2] $\tilde{O}(\phi^{-2}n^{5/3})$, assymptotically better than LSH for small enough ϕ. However, this comes at the cost of requiring much larger sketches of the input matrix rows.

2 Preliminaries

2.1 Models

We treat the observation data as defining an $n \times p$ matrix of reals, $\mathbf{M}$. Here, n denotes the number of attributes, while p indexes the different observations. Hence, each of the p columns represents an independent observation of some n-dimensional random variable. We label the columns (observations) as $\mathbf{x^{(i)}}$ for $i \in [p]$. For this data, we can apply standard definitions of covariance and correlation.

▶ **Definition 1.** Recall that:

- The *sample mean* is given by $\overline{\mathbf{x}} = \frac{1}{p}\sum_{i=1}^{p}\mathbf{x^{(i)}}$.
- The *sample covariance* is given by $\mathbf{V} = \frac{1}{p-1}(\mathbf{M} - \overline{\mathbf{x}}\mathbf{e}^T)(\mathbf{M} - \overline{\mathbf{x}}\mathbf{e}^T)^T$, where $\mathbf{e}$ is the p-dimensional vector with entries all ones.
- The *sample correlation* is given by $\mathbf{C} = \mathbf{\Sigma}^{-\frac{1}{2}}\mathbf{V}\mathbf{\Sigma}^{-\frac{1}{2}}$, where $\mathbf{\Sigma}$ is the diagonal matrix consisting of the diagonal entries of $\mathbf{V}$.

[1] For matrix $\mathbf{A}$, the Frobenius norm is $\|\mathbf{A}\|_F = (\sum_{i,j}\mathbf{A}_{i,j}^2)^{\frac{1}{2}}$.

[2] Using the convention that $\tilde{O}(\cdot)$ is the $O(\cdot)$ cost with log factors suppressed.

Essentially, the covariance is found by shifting the rows of $\mathbf{M}$ to have mean 0 and then taking inner products between them normalized by a factor of $\frac{1}{p-1}$. The definition of the correlations is similar, but further normalized by diving out the standard deviations.

It will be useful for our analysis to have notations for the rows of $\mathbf{M}$ with the shift normalization applied.

▶ **Definition 2.** For each row vector $\mathbf{y^{(i)}}$:

- Let the standardized row vector $\hat{\mathbf{y}}^{(\mathbf{i})}$ be given by $\hat{\mathbf{y}}^{(\mathbf{i})} = \frac{\hat{\mathbf{y}}^{(\mathbf{i})} - \bar{\mathbf{x}}_\mathbf{i}\mathbf{e}^T}{\|\hat{\mathbf{y}}^{(\mathbf{i})} - \bar{\mathbf{x}}_\mathbf{i}\mathbf{e}^T\|_2}$.

The observation matrix $\mathbf{M}$ is input as a stream of m updates $\langle u_1, u_2, \cdots u_m \rangle$ arriving one at a time. Starting from the zero matrix $\mathbf{M^{(0)}} = \mathbf{0}$, each update u_s describes a change to be made to $\mathbf{M^{(s-1)}}$ in order to determine $\mathbf{M^{(s)}}$. By the end of the stream, we have $\mathbf{M^{(m)}} = \mathbf{M}$. The format of the updates depends on the exact choice of stream model—we will consider three variants: row-wise permutation, column-wise permutation, and turnstile.

Row-Wise Permutation Stream (RPS). In this model, the updates are simply a list of the entries of $\mathbf{M}$, one row at a time. With each step from $\mathbf{M^{(s-1)}}$ to $\mathbf{M^{(s)}}$, one entry is changed from 0 to its final value. Entries in the same row arrive contiguously, so each row is filled out one after the other. Without loss of generality, we can assume that rows arrive in index order, so that $\mathbf{M}_{i,j} \leftarrow u_{(i-1)p+j}$ Since each entry is set exactly once, the stream has length $m = np$. The arrival of each new row corresponds to adding a new attribute to the data set.

Column-Wise Permutation Stream (CPS). This model works the same as the row-wise version, but with entries arriving as contiguous columns. Again, $m = np$ but now $\mathbf{M}_{i,j} \leftarrow u_{(j-1)n+i}$. The arrival of a new column corresponds to adding a new observation (e.g. from a new time step).

Turnstile Stream (TS). The turnstile model is the most general that we consider. Here updates are of the form $u_t = (\alpha, i, j)$ indicating that the $(i, j)^{\text{th}}$ entry should be incremented by $\alpha \in \mathbb{R}$. That is, $\mathbf{M}_{i,j}^{(s)} \leftarrow \mathbf{M}_{i,j}^{(s-1)} + \alpha$, while all other entries remain the same. Changes happen in any order, and entries can change any number of times as long as the correct state is reached by the end of the stream. Hence, the stream length m is arbitrary.

Both RPS and CPS are then special cases of this model. TS represents the situation where each of the observed values needs to be aggregated from a variety of sources. For example: suppose the entries in our observation matrix represent the number of requests for a specific resource (indexed by rows) at a specific site (indexed by columns) in a distributed system. Then, the number of requests at each node will need to be accumulated to produce the actual observation data.

2.2 Problem Statement

In what we term the *correlation outliers* problem, we are given a stream describing $\mathbf{M}$ (according to one of the three models), and three parameters: k, ϕ, and R. We make use of the following concepts:

▶ **Definition 3.** For the sample correlation matrix $\mathbf{C}$ (of $\mathbf{M}$):

- Let $\textsc{Large}_\phi \subset [n]^2$ refer to the set of index pairs of off-diagonal entries of $\mathbf{C}$ which have magnitude at least ϕ.
- Let $\mathbf{C}_{-k}$ refer to the matrix obtained by taking $\mathbf{C}$, removing all the diagonal entries, and removing the k largest magnitude off-diagonal entries (replacing them with 0's).

The problem is then to maintain a summary of the stream so that all index pairs contained in $\textsc{Large}_\phi$ can be retrieved with high probability ($o(\frac{1}{n})$ chance of failure), provided that $|\textsc{Large}_\phi| \leq k$ and $\|\mathbf{C}_{-k}\|_F \leq R$. Since the full input can be trivially maintained in $O(np)$ space, we seek solutions with space cost that is $o(np)$. Further, the summary should be quick to update (taking polylogarithmic time) and, at the end of the stream, the query routine should run in time $o(n^2)$.

Parameter Regimes of Interest. We argue that the assumption that k, the number of highly correlated pairs, is $o(n^2)$ is a reasonable one: otherwise, simply reporting all the correlated pairs would take quadratic time, and naive exhaustive solutions would suffice. Prior work has made various assumptions to limit the scale and quantity of the correlations. In particular, Valiant's 'light bulb problem' [20] considers the case when all vectors are chosen uniformly at random from the Boolean hypercube, except for one correlated pair. In this setting, the expected correlation of the uncorrelated pairs is 0, and the observed values are bounded by $O(p^{-1/2})$.

Our problem description similarly models the underlying correlation matrix as having k "large" pairs with correlation magnitude $\geq \phi$ and all other (off-diagonal) correlations have a Frobenius weight of at most R. Our results provide the most interesting bounds when we take R to be polynomially small as a function of the number of vectors n. We argue that this is consistent with analogous problems, such as in compressed sensing, sparse Fourier transform, and coding theory. In these settings, it is common to study the case when the target vector is sparse, i.e. outside of the k non-zero values, the data is exactly zero [17]; noisy with zero mean [5]; or asymptotically decaying polynomially [8] All of these cases fall within our model of constant R as long as p is some moderately high degree polynomial of n.

2.3 Our Contributions

We describe an algorithm which answers the correlation outliers problem in the turnstile streaming model. We analyze its space and time costs, and show that they meet the desiderata above. Our algorithm stores a separate sketch of each row of $\mathbf{M}$ (described in Section 2.5). Comparing these directly would still take time $\Theta(n^2)$ to perform an all-pairs comparison. Instead, we achieve an improved query time with the following three ideas:

- By randomly assigning variables into Π groups, and linearly combining the (sketched) information of all variables in the same group we can go from having to consider n^2 pairs of variables to Π^2 pairs of groups. This can be seen as a second level of sketching. This has previously been used in the offline setting for Valiant's Boolean correlation outliers algorithm [19].
- Error correcting codes are composed with the grouping step (including/excluding variables from groups based on code bits) in a way that allows us to recover the identities of large entries in a particular group pair using the decoder. This has previously been used for heavy hitters problems on sketches (see: [9, 16, 14]).
- Fast matrix multiplication algorithms allow us to quickly generate batches of sketch estimates of inner products. This speeds up the evaluation of the inner products between pairs of groups. Then checking whether the results of these computations exceeds a given threshold produces the strings of bits for the decoder.

Several other offline algorithms can be similarly implemented under this sketch-and-search strategy, notably Locality Sensitive Hashing [10] and the approximate Closest Pair algorithm of Valiant [19]. We compare these with out method below.

We note that the LSH and Closest Pair algorithm consider slightly different input assumptions from us which makes the comparison a little tricky. While in this paper we consider a bound R on the total Frobenius weight of the non-outlier pairs, these other algorithms consider an input with non-outlier pairs having a correlation magnitude smaller than a flat threshold ϕ_1.

Since we are motivated by the case when R is constant, we include in the table the costs for these other two algorithms in the case when ϕ_1 tends to 0. For details of this as well as minor modifications for the streaming setting see Section 2.4.

Our main result is stated in full in Theorem 20. Using $\theta = 2/3$ we get:

Technique	Models	Sketch Size	Query Space	Query Time
Full Search	All	$\tilde{O}(\phi^{-2}n)$	$\tilde{O}(\phi^{-2}n)$	$\tilde{O}(\phi^{-2}n^2)$
LSH	All	$\tilde{O}(n)$	$\tilde{O}(n)$	$\tilde{O}(kn^{2-\Theta(\phi)})$
Valiant	All	$\tilde{O}(n)$	$\tilde{O}(n^{2-\Theta(\sqrt{\phi})})$	$\tilde{O}(kn^{2-\Theta(\sqrt{\phi})})$
Our Approach	All	$\tilde{O}(n^{5/3}(k^2 + \frac{R^2}{\phi^2}))$	$\tilde{O}(n^{5/3}(k^2 + \frac{R^2}{\phi^2}))$	$\tilde{O}(\phi^{-2}n^{5/3}(k^2 + \frac{R^2}{\phi^2}))$

This space usage is $o(np)$ and subquadratic in n for $p \in \Omega(n^{2/3+\epsilon})$ and constant k, R, ϕ.

2.4 Related Work

Locality Sensitive Hashing. Asking for high correlation is equivalent to looking for small Euclidean distance between the standardized (normalized and centered) row vectors. A correlation of ϕ corresponds to a distance on the sphere of $\sqrt{2-2\phi}$. Hence, this problem can be solved using Euclidean Locality Sensitive Hashing (LSH). Negative correlation outlier pairs can be found by simply considering every row and its negation.

The LSH framework is parameterised by $c > 1$: the ratio $\frac{d_1}{d_2}$ between d_1, the smallest distance between "dissimilar" items and d_2, the largest distance between "similar" items. To compare the efficiency of different families of hash functions, we talk about their sensitivity ρ as a function of c. This is given by $\rho = \frac{\log 1/p_1}{\log 1/p_2}$ where p_1 and p_2 are the collision probabilities of the similar and dissimilar pairs respectively. The best known Euclidean LSH algorithms have $\rho = \frac{1}{c^2} + o(1)$ (data independent, [10]) and $\rho = \frac{1}{2c^2-1} + o(1)$ (data dependent, [4]).

For n input vectors of d-dimensions, the framework can be used to find all similar pairs with high probability in $\tilde{O}(nd)$ space and $\tilde{O}(n^{1+\rho}d)$ time. Notice that we can save space compared with the normal operation of the framework, since we can check for similar pairs and scrap the results of each hash function before moving on to the next. Assuming outlier correlations are greater than ϕ_0, and non-outlier correlations are smaller than ϕ_1, we can use a Fast Johnson-Lindenstrauss Transformation to compress input rows to length $O(\epsilon^{-2}\log n)$, distorting the pairwise distances by at most $(1 \pm \epsilon)$. Then we can do LSH with $c^2 = \left(\frac{1-\epsilon}{1+\epsilon}\right)^2 \left(\frac{1-\phi_0}{1-\phi_1}\right)$. This gives us a space cost of $\tilde{O}(n)$ and a time cost of $\tilde{O}(n^{2-\Theta(\phi_0)})$, even if ϵ goes to 1 and ϕ_1 goes to 0.

Compressed Matrix Multiplication. Pagh [16] considered the problem of efficiently computing sparse or approximate matrix products. The key idea is that by choosing a particular structure for the sketching functions, it is possible to quickly compute a sketch of the outer product $\mathbf{x}\mathbf{y}^T$, from sketches of $\mathbf{x}$ and $\mathbf{y}$, through the use of FFTs (in time $O(b \log b)$ for length b sketches). Since a matrix product $\mathbf{A}\mathbf{B}^T$ can be decomposed into a sum of such outer products between corresponding columns, this allows for efficient computation of matrix products from sketches of columns.

As the algorithm only requires access to matched columns of $\mathbf{A}$ and $\mathbf{B}$ one at a time, in the special case of $\mathbf{A} = \mathbf{B}$ this approach can be used in the CPS model to build a sketch of $\mathbf{A}\mathbf{A}^T$. In particular, we can build a sketch of the covariance matrix $\mathbf{V}$ in this streaming model, from input observation matrix $\mathbf{M}$, with update time cost $O(b \log b)$ ($O(1)$ amortized, since n dominates $b \log b$) and space usage $O(b)$. To recover dominant entries from these sketches, Pagh describes an approach (building on [9, 16, 14]) that uses $O(\log^2 n)$ sketches of sub-matrices of $\mathbf{AB}$, along with error correcting codes, to discover the identity of a small number of entries which dominate the Frobenius norm of the product, with high probability. This process runs in $O(b \log^2 n)$ time and space. Putting these pieces together provides a solution to a *covariance outliers* version of our problem in the CPS model.

Unfortunately, this approach cannot be adapted directly to the *correlation outliers* problem. Large correlations between low variance signals would be drowned out by the contribution from high variance signals that are much more weakly correlated. To apply this technique, we would need to record the whole of $\mathbf{M}$ (perhaps feasible for small np), or perform two passes over the stream—using the first to determine the variances, and then using the covariance solution on the rescaled inputs with the second pass. Instead, we will adapt the recovery process to work on different kinds of sketches.

Boolean Vectors. Valiant [19] showed how to quickly find a single correlated pair among Boolean vectors in time $O(n^{\frac{5-\omega}{4-\omega}+\epsilon} + nd) \subset O(n^{1.62} + nd)$, where $\omega < 2.4$ is the exponent of matrix multiplication. They then used this along with embeddings into the Hamming space to develop a Euclidean space approximate closest pair algorithm capable of finding a $(1+\epsilon)$ approximation to the closest pair in time $O(n^{2-\Theta(\sqrt{\epsilon})})$.

This algorithm can be adapted to solve our streaming problem in a similar manner to LSH. We keep random hyperplane projections of each of the input vectors as sketches, and then using their signs for the algorithm at query time. This takes only $\tilde{O}(n)$ space and k repetitions can be used to find k outlier pairs. Again, using the model where outliers have correlation magnitude above ϕ_0 and non-outliers below ϕ_1, we can choose $\epsilon = \sqrt{\frac{1-\phi_1}{1-\phi_0}} - 1$. This looks like $\Theta(\phi_0)$ even as ϕ_1 goes to 0.

Karppa et al. [13] improved on this work, giving a faster algorithm for Boolean vector outlier pairs, tending towards $\tilde{O}(n^{\frac{2\omega}{3}}) \subset \tilde{O}(n^{1.6})$ as the non-outlier correlations tend to 0.

Our approach uses several of the same ideas, such as Cartesian grouping and signed aggregation of the rows, but as we are interested in a slightly different problem (with vanishing rather than fixed-small-threshold non-outliers) we do not need to rely on the nice concentration properties for Boolean vectors, and can apply these ideas directly on to the Euclidean vectors - producing a fast and simple algorithm.

2.5 Sketches of Vectors

Our results make use of *sketches* of vectors. These can be thought of as random projections from the original high-dimensional space down to a lower dimensional space, such that geometric properties of the vectors are (approximately) preserved. In particular, given vectors $\mathbf{x}$ and $\mathbf{y}$, sketches exist that can estimate:

$$\text{Squared Euclidean length } \|\mathbf{x}\|_2^2 \text{ up to error } \epsilon\|\mathbf{x}\|_2^2. \tag{1}$$

$$\text{Inner product } \langle \mathbf{x}, \mathbf{y} \rangle \text{ up to error } \epsilon\|\mathbf{x}\|_2\|\mathbf{y}\|_2. \tag{2}$$

Many results for such sketches are known, from the earliest (non-constructive) results based on the Johnson-Lindenstrauss lemma [11], the tug-of-war sketches due to Alon, Matthias, Szegedy and Gibbons [3, 2], and several more [1, 15, 12]. For concreteness, we will adopt the so-called (fast) AMS sketches (explained in [7]). These create a sketch of size $O(\epsilon^{-2} \log 1/\delta)$ so that any query obtains the above claimed ϵ guarantee with probability at least $1 - \delta$, where the probability is over the random choices used to determine the random projection.

The AMS sketching procedure maps (linearly and randomly) the space of p-dimensional vectors to the space of $d \times b$ matrices. Each row of the output sketch is obtained by pre-multiplying the input vector by a diagonal matrix whose entries are Rademacher (uniformly random ± 1), and then pre-multiplying by a $b \times p$ sparse matrix where each column has a single 1, with 0 everywhere else[3]. This process generates one row of the sketch, and is repeated independently d times to generate all rows. The stated (ϵ, δ) guarantee can be achieved for $d \in \Theta(\log 1/\delta)$ and $b \in \Theta(\epsilon^{-2})$. As they are a sparse linear transformation of their input, any addition to an entry in the sketched vector can be applied to the sketch in time $O(d) = O(\log 1/\delta)$.

▶ **Definition 4.** Let:

- $\text{AMS}_{\epsilon,\delta}$ refer to a distribution of random linear maps corresponding to fast AMS sketches with the stated (ϵ, δ) norm and inner product approximation guarantees ((1) and (2)).
- (ϵ, δ)-*sketch transformation* $\mathbf{S}$ be a linear map drawn from $\text{AMS}_{\epsilon,\delta}$.
- The symbol $\odot$ represent the binary operation of performing the inner product query between two sketches. So $\mathbf{S}(\mathbf{x}) \odot \mathbf{S}(\mathbf{y}) \approx \langle \mathbf{x}, \mathbf{y} \rangle$, and $\mathbf{S}(\mathbf{x}) \odot \mathbf{S}(\mathbf{x}) \approx \|\mathbf{x}\|_2^2$.

One application of sketches is to estimate the value of a particular index in a vector. This can be achieved as a special case of an inner product query: we use the sketch to estimate $\langle \mathbf{x}, \mathbf{e}_i \rangle$, where $\mathbf{e}_i$ is the vector that is 1 at location i and 0 elsewhere. The guarantee ensures that we obtain an estimate with error at most $\epsilon \|\mathbf{x}\|_2$. This use of sketches is referred to as a Count sketch [6].

3 Algorithm and Analysis

3.1 Algorithm Overview

Our algorithm works in the most general stream model we considered, the turnstile model. At a high level, our algorithm consists of:

- An *initialization* procedure to set up the sketch data structure.
- An *update* procedure to process updates from the stream.
- A *query* procedure to recover the suspected elements of $\text{LARGE}_{\phi,k}$.

Our sketch structure is built on top of a collection of AMS sketches with standard initialization and update procedures, plus some additional variables to keep running totals. We will briefly review these procedures in Section 3.2, as well as discussing some basic properties and routines required for the query algorithm.

The query process itself is based on two main ideas. First, we can take linear combinations of AMS sketches and then perform inner product queries between them in order to estimate

[3] The construction does not require the entries to be chosen fully independently at random, so it is common to describe the sketch transformations in terms of hash functions drawn from limited independence families. This allows the transform to be stored in polylogarithmic space.

Algorithm 1: UPDATE.

Input: TS model update $u_s = (\alpha, i, j)$

1 $\mathbf{r^{(i)}} \leftarrow \mathbf{r^{(i)}} + \alpha \cdot \mathbf{S}(\mathbf{e_j})$
2 $t^{(i)} \leftarrow t^{(i)} + \alpha$

Algorithm 2: STANDARDIZE.

1 **for** $i \in [n]$ **do**
2 $\quad \mathbf{r^{(i)}} \leftarrow \mathbf{r^{(i)}} - (t^{(i)}/p) \cdot \mathbf{S}(\mathbf{e})$
3 $\quad \mathbf{r^{(i)}} \leftarrow (\mathbf{r^{(i)}} \odot \mathbf{r^{(i)}})^{-1/2} \cdot \mathbf{r^{(i)}}$

certain kinds of linear combinations of entries of $\mathbf{C}$. Further, we can perform batches of such queries quickly using fast matrix multiplication. And secondly, we can utilize error correcting codes to identify the large magnitude entries in these linear combinations of entries even with the error introduce by the AMS sketches.

Rather than fast matrix multiplication between combinations of rows, we could have hoped to employ Pagh's compressed matrix multiplication for our second layer of sketching. However, to produce a w-bucket sketch would take time $\frac{w \log w}{\epsilon^2}$ for each row of the AMS sketches. Then to control the variance of the output buckets, we would need $w = n^2\epsilon^2$, giving a time cost of $\Omega(n^2)$ to build the secondary sketch.

The outline and the discussion of the query algorithm is therefore broken up into four parts. In Section 3.3 we describe a "Cartesian sketch" which compresses a matrix by applying a pair of independent transformations (each akin to the Count sketch), one row-wise and one column-wise. Then, in Section 3.4 we show that we can use the error correcting code technique to recover large entries from Cartesian sketches. Further, we show that the recovery technique is robust to additional sources of noise per entry of the sketch. Next, in Section 3.5 we show how the AMS sketches in our structure can be used to build good enough approximations of the Cartesian sketches to satisfy the noise limits. Finally, in Section 3.6 we analyze the overall space and time costs and discuss how to amplify the probability of success. For brevity, full proofs are deferred to the Appendix, and we present informal proofs in the main body to convey the high level ideas.

3.2 Row Sketching

For our data structure we will keep an AMS sketch of each row of the observation matrix along with a running total. The choice of sketch parameters (ϵ, δ) will be made in the final analysis in Section 3.6.

To initialize the structures, we randomly pick an (ϵ, δ)-sketch transformation S, and initialize n sketches $\mathbf{r^{(i)}}$ to all zeros. We also create n counters $t^{(i)}$, initialized to zero. Algorithm 1 shows how to apply a received update in the TS model. We use $\mathbf{e_j}$ to indicate the length p-vector consisting of a 1 in entry j and 0 everywhere else. The update simply updates the ith sketch with index j, and updates the corresponding sum of weights, $t^{(i)}$. Let $\mathbf{y^{(i)}}$ refer to the i^{th} row of $\mathbf{M}$ for $i \in [n]$. By following these procedures we will have $\mathbf{r^{(i)}} = \mathbf{S}(\mathbf{y^{(i)}})$ and $t^{(i)} = \sum_{j \in [p]} \mathbf{y^{(i)}}_j$ at the conclusion of the stream.

An important operation we will need to be able to perform on these row sketches is to *standardize* them. Recalling $\overline{\mathbf{x}}$ and $\mathbf{e}$ from Definition 1, we define:

▶ **Definition 5.** For a given row vector $\mathbf{y^{(i)}}$, the *standardized vector* $\hat{\mathbf{y}}^{(\mathbf{i})}$ is given by:

$$\hat{\mathbf{y}}^{(\mathbf{i})} = (\mathbf{y^{(i)}} - \overline{\mathbf{x}}_i \mathbf{e}^T)/\|\mathbf{y^{(i)}} - \overline{\mathbf{x}}_i \mathbf{e}^T\|_2.$$

If we have a sketch $\mathbf{S}(\mathbf{y^{(i)}})$ we will refer to $\mathbf{S}(\hat{\mathbf{y}}^{(\mathbf{i})})$ as the *standardized sketch*.

In the RPS and CPS models we could keep track of the running sums of α^2 for each row, allowing us to compute the exact rescaling factor required to standardize the sketches. However, in the more general TS setting, the best we can do is an approximation. Algorithm 2 describes the procedure for computing the approximately standardized sketches.

Initially, it may appear that to perform this standardization at query time, we need to spend $\Omega(pd)$ time building the sketch $\mathbf{S}(\mathbf{e})$. However, we can amortize this cost during the update phase. As long as at least p entries of the final $\mathbf{M}$ are non-zero, then we can build up $\mathbf{S}(\mathbf{e})$ one entry per update by using a single counter to track which entries have been added. In the atypical case that $\mathbf{M}$ is extremely sparse, we will need to add $O(pd)$ to the query time to complete the construction of this sketch.

▶ **Lemma 6.** *After performing the* STANDARDIZE *routine, the inner product query between sketches* $\mathbf{r^{(i)}}$ *and* $\mathbf{r^{(j)}}$ *produces an estimate of* $\mathbf{C}_{i,j}$ *having* 4ϵ *additive error with probability at least* $1 - 3\delta$*, for* $\epsilon < 1/2$*.*

Informal Proof. Each sketch approximates the sketch of a standardized row ($\mathbf{r^{(i)}} \approx \mathbf{S}(\hat{\mathbf{y}}^{(\mathbf{i})})$) and the inner product query between sketches of standardized rows approximates the correlation ($\mathbf{S}(\hat{\mathbf{y}}^{(\mathbf{i})}) \odot \mathbf{S}(\hat{\mathbf{y}}^{(\mathbf{j})}) \approx \mathbf{C}_{i,j}$). To get a small additive error on our estimates, we then just need both sketches to be approximated well and the inner product query between them to give a good results. Each of the three events occurs with probability $(1 - \delta)$.

The correlation between two rows can be expressed as the inner product of the corresponding standardized vectors. The sketches output by STANDARDIZE approximate the true standardized sketches. To get a small additive error on our estimate, we then just need both sketches to be approximated well and the inner product query between them to give a good result. Each of the three occurs with probability $(1 - \delta)$. ◀

3.3 Cartesian Sketches

▶ **Definition 7.** For an $n \times n$ matrix $\mathbf{A}$, we call $\textsc{Cart}(\mathbf{A})$ a $\Pi \times \Pi$ Cartesian sketch of $\mathbf{A}$ if for each $(h, g) \in [\Pi]^2$ we have

$$\textsc{Cart}(\mathbf{A})_{h,g} = \sum_{P_1(x)=h} \sum_{P_2(y)=g} \left(s_1(x)s_2(y)\mathbf{A}_{x,y}\right),$$

where s_1 and s_2 are independently selected from a pairwise independent family of random sign functions $[n] \to \{-1, +1\}$, and where P_1 and P_2 are functions $[n] \to [\Pi]$ selected independently and uniformly at random from the set of functions:

$$\{f : [n] \to [\Pi] \text{ s.t. } |f^{-1}(i)| = n/\Pi \text{ for each } i \in [\Pi]\}.$$

From this definition, we can see that a Cartesian sketch transformation is very similar to a pair of independent Count sketch transformations (one performed row-wise, one column-wise). The difference is the use of fully random partitioning functions which produce exactly equal buckets. If we were performing exactly a pair of Count sketches, we would also have P_1 and P_2 expressed as limited independence hash functions. However, the $O(n \log n)$ space needed to store fully random permutations will not impact our asymptotic space usage and makes

the subsequent analysis simpler. The entries of the sketches will be referred to as buckets, and the $(i,j)^{\text{th}}$ entry of the original matrix $\mathbf{A}$ is said to be mapped to the $(h,g)^{\text{th}}$ bucket (for a given choice of sketch functions) if $P_1(i) = h$ and $P_2(j) = g$.

▶ **Definition 8.** Let $\mathcal{B}_{h,g} = \{(i,j) \in [n]^2 \text{ s.t. } P_1(i) = h \text{ and } P_2(j) = g\}$, i.e. the set of index pairs mapped to bucket (h,g).

3.4 Recovery Process

Now we will describe how to apply the recovery process to a series of Cartesian sketches of a given matrix. We will describe an algorithm which gives a constant probability of finding any given element of $\textsc{Large}_{\phi,k}$ (Definition 3) and argue that it works.

For this procedure, we apply an error correcting code to encode row and column indices (which take values in $[n]$) into a longer binary codeword. We will assume access to some family of functions (over choices of n) with the desired properties to perform the encoding and decoding. For a fixed n, let:

$$\mathcal{E} : [n] \to \{0,1\}^{L \log n} \text{ and } \mathcal{D} : \{0,1\}^{L \log n} \to [n]$$

where $L > 1$ indicates how much bigger the codeword is compared to the input size. Here, we write $\mathcal{E}$ and $\mathcal{D}$ for the encoder and decoder functions (respectively) of a scheme which can recover from up to $\lambda L \log n$ bit flip errors — i.e. an error rate of λ. That is, for any length $L \log n$ binary word $\mathbf{w}$ with at most $\lambda L \log n$ bits set to 1, we have[4] $\mathcal{D}(\mathcal{E}(i) \oplus \mathbf{w}) = i$ for every $i \in [n]$.

Error correcting codes are known to exist for $L \in O(1)$ and $\lambda \in \Omega(1)$, which can be implemented to perform encoding and decoding in $O(\log n)$ time and $O(\text{polylog}\, n)$ space (for example [18]).

▶ **Definition 9.** For each $l \in [L \log n]$ (each bit in the code words), we define a *masking matrix* $\mathbf{E}^{(\mathbf{l})}$. This is a diagonal binary matrix where entry $\mathbf{E}^{(\mathbf{l})}{}_{i,i}$ is the l^{th} bit of the code word $\mathcal{E}(i)$. That is, $\mathbf{E}^{(\mathbf{l})}{}_{i,i} = \mathcal{E}(i)_l$.

These masking matrices can be pre- or post-multiplied with $\mathbf{C}$ to mask rows or columns (respectively) based on bits of their index encodings.

The recovery process is described in Algorithm 3. It takes as input sketches $\mathbf{L}^{(\mathbf{l})} = \textsc{Cart}(\mathbf{C}\mathbf{E}^{(\mathbf{l})})$ and $\mathbf{R}^{(\mathbf{l})} = \textsc{Cart}(\mathbf{E}^{(\mathbf{l})}\mathbf{C})$ for each $l \in [L \log n]$, for a randomly selected Cartesian sketch transformation $\textsc{Cart}$.

To understand why this process should work, consider the special case where $\mathbf{C}$ is 0 on all the non-large, off-diagonal entries. That is, the only non-zero entries are the diagonals (which must be 1) and the entries corresponding to elements of $\textsc{Large}_{\phi,k}$. In this situation, the only entries contributing to the Cartesian sketches are entries of $\textsc{Large}_{\phi,k}$ corresponding to unmasked rows and columns. Now, consider what happens in a bucket with a single large entry mapped to it. Whenever the row or column of the large entry is masked, the corresponding bucket value will be 0; and when the row and column are not masked, the bucket value will have magnitude at least ϕ — in particular, greater than $\phi/2$. This means that, in the algorithm, on the outer loop corresponding to this bucket, $\mathcal{I}$ and $\mathcal{J}$ will be exactly the code words corresponding to the row and column indices of the large entry. Hence, the index pair of the large entry is added to Ω. So, isolated large entries will be correctly

[4] Here $\oplus$ represents the "exclusive-or" bitwise operation between binary words.

Algorithm 3: RecoveryStep.

Input: Cartesian sketches $\mathbf{L}^{(1)}$ and $\mathbf{R}^{(1)}$ for $l \in [L \log n]$, and Cartesian sketch transformation Cart

Output: Index multiset Ω of suspected large entries

```
Create new empty multiset Ω
for (h, g) ∈ [Π]^2 do
    Create new empty strings 𝓘 and 𝓙
    for l ∈ [L log n] do
        if |L^(1) − Cart(E^(1))| ≥ φ/2 then Append 1 to 𝓘 else Append 0 to 𝓘
        if |R^(1) − Cart(E^(1))| ≥ φ/2 then Append 1 to 𝓙 else Append 0 to 𝓙
    Append (𝒟(𝓘), 𝒟(𝓙)) to Ω
return Ω
```

recovered in this special case, and as long as k is sufficiently smaller than Π we have a good chance of any given large entry being isolated. To formalize this argument, and extend it to the more general case, we define a few different events.

▶ **Definition 10.** For fixed $\mathbf{C}$ and a fixed coding scheme define the following random events, over the random choice of sketch functions:

- Let $\text{CorrectDecode}_{h,g}$ be the event that the recovery process returns the "correct" index for bucket (h, g). If there is exactly one of $\text{Large}_{\phi,k}$ in the bucket, then the correct result is the index pair of that entry. Otherwise, any returned value is considered correct.
- Let $\text{SmallError}_{h,g,l}$ be the event that the non-large entries of $\mathbf{CE}^{(1)} - \mathbf{E}^{(1)}$ and $\mathbf{E}^{(1)}\mathbf{C} - \mathbf{E}^{(1)}$ each contribute less than $\phi/4$ to bucket (h, g) of their corresponding sketches. That is, $\text{Cart}(\mathbf{E}^{(1)}\mathbf{C}_{-\mathbf{k}})_{h,g}$ has magnitude smaller than $\phi/4$.

We begin with a lemma explaining the circumstances we are looking for to successfully find a large entry.

▶ **Lemma 11.** *If we have that:* $\mathbb{P}[\text{CorrectDecode}_{h,g}] \geq 1 - x$ *for all* $(h, g) \in [\Pi]^2$*, then for any given* $(i, j) \in \text{Large}_{\phi,k}$*, we have that* (i, j) *is in the list of index pairs produced by* RecoveryStep *with probability at least* $1 - x - 2k/\Pi$.

From this lemma, we can see that if we can get a lower bound on the probability of $\text{CorrectDecode}_{h,g}$ for every $(h, g) \in [\Pi]^2$, then we can get an overall guarantee for the recovery process.

▶ **Lemma 12.** *If we have that:* $\mathbb{P}[\text{SmallError}_{h,g,l}] \geq 1 - y$ *for all* $l \in [L \log n]$*, then* $\mathbb{P}[\text{CorrectDecode}_{h,g}] \geq 1 - y/\lambda$.

The last piece we need is a lower bound on the probability of $\text{SmallError}_{h,g,l}$.

▶ **Lemma 13.** *Recalling Definition 3, we have:*

$$\mathbb{P}[\text{SmallError}_{h,g,l}] \geq (1 - 32\|\mathbf{C}_{-\mathbf{k}}\|_F^2/(\Pi^2\phi^2)).$$

Now we have all the pieces we need to show that the recovery process works.

▶ **Lemma 14.** *If we have that* $\Pi \geq \max\{18k, 18\|\mathbf{C}_{-\mathbf{k}}\|_F/(\phi\lambda^{1/2})\}$*, then the output of* RecoveryStep *will include any fixed index pair in* $\text{Large}_{\phi,k}$ *with probability at least* $\frac{2}{3}$.

Algorithm 4: APPROXIMATE.

Input: Approximately standardized row sketches $\mathbf{r^{(1)}}, \cdots, \mathbf{r^{(n)}}$ and functions P_1, P_2, s_1, s_2 corresponding to a Cartesian sketch transformation CART

Output: $\mathbf{L^{(l)}}$ and $\mathbf{R^{(l)}}$, estimates of CART$(\mathbf{E^{(l)}C})$ and CART$(\mathbf{CE^{(l)}})$ respectively, for $l \in [L \log n]$

for $l \in [L \log n]$ **do**
  Initialize empty matrices $\mathbf{L^{(l)}}$ and $\mathbf{R^{(l)}}$
  for $h \in [\Pi]$ **do**
    Initialize LEFT$[h]$, RIGHT$[h]$, LEFTMASKED$[h]$, RIGHTMASKED$[h]$ as zero sketches $\mathbf{S(0) = 0}$
  for $i \in [n]$ **do**
    LEFT$[P_1(i)] \leftarrow$ LEFT$[P_1(i)] + s_1(i) \cdot \mathbf{r^{(i)}}$
    LEFTMASKED$[P_1(i)] \leftarrow$ LEFTMASKED$[P_1(i)] + \mathbf{E^{(l)}}_{i,i} \cdot s_1(i) \cdot \mathbf{r^{(i)}}$
    RIGHT$[P_2(i)] \leftarrow$ RIGHT$[P_2(i)] + s_2(i) \cdot \mathbf{r^{(i)}}$
    RIGHTMASKED$[P_2(i)] \leftarrow$ RIGHTMASKED$[P_2(i)] + \mathbf{E^{(l)}}_{i,i} \cdot s_2(i) \cdot \mathbf{r^{(i)}}$
  for $(h, g) \in [\Pi]^2$ **do**
    $\mathbf{L^{(l)}}_{h,g} \leftarrow$ LEFTMASKED$[h] \odot$ RIGHT$[g]$
    $\mathbf{R^{(l)}}_{h,g} \leftarrow$ LEFT$[h] \odot$ RIGHTMASKED$[g]$
return $\mathbf{L^{(1)}}, \cdots, \mathbf{L^{(L \log n)}}$ *and* $\mathbf{R^{(1)}}, \cdots, \mathbf{R^{(L \log n)}}$

Observe that we chose the definition of the event SMALLERROR$_{h,g,l}$ to leave room for an additional source of noise of similar size $\phi/4$. We will need this robustness later.

▶ **Corollary 15.** *Lemma 14 holds even when there is additional noise applied to each bucket entry, provided it has magnitude smaller than $\phi/4$ with probability at least $(1 - \lambda/18)$ on any fixed bucket.*

In the next subsection, we will show that an approximate Cartesian sketch can be constructed from row sketches within these tolerances.

3.5 Approximation from Row Sketches

We need a way of quickly approximating CART$(\mathbf{E^{(l)}C})$ and CART$(\mathbf{CE^{(l)}})$ for each $l \in [L \log n]$ for a randomly chosen Cartesian sketch transformation CART, from the row sketches described in Section 3.2. This is done by the procedure described in Algorithm 4.

For each $l \in [L \log n]$, the returned $\mathbf{L^{(l)}}$ is our approximate CART$(\mathbf{E^{(l)}C})$ and $\mathbf{R^{(l)}}$ is our approximate CART$(\mathbf{CE^{(l)}})$.

The algorithm works by observing that $\mathbf{C}$ can be approximated from the row sketches by performing all the possible inner product queries between pairs of sketches and placing the results in the corresponding positions of the matrix. We could then apply CART to the result. However, we make the further observation that since CART can be broken up into pieces that look like pre- and post-multiplication by matrices, we can rearrange the order of operation. We can perform the CART sketch first, directly on the row sketches, and then perform the all-pairs inner product query second. This simple change results in the main performance bottle-neck (the all-pairs inner product query) happening on a much smaller matrix, greatly speeding up the entire query process.

We will show that this process produces a good enough approximation of the CART sketch to act as the input to RECOVERYSTEP.

Algorithm 5: RECOVER.

Input: Row sketches $\mathbf{r^{(1)}}, \cdots, \mathbf{r^{(n)}}$ and totals $t^{(1)}, \cdots, t^{(n)}$
Output: Index set Ω of entries we are confident are large

```
Create empty multiset Ω
for γ ∈ [Γ] do
    Randomly generate functions P_1, P_2, s_1, s_2 for a Cartesian sketch CART
    Run APPROXIMATE, passing it P_1, P_2, s_1, s_2 and the row sketches
    Run RECOVERYSTEP, passing it CART and the result of APPROXIMATE
    Append the result of RECOVERYSTEP to Ω
Remove entries from Ω appearing fewer than Γ/2 times
return Ω (as a set)
```

▶ **Lemma 16.** *At the end of* APPROXIMATE*, for any given* $(h, g) \in [\Pi]^2$*, we have:*

$$|\mathbf{L^{(1)}}_{h,g} - \text{CART}(\mathbf{E^{(1)}C})_{h,g}| \leq \epsilon n \Pi^{-1} 207 \lambda^{-1/2},$$
$$\text{and } |\mathbf{R^{(1)}}_{h,g} - \text{CART}(\mathbf{CE^{(1)}})_{h,g}| \leq \epsilon n \Pi^{-1} 207 \lambda^{-1/2},$$

with probability at least $1 - \lambda/27 - \delta(2 + 12n/\Pi)$*, as long as* $\epsilon < 1/2$*.*

To meet the requirements for RECOVERY to work on these approximations, we need to set limits on the choices of ϵ and δ.

▶ **Lemma 17.** *If we have that:* $\delta \leq \lambda/(54(2 + 12n/\Pi))$*, and* $\epsilon \leq \min\{1/2, (\phi\Pi\lambda^{1/2})/(828n)\}$*, then* APPROXIMATE *produces approximations which are within the noise tolerance of* RECOVERYSTEP*.*

3.6 Analysis of Algorithm

Putting together the previous subsections, we can make the full recovery algorithm. The outline is listed in Algorithm 5.

▶ **Lemma 18.** *If we have that:*

$$\delta \leq \lambda/(54(2+12n/\Pi)),\ \epsilon \leq \min\{1/2, \phi\Pi\lambda^{1/2}/828n\},\ \Pi \geq \max\{18k, 18\|\mathbf{C_{-k}}\|_F \phi^{-1}\lambda^{-1/2})\},$$

then we can choose a $\Gamma \in O(\log n)$ *such that* RECOVER *returns every element of* $\text{LARGE}_{\phi,k}$ *with probability at least* $1 - n^{-3}$*.*

Using $\mathcal{M}(x, y)$ to represent the time required to multiply a $x \times y$ matrix by an $y \times x$ matrix, we can bound the time and space costs of the overall algorithm.

▶ **Lemma 19.** RECOVER *can be implemented to run in*

time $\tilde{O}(\Gamma(\Pi^2 + \log(1/\delta)(n\epsilon^{-2} + \mathcal{M}(\Pi, \epsilon^{-2}))))$ *and space* $\tilde{O}(\Pi^2 + n\epsilon^{-2}\log(1/\delta))$*.*

By setting $\Pi = n^\theta$ we can look for the right trade-off.

▶ **Theorem 20.** *For every* $\theta \in [0, 1]$*, there exists a sketch of size*

$$\tilde{O}\left(n^{2\theta}(k^2 + \frac{R^2}{\phi^2}) + n^{3-2\theta}(\frac{\phi^2}{k^2\phi^2 + R^2})\right)$$

from which we can extract the (up to k) entries with magnitude at least ϕ in time

$$\tilde{O}\left(n^{2\theta}(k^2+\frac{R^2}{\phi^2})+n^{3-2\theta}(\frac{\phi^2}{k^2\phi^2+R^2})+\mathcal{M}\left(n^{\theta}(k+\frac{R}{\phi}),n^{2-2\theta}(\frac{\phi^2}{k^2\phi^2+R^2}\right)\right)$$

with high probability.

▶ **Corollary 21.** *In particular, for $\theta = 2/3$, we can build a sketch of size $\tilde{O}\left(n^{5/3}(k^2+\frac{R^2}{\phi^2})\right)$ with query time $\tilde{O}\left(n^{5/3}(k^2+\frac{R^2}{\phi^2})\right)$.*

4 Concluding Remarks

We have shown how to guarantee accurate recovery of correlation outliers using a sketch-based method, beating LSH on query time for small correlation outliers and vanishing correlation non-outliers. A key part of our approach is to use sketching and coding ideas repeatedly: as well as using sketching to reduce the initial dimensionality of the data, we use a second "layer" of sketching when we combine subsets of signals, in order to speed up queries over many pairs of sketches at the cost of increased error. Where LSH tries to hash the correlated elements together, we try to separate them and then recover them from the noise. This produces a trade-off between the size of the underlying sketches and the final query time. This general approach could work in other situations where a large number of sub-queries need to be evaluated to search for large values, for example with measures of similarity/distance other than correlation.

Further, as the technique produces a linear intermediate sketch, this approach is easily adapted to recover pairs whose correlation deviates from some expected correlation matrix, or has changed compared with some previous point in time (simply perform the heavy hitters recover on the difference between two intermediate sketches built using the same permutations, signs, and codes).

Future directions would include finding ways to use alternative primitives to fast matrix multiplication (such as fast convolution via FFT, as adopted by Pagh) and trying to combine the advantages of LSH-based methods and heavy-hitters-based methods.

It would also be useful to re-analyze the algorithms of Valiant and Karppa et al. in our model of bounded total weight of the non-outliers. This could allow us to use some of their more powerful ideas to bring down the exponent in our query time cost from 5/3 to less than 1.6 (as they achieve in their Boolean algorithm for all vanishing non-outliers).

References

1. D. Achlioptas. Database-friendly random projections. In *ACM Principles of Database Systems*, pages 274–281, 2001.
2. N. Alon, P. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. In *ACM Principles of Database Systems*, pages 10–20, 1999.
3. N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *ACM Symposium on Theory of Computing*, pages 20–29, 1996.
4. Alexandr Andoni and Ilya P. Razenshteyn. Optimal data-dependent hashing for approximate near neighbors. *CoRR*, abs/1501.01062, 2015. `arXiv:1501.01062`.
5. Emmanuel Candes, Mark Rudelson, Terence Tao, and Roman Vershynin. Error correction via linear programming. In *Foundations of Computer Science, 2005. FOCS 2005. 46th Annual IEEE Symposium on*, pages 668–681. IEEE, 2005.

6 M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Procedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, 2002.

7 Graham Cormode. Sketch techniques for massive data. In Graham Cormode, Minos Garofalakis, Peter Haas, and Chris Jermaine, editors, *Synposes for Massive Data: Samples, Histograms, Wavelets and Sketches*, Foundations and Trends in Databases. NOW publishers, 2011.

8 David L Donoho. Compressed sensing. *IEEE Transactions on information theory*, 52(4):1289–1306, 2006.

9 Anna C Gilbert, Yi Li, Ely Porat, and Martin J Strauss. Approximate sparse recovery: optimizing time and measurements. *SIAM Journal on Computing*, 41(2):436–453, 2012.

10 P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *ACM Symposium on Theory of Computing*, pages 604–613, 1998.

11 W.B. Johnson and J. Lindenstrauss. Extensions of Lipshitz mapping into Hilbert space. *Contemporary Mathematics*, 26:189–206, 1984.

12 Daniel M. Kane and Jelani Nelson. Sparser johnson-lindenstrauss transforms. *Journal of the ACM*, 61(1):4:1–4:23, 2014.

13 Matti Karppa, Petteri Kaski, and Jukka Kohonen. A faster subquadratic algorithm for finding outlier correlations. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '16, pages 1288–1305, Philadelphia, PA, USA, 2016. Society for Industrial and Applied Mathematics. URL: `http://dl.acm.org/citation.cfm?id=2884435.2884525`.

14 Kasper Green Larsen, Jelani Nelson, Huy L Nguyên, and Mikkel Thorup. Heavy hitters via cluster-preserving clustering. In *Foundations of Computer Science (FOCS), 2016 IEEE 57th Annual Symposium on*, pages 61–70. IEEE, 2016.

15 P. Li, T. Hastie, and K. W. Church. Nonlinear estimators and tail bounds for dimension reduction in L_1 using cauchy random projections. *Journal of Machine Learning Research (JMLR)*, 2007.

16 Rasmus Pagh. Compressed matrix multiplication. *TOCT*, 5(3):9:1–9:17, 2013. `doi:10.1145/2493252.2493254`.

17 Eric Price and David P. Woodruff. (1 + eps)-approximate sparse recovery. In Rafail Ostrovsky, editor, *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, pages 295–304. IEEE Computer Society, 2011. `doi:10.1109/FOCS.2011.92`.

18 Daniel A. Spielman. Linear-time encodable and decodable error-correcting codes. *IEEE Trans. Information Theory*, 42(6):1723–1731, 1996. `doi:10.1109/18.556668`.

19 Gregory Valiant. Finding correlations in subquadratic time, with applications to learning parities and the closest pair problem. *J. ACM*, 62(2):13:1–13:45, 2015. `doi:10.1145/2728167`.

20 Leslie Valiant. Functionality in neural nets. In *First Workshop on Computational Learning Theory*, page 28–39, 1988.

A Detailed Proofs

Proof of Lemma 6. Recalling Definition 1, $\mathbf{C}_{i,j}$ can be expressed as $\mathbf{V}_{i,i}^{-1/2}\mathbf{V}_{i,j}\mathbf{V}_{j,j}^{-1/2}$, where each $\mathbf{V}_{h,g}$ is the scaled inner product between standardized rows $\hat{\mathbf{y}}^{(\mathbf{h})}$ and $\hat{\mathbf{y}}^{(\mathbf{g})}$:

$$\mathbf{V}_{h,g} = \frac{1}{p-1}(\mathbf{y}^{(\mathbf{h})} - \overline{\mathbf{x}}_{\mathbf{h}}\mathbf{e}^T)(\mathbf{y}^{(\mathbf{g})} - \overline{\mathbf{x}}_{\mathbf{g}}\mathbf{e}^T)^T.$$

Observe that factors in $V_{h,g}$ involving $p-1$ cancel in the expression for $\mathbf{C}_{i,j}$, so they can be ignored. What remains is the inner product between normalized (to Euclidean norm 1) versions of vectors $\mathbf{y}^{(\mathbf{i})} - \overline{\mathbf{x}}_{\mathbf{i}}\mathbf{e}^T$ and $\mathbf{y}^{(\mathbf{j})} - \overline{\mathbf{x}}_{\mathbf{j}}\mathbf{e}^T$.

Before performing STANDARDIZE, we had each $\mathbf{r}^{(\mathbf{i})} = \mathbf{S}(\mathbf{y}^{(\mathbf{i})})$. We also have that $(t^{(i)}/p) = \overline{\mathbf{x}}_i$. This means that at the end of the routine, each $\mathbf{r}^{(\mathbf{i})}$ is now a sketch of $(z^{(i)})^{-1/2}(\mathbf{y}^{(\mathbf{i})} - \overline{\mathbf{x}}_{\mathbf{i}}\mathbf{e}^T)$, where $(z^{(i)})^{-1/2}$ is the correct normalization factor to within multiplicative error in the range $[(1-2\epsilon)^{1/2}, (1+2\epsilon)^{1/2}]$ with probability at least $1-\delta$.

For the result, we require two such rescaling factors to be within their bounds, and also for the inner product query to succeed. Each of these three events holds with probability at least $1-\delta$, giving an overall probability at least $1-3\delta$ by the union bound.

To determine the overall error, consider that since $\epsilon < 1/2$ and $|\mathbf{C}_{i,j}| \leq 1$,

$$(\tilde{\mathbf{y}}^{(\mathbf{h})}(1\pm 2\epsilon)^{1/2}) \odot (\tilde{\mathbf{y}}^{(\mathbf{g})}(1\pm 2\epsilon)^{1/2}) \in (1\pm 2\epsilon)\mathbf{C}_{i,j} + \epsilon(1\pm 2\epsilon) \subset \mathbf{C}_{i,j} \pm 4\epsilon.$$

◀

Proof of Lemma 11. Let $(h,g) = (P_1(i), P_2(j))$ be the bucket (i,j) is mapped to. Since the partition functions are chosen uniformly at random, the chance that none of the other entries mapped to the same bucket are in $\text{LARGE}_{\phi,k}$ is at least $1 - 2k/\Pi$. To see this, observe that in the worst case, all index pairs in $\text{LARGE}_{\phi,k}$ have either the same row or same column index. Then, by the Markov inequality, we have less than $2k/\Pi$ probability that at least one of the remaining $k-1$ entries in $\text{LARGE}_{\phi,k}$ are mapped into one of the remaining $n/\Pi - 1$ slots in that bucket.

Now, if entry (i,j) turns out to be the only large entry in its bucket, then the event $\text{CORRECTDECODE}_{h,g}$ occurring implies that the index pair recovered from bucket (h,g) will be (i,j). The chance of both occurring is then at least $1 - x - 2k/\Pi$. ◀

Proof of Lemma 12. In the event that bucket (h,g) contains more or less than one large entry, then $\text{CORRECTDECODE}_{h,g}$ automatically holds, so we only need to consider the case of exactly one large entry in the bucket.

Now, consider the case of only one large entry $\mathbf{C}_{i,j}$ being mapped to the bucket. Observe that we can write

$$\text{CART}(\mathbf{E}^{(\mathbf{l})}\mathbf{C} - \mathbf{E}^{(\mathbf{l})})_{h,g} = \text{BIG} + \text{SMALL},$$

where $\text{BIG} = \mathbf{E}^{(\mathbf{l})}{}_{i,i} \cdot s_1(i) \cdot s_2(j) \cdot \mathbf{C}_{i,j}$ and $\text{SMALL} = \text{CART}(\mathbf{E}^{(\mathbf{l})}\mathbf{C}_{-\mathbf{k}})_{h,g}$ (see Definition 3).

When the event $\text{SMALLERROR}_{h,g,l}$ holds, we have that $|\text{SMALL}| \leq \phi/4$. Also, $|\text{BIG}|$ is either 0 (when the row of the large entry is masked) or greater than ϕ (when not masked). So, $\text{SMALLERROR}_{h,g,l}$ holding means that the l^{th} threshold bit will match the l^{th} bit of the code word for the row index we are trying to recover. An analogous argument applies to $\text{CART}(\mathbf{C}\mathbf{E}^{(\mathbf{l})} - \mathbf{E}^{(\mathbf{l})})_{h,g}$ and the column index.

For the decoder to correctly recover an index from its code word, we need at most a λ fraction of errors. So, we need less than a λ fraction of the events $\text{SMALLERROR}_{h,g,l}$ for $l \in [L \log n]$ failing to hold. By Markov's inequality, we can put the chance of more than a λ fraction of failures at less than y/λ. ◀

Proof of Lemma 13. For fixed $(h, g, l) \in [\Pi]^2 \times [L \log n]$, consider the random variable $\textsc{Cart}(\mathbf{E}^{(\mathbf{l})}\mathbf{C}_{-\mathbf{k}})_{h,g}$ (random over the choices of P_1, P_2, s_1, and s_2 that make up $\textsc{Cart}$). This can be broken down into a sum of contributions from each entry of $\mathbf{E}^{(\mathbf{l})}\mathbf{C}_{-\mathbf{k}}$, as follows:

$$\textsc{Cart}(\mathbf{E}^{(\mathbf{l})}\mathbf{C}_{-\mathbf{k}})_{h,g} = \sum_{(i,j)\in[n]^2} \beta_{i,j,l}$$

$$\text{where } \beta_{i,j,l} = \begin{cases} (\mathbf{E}^{(\mathbf{l})}{}_{i,i}) s_1(i) s_2(j) (\mathbf{C}_{-\mathbf{k}})_{i,j} & \text{if } (i,j) \in \mathcal{B}_{h,g} \\ 0 & \text{otherwise,} \end{cases}$$

recalling from definition 8 that $\mathcal{B}_{h,g}$ represents the index pairs mapped to bucket (h, g).

Due to the independently selected pairwise independent random sign functions s_1 and s_2, each term has mean $\mathbb{E}[\beta_{i,j,l}] = 0$ and covariance $\text{Cov}[\beta_{i_1,j_1,l}, \beta_{i_2,j_2,l}] = 0$ (where either $i_1 \neq i_2$ or $j_1 \neq j_2$). This means the variance of the sum (the bucket value) is simply the sum of the variances of the terms.

Each term has variance $\text{Var}[\beta_{i,j,l}] \leq (\mathbf{E}^{(\mathbf{l})}\mathbf{C}_{-\mathbf{k}})^2_{i,j}/\Pi^2$. To see this, observe that each term has at most a $1/\Pi^2$ chance of being non-zero (due to the random partition functions). Summing up all the terms gives us

$$\text{Var}\left[\textsc{Cart}(\mathbf{E}^{(\mathbf{l})}\mathbf{C}_{-\mathbf{k}})_{p,q}\right] = \|\mathbf{E}^{(\mathbf{l})}\mathbf{C}_{-\mathbf{k}}\|_F^2/\Pi^2 \leq \|\mathbf{C}_{-\mathbf{k}}\|_F^2/\Pi^2.$$

Then, by Chebyshev's inequality, we have $\textsc{Cart}(\mathbf{E}^{(\mathbf{l})}\mathbf{C}_{-\mathbf{k}})_{p,q} \geq \phi/4$, with probability less than $16\|\mathbf{C}_{-\mathbf{k}}\|_F^2/(\Pi^2\phi^2)$. An analogous argument applies to $\textsc{Cart}(\mathbf{C}_{-\mathbf{k}}\mathbf{E}^{(\mathbf{l})})_{h,g}$, giving the result by union bound. ◀

Proof of Lemma 14. Substituting $\Pi \geq 18\|\mathbf{C}_{-\mathbf{k}}\|_F/(\phi\lambda^{1/2})$ into Lemma 13 gives us that:

$$\mathbb{P}[\textsc{SmallError}_{h,g,l}] \geq 1 - 8\lambda/81.$$

Then by Lemma 11 (with $y = 8\lambda/81$), we get that:

$$\mathbb{P}[\textsc{CorrectDecode}_{h,g}] \geq 1 - 8/81.$$

Finally, using the fact that $\Pi \geq 18k$ (from the initial assumptions) along with Lemma 12 (with $x = 8/81$), we have that any fixed $(i, j) \in \textsc{Large}_{\phi,k}$ will be in the output of $\textsc{RecoveryStep}$ with probability at least $1 - 8/81 - 1/9 = 64/81 \geq 2/3$. ◀

Proof of Corollary 15. Observe that the proof of Lemma 12 still works with an additional term of magnitude no more than $\phi/4$. Then, observe that the choice of parameters in Lemma 14 leaves enough slack to condition on an additional event occurring with probability greater than $1 - \lambda/18$ per $\textsc{SmallError}_{h,g,l}$. ◀

Proof of Lemma 16. If we performed the algorithm with the exact vectors instead of AMS sketches, then $\mathbf{L}^{(\mathbf{l})}{}_{h,g}$ would be exactly $\textsc{Cart}(\mathbf{E}^{(\mathbf{l})}\mathbf{C})$. Any difference is due to the inner product approximation error which is smaller than $\epsilon\|\mathbf{h}\|_2\|\mathbf{g}\|_2$ with probability at least $1 - \delta$, where $\mathbf{h}$ and $\mathbf{g}$ are the vectors that $\textsc{LeftMasked}[h]$ and $\textsc{Right}[g]$ are sketches of. First consider

$$\mathbf{h} = \sum_{P_1(i)=h} (\mathbf{E}^{(\mathbf{l})}{}_{i,i} \cdot s_1(i) \cdot \mathcal{R}^{(i)} \cdot \tilde{\mathbf{y}}^{(\mathbf{i})}),$$

where $\mathcal{R}^{(i)}$ is the rescaling error caused by $\textsc{Standardize}$ (see Section 3.2). Recall that each $|\mathcal{R}^{(i)}| \leq 1 + 4\epsilon$ with probability at least $1 - 3\delta$ as long as $\epsilon < 1/2$.

The squared 2-norm $\|\mathbf{h}\|_2^2$ is given by:

$$\sum_{P_1(i)=P_1(j)=h} \mathcal{R}^{(i)}\mathcal{R}^{(j)}\langle(\mathbf{E^{(l)}}_{i,i}s_1(i)\tilde{\mathbf{y}}^{(\mathbf{i})}),(\mathbf{E^{(l)}}_{j,j}s_1(j)\tilde{\mathbf{y}}^{(\mathbf{j})})\rangle$$

For each $i = j$, the corresponding term is equal to $\mathcal{R}^{(i)}\mathcal{R}^{(j)}$, and for each $i < j$ there is a matching equal term with i and j swapped. So, with probability at least $1 - 3n\delta/\Pi$ the norm is at most $n(1+4\epsilon)^2/\Pi$ plus an independent random variable (random over choice of s_1) with mean 0 and variance less than

$$4\|\mathbf{C}\|_F^2(1+4\epsilon)^2/\Pi^2 \leq 4n^2(1+4\epsilon)^2/\Pi^2.$$

So by Chebyshev's inequality and a union bound, $\|\mathbf{h}\|_2^2$ is smaller than $\frac{n}{\Pi}(1+4\epsilon)^2(22\lambda^{-1/2}+1)$ with probability greater than $1 - \lambda/108 - 3n\delta/\Pi$. The same bound applies to $\|\mathbf{g}\|_2^2$, so $\epsilon\|\mathbf{h}\|_2\|\mathbf{g}\|_2 \leq \epsilon(1+4\epsilon)^2 n\Pi^{-1}23\lambda^{-1/2} \leq \epsilon n\Pi^{-1}207\lambda^{-1/2}$ with probability at least $1 - \lambda/54 - \delta(1+6n/\Pi)$.

An analogous argument works for entry $\mathbf{R^{(l)}}_{h,g}$ and $\textsc{Cart}(\mathbf{CE^{(l)}})$. A union bound over the probabilities of failure gives the result. ◀

Proof of Lemma 17. The assumptions imply that:

$\epsilon n\Pi^{-1}23\lambda^{-1/2} \leq 23\phi/828 \leq \phi/4$, and $1 - \delta(2+12n/\Pi) - \lambda/27 \leq 1 - \lambda/18$.

This tells us exactly that the errors on the approximations according to Lemma 16 are within the bounds allowed by Lemma 15. ◀

Proof of Lemma 18. From Lemmas 14 and 17 we know for $\Gamma = 1$, this algorithm succeeds at finding any one large entry with probability at least 2/3. By performing $O(\log n)$ independent repetitions and then only considering those index pairs appearing at least half the time, then by the Chernoff bound we can amplify the probability of finding any one of the large entries to $1 - n^{-5}$. There are at most n^2 such pairs, giving the result. ◀

Proof of Lemma 19. $\textsc{RecoveryStep}$ can be implemented to run in time $O(\Pi^2\text{polylog}n)$ since we have Π^2 iterations of the outer loop, $O(\log n)$ iterations of the inner loop, and all operations taking $O(\text{polylog}n)$ time (coding schemes with such fast decoding algorithms exist).

$\textsc{Approximate}$ can be implemented to run in time $O(\log n \log(1/\delta)(n\epsilon^{-2} + \mathcal{M}(\Pi, \epsilon^{-2})))$ where $\mathcal{M}(\Pi, \epsilon^{-2})$ is the time required to multiply a $\Pi \times \epsilon^{-2}$ matrix by an $\epsilon^{-2} \times \Pi$ matrix. This holds because there are $O(\log n)$ iterations of the outer loop. Then within we have $O(n)$ additions involving sketches of size $O(\epsilon^{-2}\log(1/\delta))$. We also have a series of inner products which can be expressed as a batched all-pair query. This can be performed as a series of $O(\log(1/\delta))$ matrix multiplications.

Putting this together, we get a time cost of $\tilde{O}(\Gamma(\Pi^2 + \log(1/\delta)(n\epsilon^{-2} + \mathcal{M}(\Pi, \epsilon^{-2}))))$. The filtering step adds no extra asymptotic time, since we can filter by sorting the $O(\Pi^2 \log n)$ pairs and then iterating over them counting repetitions to see if any exceed the $\Gamma/2$ threshold.

$\textsc{RecoveryStep}$ uses $O(\Pi^2 \log n + \text{polylog}n)$ space to store a pair of length $O(\log n)$ strings, a multiset of up to Π^2 index pairs, and the input of $O(\log n)$ Π-by-Π sketches. The polylog overhead is used for the encoding scheme.

$\textsc{Approximate}$ uses $O(\Pi^2 \log n + n\epsilon^{-2}\log(1/\delta))$ space to store $O(n)$ sketches and $O(\log n)$ $\Pi \times \Pi$ matrices.

All together we need $\tilde{O}(\Pi^2 + n\epsilon^{-2}\log(1/\delta))$ space, since the multiset contains at most $O(\Pi^2 \log n)$ index pairs. ◀

Proof of Theorem 20. Substitute bounds in Lemma 18 into costs in Lemma 19. ◀

Satisfiability for SCULPT-Schemas for CSV-Like Data

Johannes Doleschal
Universität Bayreuth, Bayreuth, Germany

Wim Martens
Universität Bayreuth, Bayreuth, Germany

Frank Neven
Hasselt University and transnational University of Limburg, Hasselt, Belgium

Adam Witkowski
University of Warsaw, Warsaw, Poland

Abstract

SCULPT is a simple schema language inspired by the recent working effort towards a recommendation by the World Wide Web Consortium (W3C) for tabular data and metadata on the Web. In its core, a SCULPT schema consists of a set of rules where left-hand sides select sets of regions in the tabular data and the right-hand sides describe the contents of these regions. A document (divided in cells by row- and column-delimiters) then satisfies a schema if it satisfies every rule. In this paper, we study the satisfiability problem for SCULPT schemas. As SCULPT describes grid-like structures, satisfiability obviously becomes undecidable rather quickly even for very restricted schemas. We define a schema language called L-SCULPT (Lego SCULPT) that restricts the walking power of SCULPT by selecting rectangular shaped areas and only considers tables for which selected regions do not intersect. Depending on the axes used by L-SCULPT, we show that satisfiability is PTIME-complete or undecidable. One of the tractable fragments is practically useful as it extends the structural core of the current W3C proposal for schemas over tabular data. We therefore see how the navigational power of the W3C proposal can be extended while still retaining tractable satisfiability tests.

2012 ACM Subject Classification Information systems → Semi-structured data, Theory of computation → Formal languages and automata theory, Theory of computation → Logic

Keywords and phrases CSV, Schema languages, Semi-structured data

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.14

Funding This work is supported by grant number MA 4938/2–1 from the Deutsche Forschungsgemeinschaft.

1 Introduction

Despite the availability of numerous standardized formats for semi-structured and semantic web data such as XML, RDF, and JSON, a very large percentage of data and open data published on the web remains tabular in nature.[1] Tabular data is most commonly published in the form of comma separated values (CSV) files because such files are open and therefore processable by numerous tools, and tailored for all sizes of files ranging from a number of KBs

[1] Jeni Tennison, one of the two co-chairs of the W3C CSV on the Web working group claims that over 90% of the data published on data.gov.uk is tabular data [14].

	1	2	3	4	5	6	7	8	9	10
1	subject	predicate	object	provenance						
2	:e4	type	PER							
3	:e4	mention	"Bart"	D00124	283-286					
4	:e4	mention	"JoJo"	D00124	145-149	0.9				
5	:e4	per:sibling	:e7	D00124	283-286	173-179	274-281			
6	:e4	per:age	"10"	D00124	180-181	173-179	182-191	0.9		
7	:e4	per:parent	:e9	D00124	180-181	381-380	399-406	D00101	220-225	230-233

Figure 1 Fragment of a CSV-like file (added row and column numbers), inspired by use case 13 in [13].

	1	2	3	4	5	6	7	8	9	10
1	subj	pred	obj	prov						
2	iri	pred-type	ent-type							
3	iri	pred-type	literal	doc-id	position					
4	iri	pred-type	literal	doc-id	position	certainty				
5	iri	iri	iri	doc-id	position	position	position			
6	iri	iri	literal	doc-id	position	position	position	certainty		
7	iri	iri	iri	doc-id	position	position	position	doc-id	position	position

Figure 2 Tokenized version of Figure 1, with added row and column numbers.

to several TBs. Despite these advantages, working with CSV files is often cumbersome [14] since they are typically not accompanied by a *schema* that describes the file's structure (i.e., "the second column is of integer datatype", "columns are delimited by tabs", ...) and captures its intended meaning. In fact, without schema information, already converting CSV-like data into a relational database is a challenging engineering problem [14]. In recognition of this problem, the *CSV on the Web* Working Group of the World Wide Web Consortium (W3C) [17] argues for the introduction of a schema language for tabular data to ensure higher interoperability when working with datasets using the CSV or similar formats. Inspired by the recent W3C effort towards a recommendation for tabular data and metadata on the Web, Martens et al. proposed the tabular schema language SCULPT [11]. At its core, SCULPT is a rule-based language with rules of the form $\varphi \to \rho$ where φ selects a set of regions[2] of the input table and ρ constrains the allowed structure and content of each such region. The region selection expressions φ are not limited to selecting columns but can navigate through a table, much like XPath expressions can navigate the nodes of an XML tree. This generalization beyond columns is necessary since there are natural cases in practice in which CSV-like data is not rectangular [2, 13] (see also Figure 1). In this paper, we address static optimization of SCULPT schemas, but first we present the main ideas behind the language by means of an example.

SCULPT schemas operate on *tabular documents* which are text files describing tabular data. Figure 1, coming from use case 13 in [13], shows an example CSV file, to which we added row numbers 1–7 on the left and column numbers 1–10 at the top. The original file uses tab (`\t`) as a column delimiter and newline (`\n`) as row delimiter. The rows and columns divide the document into *cells*. In this example, rows can have different numbers of cells, e.g., row two has three cells, whereas row three has five. The W3C describes the data as coming from a text extraction scenario, where "facts [are extracted] from text and [represented] as [RDF] triples along with associated metadata that include provenance and certainty values" [13]. Furthermore, "a single row in the table comprises a triple (subject-predicate-object), one or more provenance references and an optional certainty measure" [13]. In Figure 1, we see

[2] A region is a set of cells.

that the provenance information includes a document ID (e.g., the value D00124), pairs of string offsets within the document (e.g., 283–286), and an optional float representing a certainty measure (e.g., 0.9). This information can be repeated for several documents as is the case in row seven. There may not be an a priori bound on the number of columns that are needed for representing the provenance information. As we can infer from the W3C's textual description of the data, the logical organization of the data from column four to the right is in rows rather than in columns. The current W3C proposal for schemas [12, Section 5.5] does not deal with row-wise organization (and not even with different types of data in the same column) and therefore cannot adequately describe the data in their own use case 13. As we will see, SCULPT can capture the logic inherent in this example by describing the structure of the rectangle starting at cell (3,4) in the rule (†).[3]

Figure 3 shows an example SCULPT schema for CSV files of the form as depicted in Figure 1. It consists of two parts. The first part concerns *parsing information* – it defines the row and column delimiters and describes how cells should be tokenized. This allows to parse the text file and build a table-like structure consisting of rows and columns. Tokenization then proceeds as follows. The content of each cell is matched against the regexes in the schema's token rules. To each cell the first token is assigned for which the corresponding regex matches.[4] For instance, cell (4,3) in Figure 1 gets the token `literal` because it matches its regex `"[a-zA-Z0-9]*"` and none of the earlier regexes. Figure 2 depicts the tokenized CSV resulting from Figure 1, using the schema in Figure 3. The second part of the schema consists of *structural rules*. Left-hand sides select a set of regions, whereas right-hand sides are regular expressions that the tokens in each region should match. Consider the rule

```
row((1,1)) -> subj, pred, obj, prov
```

whose left-hand side selects one region (the row of the tokenized table starting at (1,1)) and requires it to match the regular expression `subj, pred, obj, prov` where the comma stands for concatenation. In our example, the right-hand side expressions always describe what *each row* in the selected region should look like. This is mostly important for the rule

```
rectangle(prov +(1,0)) -> (doc-id, position*, certainty?)*.        (†)
```

where the left-hand side selects an unbounded rectangular region for which the top left corner is the node matching the token `prov`, plus an offset $(1, 0)$, i.e., one row, zero columns. Each row in the selected rectangle should then match the expression

```
(doc-id, position*, certainty?)*
```

which it does in the example, as we can see in Figure 2. The language SCULPT is formally defined in Section 3.

In this paper, we study static optimization of SCULPT schemas. In particular, we address the satisfiability problem that asks whether for a given a SCULPT schema there is a CSV file that satisfies it. Not only is satisfiability a core problem in the foundations of database management field that has been studied in depth for a variety of formalisms, it is also particularly relevant for schema design as it allows to detect schemas that are not well-defined.

[3] We use coordinate (x, y) to refer to the cell in row x, column y.
[4] There are other options to assign tokens, e.g., as in [11], but tokenization is not our present focus.

```
%  Parsing information; Delimiters
Col Delim  = \t                                        Row Delim  = \n

% Token rules of the form <token name> = <regex>
subj       = subject                                   pred       = predicate
obj        = object                                    prov       = provenance
iri        = [a-zA-Z0-9]*:[a-zA-Z0-9]*                 pred-type  = type + mention
doc-id     = D[0-9]{5}                                 position   = [0-9]{3}-[0-9]{3}
certainty  = 0.[0-9] + 1.0                             literal    = "[a-zA-Z0-9]*"
ent-type   = PER + ORG + GPE

% Structural rules
row((1,1)) -> subj, pred, obj, prov                    col(subj)  -> iri*
col(obj)   -> (literal + iri + entity-type)*           col(pred)  -> (pred-type + iri)*
rectangle(prov +(1,0)) -> (doc-id, position*, certainty?)*
```

Figure 3 Schema for files of the type in Figure 1. (Syntax uses two columns to save space.)

Unsurprisingly, satisfiability of SCULPT quickly turns out to be undecidable, which we show by an easy reduction from the domino tiling problem [16]. Indeed, using only one rule, a region selection expression can be used to 'walk' over a grid testing all horizontal and vertical constraints, or alternatively many much simpler rules can be used to test all horizontal and vertical constraints in parallel for every domino type (cf. Section 4 for more details). Even though these observations are valid to demonstrate undecidability they use rather artificial constructions.

For this reason, we introduce a restricted variant of SCULPT called Lego SCULPT (L-SCULPT) that not only suffices to express the W3C use cases but also admits tractable satisfiability. In brief, L-SCULPT restricts region selection expressions to only select rectangular shaped areas, that is (parts of) rows, columns, and rectangles, thereby constraining the structural power of the language. A second restriction is that L-SCULPT only considers tables on which no two selected regions intersect. Specifically, in this paper, we make the following contributions:

1. We show that the safisfiability problem for the structural core of SCULPT is undecidable.
2. We define a fragment of SCULPT called L-SCULPT that is powerful enough to capture the structural core of the schemas for tabular data in the current W3C recommendation [12, Section 5.5]. Intuitively, L-SCULPT allows selections of rows, columns, and rectangles and bounded-size regions in the directions up, left, down, and right, whereas the W3C's recommendation only allows column selection.[5] As Figure 1 shows, column selection alone is too limited to describe schemas for the data fragments in the W3C's use cases, because the number of columns that the schema can describe is bounded by the number of rules in the schema. In the example, the number of columns can grow arbitrarily large. L-SCULPT strictly extends the structural core of the W3C recommendation.
3. Depending on which axes are used, we show that satisfiability of L-SCULPT is PTIME-complete or undecidable. Our main technical result shows that for L-SCULPT using only row, column, right, and rectangle selections satisfiability is in PTIME. The proof is an intricate reduction to the emptiness problem of nondeterministic tree automata where tables are encoded as trees.

[5] We only focus on the structural core of the languages. The W3C's proposal also supports key and foreign key constraints, which are out of scope here but easy to add to the language. (In fact, we implemented them in [3].)

4. Even the tractable fragments of L-SCULPT extend the structural core of the current W3C schema recommendation and are expressive enough to define a natural schema for Figure 1, one example is the schema in Figure 3. As such, our result shows how the W3C recommendation can be extended without making satisfiability intractable.

Related Work

Tabular or CSV-like data is one of many popular models for semi-structured data [1]. The schema language SCULPT was introduced in [11]. This work provides an initial formal model and considers efficient evaluation. In addition, several extensions like region semantics, token types, and transformations are considered. We implemented the system CHISEL for specifying, validating, analyzing, and debugging of SCULPT schemas and data transformations based on schema information [3]. Arenas et al. [2] propose a simple and expressive framework for adding metadata to CSV documents. They focus in particular on noisy CSV-like documents and consider the problem of annotating different elements of CSV-like files such as, for instance, cells, rows and columns. Documents are viewed as strings and regular expressions are used to select intervals. Navigation is restricted to moving to the next delimiter (any delimiter or one of a specific kind). They consider satisfiability as well as efficient evaluation. As the setting is considerably different from the one considered in this paper (grids versus strings), their and our results do not imply each other.

Labeled grids have been studied in the context of two-dimensional languages, also referred to as picture languages (cf., e.g., [5]). The bulk of the research in this area has focused on formalisms that could capture natural counterparts of string language theory, like regular languages, context-free languages, closure properties, etc. For instance, the equivalence of existential monadic-second order logic, complementation-free regular expressions, tiling systems (as projections of local languages) and two-dimensional online tessallation automata, provided enough motivation to refer to the latter class as the recognizable two-dimensional languages. Satisfiability for this class is shown to be undecidable through a direct simulation of Turing Machines [5]. Proposition 4 uses essentially the same idea but employs tiling systems. Although the proofs of Proposition 4 and Proposition 5 are rather straightforward and their novelty is limited, they do provide the necessary motivation for the introduction of L-SCULPT.

In Section 2, we introduce the necessary preliminaries. In Section 3, we formalize the structural core of SCULPT and show that the satisfiability problem is undecidable. In Section 4, we define L-SCULPT. In Section 5, we present our main technical result showing that satisfiability for L-SCULPT using only row, col, right and rectangle selections is PTIME-complete. In addition, we show that various extensions are undecidable. We conclude in Section 6.

2 Preliminaries

For numbers $n, m \in \mathbb{N}$, with $n < m$, we denote the sets $\{1, \ldots, n\}$ and $\{n, \ldots, m\}$ by $[n]$ and $[n, m]$, respectively. By Δ we denote an *alphabet*, that is, a finite, non-empty set of *symbols*. A *language* is a set of words over Δ. We assume familiarity with regular expressions but briefly describe their notation. The *regular expressions* over Δ are inductively defined as follows. Every symbol $a \in \Delta$ is a regular expression, and so is the special symbol ε, which denotes the empty word and which we assume not to be in Δ. If e_1 and e_2 are regular expressions, then so are $e_1 \cdot e_2$, $e_1 + e_2$, and e_1^*. We assume the usual precedence of operators.

The language $L(e)$ of e is defined as usual. We sometimes omit the concatenation symbol "$\cdot$", write e^+ to abbreviate ee^*, and write $e?$ to abbreviate $e + \varepsilon$.

Tables

CSV-like data consists of a text file with row and column delimiters (often newline and comma, respectively). These delimiters uniquely determine a tabular structure that can be given to the data, as we describe next. The main idea is very simple: if the file has n row delimiters, the table has $n + 1$ rows (a row delimiter is a separator between two consecutive rows). Likewise, if the file has m column delimiters in row i, then row i in the table has $m + 1$ cells. The main idea is visualised in Figure 4, which we revisit later.

We use a set $e = \{\sqcup, \triangleleft\}$ of special symbols that do not appear in any other set unless explicitly mentioned otherwise. We use $\sqcup$ to denote empty cells in the CSV file and $\triangleleft$ to denote cells that do not exist in the CSV file.[6]

If we denote a set by a single symbol (say, $\mathcal{V}$), we always assume that it does not contain any symbol from e. We use the following notation: $\mathcal{V}^e = \mathcal{V} \cup \{\sqcup, \triangleleft\}$ and $\mathcal{V}_x = \mathcal{V} \cup \{x\}$ for every symbol $x \in e$.

Let $\mathcal{V}$ be a set and $n, m \in \mathbb{N}$. A *matrix* M over $\mathcal{V}^e$ is a mapping from $[n] \times [m]$ to $\mathcal{V}^e$. We say that M has n *rows* and m *columns*. A *cell* is determined by its coordinate $(k, \ell) \in [n] \times [m]$ and its *content* is the value $M(k, \ell)$. We usually denote the later value as $M_{k,\ell}$. We denote the set $[n] \times [m]$ of all *coordinates* of M by $\mathrm{Coords}(M)$. A *region* in M is a set of coordinates, that is, a subset of $[n] \times [m]$.

▶ **Definition 1** (Core Tabular Data Model, [11, 15])**.** A *table* T *over* $\mathcal{V}^e$ is a matrix over $\mathcal{V}^e$ that satisfies the requirement that whenever $T_{i,j} = \triangleleft$ then $T_{i,j'} = \triangleleft$ for all $j' > j$ (i.e., $\triangleleft$ is only used for padding to the right).

The purpose of the $\triangleleft$ symbol is just to indicate how many cells there are in a row in the underlying CSV-like text file. The text file in Figure 4 has two row separators ($\hookleftarrow$), so the corresponding table has three rows. The number of columns in the table is the maximal number of column separators (comma) in a row of the text file, plus one. The first row of the CSV-like file has three cells, for which the first two are empty (do not even contain whitespace). We use $\sqcup$ to denote this in the table. Furthermore, the last cell in the first row in the table is labeled $\triangleleft$ to indicate that this cell does not exist in the underlying text file on the left. (Since rows are left-aligned in CSV-like files, there can only be $\triangleleft$ to the right of $\triangleleft$.) The second row in the text file on the left has one cell, which is empty. The third and final row has four cells. The last cell contains $\sqcup$ since the column separator is the last symbol of the CSV-like text file.

We assume the natural *table order* on coordinates. That is, we say that coordinate (k, ℓ) precedes coordinate (k', ℓ') (denoted $(k, \ell) < (k', \ell')$) if (k, ℓ) precedes (k', ℓ') lexicographically, that is: either (1) $k < k'$ or (2) $k = k'$ and $\ell < \ell'$. When depicting tables, we always put cell (1,1) on the top left.

[6] These symbols play a similar role as the "end of tape" marker and the blank symbol in some definitions of Turing Machines. Here we only focus on $\sqcup$ and $\triangleleft$ and their correspondence to the underlying CSV-like text files.

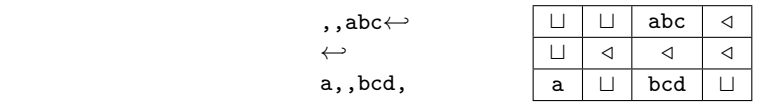


Figure 4 A CSV-like text file (left) and its corresponding tabular representation (right).

3 A Structural Core of SCULPT

We present a formal model for the structural core of SCULPT. In our formalization, we will work with tables corresponding to *tokenized* CSV files as exemplified in Figure 2. Formally, a SCULPT schema is a tuple $S = (\Delta^e, R)$ where

- Δ^e is a finite set of elements which we call *tokens*, and
- R is a set of *structural rules* of the form $s \to \rho$ that constrain the admissible table content:
  - s is a *region selection expression* that maps every table over Δ^e to a set of regions; and,
  - ρ is a *content expression* that defines the permitted content of regions selected by s.

The schema in Figure 3 has, in addition to the structural rules R, a set of *token rules* that associate tokens to cell contents. We omitted these here because we focus on the structural core of the language. In what follows we just use the term *rules* to refer to *structural rules*.

In brief, a SCULPT schema defines a set of tables over Δ^e. Intuitively, such tables should satisfy all rules $s \to \rho$ in the schema. Let T be a table over Δ^e and z be a region of T. In the different versions of SCULPT that we consider, we will define when z satisfies ρ, which we denote by $z \models \rho$. The region selection expressions s, when applied to a table T, returns a *set of regions*, i.e., $[\![s]\!]_T$ is a subset of $2^{\mathrm{Coords}(T)}$.

▶ **Definition 2.** A table T over Δ^e *satisfies* a SCULPT schema $S = (\Delta^e, R)$, denoted $T \models S$, if $z \models \rho$ for every rule $s \to \rho$ in R and $z \in [\![s]\!]_T$.

We now define the region selection and content expressions for SCULPT. The full language SCULPT will only be used in this section, where the main goal is to understand which properties of SCULPT make satisfiability undecidable. In Section 4, we introduce L-SCULPT which avoids these properties and for which satisfiability becomes tractable for some fragments.

Region Selection Expressions

We now define *region selection expressions* for the most general schemas we consider. Intuitively, a region selection expression is of the form $f(\varphi)$ where φ is a formula that returns a region z and f is an operator in $\{\mathsf{region}, \mathsf{rows}\}$ that maps regions to sets of regions.[7] The formulas φ in SCULPT are formulas in propositional dynamic logic (PDL for short) [4], tweaked for navigation over grids. We refer to Appendix A for details. Then, for a region z, we define $\mathsf{region}(z)$ as $\{z\}$ and $\mathsf{rows}(z)$ as the set of rows in z, more specifically $\mathsf{rows}(z) = \{\{(i,j) \in z \mid j \in \mathbb{N}\} \mid i \in \mathbb{N}\}$.[8]

[7] In [11, 3], this operator is encoded in rules by using different arrows: rules with => use $f = \mathsf{region}$ and rules with -> use $f = \mathsf{rows}$. We use the same convention in Figure 3, where $f = \mathsf{row}$ for every rule.

[8] SCULPT as defined in [11] only uses regions and rows. We also implemented cols [3], that cuts z into its set of columns (and can be defined analogously), but do not use it in the present paper.

Content Expressions

As content expressions, we simply use regular expressions over Δ^e. Let T be a table, let z be a region of T, and let ρ be a content expression. Then z *satisfies* ρ *(denoted* $z \models \rho$*)* if there exist tokens $a_1, \ldots, a_n \in \Delta^e$ such that $a_1 \cdots a_n \in L(\rho)$ and $a_1 \cdots a_n$ is the enumeration of all tokens in z in table order. Recall that Definition 2 now implies that $T \models (\Delta^e, R)$ if, for every rule $f(\varphi) \to \rho$ in R, we have that $z \models \rho$ for every $z \in f(Z)$, where Z is the region selected by φ in T.

Decision Problems

We recall that validation of SCULPT schemas is in linear time:

▶ **Theorem 3** ([11]). *Given a table T and* SCULPT *schema S, testing if $T \models S$ can be done in time $O(|T| \cdot |S|)$.*

In this paper, we study satisfiability problems for SCULPT. The most straightforward variant is defined as follows:

Problem:	SAT
Input:	A SCULPT schema S.
Question:	Is there a table T such that $T \models S$?

In its full generality, SAT is easily seen to be undecidable. The proof is a simple reduction from Domino Tiling where only *one* region selection expression is used to 'walk' over the grid checking all horizontal and vertical constraints.

▶ **Proposition 4.** SAT *is undecidable for* SCULPT*, even if schemas use only one rule that selects only one cell.*

We note that a similar result was obtained by Göller et al. [7, Theorem 4.11], where satisfiability of PDL with restricted negation was shown to be undecidable. The main difference is that Göller et al. consider infinite satisfiability whereas we consider finite satisfiability. Göller et al. encode the grid structure in their formula but, from there on, their proof and ours use a similar main idea.

Restricting the 'walking' power of region selection expression does not suffice for decidability as the next proposition shows. Again, a reduction from Domino Tiling is used but now the schema needs to select 'intersecting' regions: every cell containing a certain domino is in a selected region that checks the horizontal constraints and another region that tests the vertical constraints for this domino.

▶ **Proposition 5.** SAT *is undecidable for* SCULPT*, even if rules only use left-hand-sides that select "the row of cell (1,1)", "the column of cell (1,1)", "the cell(s) to the immediate right of a given token", and "the cell(s) immediately below a given token".*

4 Lego SCULPT

The use cases put forward by W3C [13] do not use powerful 'walking' expressions or select 'intersecting' regions as is done in the proofs of Propositions 4 and 5. As in addition the schemas used in the proofs of the mentioned propositions are rather artificial, we introduce a restricted variant of SCULPT called Lego SCULPT (L-SCULPT). From a structural perspective, this language is still more powerful than the W3C's proposal for a schema

language, can more accurately describe the data in the W3C use cases (see Figures 1–3), and admits tractable satisfiability. In brief, L-SCULPT restricts region selection expressions to select rows, columns, and rectangles. A second restriction is that L-SCULPT only considers tables on which no two selected regions intersect.[9] Formally, an L-SCULPT schema $S = (\Delta^e, R)$ is a pair as before but with some restrictions that we explain next.

Region Selection Expressions

We first discuss the region selection expressions occurring as left-hand sides of rules in R:

$$s \quad := \quad c \mid \mathsf{up}(d) \mid \mathsf{down}(d) \mid \mathsf{left}(d) \mid \mathsf{right}(d) \mid \mathsf{row}^\bullet(d) \mid \mathsf{col}^\bullet(d) \mid \mathsf{rect}(d+o)$$

Here, $\bullet \in \{*, +\}$, c is a coordinate, d is a coordinate or a token (different from $\triangleleft$ or $\sqcup$), and $o \in \{0,1\} \times \{0,1\}$ is an offset. We allow offsets in rectangles for flexibility. In Figure 3, it is convenient to use the offset (1,0), for example. For the definition of $[\![s]\!]_T$, we use the following shorthand: If c is a coordinate and a is a token, we write $c \models a$ if $T_c = a$. Additionally, we define $c \models c$ for each coordinate c of T. Furthermore, we denote by R_T the region consisting of all cells of T. Then, $[\![s]\!]_T$ defines a set of regions as follows:[10]

$$\begin{aligned}
[\![c]\!]_T &:= \{\{c\}\} \\
[\![\mathsf{rect}(d+o)]\!]_T &:= \{\{c + o + (k,\ell) \mid k,\ell \in \mathbb{N}\} \cap R_T \mid c \models d\} \\
\\
[\![\mathsf{up}(d)]\!]_T &:= \{\{(i-1,j)\} \cap R_T \mid (i,j) \models d\} \\
[\![\mathsf{down}(d)]\!]_T &:= \{\{(i+1,j)\} \cap R_T \mid (i,j) \models d\} \\
\\
[\![\mathsf{left}(d)]\!]_T &:= \{\{(i,j-1)\} \cap R_T \mid (i,j) \models d\} \\
[\![\mathsf{right}(d)]\!]_T &:= \{\{(i,j+1)\} \cap R_T \mid (i,j) \models d\} \\
\\
[\![\mathsf{row}^*(d)]\!]_T &:= \{\{c + (0,k) \mid k \in \mathbb{N}\} \cap R_T \mid c \models d\} \\
[\![\mathsf{row}^+(d)]\!]_T &:= \{\{c + (0,k+1) \mid k \in \mathbb{N}\} \cap R_T \mid c \models d\} \\
\\
[\![\mathsf{col}^*(d)]\!]_T &:= \{\{c + (k,0) \mid k \in \mathbb{N}\} \cap R_T \mid c \models d\} \\
[\![\mathsf{col}^+(d)]\!]_T &:= \{\{c + (k+1,0) \mid k \in \mathbb{N}\} \cap R_T \mid c \models d\}
\end{aligned}$$

Again, c is a coordinate, d is a coordinate or token, and o an offset. We give some intuition. Rule $\mathsf{col}^*(c)$ selects a singleton region consisting of c and all cells below c. Rule $\mathsf{row}^+(a)$ selects a set of regions, namely, for each cell c with token a, the region having all cells to the right of c. The rule $\mathsf{right}(a)$ contains, for each cell c with token a, the region $\{c + (0,1)\}$. Finally, $\mathsf{rect}(c + (1,0))$ contains the set of rows starting in coordinates below c.

We refer to rules with a coordinate in their left-hand side as *coordinate rules* and to the other rules as *token rules*. We say that the expressions of the form rect are *rectangular*. Although rect selects sets of rows, the terminological intuition is the following: since the rows we select are consecutive and all start in the same column, their union in T is always rectangular.

Observe that, in the case of coordinate rules, $\mathsf{row}^+((x,y))$ is syntactic sugar for $\mathsf{row}^*((x, y+1))$, whereas $\mathsf{row}^*((1,1))$ ("select the first row") cannot be expressed with row^+. In the

[9] The restriction to brick-like regions together with the disjointness requirement explains why we refer to this fragment as Lego SCULPT.

[10] For simplicity, we do not make use of a 'slice' operator f as in Section 3, but rather defined the set of regions directly.

case of token rules, although one can use $\mathsf{row}^*(a)$, it introduces redundancy: its content expression will have to repeat that the first cell in the region contains a. In the remainder of the paper, we therefore do not consider $\mathsf{row}^+(c)$ or $\mathsf{row}^*(a)$. Furthermore, we just write $\mathsf{row}(c)$ for $\mathsf{row}^*(c)$ and $\mathsf{row}(a)$ for $\mathsf{row}^+(a)$ and follow the same conventions for columns. The rules in Figure 3 are all in L-SCULPT, with the semantics as define here.

We assume in the sequel that the coordinates c are encoded in unary. We feel that this assumption is reasonable because we have not yet encountered data for which large coordinates are needed. In W3C schemas for tabular data, such numbers (the number of columns of the data) are encoded in unary as well.

Content Expressions

Content expressions define the allowed content of cells but can no longer force cells in the underlying CSV-file to be missing. We therefore restrict content expressions in L-SCULPT, by disallowing the explicit use of $\triangleleft$, i.e., content expressions are now regular expressions over $\Delta_{\sqcup}$. In addition, when matching a region against a content expression, we allow arbitrarily long padding at the end. Formally, for a content expression ρ, denote by $L^e(\rho)$ the language $L(\rho \cdot \sqcup^* \cdot \triangleleft^*)$. We say that a region z *satisfies* ρ (denoted $z \models \rho$) if $a_1 \cdots a_n \in L^e(\rho)$, where $a_1, \ldots, a_n$ are the tokens in z, in table order.

Padding with $\triangleleft$ allows us to take non-rectangular CSV-like files into account. Padding with $\sqcup$ ensures that content expressions cannot "force" a row in a CSV-file to be short. Indeed, consider the following contrived example with a rule $\mathsf{right}(a) \to \varepsilon$. If we would define $L^e(\rho) = L(\rho \cdot \triangleleft^*)$ then the content of the cell to the right of a would need to be $\triangleleft$, therefore forcing the cell in the underlying CSV data to be missing. In our definition, we allow the cell to be missing or empty.

Region Disjointness

Let $S = (\Delta^e, R)$ be an L-SCULPT schema and T a table. Intuitively, we say that S is *region-disjoint* on T if all regions selected by S are pairwise disjoint. Formally, S is *region-disjoint* on T if, for every pair of rules $r_1 = s_1 \to \rho_1$ and $r_2 = s_2 \to \rho_2$ and every pair of regions $z_1 \in [\![s_1]\!]_T$ and $z_2 \in [\![s_2]\!]_T$, if $z_1 \cap z_2 \neq \emptyset$, then $r_1 = r_2$ and $z_1 = z_2$. Finally, we say that $T \models S$ if S is region-disjoint on T and $r \models \rho$ for every rule $s \to \rho$ in R and for *every* region $r \in [\![s]\!]_T$.

Notice that, given a table T and schema S, it is easy to check whether T is region-disjoint on S. This can be checked during evaluation, which is in in time $O(|T| \cdot |S|)$, even for full SCULPT (Theorem 3).

Recall that we introduced region-disjointness to avoid the undecidability problems in Proposition 5. An alternative, more restrictive way of introducing region-disjointness would be to require that a schema is in L-SCULPT only if it is region-disjoint on *every* table for which the rules match. But as we discuss next, this severely limits the power of L-SCULPT and makes schemas more difficult to design. For instance, already the schema consisting of the two rules $\mathsf{row}(a_1) \to b^*$ and $\mathsf{col}(a_2) \to b^*$ would be disallowed, because there exist tables (say, with a_1 in cell (2,1) and a_2 in cell (1,2)) in which the row and column intersect. (In our semantics, the schema for instance is satisfied by putting a_1 in (1,1), a_2 in (2,1), and no b at all.) Furthermore, the alternative semantics would introduce strange behavior. While the schema with the rule $\mathsf{row}(a) \to b^*$ is clearly satisfiable, the rule $\mathsf{row}(a) \to b^* + c^*(ac + d)^*c^*$ would not be allowed due to the occurrence of a in the content expression. The burden then lies with the user to rewrite the rule to $\mathsf{row}(a) \to b^* + c^*d^*c^*$ to become allowed again. In the semantics we propose, the last two rules are both allowed and define the same set of tables.

Finally, notice that the problem of testing whether there is a table T such that S is region-disjoint on T and all rules of S match is precisely SAT for L-SCULPT, which we study in Section 5.

Comparison With W3C Schemas for Tabular Data

The W3C proposes a schema language for tabular data in [12, Section 5.5]. From a structural perspective, this language is a strict subset of L-SCULPT, since it can be expressed as L-SCULPT using only rules of the form $\mathsf{col}(c) \to a^*$, where c is a coordinate and a is a token. Furthermore, the W3C schemas also do not admit selection of intersecting regions. Concerning our example in the introduction, although it is possible to define a schema using only rules of the form $\mathsf{col}(c) \to a^*$ for the data in Figure 1, we feel that such an approach leads to a much less accurate description of the data than our example in Figure 3.

▶ **Example 6.** The set of rules in Figure 3 are written in L-SCULPT. Furthermore, the schema is region-disjoint on the table T corresponding to the CSV file of Figure 2. (Therefore, T witnesses that the schema is satisfiable.) Recall that it is impossible to describe the data in such CSV files using column navigation only, as is currently the case in the W3C recommendation.

5 Satisfiability for L-SCULPT

In this section, we discuss the complexity of the satisfiability problem for L-SCULPT schemas. We distinguish between L-SCULPT fragments by explicitly listing the allowed operators in region selection expressions. For instance, L-SCULPT(row, col, right) denotes the fragment of L-SCULPT that only uses the operators row, col and right as region selection expressions.

In Section 5.1, we obtain the main technical result of the paper by delineating a relevant L-SCULPT fragment for which SAT is tractable, namely L-SCULPT(row, col, rect, right). In Section 5.2, we show that various extensions of this fragment become undecidable.

5.1 Polynomial-Time Fragments

We first show that satisfiability is in polynomial time for L-SCULPT(row, col, rect, right):

▶ **Theorem 7.** SAT *for* L-SCULPT(*row*, *col*, *rect*, *right*) *is PTIME-complete.*

The lower bound is a straightforward reduction from the emptiness problem of context-free grammars. The upper bound is significantly more challenging and is a reduction to the emptiness problem of nondeterministic tree automata where we represent tables T satisfying L-SCULPT schemas S as trees and use tree automata to match the rules of S on T and to test for region-disjointness.

We start by introducing the necessary terminology concerning trees, tree automata and the embedding of trees into tables in Section 5.1.1. When a schema has no coordinate rule, it is trivally satisfiable as it is satisfied by any table containing only $\triangleleft$-entries. So, we prove in Section 5.1.2 the upper bound for Theorem 7 for the special case where schemas consist of *exactly* one coordinate rule and do not make use of rectangular regions. Hereafter, we extend the proof to multiple coordinate rules and rectangles in Section 5.1.3 thereby completing the proof of Theorem 7. Finally, we discuss at the end of Section 5.1 how to obtain PTIME satisfiability for other L-SCULPT fragments.

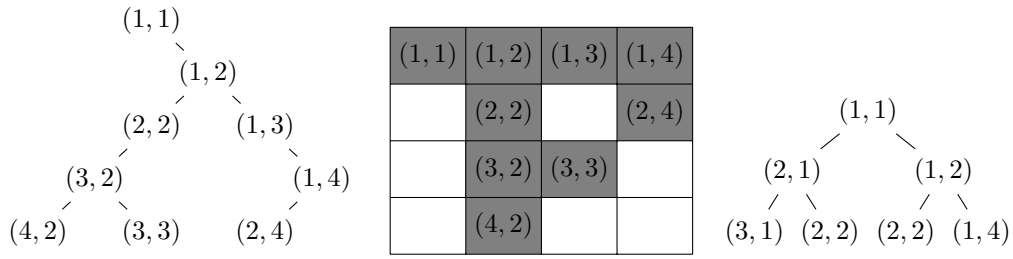


Figure 5 A binary tree (left), a table embedding for the tree on the left (middle), and a tree that does not have a table injection (right).

5.1.1 Preliminaries regarding trees

A (rooted, ordered, finite, labeled) *binary tree* is a finite tree where every node has at most two children (*left child* and *right child*). We allow nodes to have a left (resp., right) child only. We denote the empty tree by ε. Non-empty trees are denoted as $a(t_1, t_2)$. Here, the root carries the label a, has left subtree t_1, and right subtree t_2. Notice that t_1 or t_2 can be empty. For instance, $a(\varepsilon, \varepsilon)$ is a tree that consists of a single node, labeled a. We denote by $\mathrm{Nodes}(t)$ the set of nodes in the tree t. Every node u in the tree has a single *label* from Δ. For a formal introduction into tree automata we refer to Appendix B.1.

Table Embeddings and Table Trees

As we want to use tree automata to reason about tables, we define a correspondence between tables and trees. A *table embedding* of a binary tree t in a table T is a mapping $\mu : \mathrm{Nodes}(t) \to \mathrm{Coords}(T)$ such that, for each node v of t,

- the left child v_1 is mapped directly below its parent, that is, $\mu(v_1) = \mu(v) + (1, 0)$, and
- the right child v_2 is mapped directly to the right of its parent, that is, $\mu(v_2) = \mu(v) + (0, 1)$.

A *table injection* is an injective table embedding.

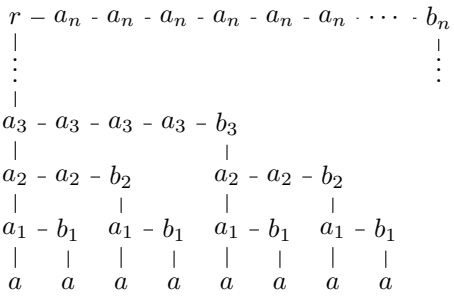
Notice that a table embedding is always completely determined by the cell on which the root of t is mapped. We illustrate table embeddings in Figure 5. The tree on the left has a table injection which is depicted in the middle. The tree on the right does not have a table injection: its canonical embedding is not injective on the nodes labeled $(2, 2)$.

In the remainder, we use the term *table tree* for a tree that has a table injection. Since we use trees to reason about tables, it will be more natural to speak of the *downward* and the *right* child of nodes in trees (as opposed to the *left* and the *right* child). Similarly, a *right path* (resp., *downward path*) is a path consisting only of right (resp., downward) children.

We note that other variants of table (or grid) embeddings have been studied in the literature (see, e.g., [8, 9]). These works allow children of nodes to be mapped to neighboring cells in the table, but they leave freedom as to which neighboring cells can be chosen. Under this alternative definition of embeddings, it is NP-complete to decide if an injective embedding exists for a given tree [8]. For table injections, however, this question is trivially in PTIME because it only amounts to testing if the canonical table embedding is injective.

$$
\begin{array}{lcl}
(1,1) & \to & r \\
\mathsf{col}(r) & \to & a_{n-1}\cdots a_1 a \\
\mathsf{right}(r) & \to & a_n \\
\mathsf{right}(a_i) & \to & a_i + b_i \\
 & & \quad (\text{for all } i = 1,\ldots,n) \\
\mathsf{col}(b_i) & \to & a_{i-1}a_{i-2}\cdots a_1 a \\
 & & \quad (\text{for all } i = 1,\ldots,n)
\end{array}
$$

(a) SCULPT schema consisting of $2n+1$ rules, each of size $O(n)$.



(b) Smallest table tree satisfying the schema in Figure 6a in a region disjoint manner. The path at the top contains 2^n nodes labeled a_n.

Figure 6 Schema for which the smallest region-disjoint table has an exponentially long path.

5.1.2 One Coordinate and No Rectangles

We assume that an L-SCULPT schema has at least one coordinate rule. As already mentioned above, when a schema has no coordinate rule, it is trivially satisfiable. We first prove that SAT is in PTIME for schemas with a single coordinate rule and that uses no rectangular regions.

Given an L-SCULPT(row, col, right) schema S, we construct a tree automaton $\mathcal{A}_S$ of size polynomial in S such that $L(\mathcal{A}_S) \neq \emptyset \Leftrightarrow S \in \text{SAT}$. Ideally, $\mathcal{A}_S$ would accept precisely those trees whose table injection satisfies S. But as the following example shows the latter condition could require $\mathcal{A}_S$ to be of size exponential in S. Indeed, consider the class of schemas in Figure 6a (which depends on a number n). Figure 6b shows that the smallest trees corresponding to tables satisfying S in a region-disjoint manner have an exponentially long path. Therefore, we construct $\mathcal{A}_S$ to accept precisely those trees that *can be pumped* to a tree that has a table injection satisfying S.

Pumping trees

We describe how trees are pumped, but first we need some terminology. We say that a token a is a col token if $\mathsf{col}(a)$ is a left-hand side of a rule in S. We define row tokens and right tokens analogously. A *horizontal token* is a row token or a right token. We sometimes also call col tokens *vertical tokens* for consistency in terminology.

Consider a tree t with a long right path containing node u_1 and, further right, u_2. Moreover, u_1 and u_2 have col tokens and no node between them has a col token. (One can think of u_1 being the root node in Figure 6b and u_2 the b_n-labeled node.) Furthermore, assume that t has a long downward path rooted at u_2. In any table embedding e, nodes u_1 and u_2 would be mapped to cells on the same row, i.e., with coordinates (i, j_1) and (i, j_2) for some i and $j_1 < j_2$. So, for e to be injective, the entire subtree rooted at u_1's downward child must be mapped into the region $\{(x, y) \mid x > i \text{ and } j_1 \leq y < j_2\}$, in order to avoid intersection with downward path below u_2.

A crucial observation is that we do not need to check this for very long paths. Since content expressions are regular, if $j_2 - j_1$ exceeds the size of the largest content expression in S, then the gap between u_1 and u_2 can be made arbitrarily large by pumping, avoiding the use of any column tokens. This means that, in this case, the size of the subtree at u_1's downward child does not matter for testing if the embedding e can be made injective. This

will later help us to ensure region-disjointness. The tree automaton will therefore count the distance between such branches up to a certain length, exceeding the size of the largest content expression in S. Beyond that, it classifies the distance as 'arbitrarily large'.

We also need to reason about the *width* of a tree t, which intuitively corresponds to the number of columns that will be used in its table embedding. Formally, if $t = \varepsilon$ then $\text{width}(t) = 0$. If $t = a(t^{\downarrow}, t^{\rightarrow})$, then $\text{width}(t) = \max\{\text{width}(t^{\downarrow}), 1 + \text{width}(t^{\rightarrow})\}$.

As we only consider row, col, and right axes, testing if a schema is region-disjoint on a table that corresponds to a table tree t amounts to the following. We need to test if, for every subtree $a(t^{\downarrow}, t^{\rightarrow})$ with $\text{width}(t^{\downarrow}) = k$, the tree $t^{\rightarrow}$ starts with a right path of length at least k before we see the first node with a col token or a downward child. We formalize this as follows. For a tree $t = a(t^{\downarrow}, t^{\rightarrow})$, we define path-length$(t)$ to be the number of nodes on the right path starting at the root, before we reach a node with a col token or a downward child. We set the value to ∞ if such a node does not exist. Formally, taking $\infty + 1 = \infty$, we define path-length as follows:

- if $t^{\downarrow} \neq \varepsilon$ or a is a col token, then path-length$(t) = 0$,
- else, if $t^{\rightarrow} = \varepsilon$, then path-length$(t) = \infty$, and
- otherwise, path-length$(t) = 1 +$ path-length$(t^{\rightarrow})$.

Main Construction

We construct a tree automaton $\mathcal{A}_S$ from a given L-SCULPT(row, col, right) schema S, such that $L(\mathcal{A}_S) \neq \emptyset \Leftrightarrow S \in \text{SAT}$. We need one more technicality. We call a token a *forbidden* if there are either two rules $\mathsf{right}(a) \to \rho_1$ and $\mathsf{row}(a) \to \rho_2$, two distinct rules $\mathsf{right}(a) \to \rho_1$ and $\mathsf{right}(a) \to \rho_2$, or two distinct rules $s_1(a) \to \rho_1$ and $s_2(a) \to \rho_2$ where $s_1 = s_2$. Notice that forbidden tokens can never appear in a table satisfying S. Observe that the set of forbidden tokens can be trivially computed in PTIME.

The tree automaton $\mathcal{A}_S$ is the intersection of the following automata:

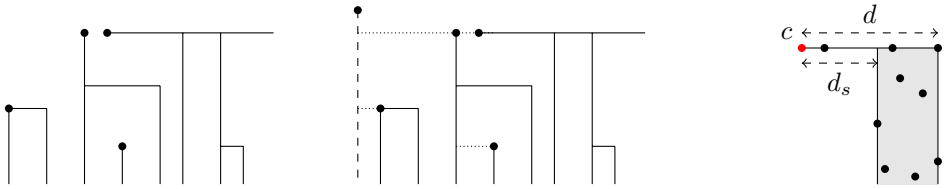
- $\mathcal{A}_{forbidden}$, which accepts the trees without forbidden tokens;
- $\mathcal{A}_{content}$, which checks conformity w.r.t. the content expressions and for region disjointness in the same dimension. More precisely, $\mathcal{A}_{content}$ checks
  1. whether the table induced by the tree matches the rules in the schema, and
  2. whether two row rules, a row and a right rule or two col rules select a common cell;[11]
- $\mathcal{A}_{\mathsf{row/col}}$, which checks that there are no row and col rules that select a common cell; and
- $\mathcal{A}_{\mathsf{right/col}}$, which checks that the input tree can be pumped to a tree where no right and col rule select a common cell.

Formally, the invariants of the automata are captured by the four Lemmas 12, 13, 14, 15, which can be found in Appendix B.2. Using these lemmas, it can be proved that the product automaton $\mathcal{A}_S$ accepts a tree if and only if it can be pumped to a table tree that satisfies S. By construction, in $L(\mathcal{A}_S)$, we will only be able to pump paths that are longer than $|\mathcal{A}_{content}|$.

▶ **Theorem 8.** *Let S be an* L-SCULPT(*row*, *col*, *right*) *schema that contains exactly one coordinate rule and no rectangular regions. Then we can construct in time polynomial in $|S|$ a tree automaton $\mathcal{A}_S$ such that $L(\mathcal{A}_S) \neq \emptyset \Leftrightarrow S \in \text{SAT}$.*

As testing for non-emptiness of tree automata is in PTIME, SAT for L-SCULPT(row, col, right) is in PTIME for schemas that contain at most one coordinate rule.

[11] A col rule is a rule with col(d) as a left-hand side. Similar for row and right rule.



(a) Example structure of a model for a schema with multiple coordinates.

(b) How the tree automaton sees it similar to one coordinate case.

(c) Area of responsibility of a state (c, d_s, d) in the automaton.

Figure 7 Dealing with the multiple coordinate case.

5.1.3 Multiple Coordinate Rules and Rectangles

We now show how to extend the result from Theorem 8 to the case where the schema may contain multiple coordinate rules. The main idea is summarized in Figure 7. Figure 7a depicts an instance of a region-disjoint forest with four roots. To turn it into a tree, we compute an equisatisfiable L-SCULPT schema, where every coordinate is shifted one cell to the right, add a new coordinate at (1,1) and select its entire column; see the dashed line in Figure 7b. Since all coordinates in the new L-SCULPT schema are at column two or higher, every coordinate in Figure 7a now has at least one selected cell somewhere to its left. We then allow the tree automaton to read a branch to the right at every node, provided this branch ends in a coordinate. (These are the dotted lines in Figure 7b.) The tree automaton then tests if it finds every coordinate in the tree.

Naïvely, the latter test requires an exponential number of states (see Example 11 in Appendix B.1), since it seems that the tree automaton needs a state for every subset of C, the set of coordinates occurring in the schema. However, let $n \in \mathbb{N}$ be the smallest number such that $C \subseteq [n]^2$. We can do the test with a polynomial number of states if we exploit that the schema is in L-SCULPT(row, col, right). We prove that an automaton for finding all coordinates in C can use states (c, d_s, d), where c is a coordinate in $[n]^2$, and d_s and d are in $[n] \cup \{0\}$. Formally, if a state (c, d_s, d) is assigned to a node u in an accepting run, it means that the automaton finds in the subtree at u all coordinates in $(a, b) \in C$ of the form (1) $x = a$ and $y \leq b \leq y + d$ and (2) $a \geq x$ and $y + d_s \leq b \leq y + d$. See Figure 7c for an illustration. Furthermore, we can also deal with rectangular regions:

▶ **Theorem 9.** *Let S be an* L-SCULPT(*row*, *col*, *rect*, *right*) *schema. Then we can construct in time polynomial in $|S|$ a tree automaton $\mathcal{A}_S$ such that $L(\mathcal{A}_S) \neq \emptyset \Leftrightarrow S \in$ SAT.*

5.2 Undecidable Fragments

L-SCULPT(row, col, rect, right) is asymmetric in that it only allows bounded navigation in one direction (i.e., the right operator). Next, we show that bounded navigation in two directions makes SAT undecidable. The first cases in the following theorem are the most natural ones. Here, we need at most one direction for unbounded navigation. The last case needs two unbounded directions, but we only need them once in the proof, for navigating to the bottom right corner of the data and start the tiling encoding from there. The undecidability results follow by reductions from Domino Tiling.

▶ **Theorem 10.** SAT *is undecidable for* L-SCULPT(*row*, *up*, *down*), L-SCULPT(*right*, *down*), L-SCULPT(*col*, *left*, *right*), *and* L-SCULPT(*row*, *col*, *up*, *left*).

6 Discussion

In this paper, we considered static optimization of SCULPT. As the satisfiability problem for SCULPT becomes undecidable rather quickly, we defined the restriction L-SCULPT and show that it admits a PTIME satisfiability test for the fragment L-SCULPT(row, col, rect, right). This fragment contains selection of rows, columns, and rectangular regions, starting from a given cell, plus bounded regions to the right (i.e., it can simulate "select three cells to the right of coordinate (5,2)" or "select the five cells to the right of each a"). Although SAT is still in PTIME if we replace right with down (by symmetry), we show that it becomes undecidable when we extend this fragment with bounded navigation in two directions. Interestingly, the just mentioned PTIME fragments contain the structural core of the W3C recommendation [12, Section 5.5] and, in addition, extend it with features allowing to deal with cases like, for instance, Use Case 13 in [13] that can not be addressed with the current recommendation. To summarize, even though L-SCULPT(row, col, rect, right) seems to be very restrictive when one starts from the full language SCULPT, it still extends the current W3C recommendation and seems powerful enough to describe many data sets in practice.

Our result is still useful if one would omit the region disjointness condition in the semantics of L-SCULPT. The PTIME algorithm would still be sound but incomplete. The only case where the algorithm would return 'no', although the schema would be satisfiable is when it can only be satisfied using tables in which selected regions are not disjoint.

References

1 Serge Abiteboul, Marcelo Arenas, Pablo Barceló, Meghyn Bienvenu, Diego Calvanese, Claire David, Richard Hull, Eyke Hüllermeier, Benny Kimelfeld, Leonid Libkin, Wim Martens, Tova Milo, Filip Murlak, Frank Neven, Magdalena Ortiz, Thomas Schwentick, Julia Stoyanovich, Jianwen Su, Dan Suciu, Victor Vianu, and Ke Yi. Research directions for principles of data management (abridged). *SIGMOD Record*, 45(4):5–17, 2016.

2 Marcelo Arenas, Francisco Maturana, Cristian Riveros, and Domagoj Vrgoč. A framework for annotating csv-like data. *Proceedings of the VLDB Endownment (PVLDB)*, 9, 2016.

3 Johannes Doleschal, Nico Höllerich, Martens, and Frank Neven. CHISEL: Sculpting tabular and non-tabular data on the Web. In *World Wide Web Conference (WWW)*, 2018. To appear.

4 Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.

5 Dora Giammarresi and Antonio Restivo. Two-dimensional languages. In *Handbook of Formal Languages: Volume 3 Beyond Words*, chapter 4. Springer, 1997.

6 Ian Glaister and Jeffrey Shallit. A lower bound technique for the size of nondeterministic finite automata. *Information Processing Letters*, 59(2):75–77, 1996.

7 Stefan Göller, Markus Lohrey, and Carsten Lutz. PDL with intersection and converse: satisfiability and infinite-state model checking. *The Journal of Symbolic Logic*, 74(1):279–314, 2009.

8 Angelo Gregori. Unit-length embedding of binary trees on a square grid. *Information Processing Letters*, 31:167–173, 1989.

9 Ralf Heckmann, Ralf Klasing, Burkhard Monien, and Walter Unger. Optimal embedding of complete binary trees into lines and grid. *Journal of Parallel and Distributed Computing*, 49(1):40–56, 1998.

10 Leonid Libkin, Wim Martens, and Domagoj Vrgoč. Querying graphs with data. *Journal of the ACM*, 63(2):14:1–14:53, 2016.

11 Wim Martens, Frank Neven, and Stijn Vansummeren. SCULPT: A schema language for tabular data on the web. In *World Wide Web Conference (WWW)*, pages 702–712, 2015.

12 Rufus Pollock, Jeni Tennison, Gregg Kellogg, and Ivan Herman. Metadata vocabulary for tabular data. Technical report, World Wide Web Consortium (W3C), December 2015. `https://www.w3.org/TR/2015/REC-tabular-metadata-20151217/`.

13 Jeremy Tandy, Davide Ceolin, and Eric Stephan. CSV on the Web: Use cases and requirements. Technical report, World Wide Web Consortium (W3C), Februrary 2016. `http://www.w3.org/TR/2016/NOTE-csvw-ucr-20160225/`.

14 Jeni Tennison. 2014: The year of CSV. `http://theodi.org/blog/2014-the-year-of-csv`. Visited on Sept. 18, 2017.

15 Jeni Tennison and Gregg Kellogg. Model for tabular data and metadata on the web. Technical report, World Wide Web Consortium (W3C), July 2014. `https://www.w3.org/TR/2015/REC-tabular-data-model-20151217/`.

16 Peter van Emde Boas. The convenience of tilings. In *Complexity, Logic and Recursion Theory*, volume 187 of *Lecture Notes in Pure and Applied Mathematics*, pages 331–363. Marcel Dekker Inc., 1997.

17 World Wide Web Consortium (W3C). CSV on the web working group charter. `https://www.w3.org/2013/05/lcsv-charter.html`. Visited on Sept. 18, 2017.

A SCULPT Region Selection Expressions

SCULPT *node expressions* are formulas φ in propositional dynamic logic (PDL for short) [4], tweaked for navigation over grids:

$$\begin{aligned}\varphi, \psi &:= a \mid \mathsf{root} \mid \mathsf{true} \mid \langle\alpha\rangle \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \neg\varphi \mid \alpha(\varphi)\\ \alpha, \beta &:= \varepsilon \mid \mathsf{up} \mid \mathsf{down} \mid \mathsf{left} \mid \mathsf{right} \mid [\psi] \mid (\alpha \cdot \beta) \mid (\alpha + \beta) \mid (\alpha^*)\end{aligned}$$

Here, φ and ψ are node expressions and α and β are navigational expressions. Furthermore, a ranges over tokens in Δ^e and root is a constant referring to the cell at coordinate $(1,1)$. Node expressions map a table to a region, whereas navigational expressions are mappings from regions to regions.

When evaluated over a table T, a node expression φ defines a region $(\!|\varphi|\!)_T \subseteq \mathrm{Coords}(T)$, as follows:

$$\begin{aligned}(\!|a|\!)_T &:= \{(i,j) \in \mathrm{Coords}(T) \mid a \in T_{i,j}\}\\ (\!|\langle\alpha\rangle|\!)_T &:= \{c \in \mathrm{Coords}(T) \mid (\!|\alpha(\{c\})|\!)_T \neq \emptyset\}\\ (\!|(\varphi \vee \psi)|\!)_T &:= (\!|\varphi|\!)_T \cup (\!|\psi|\!)_T\\ (\!|(\neg\varphi)|\!)_T &:= \mathrm{Coords}(T) \setminus (\!|\varphi|\!)_T\end{aligned} \qquad \begin{aligned}(\!|\mathsf{root}|\!)_T &:= \{(1,1)\}\\ (\!|\mathsf{true}|\!)_T &:= \mathrm{Coords}(T)\\ (\!|(\varphi \wedge \psi)|\!)_T &:= (\!|\varphi|\!)_T \cap (\!|\psi|\!)_T\end{aligned}$$

We define $(\!|\alpha(\varphi)|\!)_T$ as $(\!|\alpha(z)|\!)_T$, where $z = (\!|\varphi|\!)_T$ and $(\!|\alpha(z)|\!)_T$ is recursively defined as follows (for an arbitrary region $z \subseteq \mathrm{Coords}(T)$):

$$\begin{aligned}(\!|\varepsilon(z)|\!)_T &:= z\\ (\!|(\alpha \cdot \beta)(z)|\!)_T &:= (\!|\beta((\!|\alpha(z)|\!)_T)|\!)_T\\ (\!|(\alpha + \beta)(z)|\!)_T &:= (\!|\alpha(z)|\!)_T \cup (\!|\beta(z)|\!)_T\\ (\!|(\alpha^*)(z)|\!)_T &:= \textstyle\bigcup_{i \geq 0} (\!|\alpha^i(z)|\!)_T\\ (\!|[\psi](z)|\!)_T &:= (\!|\psi|\!)_T \cap z\end{aligned} \qquad \begin{aligned}(\!|\mathsf{up}(z)|\!)_T &:= \{(i-1,j) \mid i > 1, (i,j) \in z\}\\ (\!|\mathsf{down}(z)|\!)_T &:= \{(i+1,j) \mid i < n, (i,j) \in z\}\\ (\!|\mathsf{left}(z)|\!)_T &:= \{(i,j-1) \mid j > 1, (i,j) \in z\}\\ (\!|\mathsf{right}(z)|\!)_T &:= \{(i,j+1) \mid j < m, (i,j) \in z\}\end{aligned}$$

Here, α^i abbreviates the i-fold composition $\alpha \cdots \alpha$. We also use this abbreviation in the remainder. Notice that every coordinate (k, ℓ) of T can be expressed as $\mathsf{down}^{k-1}\mathsf{right}^{\ell-1}(\mathsf{root})$. Henceforth, we therefore use coordinates (k, ℓ) as syntactic sugar. Due to the close connection to PDL, region selection expressions are also very close to some fragments of Graph XPath [10].

To sum up, $f(\varphi)$ on a table T defines the set of regions $\mathsf{region}(\langle\!|\varphi|\!\rangle_T)$ and $\mathsf{rows}(\langle\!|\varphi|\!\rangle_T)$ when f is region or rows, respectively.

B Polynomial-Time Fragments

B.1 Trees and Tree Automata

A *nondeterministic tree automaton (over alphabet* Δ*)* or *NTA* is a tuple $N = (Q, \Delta, \delta, F, e)$ where Q is a finite set of states, $F \subseteq Q$ is the set of accepting states, e is a bit indicating if N accepts the empty tree or not, and δ is a set of transition rules of the form $[q_1, q_2] \xrightarrow{a} q$, where $a \in \Delta$ and $q_1, q_2 \in Q \uplus \{\varepsilon\}$. A *run* of N on a labeled binary tree t is an assignment of nodes to states $\lambda : \mathrm{Nodes}(t) \to Q$ such that for every $v \in \mathrm{Nodes}(t)$ the following holds. Let ℓ_v be the label of v. If v is a leaf, then $[\varepsilon, \varepsilon] \xrightarrow{\ell_v} \lambda(v) \in \delta$; if v only has a left child v_1 then $[\lambda(v_1), \varepsilon] \xrightarrow{\ell_v} \lambda(v) \in \delta$; if v only has a right child v_2 then $[\varepsilon, \lambda(v_2)] \xrightarrow{\ell_v} \lambda(v) \in \delta$ and, finally, if v has left child v_1 and right child v_2 then $[\lambda(v_1), \lambda(v_2)] \xrightarrow{\ell_v} \lambda(v) \in \delta$. A run is *accepting* if $\lambda(r) \in F$ for the root r of t. A non-empty tree t is *accepted* if there exists an accepting run on t. The empty tree is accepted if $e = \text{true}$. The set of all accepted trees is denoted by $L(N)$.

Transition rules suggest that NTAs read trees in a *bottom-up* manner. For this reason, NTAs are usually referred to as *bottom-up* nondeterministic tree automata. However, the semantics of NTAs does not depend on whether we write rules "bottom-up" as $[q_1, q_2] \xrightarrow{a} q$ or "top-down" as $q \xrightarrow{a} [q_1, q_2]$. In our proofs, we mix notation depending on the situation at hand.

▶ **Example 11.** We give an example of a tree automaton (that is useful in Section 5.1.3). Let $C \subseteq \Delta$. The tree automaton $N_C = (Q, \Delta, \delta, C, \text{false})$ accepts precisely those trees in which every symbol from C occurs. Define $Q = 2^C$. We define δ as follows:

$$[\varepsilon, \varepsilon] \xrightarrow{a} \{a\} \cap C \qquad [C_1, \varepsilon] \xrightarrow{a} (C_1 \cup \{a\}) \cap C$$
$$[C_1, C_2] \xrightarrow{a} (C_1 \cup C_2 \cup \{a\}) \cap C \qquad [\varepsilon, C_2] \xrightarrow{a} (C_2 \cup \{a\}) \cap C$$

In an accepting run, N_C visits a node u in a state $C' \subseteq C$ iff C' is the largest subset from C such that the subtree rooted at u contains every symbol from C'. We note that N_C is exponentially larger than $|C|$. This exponential size in C cannot be avoided as even on words, the smallest NFA recognizing the words in which all symbols from C occur has exponential size. The latter can be easily proved using Glaister and Shallit's lower bound technique for the size of NFAs [6]. Intuitively, the blow-up is due to the fact that the symbols from C may occur in any order and therefore the automaton needs to remember which symbols have been already encountered.

B.2 Invariants for the Main Construction

The invariants of the four tree automata used in Theorem 8 are captured by the following four Lemmas.

▶ **Lemma 12.** *An NTA* $\mathcal{A}_{forbidden}$ *can be constructed in time polynomial in* $|S|$ *such that* $L(\mathcal{A}_{forbidden})$ *is the set of trees containing no forbidden token.*

For a node u in t we define its set of *triggering rules in* S as follows:

- if $\mathrm{lab}(u) = a$, then its triggering rules are all rules of the form $\mathsf{row}(a) \to \rho$, $\mathsf{col}(a) \to \rho$, or $\mathsf{right}(a) \to \rho$ in S and
- if u is the root of t, then, additionally, the unique coordinate rule of S is also triggering.

▶ **Lemma 13.** *An NTA $\mathcal{A}_{content}$ can be constructed in time polynomial in $|S|$ such that $L(\mathcal{A}_{content}) \cap L(\mathcal{A}_{forbidden})$ is the set of trees t such that for every node u all the following hold:*

1. *If u has a triggering rule $\mathsf{row}(d) \to \rho$ then the right path of t rooted at u forms a word in $L(\rho)$ that does not contain any horizontal tokens.*
2. *If u has a triggering rule $\mathsf{col}(d) \to \rho$ then the downward path of t rooted at u forms a word in $L(\rho)$ that does not contain any vertical tokens.*
3. *If u has a triggering rule $\mathsf{right}(d) \to \rho$ then the label of the right child of u either is $\sqcup$, or a word in $L(\rho)$.*
4. *u lies on a path described in cases 1-3.*

In the above, d can be a coordinate or a token. For the unique rule where d is a coordinate, node u is included in the respective path. In all other cases it is excluded.

We note that property 4 ensures that a tree accepted by $\mathcal{A}_{content}$ is minimal in the sense that it only contains nodes required to match the schema.

▶ **Lemma 14.** *An NTA $\mathcal{A}_{\mathsf{row}/\mathsf{col}}$ can be constructed in time polynomial in $|S|$ such that $t \in L(\mathcal{A}_{\mathsf{row}/\mathsf{col}})$ if and only if, for every subtree $a(t^{\downarrow}, t^{\rightarrow})$ of t, there is no row token in $t^{\downarrow}$ or there is no col token in $t^{\rightarrow}$.*

▶ **Lemma 15.** *An NTA $\mathcal{A}_{\mathsf{right}/\mathsf{col}}$ can be constructed in time polynomial in $|S|$ such that $t \in L(\mathcal{A}_{\mathsf{right}/\mathsf{col}})$ if and only if, for every node u of t, path-length$(t^{\rightarrow}) \leq |\mathcal{A}_{content}|$ implies that*

$$\mathit{width}(t^{\downarrow}) \leq \mathit{path\text{-}length}(t^{\rightarrow}) + 1,$$

where $a(t^{\downarrow}, t^{\rightarrow})$ is the subtree of t rooted in u.

Querying the Unary Negation Fragment with Regular Path Expressions

Jean Christoph Jung
University of Bremen, Bremen, Germany
jeanjung@informatik.uni-bremen.de

Carsten Lutz
University of Bremen, Bremen, Germany
clu@informatik.uni-bremen.de

Mauricio Martel
University of Bremen, Bremen, Germany
martel@informatik.uni-bremen.de

Thomas Schneider
University of Bremen, Bremen, Germany
ts@informatik.uni-bremen.de

Abstract

The unary negation fragment of first-order logic (UNFO) has recently been proposed as a generalization of modal logic that shares many of its good computational and model-theoretic properties. It is attractive from the perspective of database theory because it can express conjunctive queries (CQs) and ontologies formulated in many description logics (DLs). Both are relevant for ontology-mediated querying and, in fact, CQ evaluation under UNFO ontologies (and thus also under DL ontologies) can be 'expressed' in UNFO as a satisfiability problem. In this paper, we consider the natural extension of UNFO with regular expressions on binary relations. The resulting logic UNFO$^{\text{reg}}$ can express (unions of) conjunctive two-way regular path queries (C2RPQs) and ontologies formulated in DLs that include transitive roles and regular expressions on roles. Our main results are that evaluating C2RPQs under UNFO$^{\text{reg}}$ ontologies is decidable, 2ExpTime-complete in combined complexity, and coNP-complete in data complexity, and that satisfiability in UNFO$^{\text{reg}}$ is 2ExpTime-complete, thus not harder than in UNFO.

2012 ACM Subject Classification Theory of computation → Logic

Keywords and phrases Query Answering, Regular Path Queries, Decidable Fragments of First-Order Logic, Computational Complexity

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.15

Funding A full version of this paper with an appendix that contains detailed proofs is available at `http://www.informatik.uni-bremen.de/tdki/research/papers.html`

Acknowledgements ERC Consolidator Grant 647289 CODA and DFG grant SCHN 1234/3.

1 Introduction

In ontology-mediated querying, queries against incomplete and heterogeneous data are supported by an ontology that provides domain knowledge and assigns a semantics to the data [15, 19, 37, 41]. The ontologies are often formulated in a specialized language such as a description logic [4, 5] or an existential rule language [6, 7, 22, 33] while the actual query is typically a conjunctive query (CQ) or a mild extension thereof such as a union of CQs

(UCQ). However, it can also be useful to consider more expressive decidable fragments of first-order logic (FO) as an ontology language as this serves to explore the limits of the ontology-mediated querying approach, to provide maximum expressive power for ontology formulation, and to put ontology-mediated querying into a more general logical perspective. Notably, this has been done in [8, 9, 19], where the guarded fragment (GF), the unary negation fragment (UNFO), and the guarded negation fragment (GN) of FO have been used as ontology languages. These fragments originate from the attempt to explain the good computational behaviour of modal and description logics and to extend their expressive power in a natural way. While GF and UNFO are orthogonal in expressive power, GN subsumes both of these fragments [9] and all of them subsume many common modal and description logics. It is an important result that, for all these fragments, ontology-mediated querying with UCQs remains decidable and that the complexity stays within the expected, namely 2ExpTime combined and coNP data.

From the perspective of database theory, it is an attractive property of both UNFO and GN (but not of GF) that they can express CQs and UCQs. In ontology-mediated querying, this allows to 'express' the evaluation of ontology-mediated queries in terms of satisfiability in a natural way. It is easiest to state this for Boolean queries: if $(\mathcal{O}, \Sigma, q)$ is an ontology-mediated query (OMQ) where $\mathcal{O}$ is an ontology, Σ a set of predicate symbols (that is, relation names) that may occur in the data, and q a UCQ, and D is a Σ-database, then $D \models (\mathcal{O}, \Sigma, q)$ iff $\mathcal{O} \wedge D \wedge \neg q$ is unsatisfiable. When $\mathcal{O}$ is formulated in UNFO or in GN, then so is $\mathcal{O} \wedge D \wedge \neg q$. What is more, the containment of OMQs can also be 'expressed' as a satisfiability problem in the natural case where both OMQs contain the same ontology and Σ is the set of all predicates symbols; from now on, we generally mean this case when speaking of OMQ containment. But also beyond ontology-mediated querying, we believe that the ability to express UCQs makes UNFO and GN attractive as an expressive logical backdrop for database theory.

In this paper, we study the natural extension UNFO^{reg} of UNFO with regular path expressions on binary relations. The resulting logic has the attractive property that it allows to express regular path queries [29] and conjunctive two-way regular path queries (C2RPQs) [25] as well as unions thereof (UC2RPQs). Such queries play a central role in the area of graph databases [2, 10] and they have also received considerable attention in ontology-mediated querying [12, 17, 18, 26, 27, 28, 40]. An additional reason to consider UNFO^{reg} is provided by the observation that transitive roles are an important feature of many common description logics (a role is a binary relation), but that transitive roles cannot be expressed in UNFO. In UNFO^{reg}, even transitive closure of roles and regular expressions on roles are expressible, two features that are provided by several expressive description logics [3, 24]. As a concrete example, every ontology formulated in $\mathcal{ALCI}^{\text{reg}}$, the extension of the common description logic $\mathcal{ALCI}$ with regular expressions on roles [44], can be expressed in UNFO^{reg} and thus the evaluation of ontology-mediated queries $(\mathcal{O}, \Sigma, q)$ where $\mathcal{O}$ is formulated in $\mathcal{ALC}^{\text{reg}}$ and q is a UC2RPQ can be 'expressed' as a satisfiability problem in UNFO^{reg}; of course, the same is true when $\mathcal{O}$ is formulated in UNFO^{reg} itself. We remark that transitive roles cannot be expressed in GF and GNF either, and that adding transitive relations to GF without losing decidability requires to impose rather strong syntactical restrictions [45], especially so in an ontology-mediated querying context [34]. Adding transitive relations to GNF has, to the best of our knowledge, not yet been studied.

The main problem that we are interested in is evaluating OMQs in which the ontology is formulated in UNFO^{reg} and the actual query is a UC2RPQ. We show that this problem is decidable, 2ExpTime-complete in combined complexity and coNP-complete in data

complexity. We further consider the OMQ containment problem and show that it is 2ExpTime-complete as well. We additionally show that the complexity of model checking in UNFO$^{\text{reg}}$ is the same as in UNFO, namely complete for $\text{P}^{\text{NP}[O(\log^2 n)]}$.

As explained above, both OMQ evaluation and OMQ containment can be reduced to satisfiability in polynomial time. For studying the combined complexity of the former and the complexity of the latter, we thus concentrate on the satisfiability problem and prove a 2ExpTime upper bound. Note that the addition of regular expressions does thus not increase the complexity of this problem as satisfiability in UNFO is also 2ExpTime-complete [46] and that the lower bound holds already when the arity of predicates is bounded by two, as a consequence of the results in [39]. Our proof proceeds by first showing that every satisfiable UNFO$^{\text{reg}}$ formula φ has a model whose treewidth is bounded by the size of φ, then establishing a characterization of the satisfaction of C2RPQs (that occur as a building block in φ) in such models in terms of certain witness trees, and finally showing that this infrastructure gives rise to a decision procedure based on two-way alternating tree automata. This 'direct approach' is in contrast to the reduction to satisfiability in the μ-calculus used for UNFO in [46] which seems unwieldy in the presence of regular path expressions. Note in particular that an important reason for the relative simplicity of the reduction in [46] is that there is always a model of bounded treewidth in which any two bags overlap in at most one element; this is no longer true in UNFO$^{\text{reg}}$. To establish the coNP upper bound on data complexity, we first observe that it suffices to consider a database satisfiability problem (given a database D, is there a model of the fixed UNFO$^{\text{reg}}$ sentence φ that extends D?) and then establish a certain kind of decoration of D as a witness for D being a positive instance, in a way such that witnesses can be guessed and verified in polynomial time.

Related work. For general background on ontology-mediated querying, we refer to [15, 19, 37, 41] and the references therein. OMQ containment was considered in [11, 13, 14, 21]. UNFO was introduced and studied by ten Cate and Segoufin in [46] and it was considered as an ontology language for OMQs in [19]. Regular path queries, C2RPQs, and variations thereof emerge from graph databases, see the surveys [2, 10] and references therein. We use C2RPQs that admit nesting via node tests, as considered in [16], see also [20]. Sometimes, this is referred to as 'nested' C2RPQs. There are several further extensions of C2RPQs that are not considered in this paper. We still mention two of them. A more powerful form of nesting is obtained by allowing to use C2RPQs with two answer variables in place of binary predicates in regular expressions, giving rise to regular queries [42]. Another expressive extension of C2RPQs is defined by monadically defined queries, which implement a certain 'flag and check' paradigm [43]. OMQs in which the actual query is some form of regular path queries are considered in [12, 17, 18, 26, 27, 28, 40]. As discussed in more detail later, UNFO$^{\text{reg}}$ is also related (but orthogonal in expressive power) to propositional dynamic logic with intersection and converse (ICPDL) [31] and to UNFO extended with fixed points [46].

2 Preliminaries

We assume that a countably infinite supply of predicate symbols of each arity is available. In the *unary negation fragment of first-order logic extended with regular path expressions* (UNFO$^{\text{reg}}$), formulas φ are formed according to the following grammar:

$$\begin{array}{lcl} \varphi & ::= & P(\mathbf{x}) \mid E(x,y) \mid x = y \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \exists x \varphi \mid \neg\varphi(x) \\ E & ::= & R \mid R^- \mid E \cup E \mid E \cdot E \mid E^* \mid \varphi(x)? \end{array}$$

where P ranges over predicate symbols, R over binary predicate symbols, and, in the $\neg\varphi(x)$ clause, φ has no free variables besides (possibly) x. Expressions E formed according to the second line are called *(regular) path expressions* and expressions $\varphi(x)?$ according to the last clause in that line are called *tests*. Tests are similar to the test operator in propositional dynamic logic (PDL) [30] and to node tests in XPath [32] and in some versions of regular path queries [16, 20]. When we write $\varphi(\mathbf{x})$, we generally mean that the free variables of φ are among $\mathbf{x}$, but not all variables from $\mathbf{x}$ need actually be free in φ. For a UNFO$^{\mathsf{reg}}$ formula $\varphi(x)$, we use $\forall x\varphi$ to abbreviate $\neg\exists x\neg\varphi(x)$.

▶ **Example 1.** The following are UNFO$^{\mathsf{reg}}$ formulas: $\forall x\big(\exists y R(x,y) \wedge \neg(R \cdot R^*)(x,x)\big)$ and $\exists y\big(R^*(x,y) \wedge S^*(x,y)\big)$.

A *structure* $\mathfrak{A}$ takes the form $(A, R_1^{\mathfrak{A}}, R_2^{\mathfrak{A}}, \dots)$ where A is a non-empty set called the *domain* and R_i is an n_i-ary relation over A if R_i is a predicate symbol of arity n_i.Whenever convenient, we use $\mathsf{dom}(\mathfrak{A})$ to refer to A. Every path expression E is interpreted as a binary relation $E^{\mathfrak{A}}$ over A: $R^{\mathfrak{A}}$ is part of $\mathfrak{A}$, $(R^-)^{\mathfrak{A}}$ is the converse of $R^{\mathfrak{A}}$, $(E_1 \cup E_2)^{\mathfrak{A}} = E_1^{\mathfrak{A}} \cup E_2^{\mathfrak{A}}$, $(E_1 \circ E_2)^{\mathfrak{A}} = E_1^{\mathfrak{A}} \circ E_2^{\mathfrak{A}}$, $(E^*)^{\mathfrak{A}}$ is the reflexive-transitive closure of $E^{\mathfrak{A}}$, and $(\varphi(x)?)^{\mathfrak{A}} = \{(a,a) \mid \mathfrak{A} \models \varphi(a)\}$. UNFO$^{\mathsf{reg}}$ formulas are then interpreted under the standard first-order semantics with path expressions being treated in the same way as binary predicates. A UNFO$^{\mathsf{reg}}$ sentence $\varphi(\mathbf{x})$ is *satisfiable* if there is a structure $\mathfrak{A}$ such that $\mathfrak{A} \models \varphi$. Such an $\mathfrak{A}$ is called a *model* of $\varphi(\mathbf{x})$.

▶ **Example 2.** Reconsider the UNFO$^{\mathsf{reg}}$ formulas from Example 1. It can be verified that the first sentence is satisfiable, but not in a finite model. Thus, in contrast to UNFO (and to propositional dynamic logic), UNFO$^{\mathsf{reg}}$ lacks the finite model property. The second sentence expresses a property that cannot be expressed in UNFO extended with fixed points, as studied in [46], which can formally be shown using UN-bisimulations, also defined in [46]. In fact, UNFO$^{\mathsf{reg}}$ and UNFO with fixed points are orthogonal in expressive power. Another related logic is ICPDL, that is, PDL extended with intersection and converse [31]. This logic, too, is orthogonal in expressive power to UNFO$^{\mathsf{reg}}$. For example, the existence of a 4-clique can be expressed as a UNFO sentence, but not in ICPDL since every satisfiable ICPDL formula is satisfiable in a structure of tree width two.

The expressive power of UNFO$^{\mathsf{reg}}$ is closely related to that of conjunctive 2-way regular path queries. A *database* D is a finite structure such that for every $a \in \mathsf{dom}(D)$, there is an $\mathbf{a} \subseteq \mathsf{dom}(D)$ and a predicate symbol P such that $a \in \mathbf{a} \in P^D$. Since a database is a syntactic object, we refer to the elements of $\mathsf{dom}(D)$ as *constants* whereas we speak about *elements* in the context of (semantic) structures. A *conjunctive 2-way regular path query (C2RPQ)* is a formula of the form $q(\mathbf{x}) = \exists\mathbf{y}\,\varphi(\mathbf{x},\mathbf{y})$ where $\varphi(\mathbf{x},\mathbf{y})$ is a conjunction of *atoms* of the form $R(\mathbf{z})$ and $E(z_1,z_2)$, R a predicate symbol and E a two-way regular path query, that is, an expression formed according to the second line of the syntax definition of UNFO$^{\mathsf{reg}}$, but allowing only formulas $\varphi(x)$ that are C2RPQs in tests. The variables $\mathbf{x}$ are the *answer variables* of $q(\mathbf{x})$ and $q(\mathbf{x})$ is *Boolean* if $\mathbf{x} = \emptyset$. A *union of C2RPQs (UC2RPQ)* is a disjunction of C2RPQs that all have the same answer variables. A *conjunctive query (CQ)* is a C2RPQ that does not use atoms of the form $E(z_1,z_2)$. The *answers* to a UC2RPQ $q(\mathbf{x})$ on a database D, denoted $\mathsf{ans}(q,D)$, are defined in the standard way, see for example [42]. Note that every UC2RPQ is a UNFO$^{\mathsf{reg}}$ formula.

▶ **Example 3.** Consider the following database about family relationships, using binary predicates Child and Spouse, and written as a set of facts.

$$\begin{aligned} D \;=\; \{\, &\mathsf{Child}(\mathsf{Nívea},\mathsf{Clara}),\ \mathsf{Child}(\mathsf{Clara},\mathsf{Blanca}),\ \mathsf{Child}(\mathsf{Blanca},\mathsf{Alba}),\\ &\mathsf{Spouse}(\mathsf{Nívea},\mathsf{Severo}),\ \mathsf{Spouse}(\mathsf{Esteban},\mathsf{Clara}) \qquad\qquad \}\end{aligned}$$

The following C2RPQ asks for all pairs (x, y) such that x is an ancestor of y in a line of only married ancestors (using the shorthand $R^+ = R \cdot R^*$).

$$q(x,y) = (m(z)? \cdot \mathsf{Child})^+(x,y) \quad \text{where} \quad m(z) = \exists z' \, (\mathsf{Spouse} \cup \mathsf{Spouse}^-)(z,z')$$

We have $\mathsf{ans}(q, D) = \{(\mathsf{Nívea}, \mathsf{Clara}), (\mathsf{Nívea}, \mathsf{Blanca}), (\mathsf{Clara}, \mathsf{Blanca})\}$.

Let $q(\mathbf{x}) = \exists \mathbf{y} \, \varphi(\mathbf{x}, \mathbf{y})$ be a C2RPQ. We use $\mathsf{var}(q)$ to denote the variables that occur in q outside of tests, that is, $\mathbf{x} \cup \mathbf{y}$. We do not distinguish between $q(\mathbf{x})$ and the set of all atoms in φ, writing e.g. $R(x, y, z) \in q(\mathbf{x})$ to mean that $R(x, y, z)$ is an atom in φ. For simplicity, we treat an atom $E(x, x)$ in a C2RPQ $q(\mathbf{x})$ where E is the test $\varphi(y)?$ as an atom of the form $\varphi(x)$; that is, w.l.o.g. we use tests not only in path expressions but also directly as atoms of a C2RPQ. A C2RPQ $q(\mathbf{x})$ can be viewed as a finite hypergraph in the expected way, that is, every atom $R(\mathbf{z})$ and $E(z_1, z_2)$ is viewed as a hyperedge. We say that $q(\mathbf{x})$ is *connected* if the Gaifman graph of this hypergraph is connected. It is interesting to observe that foundational problems concerning UC2RPQs can be phrased as (un)satisfiability problems in UNFO$^{\mathsf{reg}}$.

▶ **Example 4.**

1. The problem whether a Boolean UC2RPQ $q()$ evaluates to true on a database D (i.e., whether the empty tuple is in $\mathsf{ans}(q, D)$) corresponds to the unsatisfiability of $\varphi_D() \wedge \neg q()$ where $\varphi_D()$ is D viewed as a Boolean CQ in the obvious way.
2. The problem whether a Boolean UC2RPQ $q_1()$ is contained in a Boolean UC2RPQ $q_2()$ (defined in the usual way) corresponds to the unsatisfiability of $q_1() \wedge \neg q_2()$.

Both reductions extend to the case of non-Boolean queries by simulating answer variables using fresh unary predicates, see the proof of Lemma 6 below.

An *ontology-mediated query (OMQ)* is a triple $(\mathcal{O}, \Sigma, q)$ where $\mathcal{O}$ is a logical sentence called the *ontology*, Σ is a set of predicate symbols called the *data signature*, and q is a query. In this paper, we shall primarily be interested in the case where $\mathcal{O}$ is an UNFO$^{\mathsf{reg}}$ sentence and q is a UC2RPQ. We use (UNFO$^{\mathsf{reg}}$, UC2RPQ) to denote the set of OMQs of this form and similarly for other ontology languages and query languages. Let $Q = (\mathcal{O}, \Sigma, q)$ be from (UNFO$^{\mathsf{reg}}$, UC2RPQ) and D a database that uses only symbols from Σ. We call $\mathbf{a} \subseteq \mathsf{dom}(D)$ a *certain answer to Q on D* if $\mathbf{a} \in \mathsf{ans}(q, \mathfrak{A})$ for every structure $\mathfrak{A}$ that extends D and is a model of $\mathcal{O}$, where $\mathfrak{A}$ *extends* D if $\mathsf{dom}(D) \subseteq \mathsf{dom}(\mathfrak{A})$ and $P^D \subseteq P^{\mathfrak{A}}$ for all predicate symbols P. Note that this semantics embodies a 'standard names assumption', that is, constants in D are interpreted as themselves. The set of all certain answers to Q on D is denoted $\mathsf{cert}(Q, D)$. We say that Q is *Boolean* if q is. For a Boolean OMQ Q, we write $D \models Q$ to indicate that Q is true on D, meaning that the empty tuple is in $\mathsf{cert}(Q, D)$.

▶ **Example 5.** Consider the OMQ $Q = (\mathcal{O}, \Sigma, q')$ based on an extension of the C2RPQ q from Example 3, where $\mathcal{O}$ defines a single mother as an unmarried woman who has a child, using additional unary predicates Female and $\mathsf{SingleMother}$, and q' has an additional conjunct requiring that y is a single mother, that is:

$$\begin{aligned} \mathcal{O} &= \forall x \, \big(\mathsf{SingleMother}(x) \leftrightarrow \mathsf{Female}(x) \wedge \mathsf{Single}(x) \wedge \exists y \, \mathsf{Child}(x, y)\big) \\ q'(x,y) &= q(x,y) \wedge \mathsf{SingleMother}(y) \\ \Sigma &= \{\mathsf{Child}, \mathsf{Spouse}, \mathsf{Female}, \mathsf{Single}\} \end{aligned}$$

Note that $\mathcal{O}$ is equivalent to a UNFO$^{\mathsf{reg}}$ (even plain UNFO) formula obtained by eliminating $\leftrightarrow$ in the usual way. Let $D' = D \cup \{\mathsf{Female}(\mathsf{Blanca}), \mathsf{Single}(\mathsf{Blanca})\}$, where D is the database from Example 3. Then $\mathsf{cert}(Q, D') = \{(\mathsf{Nívea}, \mathsf{Blanca}), (\mathsf{Clara}, \mathsf{Blanca})\}$, but $\mathsf{cert}(Q, D) = \emptyset$.

OMQ evaluation in $(\text{UNFO}^{\text{reg}}, \text{UC2RPQ})$ is the problem to decide, given an OMQ Q from $(\text{UNFO}^{\text{reg}}, \text{UC2RPQ})$, a database D, and an $\mathbf{a} \subseteq \mathsf{dom}(D)$, whether $\mathbf{a} \in \mathsf{cert}(Q, D)$. This is a relevant problem since ontologies formulated in many logics used as ontology languages can be translated into an equivalent UNFO^{reg} sentence in polynomial time. In particular, this is the case for the basic description logics $\mathcal{ALC}$ and $\mathcal{ALCI}$ [19] and for their extensions with transitive closure of roles [3] and with regular expressions over roles [24]. For any of these logics $\mathcal{L}$, this of course also yields a polynomial time reduction of OMQ evaluation in $(\mathcal{L}, \text{UC2RPQ})$ to OMQ evaluation in $(\text{UNFO}^{\text{reg}}, \text{UC2RPQ})$. Even UNFO itself has occasionally been considered as an ontology language [19].

For the rather common extension of the description logic $\mathcal{ALC}$ with transitive roles [5], an equivalence preserving translation of ontologies into UNFO^{reg} sentences is not possible since UNFO^{reg} cannot enforce that a binary predicate is transitive. However, a transitive role can be simulated using the transitive closure of a binary predicate R (and never using R without transitive closure). In this way, one still obtains the desired polynomial time reduction of OMQ evaluation. The same reduction can be applied even to UNFO^{reg} extended with transitive relations. We use $\text{UNFO}^{\text{reg}}_{\mathsf{trans}}$ to denote the extension of UNFO^{reg} where sentences take the form $\varphi_{\mathsf{trans}} \wedge \varphi$ with φ_{trans} a conjunction of atoms of the form $\mathsf{trans}(R)$, R a binary predicate symbol, and φ a UNFO^{reg} sentence. An atom $\mathsf{trans}(R)$ is satisfied in a structure $\mathfrak{A}$ if $R^{\mathfrak{A}}$ is transitive.

Evaluation of Boolean OMQs in $(\text{UNFO}^{\text{reg}}_{\mathsf{trans}}, \text{UC2RPQ})$ reduces in polynomial time to satisfiability in $\text{UNFO}^{\text{reg}}_{\mathsf{trans}}$ since $D \models (\mathcal{O}, \Sigma, q)$ iff $\varphi_D() \wedge \mathcal{O} \wedge \neg q()$ is unsatisfiable. The reduction can be extended to non-Boolean queries by simulating answer variables using fresh unary predicates. Because of this observation, in the main body of the paper we concentrate on deciding satisfiability rather than OMQ evaluation.

▶ **Lemma 6.** *Lemma OMQ evaluation in* $(\text{UNFO}^{\text{reg}}_{\mathsf{trans}}, \text{UC2RPQ})$ *reduces in polynomial time to satisfiability in* UNFO^{reg}*, and so does satisfiability in* $\text{UNFO}^{\text{reg}}_{\mathsf{trans}}$*.*

Together with Theorem 14 it thus follows that UNFO can be extended with transitive relations without losing decidability or affecting the complexity of satisfiability and of OMQ evaluation. This is in contrast to the guarded fragment, where in both cases decidability can only be obtained by adopting additional syntactic restrictions. While for satisfiability it suffices to assume that transitive relations are only used in guard positions, even stronger restrictions are necessary for OMQ evaluation [35, 45, 34].

There are also other interesting reasoning problems that can be reduced to satisfiability in UNFO^{reg}. Here we consider OMQ containment, leaving out transitive roles for simplicity. Let $Q_1 = (\mathcal{O}, \Sigma_{\mathsf{full}}, q_1)$ and $Q_2 = (\mathcal{O}, \Sigma_{\mathsf{full}}, q_2)$ be OMQs from $(\text{UNFO}^{\text{reg}}, \text{UC2RPQ})$ with the same number of answer variables and where Σ_{full} is the *full* data signature, that is, the set of all predicate symbols. We say that Q_1 *is contained in* Q_2 and write $Q_1 \subseteq Q_2$ if for every database D, $\mathsf{cert}(Q_1, D) \subseteq \mathsf{cert}(Q_2, D)$. We observe that OMQ containment can also be reduced to satisfiability in polynomial time.

▶ **Lemma 7.** *OMQ containment in* $(\text{UNFO}^{\text{reg}}, \text{UC2RPQ})$ *reduces in polynomial time to satisfiability in* UNFO^{reg}*.*

There are also versions of OMQ containment that admit different ontologies in the two involved OMQs and more restricted data signatures in place of Σ_{full} [11, 13, 14, 21]. These are computationally harder and a polynomial time reduction to satisfiability cannot be expected. In fact, it follows from results in [21] that this more general form of OMQ containment is 2NExpTime-hard already when the ontologies are formulated in the description logic $\mathcal{ALCI}$,

a fragment of UNFO, and when the actual queries are CQs. Decidability remains an open problem. We remark that when the actual queries in OMQs are CQs, then OMQ containment under the full data signature can be reduced to query evaluation in a straightforward way, essentially by viewing the query from the left-hand OMQ as a database. In the presence of regular path queries, however, this does not seem to be easily possible.

We next introduce a normal form for UNFO$^{\text{reg}}$ sentences, similar but not identical to the normal form used for UNFO in [46]. For a set $\mathcal{L}$ of UNFO$^{\text{reg}}$ formulas with one free variable, a *C2RPQ extended with $\mathcal{L}$-formulas* is a C2RPQ in which all tests $\varphi(x)?$ in atoms $E(z_1, z_2)$ have been replaced with tests $\psi(x)?$, $\psi(x)$ a formula from $\mathcal{L}$. The set of *normal* UNFO$^{\text{reg}}$ *formulas* is the smallest set of formulas such that

1. every connected C2RPQ with exactly one free variable, extended with normal UNFO$^{\text{reg}}$ formulas, is a normal UNFO$^{\text{reg}}$ formula;
2. if $\varphi(x)$ and $\psi(x)$ are normal UNFO$^{\text{reg}}$ formulas, then $\neg\varphi(x)$, $\varphi(x) \vee \psi(x)$, and $\exists x\, \varphi(x)$ are normal UNFO$^{\text{reg}}$ formulas.

Observe that Item 1 serves as an induction start since every connected C2RPQ without tests (and with one free variable) is a normal UNFO$^{\text{reg}}$ formula. Note that normal formulas are closed under conjunction in the sense that the conjunction of normal formulas $\varphi_1(x)$ and $\varphi_2(x)$ is a C2RPQ extended with normal UNFO$^{\text{reg}}$ formulas and thus a normal formula. Thus, unary disjunction could be eliminated, but for our purposes it is more convenient to keep it. We note in passing that using this normal form, it is easy to observe that UNFO$^{\text{reg}}$ has the same expressive power as C2RPQs that admit both tests and negated tests.

The *width* of a normal UNFO$^{\text{reg}}$ formula is the maximal number of variables in a C2RPQ that occurs in it (not counting the variables that occur in the C2RPQ only inside tests). The *atom width* is defined analogously, but referring to the number of atoms instead of the number of variables. In the context of normal UNFO$^{\text{reg}}$ formulas, for brevity we speak of C2RPQs when meaning C2RPQs extended with normal UNFO$^{\text{reg}}$ formulas. The *size* of a UNFO$^{\text{reg}}$ formula is the number of symbols needed to write it, with variable symbols and predicate symbols being counted as a single symbol.

▶ **Lemma 8.** *Every* UNFO$^{\text{reg}}$ *sentence φ can be transformed into an equivalent normal* UNFO$^{\text{reg}}$ *sentence φ' in single exponential time. Moreover, the width and the atom width of φ' are at most polynomial in the size of φ and the path expressions that occur in φ' are exactly those in φ.*

In the following sections, we replace atoms $E(z_1, z_2)$ in the C2RPQs that occur in a normal UNFO$^{\text{reg}}$ formula with atoms of the form $\mathcal{A}(z_1, z_2)$ where $\mathcal{A}$ is a *nondeterministic automaton on finite words (NFA)* over a suitable alphabet; we call such atoms *NFA atoms*. Formally, an NFA is a tuple $(Q, \Sigma, \Delta, q_0, F)$ where Q is a finite set of states, Σ a finite alphabet, $\Delta \subseteq Q \times \Sigma \times Q$ a transition relation, $q_0 \in Q$ an initial state and $F \subseteq Q$ a set of final states. When deciding the satisfiability of a UNFO$^{\text{reg}}$ sentence φ_0, we will generally take Σ to be $\{R, R^- \mid R \text{ a binary predicate in } \varphi_0\} \cup \{\varphi(x)? \mid \varphi(x)? \text{ a test in } \varphi_0\}$. Clearly, all path expressions in φ_0 are regular expressions over this alphabet. Since every regular expression can be converted into an equivalent NFA in polynomial time, we can thus w.l.o.g. assume the NFA-based presentation. Let $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$ be an NFA. Then we use $\mathcal{A}[F/F']$ to denote the NFA obtained from $\mathcal{A}$ by replacing F with $F' \subseteq Q$ and $\mathcal{A}[q_0/q]$ for the NFA obtained from $\mathcal{A}$ by replacing q_0 with $q \in Q$. For a structure $\mathfrak{A}$, an NFA $\mathcal{A}$, and $a, b \in A$, we write $\mathfrak{A} \models \mathcal{A}(a, b)$ if there are $a_1, \dots, a_n \in A$ and a word $w \in L(\mathcal{A})$ of length $n-1$ such that $a = a_1$, $a_n = b$, $(a_i, a_{i+1}) \in R^{\mathfrak{A}}$ if the ith symbol in w is R, $(a_{i+1}, a_i) \in R^{\mathfrak{A}}$ if the ith symbol in w is R^-, and $a_i = a_{i+1}$ and $\mathfrak{A} \models \varphi(a_i)$ if the ith symbol in w is $\varphi(x)?$. This gives a

semantics to NFA atoms. The *size* of a normal UNFO$^{\mathsf{reg}}$ formula with NFA atoms is defined in the same way as the size of a UNFO formula, where every NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$ contributes the cardinality of Q plus the cardinality of Δ plus the cardinality of F.

3 Tree-like Structures and Witness Trees

We give a characterization of satisfiability in UNFO$^{\mathsf{reg}}$ that is tailored towards implementation by tree automata. In particular, we show that every satisfiable UNFO$^{\mathsf{reg}}$ formula φ has a model whose treewidth is bounded by the width of φ, introduce a representation of such models in terms of labeled trees, and characterize the satisfaction of C2RPQs in models represented in this way in terms of tree-shaped witnesses. To simplify the technical development, in this section and the subsequent one we disallow predicates of arity zero. Note that an atom $P()$ can be simulated by the formula $\exists x\, P(x)$, so this assumption is w.l.o.g. We work with normal UNFO$^{\mathsf{reg}}$ sentences throughout the section.

A *(directed) tree* is a prefix-closed subset $T \subseteq (\mathbb{N} \setminus \{0\})^*$. A node $w \in T$ is a *successor* of $v \in T$ and v is a *predecessor* of w if $w = v \cdot i$ for some $i \in \mathbb{N}$. Moreover, w is a *neighbor* of v if it is a successor or predecessor of v. A *tree-like structure* is a pair (T, bag) where T is a tree and bag a function that assigns to every $w \in T$ a finite structure $\mathsf{bag}(w)$ such that

$$\text{the set of nodes } \{w \in T \mid a \in \mathsf{dom}(w)\} \text{ is connected in } T, \text{ for each } a \in \bigcup_{w \in T} \mathsf{dom}(w)$$

where, here and in the remainder of the paper, $\mathsf{dom}(w)$ is a shorthand for $\mathsf{dom}(\mathsf{bag}(w))$. The *width* of (T, bag) is the maximum domain size of structures that occur in the range of bag. Its *outdegree* is the outdegree of T. A tree-like structure (T, bag) defines the associated structure $\mathfrak{A}_{(T,\mathsf{bag})}$ which is the (non-disjoint) union of all structures $\mathsf{bag}(w)$, $w \in T$. We use $\mathsf{dom}(T, \mathsf{bag})$ as a shorthand for $\mathsf{dom}(\mathfrak{A}_{(T,\mathsf{bag})})$. As witnessed by its representation (T, bag), the treewidth of the structure $\mathfrak{A}_{(T,\mathsf{bag})}$ is bounded by the maximum cardinality of $\mathsf{dom}(\mathsf{bag}(w))$, $w \in T$. We will show that every satisfiable UNFO$^{\mathsf{reg}}$ sentence φ_0 is satisfiable in a tree-like structure whose width is bounded by the width of φ_0. In UNFO, it suffices to consider structures of this form in which bags overlap in at most one element; this is not the case in UNFO$^{\mathsf{reg}}$.

Let φ_0 be a normal UNFO$^{\mathsf{reg}}$ sentence. We use $\mathsf{sub}(\varphi_0)$ to denote the subformulas of φ_0 with at most one free variable, and where the free variable is renamed to x. Then $\mathsf{cl}(\varphi_0)$ denotes the smallest set of normal UNFO$^{\mathsf{reg}}$ formulas that contains $\mathsf{sub}(\varphi_0)$ and is closed under single negation. A *1-type for* φ_0 is a subset $t \subseteq \mathsf{cl}(\varphi_0)$ that satisfies the following conditions:

1. $\varphi \in t$ iff $\neg\varphi \notin t$ for all $\neg\varphi \in \mathsf{cl}(\varphi_0)$;
2. $\varphi \vee \psi \in t$ iff $\varphi \in t$ or $\psi \in t$ for all $\varphi \vee \psi \in \mathsf{cl}(\varphi_0)$.

We use $\mathsf{TP}(\varphi_0)$ to denote the set of all 1-types for φ_0.

A *type decorated tree-like structure for* φ_0 is a triple (T, bag, τ) with (T, bag) a tree-like structure such that only predicates from φ_0 occur in the range of bag and $\tau : \mathsf{dom}(T, \mathsf{bag}) \to \mathsf{TP}(\varphi_0)$. Let (T, bag, τ) be such a structure, $\mathcal{A}$ an NFA, and $a, b \in \mathsf{dom}(T, \mathsf{bag})$. We write $\mathfrak{A}_{(T,\mathsf{bag})}, \tau \models \mathcal{A}(a, b)$ if $\mathfrak{A}_{(T,\mathsf{bag})} \models \mathcal{A}(a, b)$ with the semantics of tests reinterpreted: instead of demanding that $\mathfrak{A} \models \varphi(a')$ for a test $\varphi_0(x)$? to hold at an element a', we now require that $\varphi \in \tau(a')$. Let $\varphi(x) = \exists \mathbf{y}\, \psi(x, \mathbf{y})$ be a C2RPQ and $a \in \mathsf{dom}(T, \mathsf{bag})$. A *homomorphism from* $\varphi(x)$ *to* (T, bag, τ) is a function $h : \{x\} \cup \mathbf{y} \to \mathsf{dom}(T, \mathsf{bag})$ such that the following conditions are satisfied:

- $h(\mathbf{x}) \in R^{\mathfrak{A}_{(T,\mathsf{bag})}}$ for each $R(\mathbf{x}) \in \varphi(x)$;
- $\mathfrak{A}_{(T,\mathsf{bag})}, \tau \models \mathcal{A}(h(y), h(z))$ for each $\mathcal{A}(y, z) \in \varphi(x)$.

A type decorated tree-like structure (T, bag, τ) for φ_0 is *proper* if:

1. for all $\exists x\, \varphi(x) \in \mathsf{cl}(\varphi_0)$, $\exists x\, \varphi(x) \in \tau(a)$ iff there is a $b \in \mathsf{dom}(T, \mathsf{bag})$ with $\varphi(x) \in \tau(b)$;
2. for all C2RPQs $\varphi(x) \in \mathsf{cl}(\varphi_0)$, $\varphi(x) \in \tau(a)$ iff there is a homomorphism h from $\varphi(x)$ to (T, bag, τ) such that $h(x) = a$.

The following lemma establishes proper type decorated tree-like structures for φ_0 as witnesses for the satisfiability of φ_0. The proof of the 'only if' direction is via an unraveling procedure that constructs a type decorated tree-like structure in a top-down manner, introducing fresh bags to satisfy C2RPQs and to implement a step-by-step chase of paths that witness satisfaction of NFA atoms in C2RPQs.

▶ **Lemma 9.** *A normal* $\mathrm{UNFO}^{\mathsf{reg}}$ *sentence* φ_0 *of size* n *and width* m *is satisfiable iff there is a proper type decorated tree-like structure* (T, bag, τ) *for* φ_0 *of width at most* m *and outdegree at most* $n^2 + n$ *such that* $\varphi_0 \in \tau(a)$ *for some* a*.*

As the next step, we take a closer look at Point 2 of properness, that is, we characterize carefully the existence of a homomorphism h from $\varphi(x)$ to (T, bag, τ) such that $h(x) = a$ in a way that is tailored towards implementation by tree automata. This gives rise to the notion of a witness tree below. We start with introducing the notions of subdivisions and splittings which shall help us to take care of the fact that the homomorphic image of a query $q(\mathbf{x})$ may be spread over several bags of a tree-like structure, and in fact this might even be the case for a single NFA atom.

An *instantiated C2RPQ* is a C2RPQ in which all free variables have been replaced with constants. We write $\varphi(\mathbf{a})$ to indicate that the constants in the instantiated C2RPQ are exactly $\mathbf{a}$. When working with instantiated C2RPQs, we drop existential quantifiers, assuming that all variables are implicitly existentially quantified. For brevity, we often omit the word 'instantiated' and only speak of C2RPQs. We speak of *terms* to mean both variables and constants, and we denote terms with t.

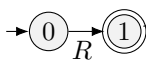
Let $\varphi(\mathbf{a})$ be a connected C2RPQ, Δ be a domain, and $s \geq 1$. A (Δ, s)-*subdivision of* of an atom $\mathcal{A}(t, t') \in \varphi(\mathbf{a})$ is a set of atoms

$$\mathcal{A}[F/\{q_1\}](t, b_1), \mathcal{A}[q_0/q_1, F/\{q_2\}](b_1, b_2), \ldots, \mathcal{A}[q_0/q_{k-1}, F/\{q_k\}](b_{k-1}, b_k), \mathcal{A}[q_0/q_k](b_k, t')$$

where $q_1, \ldots, q_k$ are states of $\mathcal{A}$, $k \leq s$, and $b_1, \ldots, b_k$ are constants from Δ. A C2RPQ $\psi(\mathbf{a}')$ is a (Δ, s)-*subdivision* of $\varphi(\mathbf{a})$ if it is obtained from $\varphi(\mathbf{a})$ by replacing zero or more NFA atoms with (Δ, s)-subdivisions. Let $\psi(\mathbf{a}')$ be a (Δ, s)-subdivision of $\varphi(\mathbf{a})$. A *splitting* of $\psi(\mathbf{a}')$ is a sequence $\psi_0(\mathbf{a}_0), \ldots, \psi_\ell(\mathbf{a}_\ell)$, $\ell \geq 0$, of C2RPQs that is a partition of $\psi(\mathbf{a}')$ (viewed as a set of atoms) where we also allow the special case that $\psi_0(\mathbf{a}_0)$ is empty (and thus $\psi_1(\mathbf{a}_1), \ldots, \psi_\ell(\mathbf{a}_\ell)$ is the actual partition). We require that the following conditions are satisfied:

1. $\psi_1(\mathbf{a}_1), \ldots, \psi_\ell(\mathbf{a}_\ell)$ are connected;
2. $\mathsf{var}(\psi_i(\mathbf{a}_i)) \cap \mathsf{var}(\psi_j(\mathbf{a}_j)) \subseteq \mathsf{var}(\psi_0(\mathbf{a}_0))$ for $1 \leq i < j \leq \ell$;
3. each of $\psi_1(\mathbf{a}_1), \ldots, \psi_\ell(\mathbf{a}_\ell)$ contains at most one atom from each subdivision of an atom in $\varphi(\mathbf{a})$.

Intuitively, the $\vartheta_0(\mathbf{a})$ component of a splitting is the part of $\psi(\mathbf{a}')$ that maps into a bag that we are currently focussing on while the other components are pushed to neighboring bags.

▶ **Example 10.** Consider $q(a) = \{\mathcal{A}(a, y), T(a, z), Q(a, y, z)\}$ with $\mathcal{A} =$ 

Let $\Delta = \{a, b, c\}$ and $\mathcal{A}_{ij} = \mathcal{A}[0/i, F/j]$. An example for a $(\Delta, 2)$-subdivision of $\mathcal{A}(a, y)$ is $\{\mathcal{A}_{01}(a, b), \mathcal{A}_{11}(b, b), \mathcal{A}_{11}(b, y)\}$, which yields the following $(\Delta, 2)$-subdivision of $\varphi(a)$:

$\psi(a,b) = \{\mathcal{A}_{01}(a,b), \mathcal{A}_{11}(b,b), \mathcal{A}_{11}(b,y), T(a,z), Q(a,y,z)\}$. $\psi(a,b)$ admits a splitting into ψ_0, ψ_1 as follows: $\psi_0(a,b) = \{\mathcal{A}_{01}(a,b), \mathcal{A}_{11}(b,b), T(a,z)\}$ and $\psi_1(a,b) = \{\mathcal{A}_{11}(b,y), Q(y,z,a)\}$.

The *query closure* $\mathsf{qcl}(\varphi_0, \Delta, s)$ is defined as the smallest set such that the following conditions are satisfied:

- if $\varphi(x) \in \mathsf{cl}(\varphi_0)$ is a C2RPQ and $a \in \Delta$, then $\varphi(a) \in \mathsf{qcl}(\varphi_0, \Delta, s)$;
- if $\varphi(\mathbf{a}) \in \mathsf{qcl}(\varphi_0, \Delta, s)$, $\psi(\mathbf{a}')$ is a (Δ, s)-subdivision of $\varphi(\mathbf{a})$, $\psi_0(\mathbf{a}_0), \dots, \psi_\ell(\mathbf{a}_\ell)$ is a splitting of $\psi(\mathbf{a}')$, $1 \leq i \leq \ell$, and $\psi_i'(\mathbf{a}_i')$ is obtained from $\psi_i(\mathbf{a}_i)$ by consistently replacing zero or more variables with constants from Δ, then $\psi_i'(\mathbf{a}_i') \in \mathsf{qcl}(\varphi_0, \Delta, s)$.

▶ **Lemma 11.** *The cardinality of* $\mathsf{qcl}(\varphi_0, \Delta, s)$ *is bounded by* $p \cdot (a^2 d^m)^{m'}$*, where* p *is the number of C2RPQs in* φ_0*,* a *the maximal number of states in an NFA in* φ_0*,* d *the cardinality of* Δ*,* m *the width of* φ_0*, and* m' *the atom width of* φ_0*.*

We are almost ready to define witness trees. The following notion of a homomorphism is more local than the ones used so far as it only concerns a single bag rather than the entire tree-like structure. Let (T, bag, τ) be a type decorated tree-like structure, $w \in T$, $\mathcal{A}$ an NFA, and $a, b \in \mathsf{dom}(w)$. We write $\mathsf{bag}(w), \tau \models \mathcal{A}(a,b)$ if $\mathsf{bag}(w) \models \mathcal{A}(a,b)$ with the semantics of tests reinterpreted: instead of demanding $\mathsf{bag}(w) \models \varphi(a')$ for a test $\varphi(x)$? to hold at an element a', we now require that $\varphi(x) \in \tau(a')$. Let $\varphi(\mathbf{a})$ be a C2RPQ. A *homomorphism from* $\varphi(\mathbf{a})$ *to* $\mathsf{bag}(w)$ *given* τ is a function $h : \mathbf{a} \cup \mathsf{var}(\varphi) \to \mathsf{dom}(w)$ such that the following conditions are satisfied:

- $h(a) = a$ for each $a \in \mathbf{a}$;
- $h(\mathbf{t}) \in R^{\mathsf{bag}(w)}$ for each $R(\mathbf{t}) \in \varphi(\mathbf{a})$;
- $\mathsf{bag}(w), \tau \models \mathcal{A}(h(t), h(t'))$ for each $\mathcal{A}(t,t') \in \varphi(a)$.

Let n be the size of φ_0, $a \in \mathsf{dom}(T, \mathsf{bag})$, and $\varphi(x) \in \mathsf{cl}(\varphi_0)$ a C2RPQ. A *witness tree for* $\varphi(a)$ *in* (T, bag, τ) is a finite labeled tree (W, σ) with $\sigma : W \to T \times \mathsf{qcl}(\varphi_0, \mathsf{dom}(T, \mathsf{bag}), n^2)$ such that the root is labeled with $\sigma(\varepsilon) = (w, \varphi(a))$ for some $w \in T$ with $a \in \mathsf{dom}(w)$ and the following conditions are satisfied for all $u \in W$:

$(*)$ if $\sigma(u) = (w, \psi(\mathbf{a}))$, then there is a $(\mathsf{dom}(w), n^2)$-subdivision $\vartheta'(\mathbf{a})$ of $\psi(\mathbf{a})$, a splitting $\vartheta_0(\mathbf{a}_0), \dots, \vartheta_\ell(\mathbf{a}_\ell)$ of $\vartheta'(\mathbf{a})$, a homomorphism h from $\vartheta_0(\mathbf{a}_0)$ to $\mathsf{bag}(w)$ given τ, and successors $u_1, \dots, u_\ell$ of u such that $\sigma(u_i) = (w_i, \vartheta_i'(\mathbf{a}_i'))$ for $1 \leq i \leq \ell$, where each w_i is a neighbor of w in T with $\mathbf{a}_i' \subseteq \mathsf{dom}(w_i)$ and $\vartheta_i'(\mathbf{a}_i')$ is obtained from $\vartheta_i(\mathbf{a}_i)$ by replacing each variable x in the domain of h with the constant $h(x)$.

Informally, a witness tree decomposes a homomorphism h from $\varphi(x)$ to $\mathfrak{A}_{T,\mathsf{bag}}$ into local 'chunks', each of which concerns only a single bag. In particular, the splitting $\vartheta_0(\mathbf{a}_0), \dots, \vartheta_k(\mathbf{a}_k)$ in $(*)$ breaks the current C2RPQ down into components that are satisfied in different parts of the tree-like structure.We need to first subdivide since satisfaction of NFA atoms is witnessed by an entire path, and this path can pass through the current node several times. Fortunately, the number of points introduced in a subdivision can be bounded: we can w.l.o.g. choose a shortest path and such a path can pass through w at most once for each element in $\mathsf{dom}(w)$ and each state of the automaton $\mathcal{A}$, thus we need at most n^2 points in subdivisions.

▶ **Lemma 12.** *Let* (T, bag, τ) *be a type decorated tree-like structure,* $\varphi(x) \in \mathsf{cl}(\varphi_0)$ *a C2RPQ, and* $a \in \mathsf{dom}(T, \mathsf{bag})$*. Then there is a homomorphism* h *from* $\varphi(x)$ *to* (T, bag, τ) *with* $h(x) = a$ *iff there is a witness tree for* $\varphi(a)$ *in* (T, bag, τ)*.*

4 Automata-Based Decision Procedure

We now reduce satisfiability of UNFO$^{\mathsf{reg}}$ sentences to the nonemptiness problem of two-way alternating tree automata. We start with recalling this automata model and discuss the encoding of tree-like structures as an input to automata.

Two-way alternating tree automata. A tree is *k-ary* if each node has *exactly* k successors. As a convention, we set $w \cdot 0 = w$ and $wc \cdot (-1) = w$, leave $\varepsilon \cdot (-1)$ undefined, and for any $k \in \mathbb{N}$, set $[k] = \{-1, 0, \dots, k\}$. Let Σ be a finite alphabet. A *Σ-labeled tree* is a pair (T, L) with T a tree and $L : T \to \Sigma$ a node labeling function.

An *alternating 2-way tree automaton (2ATA)* over Σ-labeled k-ary trees is a tuple $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ where Q is a finite set of *states*, $q_0 \in Q$ is an *initial state*, δ is the *transition function*, and F is the *(parity) acceptance condition*, that is, a finite sequence $G_1, \dots, G_k$ with $G_1 \subseteq G_2 \subseteq \dots \subseteq G_k = Q$. The transition function maps a state q and an input letter $a \in \Sigma$ to a positive Boolean formula over the constants true and false, and variables from $[k] \times Q$. The semantics is given in terms of runs in the appendix of the long version. As usual, $L(\mathcal{A})$ denotes the set of trees accepted by $\mathcal{A}$. The *nonemptiness problem* for 2ATAs is the problem to decide, given a 2ATA $\mathcal{A}$, whether $L(\mathcal{A})$ is nonempty. It can be solved in time single exponential in the number of states and the number of sets in the parity condition, and linear in the size of the transition function [47].

Encoding of tree-like structures. Let φ_0 be a normal UNFO$^{\mathsf{reg}}$ sentence whose satisfiability we want to decide. By Lemma 9, this corresponds to deciding the existence of a proper type decorated tree-like structure for φ_0 (of certain dimensions) and thus our aim is to build a 2ATA $\mathcal{A}$ such that $L(\mathcal{A}) \neq \emptyset$ if and only if there is such a structure. 2ATAs cannot run directly on tree-like structures because the labeling of the underlying trees is not finite: we have already shown that UNFO$^{\mathsf{reg}}$ does not have the finite model property and thus it might be necessary that infinitely many elements occur in the bags. We therefor use an appropriate encoding that 'reuses' element names so that we can make do with finitely many element names overall, similar to what has been done, for example, in [36, 1].

Let $R_1, \dots, R_\ell$ be the predicate symbols that occur in φ_0 and let m be the width of φ_0. Fix a finite set Δ with $2m$ elements and define Σ to be the set of all pairs (bag, τ) such that $\mathsf{bag} = (A, R_1^{\mathsf{bag}}, \dots, R_\ell^{\mathsf{bag}})$ is a structure that satisfies $A \subseteq \Delta$ and $|A| \leq m$, and $\tau : A \to \mathsf{TP}(\varphi_0)$ is a map that assigns a 1-type to every element in bag.

Let (T, L) be a Σ-labeled tree. For convenience, we use bag_w to refer to the first component of $L(w)$ and τ_w to refer to the second component, that is, $L(w) = (\mathsf{bag}_w, \tau_w)$. Moreover, dom_w is shorthand for $\mathsf{dom}(\mathsf{bag}_w)$. For an element $d \in \Delta$, we say that $v, w \in T$ are *d-equivalent* if $d \in \mathsf{dom}_u$ for all u on the unique shortest path from v to w. Informally, this means that d represents the same element in bag_v and in bag_w. In case that $d \in \mathsf{dom}_w$, we use $[w]_d$ to denote the set of all v that are d-equivalent to w. We say that (T, L) is *type consistent* if, for all $d \in \Delta$ and all d-equivalent $v, w \in T$, $\tau_v(d) = \tau_w(d)$. Each type consistent (T, L) *represents* a type decorated tree-like structure $(T, \mathsf{bag}', \tau')$ of width at most m as follows. The domain of $\mathfrak{A}_{(T', \mathsf{bag}')}$ is the set of all equivalence classes $[w]_d$ with $w \in T$ and $d \in \mathsf{dom}_w$. The function τ' maps each domain element $[w]_d$ to $\tau_w(d)$, which is well-defined since (T, L) is type consistent. Finally, for every $w \in T$, the structure $\mathsf{bag}'(w) = (A(w), R_1^{\mathsf{bag}(w)}, \dots, R_\ell^{\mathsf{bag}(w)})$ is defined by:

$$\begin{aligned} A(w) &= \{[w]_d \mid d \in \mathsf{dom}_w\}, \\ R_i^{\mathsf{bag}'(w)} &= \{([w]_{d_1}, \dots, [w]_{d_j}) \mid (d_1, \dots, d_j) \in R_i^{\mathsf{bag}_w}\} \quad \text{for } 1 \leq i \leq \ell. \end{aligned}$$

Conversely, for every type decorated tree-like structure (T, bag, τ) of width m, there is a Σ-labeled tree (T, L) that represents a type decorated tree-like structure $(T, \mathsf{bag}', \tau')$ such that there is an isomorphism π between $\mathfrak{A}_{(T,\mathsf{bag})}$ and $\mathfrak{A}_{(T,\mathsf{bag}')}$ that satisfies $\tau(d) = \tau'(\pi(d))$, for all $d \in \mathsf{dom}(T, \mathsf{bag})$. In fact, since Δ is of size $2m$, it is possible to select a mapping $\pi : \mathsf{dom}(T, \mathsf{bag}) \to \Delta$ such that for each $w \in T \setminus \{\varepsilon\}$ and each $d \in \mathsf{dom}(w) \setminus \mathsf{dom}(w \cdot -1)$, we have $\pi(d) \notin \{\pi(e) \mid e \in \mathsf{dom}(w \cdot -1)\}$. Define the Σ-labeled tree (T, L) by setting, for all $w \in T$, bag_w to the image of $\mathsf{bag}(w)$ under π and τ_w to the map defined by $\tau_w(h(d)) = \tau(d)$, for all $d \in \mathsf{dom}_w$. Clearly, π satisfies the desired properties.

The notion of a witness tree carries over straightforwardly from type decorated tree-like structures to type consistent Σ-labeled trees. In fact, one only needs to replace τ with τ_w in Condition $(*)$. Then, there is a witness tree for $\varphi(a)$ in a type consistent (T, L) iff there is a witness tree for $\varphi(a)$ in the type decorated tree-like structure $(T, \mathsf{bag}', \tau')$ represented by (T, L). The notion of properness also carries over straightforwardly. For easier reference, we spell it out explicitly below, and also replace the homomorphisms from the original formulation by witness trees as suggested by Lemma 12. A type consistent Σ-labeled tree (T, L) is *proper* if for all $w \in T$ and $a \in \mathsf{dom}_w$,

1′. for all $\exists x\, \varphi(x) \in \mathsf{cl}(\varphi_0)$, $\exists x\, \varphi(x) \in \tau_w(a)$ iff there is a $v \in T$, $b \in \mathsf{dom}_v$ with $\varphi(x) \in \tau_v(b)$;
2′. for all C2RPQs $\varphi(x) \in \mathsf{cl}(\varphi_0)$, $\varphi(x) \in \tau_w(a)$ iff there is a witness tree for $\varphi(a)$ in (T, L).

It is straightforward to verify that (T, L) is proper iff the type decorated tree-like structure $(T, \mathsf{bag}', \tau')$ represented by (T, L) is proper. Thus, our aim is to build a 2ATA $\mathcal{A}$ that accepts exactly the proper type consistent Σ-labeled trees (T, L) such that $\varphi_0 \in \tau_w(a)$ for some $w \in T$ and $a \in \mathsf{dom}_w$.

Automata construction. Let n be the size of φ_0, $k = n^2 + n$ the bound on the outdegree from Lemma 9, and assume from now on that the automata run over k-ary Σ-labeled trees. It is straightforward to construct a 2ATA $\mathcal{A}_0$ that accepts (T, L) iff it is type consistent and satisfies Condition 1′ of properness and the condition that $\varphi_0 \in \tau_w(a)$ for some $w \in T$ and $a \in \mathsf{dom}_w$. The number of states of the automaton is linear in the size of φ_0; details are omitted. We next show how to construct a 2ATA $\mathcal{A}_1 = (Q, \Sigma, q_0, \delta, F)$ that accepts a type consistent (T, L) iff Condition 2′ is satisfied. The automaton uses the set of states

$$Q = \{q_0\} \cup \{\varphi(\mathbf{a}), \overline{\varphi}(\mathbf{a}) \mid \varphi(\mathbf{a}) \in \mathsf{qcl}(\varphi_0, \Delta, n^2)\}.$$

where states of the form $\varphi(\mathbf{a})$ are used to verify the 'only if' part of Condition 2′ while states of the form $\overline{\varphi}(\mathbf{a})$ are used to verify the contrapositive of the 'if' part, that is, whenever a C2RPQ $\varphi(x) \in \mathsf{cl}(\varphi_0)$ is not in $\tau_w(a)$, then there is no witness tree for $\varphi(a)$ in (T, L).

Starting from the initial state, $\mathcal{A}_1$ loops over all nodes and domain elements using the following transitions, for all $(\mathsf{bag}, \tau) \in \Sigma$:

$$\delta(q_0, (\mathsf{bag}, \tau)) = \bigwedge_{1 \le i \le k} (i, q_0) \wedge \bigwedge_{a \in \mathsf{dom}(\mathsf{bag})} \Big(\bigwedge_{\substack{\varphi(x) \in \tau(a), \\ \varphi(x) \text{ a C2RPQ}}} \varphi(a) \wedge \bigwedge_{\substack{\neg\varphi(x) \in \tau(a), \\ \varphi(x) \text{ a C2RPQ}}} \overline{\varphi}(a) \Big)$$

We next give transitions for states of the form $\varphi(\mathbf{a}) \in \mathsf{qcl}(\varphi_0, \Delta, n^2)$. Informally, if the automaton visits a node w in state $\varphi(\mathbf{a})$, then this is an obligation to show that there is a witness tree whose root is labeled with $(w, \varphi(\mathbf{a}))$. In particular, the automaton has to demonstrate that there are suitable successors for the root of the witness tree, implementing Condition $(*)$. For a more concise definition of the transitions, we first establish a suitable notation. Let $\varphi(\mathbf{a}), \vartheta_1(\mathbf{a}_1), \dots, \vartheta_\ell(\mathbf{a}_\ell) \in \mathsf{qcl}(\varphi_0, \Delta, n^2)$ and $(\mathsf{bag}, \tau) \in \Sigma$. We write $\varphi(\mathbf{a}) \to_{(\mathsf{bag},\tau)} \vartheta_1(\mathbf{a}_1), \dots, \vartheta_\ell(\mathbf{a}_\ell)$ if there is a (Δ, n^2)-subdivision $\vartheta(\mathbf{a}')$ of $\varphi(\mathbf{a})$, a splitting

$\vartheta'_0(\mathbf{a}'_0), \dots, \vartheta'_\ell(\mathbf{a}'_\ell)$ of $\vartheta(\mathbf{a}')$, a homomorphism h from $\vartheta'_0(\mathbf{a}'_0)$ to bag given τ, and $\vartheta_i(\mathbf{a}_i)$ is obtained from $\vartheta'_i(\mathbf{a}'_i)$ by replacing each variable x in the domain of h with the constant $h(x)$; please note that this is an essential part of Condition $(*)$. Then, we include for each $\varphi(\mathbf{a}) \in \mathsf{qcl}(\varphi_0, \Delta, n^2)$ and each $(\mathsf{bag}, \tau) \in \Sigma$ the transition

$$\delta(\varphi(\mathbf{a}), (\mathsf{bag}, \tau)) = \bigvee_{\varphi(\mathbf{a}) \to_{(\mathsf{bag},\tau)} \vartheta_1(\mathbf{a}_1), \dots, \vartheta_\ell(\mathbf{a}_\ell)} \bigwedge_{1 \leq i \leq \ell} \bigvee_{j \in [k] \setminus \{0\}} (j, \vartheta_i(\mathbf{a}_i))$$

if $\mathbf{a} \subseteq \mathsf{dom}(\mathsf{bag})$ and set $\delta(\varphi(\mathbf{a}), (\mathsf{bag}, \tau)) = \mathsf{false}$ otherwise. States of the form $\overline{\varphi}(\mathbf{a})$ are treated dually, that is, using the transitions

$$\delta(\overline{\varphi}(\mathbf{a}), (\mathsf{bag}, \tau)) = \bigwedge_{\varphi(\mathbf{a}) \to_{(\mathsf{bag},\tau)} \vartheta_1(\mathbf{a}_1), \dots, \vartheta_\ell(\mathbf{a}_\ell)} \bigvee_{1 \leq i \leq \ell} \bigwedge_{j \in [k] \setminus \{0\}} (j, \overline{\vartheta}_i(\mathbf{a}_i))$$

if $\mathbf{a} \subseteq \mathsf{dom}(\mathsf{bag})$ and setting $\delta(\overline{\varphi}(\mathbf{a}), (\mathsf{bag}, \tau)) = \mathsf{true}$ otherwise.

To ensure that the witness trees constructed by the states of the form $\varphi(\mathbf{a})$ are finite, we use the parity condition $F = G_1, G_2$ with $G_1 = \mathsf{qcl}(\varphi_0, \Delta, n^2)$ and $G_2 = Q$. From an accepting run of $\mathcal{A}_1$ on an input tree (T, L), one can extract the witness trees that are required to show that the 'only if' direction of Condition 2′ is satisfied. Moreover, the run demonstrates that the witness trees forbidden by the 'if' direction do not exist. We thus obtain the following.

▶ **Lemma 13.** *The* UNFO$^{\mathsf{reg}}$ *sentence* φ_0 *is satisfiable iff* $L(\mathcal{A}_0) \cap L(\mathcal{A}_1)$ *is not empty.*

Putting together Lemmas 8, 11, and 13, it follows that satisfiability in UNFO$^{\mathsf{reg}}$ is in 2ExpTime. The corresponding lower bound is inherited from UNFO [46].

▶ **Theorem 14.** *In* UNFO$^{\mathsf{reg}}$*, satisfiability is* 2ExpTime*-complete.*

5 OMQ Evaluation and Containment

We study the complexity of OMQ evaluation and OMQ containment in (UNFO$^{\mathsf{reg}}$, UC2RPQ). Recall that the complexity of OMQ evaluation can be measured in different ways. In *combined complexity*, both the OMQ and the database on which it is evaluated are considered to be an input. In *data complexity*, the OMQ is fixed and the database is the only input. We first state our main result regarding the combined complexity of OMQ evaluation and the complexity of OMQ containment.

▶ **Theorem 15.** *In* (UNFO$^{\mathsf{reg}}$, UC2RPQ)*,*

1. *OMQ evaluation is* 2ExpTime*-complete in combined complexity and*
2. *OMQ containment is* 2ExpTime*-complete.*

The upper bounds in Theorem 15 are a consequence of Lemmas 6 and 7 and Theorem 14. The lower bounds hold already when predicates are at most binary. For Point 1 this follows from the fact that OMQ evaluation is 2ExpTime-hard even for OMQs from the class $(\mathcal{ALCI}, \mathrm{CQ})$ where the ontology is formulated in the description logic $\mathcal{ALCI}$, a fragment of UNFO with only unary and binary predicates, and the actual query is a CQ [39]. The same is true for Point 2 since in $(\mathcal{ALCI}, \mathrm{CQ})$, OMQ evaluation can be reduced in polynomial time to OMQ containment in a straightforward way.

We next study the data complexity of (UNFO$^{\mathsf{reg}}$, UC2RPQ). A coNP lower bound is again inherited from (rather small) fragments of (UNFO$^{\mathsf{reg}}$, C2RPQ) [38, 23]. We give a coNP upper bound, thus establishing the following.

▶ **Theorem 16.** *OMQ evaluation in* (UNFO$^{\mathsf{reg}}$, UC2RPQ) *is* coNP*-complete in data complexity.*

Instead of directly considering OMQ evaluation, we work with a problem that we call database satisfiability. A database D is *satisfiable* with an UNFO$^{\mathsf{reg}}$ sentence φ if there is a model of φ that extends D. Let φ be an UNFO$^{\mathsf{reg}}$ sentence and Σ a set of predicate symbols. The *database satisfiability problem associated with* φ *and* Σ is to decide, given a Σ-database D, whether D is satisfiable with φ. Note that OMQ evaluation can be reduced in polynomial time to Boolean OMQ evaluation as in the proof of Lemma 6. Moreover, for a Boolean OMQ $Q = (\mathcal{O}, \Sigma, q)$ and a Σ-database D, $D \not\models Q$ iff D is satisfiable with $\mathcal{O} \wedge \neg q$. Consequently, a coNP upper bound for OMQ evaluation in (UNFO$^{\mathsf{reg}}$, UC2RPQ) can be proved by establishing an NP upper bound for database satisfiability in UNFO$^{\mathsf{reg}}$.

Let φ_0 be an UNFO$^{\mathsf{reg}}$ formula and Σ a set of predicate symbols. We may assume w.l.o.g. that φ_0 is normal and that every symbol from Σ occurs in φ_0. Subdivisions and splittings, defined as in Section 3, shall again play an inportant role. However, instead of subdividing an atom $\mathcal{A}(t, t')$ into at most n^2 many atoms, we use at most *two* intermediary points. Informally, this splits a witnessing path for $\mathcal{A}(t, t')$ into three parts: the first part is from t to the first element from D that appears on the path, the third subdivision atom represents the part from the last element from D that appears on the path to t', and the second atom represents the remaining middle part of the path.

We use $\mathsf{ecl}(\varphi_0)$ to denote the union of $\mathsf{cl}(\varphi_0)$ and $\mathsf{qcl}(\varphi_0)$, closed under single negation, where $\mathsf{qcl}(\varphi_0)$ is $\mathsf{qcl}(\varphi_0, \{x\}, 2)$ extended with the set of all $\mathcal{A}[q_0/s, F/\{s'\}](x, x)$ such that $\mathcal{A}$ is an NFA that occurs in φ_0 and s, s' are states in $\mathcal{A}$. An *extended* 1*-type for* φ_0 is a subset $t \subseteq \mathsf{ecl}(\varphi_0)$ such that t satisfies the conditions for being a 1-type from Section 3. We denote with $\mathsf{eTP}(\varphi_0)$ the set of all extended 1-types for φ_0.

Let D be a Σ-database. A *type decoration for* D is a mapping $\tau : \mathsf{dom}(D) \to \mathsf{eTP}(\varphi_0)$. We write $D, \tau \models \mathcal{A}(a, b)$ if $D \models \mathcal{A}(a, b)$ with the semantics of tests reinterpreted: instead of demanding $D \models \varphi(a')$ for a test $\varphi(x)?$ to hold at an element a', we now require that $\varphi(x) \in \tau(a')$. Let $\varphi(\mathbf{a})$ be an (instantiated) C2RPQ. A *homomorphism from* $\varphi(\mathbf{a})$ *to* D *given* τ is a function $h : \mathbf{a} \cup \mathsf{var}(\varphi) \to \mathsf{dom}(D)$ such that the following conditions are satisfied: $h(\mathbf{a}) = \mathbf{a}$, $h(\mathbf{t}) \in R^D$ for each $R(\mathbf{t}) \in \varphi(\mathbf{a})$, and for each $\mathcal{A}(t, t') \in \varphi(\mathbf{a})$, there are $a_1, \dots, a_n \in \mathsf{dom}(D)$ and states $s_0, \dots, s_n$ from $\mathcal{A}$, and a word $\nu_1 \cdots \nu_{n-1}$ from the alphabet of $\mathcal{A}$ such that

(a) $a_1 = h(t)$, $a_n = h(t')$, $s_0 = q_0$, and $s_n \in F$,
(b) $(a_i, a_{i+1}) \in R^D$ if $\nu_i = R$, $(a_{i+1}, a_i) \in R^D$ if $\nu_i = R^-$, and $\theta(x) \in \tau(a_i)$ and $a_{i+1} = a_i$ if $\nu_i = \theta(x)?$, for $1 \leq i < n$, and
(c) $(s, \nu_i, s_{i+1}) \in \Delta$ for some s with $\mathcal{A}[q_0/s_i, F/\{s\}](x, x) \in \tau(a_{i+1})$, for $0 \leq i < n$.

Note that Condition (c) admits the spontaneous change from state s_i to state s at a_{i+1}, without reading any of the ν_j symbols, when the atom $\mathcal{A}[q_0/s_i, F/\{s\}](x, x)$ is contained in $\tau(a_{i+1})$, asserting that we can indeed get from s_i to s starting at a_{i+1} and cycling back there while reading some unknown subword.

A type decoration τ is called *proper* if for all $a \in \mathsf{dom}(D)$, the following hold:

1. $\bigwedge_{\psi(x) \in \tau(a)} \psi(a)$ is satisfiable;
2. $\exists x \varphi(x) \in \tau(a)$ iff $\exists x \varphi(x) \in \tau(b)$, for all $a, b \in \mathsf{dom}(D)$ and all $\exists x \varphi(x) \in \mathsf{cl}(\varphi_0)$;
3. if $\neg\psi(x) \in \tau(a)$ for some $\psi(x) \in \mathsf{qcl}(\varphi_0)$, then for each $(\mathsf{dom}(D), 2)$-subdivision $\vartheta(\mathbf{a})$ of $\psi(a)$ and each splitting $\vartheta_0(\mathbf{a}_0), \vartheta_1(a_1), \dots, \vartheta_\ell(a_\ell)$ of $\vartheta(\mathbf{a})$ such that there is a homomorphism h from $\vartheta_0(\mathbf{a}_0)$ to D given τ, there is an $i \in \{1, \dots, k\}$ such that $\neg\vartheta_i(x) \in \tau(a_i)$.

Our NP procedure for database satisfiability is, given a Σ-database D, to guess a type decoration τ for D and to then verify in deterministic polynomial time that D is proper.

Note that the size of a type decoration is $\mathcal{O}(c \cdot |D|)$ for some constant c. The satisfiability checks in Point 1 of properness concern sentences whose size is independent of D, thus they need only constant time. Point 2 can be checked in time quadratic in the size of D. For Point 3, note that there are only polynomially many $(\mathsf{dom}(D), 2)$-subdivisions and splittings (in the size of D). To check the existence of the required homomorphism h, we can go through all candidates, directly verifying the homomorphism condition for relational atoms and proceedings as follows for NFA atoms: first extend D by exhaustively adding 'implied facts' of the form $\mathcal{A}(a, b)$, also taking into account assertions of the form $\mathcal{A}[q_0/s_i, F/\{s\}](x, x)$ that occur in τ-labels, as in Condition (c) above, and then treat NFA atoms like relational atoms. The following lemma finishes the proof of Theorem 16.

▶ **Lemma 17.** *D is satisfiable with φ_0 iff D has a proper type decoration τ such that $\varphi_0 \in \tau(a_0)$ for some $a_0 \in \mathsf{dom}(D)$.*

6 Model Checking

We show that model checking in UNFO$^{\mathsf{reg}}$ is complete for $\mathrm{P}^{\mathrm{NP}[O(\log^2 n)]}$, the class of problems that can be solved in polyniomial time given access to an NP oracle, but with only $O(\log^2 n)$ many oracle calls admitted. It thus has the same complexity as model checing in UNFO. Formally, the model checking problem for UNFO$^{\mathsf{reg}}$ is as follows: given a finite structure $\mathfrak{A}$ and a UNFO$^{\mathsf{reg}}$ sentence φ, does $\mathfrak{A} \models \varphi$ hold? Without tests in path expressions, UNFO$^{\mathsf{reg}}$ model checking can easily be reduced to model checking in UNFO: simply extend the input structure by exhaustively adding 'implied facts' of the form $\mathcal{A}(a, b)$ and then replace every $\mathcal{A}$ with a fresh binary relation symbol in both φ and $\mathfrak{A}$, obtaining an instance of UNFO model checking. With tests, this does not work. We would need multiple calls to UNFO model checking, essentially one call for every subformula inside a test in the input formula, but this bring us outside of $\mathrm{P}^{\mathrm{NP}[O(\log^2 n)]}$. We thus resort to expanding the $\mathrm{P}^{\mathrm{NP}[O(\log^2 n)]}$upper bound proof from [46], which is by reduction to a $\mathrm{P}^{\mathrm{NP}[O(\log^2 n)]}$-complete circuit value problem.

▶ **Theorem 18.** *The* UNFO$^{\mathsf{reg}}$ *model checking problem is* $\mathrm{P}^{\mathrm{NP}[O(\log^2 n)]}$*-complete.*

7 Conclusion

We have proved that OMQ evaluation in (UNFO$^{\mathsf{reg}}$, UC2RPQ) is decidable, 2ExpTime-complete in combined complexity, and coNP-complete in data complexity, and that OMQ containment and satisfiability are also 2ExpTime-complete. There are several interesting topics for future work. First, in contrast to UNFO, UNFO$^{\mathsf{reg}}$ does not have the finite model property and thus it would be interesting to study OMQ evaluation over finite models as well as finite satisfiability. Second, there are various natural directions for further increasing the expressive power. For example, one could allow any UNFO$^{\mathsf{reg}}$ formula with two free variables as a base case in regular path expressions instead of only atomic formulas. Such a logic would be strictly more expressive than propositional dynamic logic (PDL) with converse and intersection [31] and it would push the expressive power of UNFO$^{\mathsf{reg}}$ into the direction of regular queries, which have recently been proposed as an extension of C2RPQs [42]. Another natural extension was proposed by a reviewer of this paper: replace C2RPQs with linear Datalog to remove the asymmetry between binary relations and relations of higher arity in UNFO$^{\mathsf{reg}}$. Additional relevant extensions could arise from the aim to capture additional description logics. From this perspective, it would for example be natural to extend UNFO$^{\mathsf{reg}}$ with constants, with fixed points, and with so-called role inclusions, please

see [5]. Since functional relations and similar forms of counting play an important role in description logics, we remark that it is implicit in [46] that satisfiability (and thus OMQ evaluation) is undecidable in UNFO extended with two functional relations. Finally, it would be interesting to inverstigate the complexity of OMQ containment in $(\text{UNFO}^{\text{reg}}, \text{C2RPQ})$ without the restriction to a single ontology and to the full data signature. For (UNFO, CQ), a 2NExpTime upper bound can be proved by a slight adaptation of the technique in [21], also using (a slightly refined version of) the translation from (UNFO, CQ) to monadic disjunctive Datalog from [19]. However, accommodating C2RPQs in this approach seems nontrivial.

References

1 Antoine Amarilli, Michael Benedikt, Pierre Bourhis, and Michael Vanden Boom. Query answering with transitive and linear-ordered data. In *Proc. IJCAI*, pages 893–899, 2016.

2 Renzo Angles and Claudio Gutiérrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, 2008.

3 Franz Baader. Augmenting concept languages by transitive closure of roles: An alternative to terminological cycles. In *Proc. IJCAI*, pages 446–451, 1991.

4 Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge Univ. Press, 2nd edition, 2007.

5 Franz Baader, Ian Horrocks, Carsten Lutz, and Ulrike Sattler. *An Introduction to Description Logic*. Cambride University Press, 2017.

6 Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, and Eric Salvat. On rules with existential variables: Walking the decidability line. *Artif. Intell.*, 175(9-10):1620–1654, 2011.

7 Jean-François Baget, Marie-Laure Mugnier, Sebastian Rudolph, and Michaël Thomazo. Walking the complexity lines for generalized guarded existential rules. In *Proc. IJCAI*, pages 712–717, 2011.

8 Vince Bárány, Georg Gottlob, and Martin Otto. Querying the guarded fragment. *Logical Methods in Computer Science*, 10(2), 2014.

9 Vince Bárány, Balder ten Cate, and Luc Segoufin. Guarded negation. *J. ACM*, 62(3):22:1–22:26, 2015.

10 Pablo Barceló. Querying graph databases. In *Proc. PODS*, pages 175–188, 2013.

11 Pablo Barceló, Gerald Berger, and Andreas Pieris. Containment for rule-based ontology-mediated queries. In *Proc. PODS*, 2018.

12 Meghyn Bienvenu, Diego Calvanese, Magdalena Ortiz, and Mantas Šimkus. Nested regular path queries in description logics. In *Proc. KR*, 2014.

13 Meghyn Bienvenu, Peter Hansen, Carsten Lutz, and Frank Wolter. First order-rewritability and containment of conjunctive queries in Horn description logics. In *Proc. IJCAI*, pages 965–971. IJCAI/AAAI Press, 2016.

14 Meghyn Bienvenu, Carsten Lutz, and Frank Wolter. Query containment in description logics reconsidered. In *Proc. KR*. AAAI Press, 2012.

15 Meghyn Bienvenu and Magdalena Ortiz. Ontology-mediated query answering with data-tractable description logics. In *Proc. Reasoning Web*, volume 9203 of *LNCS*, pages 218–307. Springer, 2015.

16 Meghyn Bienvenu, Magdalena Ortiz, and Mantas Šimkus. Conjunctive regular path queries in lightweight description logics. In *Proc. IJCAI*, pages 761–767. IJCAI/AAAI, 2013.

17 Meghyn Bienvenu, Magdalena Ortiz, and Mantas Šimkus. Navigational queries based on frontier-guarded datalog: Preliminary results. In *Proc. AMW*, volume 1378 of *CEUR Workshop Proceedings*, 2015.

18 Meghyn Bienvenu, Magdalena Ortiz, and Mantas Šimkus. Regular path queries in lightweight description logics: Complexity and algorithms. *J. Artif. Intell. Res.*, 53:315–374, 2015.

19 Meghyn Bienvenu, Balder ten Cate, Carsten Lutz, and Frank Wolter. Ontology-based data access: A study through disjunctive datalog, CSP, and MMSNP. *ACM Trans. Database Syst.*, 39(4):33:1–33:44, 2014.

20 Pierre Bourhis, Markus Krötzsch, and Sebastian Rudolph. How to best nest regular path queries. In *Proc. DL*, volume 1193 of *CEUR Workshop Proceedings*, pages 404–415, 2014.

21 Pierre Bourhis and Carsten Lutz. Containment in monadic disjunctive datalog, MMSNP, and expressive description logics. In *Proc. KR*, pages 207–216. AAAI Press, 2016.

22 Andrea Calì, Georg Gottlob, and Michael Kifer. Taming the infinite chase: Query answering under expressive relational constraints. In *Proc. KR*, pages 70–80, 2008.

23 Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Data complexity of query answering in description logics. *Artif. Intell.*, 195:335–360, 2013.

24 Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Daniele Nardi. Reasoning in expressive description logics. In *Handbook of Automated Reasoning*, pages 1581–1634. Elsevier and MIT Press, 2001.

25 Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Containment of conjunctive regular path queries with inverse. In *Proc. KR*, pages 176–185, 2000.

26 Diego Calvanese, Thomas Eiter, and Magdalena Ortiz. Answering regular path queries in expressive description logics: An automata-theoretic approach. In *Proc. AAAI*, pages 391–396, 2007.

27 Diego Calvanese, Thomas Eiter, and Magdalena Ortiz. Regular path queries in expressive description logics with nominals. In *Proc. IJCAI*, pages 714–720, 2009.

28 Diego Calvanese, Thomas Eiter, and Magdalena Ortiz. Answering regular path queries in expressive description logics via alternating tree-automata. *Inf. Comput.*, 237:12–55, 2014.

29 Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion. In *Proc. SIGMOD*, pages 323–330, 1987.

30 Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18(2):194–211, 1979.

31 Stefan Göller, Markus Lohrey, and Carsten Lutz. PDL with intersection and converse: satisfiability and infinite-state model checking. *J. Symb. Log.*, 74(1):279–314, 2009.

32 Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. In *Proc. VLDB*, pages 95–106, 2002.

33 Georg Gottlob, Giorgio Orsi, Andreas Pieris, and Mantas Šimkus. Datalog and its extensions for semantic web databases. In *Proc. Reasoning Web*, volume 7487 of *LNCS*, pages 54–77. Springer, 2012.

34 Georg Gottlob, Andreas Pieris, and Lidia Tendera. Querying the guarded fragment with transitivity. In *Proc. ICALP II*, volume 7966 of *LNCS*, pages 287–298. Springer, 2013.

35 Erich Grädel. On the restraining power of guards. *J. Symb. Log.*, 64(4):1719–1742, 1999.

36 Erich Grädel and Igor Walukiewicz. Guarded fixed point logic. In *Proc. LICS-99*, pages 45–54, 1999.

37 Roman Kontchakov and Michael Zakharyaschev. An introduction to description logics and query rewriting. In *Proc. Reasoning Web*, volume 8714 of *LNCS*, pages 195–244. Springer, 2014.

38 Adila Krisnadhi and Carsten Lutz. Data complexity in the $\mathcal{EL}$ family of description logics. In *Proc. LPAR*, volume 4790 of *LNCS*, pages 333–347. Springer, 2007.

39 Carsten Lutz. The complexity of conjunctive query answering in expressive description logics. In *Proc. IJCAR*, volume 5195 of *LNCS*, pages 179–193. Springer, 2008.

40 Magdalena Ortiz, Sebastian Rudolph, and Mantas Šimkus. Query answering in the Horn fragments of the description logics $\mathcal{SHOIQ}$ and $\mathcal{SROIQ}$. In *Proc. IJCAI*, pages 1039–1044, 2011.

41 Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. *J. Data Semantics*, 10:133–173, 2008.

42 Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. Regular queries on graph databases. *Theory Comput. Syst.*, 61(1):31–83, 2017.

43 Sebastian Rudolph and Markus Krötzsch. Flag & check: data access with monadically defined queries. In *Proc. PODS*, pages 151–162, 2013.

44 Klaus Schild. A correspondence theory for terminological logics: Preliminary report. In *Proc. IJCAI*, pages 466–471, 1991.

45 Wiesław Szwast and Lidia Tendera. The guarded fragment with transitive guards. *Ann. Pure Appl. Logic*, 128(1-3):227–276, 2004.

46 Balder ten Cate and Luc Segoufin. Unary negation. *Logical Methods in Computer Science*, 9(3), 2013.

47 Moshe Y. Vardi. Reasoning about the past with two-way automata. In *Proc. ICALP-98*, pages 628–641, 1998.

Covers of Query Results

Ahmet Kara
Department of Computer Science, University of Oxford, Oxford, UK
ahmet.kara@cs.ox.ac.uk

Dan Olteanu
Department of Computer Science, University of Oxford, Oxford, UK
dan.olteanu@cs.ox.ac.uk

Abstract

We introduce succinct lossless representations of query results called covers. They are subsets of the query results that correspond to minimal edge covers in the hypergraphs of these results.

We first study covers whose structures are given by fractional hypertree decompositions of join queries. For any decomposition of a query, we give asymptotically tight size bounds for the covers of the query result over that decomposition and show that such covers can be computed in worst-case optimal time up to a logarithmic factor in the database size. For acyclic join queries, we can compute covers compositionally using query plans with a new operator called cover-join. The tuples in the query result can be enumerated from any of its covers with linearithmic pre-computation time and constant delay.

We then generalize covers from joins to functional aggregate queries that express a host of computational problems such as aggregate-join queries, in-database optimization, matrix chain multiplication, and inference in probabilistic graphical models.

2012 ACM Subject Classification Information systems → Relational database model, Information systems → Join algorithms, Information systems → Relational database query languages

Keywords and phrases factorized database, fractional hypertree decomposition, functional aggregate query, minimal edge cover, query plan

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.16

Funding This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement 682588.

Acknowledgements The authors would like to thank Milos Nikolic, Max Schleich, and the anonymous reviewers for their feedback on drafts of this paper, and Yu Tang for inspiring discussions that led to the concept of cover as a relational alternative to factorized representations.

1 Introduction

This paper introduces succinct lossless representations of query results called covers. Given a database and a join query or, more generally, a functional aggregate query (FAQ) [18], a cover is a subset of the query result that, together with a (fractional hypertree) decomposition of the query [13], recovers the query result. Covers enjoy desirable properties.

First, they can be more succinct than the listing representation of the query result. For a join query Q, database $\mathbf{D}$, and a decomposition $\mathcal{T}$ of Q with fractional hypertree width w [20], a cover over $\mathcal{T}$ has size $\mathcal{O}(|\mathbf{D}|^w)$. In contrast, there are arbitrarily large databases for which the listing representation of the query result has size $\Omega(|\mathbf{D}|^{\rho^*})$, where ρ^* is the fractional edge cover number of Q [4]. The gap between the fractional hypertree width and the fractional edge cover number can be as large as the number of relation symbols in Q.

For an FAQ (and the special case of a join query) φ, any cover of its result can be computed in time $\mathcal{O}(|\mathbf{D}|^w \log |\mathbf{D}|)$, where w is the FAQ-width [18] of φ. FAQs can express aggregates over database joins [6], in-database optimization [24, 2], matrix chain multiplication, and inference in probabilistic graphical models.

Second, the tuples in the query result can be enumerated from one of its covers with linearithmic pre-computation time and constant delay. This is not the case for the representation defined by the pair of database and join query (unless W[1]=FPT) [25]. The benefits of covers over the latter representation are less apparent for acyclic queries, for which both representations share the same linear-size bound and desirable enumeration complexity [5]. For acyclic joins, the question thus becomes why to succinctly represent a query result by one relation instead of the pair of a set of relations and the query. We next highlight three practical benefits. Covers readily provide a subset of the query result without the need to compute the join. This improves cache locality for subsequent operations, e.g., aggregates, since we only need to read in tuple by tuple from the cover instead of reading tuples from different relations stored at different locations in memory and then joining them. Similarly, covers provide access locality for disk operations since tuples from the cover are stored on the same disk page, whereas tuples from different relations are stored on different pages. Furthermore, covers are samples of the query result that disregard the uninformative yet exhaustive pairings brought by Cartesian products. In exploratory data analysis, the explicit listing of Cartesian products is overwhelming to the user since it may be very large. An alternative approach that would present the user with many relations and the query, would have to rely on the user to figure out possible tuples in the query result, which is not desirable. A cover, in contrast, is a compact relation that absolves the user from ad-hoc joining of relations and from re-discovering Cartesian products in a large listing of tuples. Finally, processing following the in-database joins may require a single relation as input, as it is the case for machine learning over joins [24]. Indeed, instead of learning regression models over the result of a join we can instead learn them over one of its covers.

Third, covers use the standard listing representation. Prior work introduced lossless representations of query results called *factorized databases* that achieve the same succinctness as covers, yet they are directed acyclic graphs that represent the query result as circuits whose nodes are data values or the relational operators Cartesian product and union [23]. The graph representation makes difficult their adoption as a data representation model by mainstream database systems that rely on relational storage (factorized *computation* is however used in relational systems [2]). A relational alternative to factorized databases, as metamorphosed in covers, can prove useful in a variety of settings. The intermediate results in query plans can be represented as covers. In distributed query plans, covers can encode succinctly the otherwise expensive intermediate query results that are communicated among servers in each round [26] and can be processed as soon as each of their tuples is received.

The contributions of this paper are as follows:

- Section 3 introduces covers of join query results and their correspondence to minimal edge covers in the hypergraphs of the query results. We also give tight size bounds for covers and show that the tuples in the query result can be enumerated from any cover with linearithmic pre-computation time and constant delay.
- Given a database and a join query, covers of its result can be computed in worst-case optimal time (modulo a log factor). Section 4 focuses on the compositionality of cover computation for acyclic join queries. We introduce cover-join plans to compute covers in time linearithmic in their sizes and the size of the input database. A cover-join plan is a binary plan that follows the structure of a join tree of the acyclic query. It uses a

cover-join operator that computes covers of the join of two relations, which may be input relations or covers for subqueries. Different plans may lead to different sets of covers. There are covers that cannot be obtained using binary plans.

- Section 5 generalizes our notion of covers from joins to functional aggregate queries by representing succinctly both tuples and aggregates in the query result.

We consider natural join queries where each relation is used at most once. The appendix extends our results to arbitrary equi-join queries and provides further details and examples. The proofs of the formal statements are given in an extended version of this paper [17].

Related work. There are three strands of directly related work: cores in databases and graph theory; succinct representations of query results; and normal forms for relational data.

Cores of graphs, queries, and universal solutions to data exchange problems revolve around smaller yet lossless representations that are homomorphically minimal subgraphs [16], subqueries [8], and universal solutions [11], respectively. A further application of graph cores is in the context of the Semantic Web, where cores of RDF graphs are used to obtain minimal representations and normal forms of such graphs [15]. Our notion of covers is different. Covers rely on query decompositions to achieve succinctness, and they only become lossless *in conjunction* with a decomposition. If we ignore the decomposition, the covers become lossy as they are subsets of the result. Whereas in data exchange all universal solutions have the same core (up to isomorphism), the result of a query may have exponentially many incomparable covers. While not a defining component of cores in data exchange, generalized hypertree decompositions can help derive improved algorithms for computing the core of a relational instance with labeled nulls under different classes of dependencies [12].

Covers are relational encodings of d-representations, a lossless graph-based factorization of the query result [23]. The structure of d-representations is given by variable orders called d-trees, which are an alternative syntax for fractional hypertree decompositions. Whereas d-representations are lossless on their own, covers need the decomposition to derive the missing tuples. Decompositions are the data-independent price to pay for achieving the data-dependent succinctness of factorized representations using the listing representation. Both d-representations and covers achieve succinctness by avoiding the materialization of Cartesian products. Whereas the former encode the products symbolically and losslessly, the covers only keep a minimal subset of the product that is enough to reconstruct it entirely.

The goal of database design is to avoid redundancy in the *input* database. Existing normal forms achieve this by *decomposing* one relation into several relations guided by functional and join dependencies [9]. Covers exploit the join dependencies to avoid redundancy in the query *output*. They do not decompose the result back into the (now globally consistent) input database. Like factorized representations, covers are a normal form for relations representing query results. From a cover of a join result over a decomposition, we can obtain a decomposition of the join result in project-join normal form (5NF) [10] by taking one projection of the cover onto the attributes of each bag of the decomposition.

2 Preliminaries

Databases. We assume an ordered domain of data values. A relation schema is a finite set of attributes. For an attribute A, we denote by $\mathsf{dom}(A)$ its domain. A database schema is a finite set of relation symbols. A tuple t over a relation schema S is a mapping from the attributes in S to values in their respective domains. A relation over a relation schema S is a finite set of tuples over S. A database $\mathbf{D}$ over a database schema $\mathcal{S}$ contains for each relation

symbol in $\mathcal{S}$, a relation over the same schema. For a relation (symbol) R and tuple t, we use $\mathcal{S}(R)$ and $\mathcal{S}(t)$ to refer to their schemas and write $R(S)$ to express that the schema of R is S. The tuples $t_1, \ldots, t_n$ are joinable if $\pi_{S_{i,j}} t_i = \pi_{S_{i,j}} t_j$ for all $i, j \in [n]$ and $S_{i,j} = \mathcal{S}(t_i) \cap \mathcal{S}(t_j)$. The size $|R|$ of a relation R is the number of its tuples. The size $|\mathbf{D}|$ of a database $\mathbf{D}$ is the sum of the sizes of its relations.

Natural Join Queries. We consider natural join queries of the form $Q = R_1(S_1) \bowtie \ldots \bowtie R_n(S_n)$, where each R_i is a relation symbol over relation schema S_i and refers to a database relation over the same schema. Notation-wise we do not distinguish between a relation symbol and the corresponding relation. The joins in Q are expressed by sharing attributes across relation schemas. The schema $\mathcal{S}(Q)$ of Q is the set of relation symbols in Q: $\mathcal{S}(Q) = \{R_i\}_{i \in [n]}$. The set $att(Q)$ of attributes of Q is the union of the schemas of its relation symbols: $att(Q) = \bigcup_{i \in [n]} S_i$. The size $|Q|$ of Q is the number of its relation symbols: $|Q| = n$. A database is *globally consistent* with respect to a query Q if there are no (dangling) tuples that do not contribute to the result of Q [1]. Two relations R_1 and R_2 are *consistent* if the database $\{R_1, R_2\}$ is globally consistent with respect to the query $R_1 \bowtie R_2$. We assume that the relation symbols in Q are non-repeating and each relation symbol corresponds to a distinct relation. Appendix C extends our contributions to arbitrary equi-join queries.

Hypergraphs. Let H be a multi-hypergraph (hypergraph for short) whose edge multiset E may contain multiple hyperedges (edges for short) with the same node set. A fractional edge cover for H is a function γ mapping each edge in H to a positive number such that $\Sigma_{e \ni v} \gamma(e) \geq 1$ for each node v of H, i.e., the sum of the function values for all edges incident to v is at least 1. We define the weight of a fractional edge cover γ as $weight(\gamma) = \Sigma_{e \in E} \gamma(e)$. The fractional edge cover number $\rho^*(H)$ of H is the minimum weight of fractional edge covers of H. It can be obtained from a fractional edge cover where the edge weights are rational numbers of bit-length polynomial in the size of H [4].

We use hypergraphs for queries and for relations representing their results. The hypergraph H of a query Q consists of one node A for each attribute A in Q and one edge $\mathcal{S}(R)$ for each relation symbol $R \in \mathcal{S}(Q)$. We define $\rho^*(Q) = \rho^*(H)$.

Let R be a relation and $\mathcal{P}$ a set of (possibly overlapping) subsets of $\mathcal{S}(R)$ such that $\bigcup_{S \in \mathcal{P}} S = \mathcal{S}(R)$. The hypergraph H of R over $\mathcal{P}$ consists of one node for each distinct tuple in $\pi_S R$ for each attribute set $S \in \mathcal{P}$ and one edge for each tuple in R. The edge for a tuple t thus consists of all nodes for tuples $\pi_S(t)$ with $S \in \mathcal{P}$. We use $tuple(v)$ to denote the tuple represented by a node or edge v in H. Given a subset M of the edges in H, we define $rel(M) = \{tuple(e)\}_{e \in M}$ as the relation represented by M. The set M is an edge cover of H if each node in H is contained in at least one edge in M. The set M is a minimal edge cover if it is an edge cover and any of its strict subsets is not.

▶ **Example 1.** Consider the path query $Q = R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(C, D)$. Figure 1 depicts in the top row a database of the three relations R_1, R_2 and R_3, the query result and a subset of it. In the bottom row, the figure depicts the hypergraph of Q (and its decomposition defined below), the hypergraph of its result over the attribute sets $\{\{A, B\}, \{B, C\}, \{C, D\}\}$, and the hypergraph of a subset of the query result over the same attribute sets.

Decompositions. A *hypertree decomposition* $\mathcal{T}$ of (the hypergraph H of) a query Q is a pair (T, χ), where T is a tree and χ a function mapping each node in T to a subset of the nodes of H. For a node $t \in T$, the set $\chi(t)$ is called a bag. A hypertree decomposition satisfies

R_1
A B
a_1 b_1
a_1 b_2
a_2 b_1
a_2 b_2
a_1 b_3

R_2
B C
b_1 c_1
b_2 c_2
b_3 c_3
b_4 c_4

R_3
C D
c_1 d_1
c_1 d_2
c_2 d_1
c_2 d_2
c_4 d_1

$Q(\mathbf{D})$
A B C D
a_1 b_1 c_1 d_1
a_1 b_1 c_1 d_2
a_2 b_1 c_1 d_1
a_2 b_1 c_1 d_2
a_1 b_2 c_2 d_2
a_1 b_2 c_2 d_1
a_2 b_2 c_2 d_2
a_2 b_2 c_2 d_1

$rel(M)$
A B C D
a_1 b_1 c_1 d_1
a_2 b_1 c_1 d_2
a_1 b_2 c_2 d_1
a_2 b_2 c_2 d_2

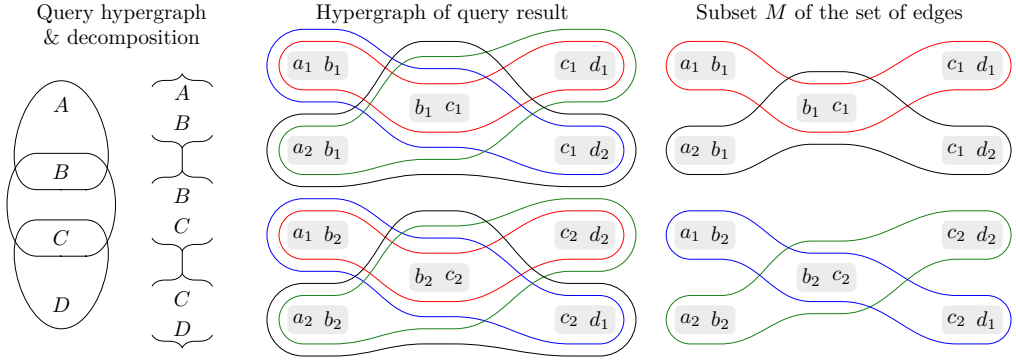


Figure 1 Top row: database $\mathbf{D} = \{R_1, R_2, R_3\}$, the result $Q(\mathbf{D})$ of the path query Q in Example 1, and a subset of $Q(\mathbf{D})$; bottom row: the hypergraph of Q, the tree of a decomposition $\mathcal{T}$ of Q, the hypergraph of $Q(\mathbf{D})$ over attribute sets $\mathcal{S}(\mathcal{T})$, and a minimal edge cover M of this hypergraph.

two properties. *Coverage*: For each edge e in H, there must be a node t in T with $e \subseteq \chi(t)$. *Connectivity*: For each node v in H, the set $\{t \mid t \in T, v \in \chi(t)\}$ must be non-empty and form a connected subtree in T. The schema of $\mathcal{T}$ is the set of its bags: $\mathcal{S}(\mathcal{T}) = \{\chi(t) \mid t \in T\}$. The attributes of $\mathcal{T}$ are defined by $att(\mathcal{T}) = \bigcup_{B \in \mathcal{S}(\mathcal{T})} B$.

A *fractional hypertree decomposition* [14] of (the hypergraph H of) a query Q is a triple $(T, \chi, \{\gamma_t\}_{t \in T})$ where (T, χ) is a hypertree decomposition of H and for each node $t \in T$, γ_t is a fractional edge cover of minimal weight for the subgraph of H restricted to $\chi(t)$. We define the *fractional hypertree width* of $\mathcal{T} = (T, \chi, \{\gamma_t\}_{t \in T})$ as $\max_{t \in T}\{weight(\gamma_t)\}$ and we denote it by $\mathsf{fhtw}(\mathcal{T})$. The fractional hypertree width $\mathsf{fhtw}(H)$ of the hypergraph H is the minimal possible such width of any fractional hypertree decomposition of H. The fractional hypertree width $\mathsf{fhtw}(Q)$ of a query Q is the fractional hypertree width $\mathsf{fhtw}(H)$ of its hypergraph H. For simplicity, we use the terms decomposition and width in place of fractional hypertree decomposition and fractional hypertree width, respectively.

A hypergraph H is *α-acyclic* (acyclic for short) if it has a decomposition in which each bag is contained in an edge of H [7]. A query whose hypergraph is acyclic is also called acyclic. The width of any acyclic hypergraph or query is one. A *join tree* of a query Q is a labelled tree (T, ℓ) where $T = (\mathcal{S}(Q), E)$ is a tree and ℓ is an edge labelling such that (i) each edge $e = (R, R') \in E$ is labelled by $\ell(e) = \mathcal{S}(R) \cap \mathcal{S}(R')$ and (ii) for every pair R, R' of distinct nodes and for each attribute $A \in \mathcal{S}(R) \cap \mathcal{S}(R')$, the label of each edge along the unique path between R and R' includes A (Section 6.4 in [1]). A query is acyclic if and only if it admits a join tree (Theorem 6.4.5 in [1]). The decomposition $\mathcal{T}$ *corresponding* to the join tree $\mathcal{J}$ of a query Q is constructed as follows. Each node in $\mathcal{J}$, which corresponds to a

relation symbol R, is mapped to a node in $\mathcal{T}$, which has the bag $\mathcal{S}(R)$. For each node t in $\mathcal{T}$ with bag $\mathcal{S}(R)$, the function γ_t maps the hyperedge for R to 1.

▶ **Example 2.** Figure 1 gives the hypergraph (left, bottom row) of the path query in Example 1 along with one of its decompositions. This decomposition has width one, since each bag is included in one edge of the hypergraph; the path query is acyclic. The decomposition, where the top two bags are merged into one, has width two. For queries with cycles, the width can be larger than one. For instance, the width of the triangle query is 3/2 [4].

Computational Model. We use the uniform-cost RAM model [3] where data values as well as pointers to databases are of constant size. Our analysis is with respect to data complexity where the query is assumed fixed. We use $\widetilde{\mathcal{O}}$ to hide a $\log |\mathbf{D}|$ factor.

Result-preserving Transformation. Let $(Q, \mathcal{T}, \mathbf{D})$ denote a triple of a natural join query Q, a decomposition $\mathcal{T}$ of Q, and a database $\mathbf{D}$.

▶ **Proposition 3.** *Given $(Q, \mathcal{T}, \mathbf{D})$, we can compute $(Q', \mathcal{T}, \mathbf{D}')$ with size $\mathcal{O}(|\mathbf{D}|^{fhtw(\mathcal{T})})$ and in time $\widetilde{\mathcal{O}}(|\mathbf{D}|^{fhtw(\mathcal{T})})$ such that Q' is an acyclic natural join query, $\mathcal{T}$ corresponds to a join tree of Q', $\mathbf{D}'$ is globally consistent with respect to Q' and $Q'(\mathbf{D}') = Q(\mathbf{D})$.*

▶ **Example 4.** Consider the path query Q, decomposition $\mathcal{T}$, and database $\mathbf{D}$ in Example 2. The application of Proposition 3 leaves Q unchanged, since Q is already acyclic and $\mathcal{T}$ corresponds to a join tree of Q. The database in Figure 1 is not globally consistent with respect to Q, since it contains tuples (under the thin lines) that do not contribute to the result. We remove these dangling tuples to make it consistent.

Consider now the bowtie query $Q_{\bowtie} = R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(A, C) \bowtie R_4(A, D) \bowtie R_5(D, E) \bowtie R_6(A, E)$. A decomposition $\mathcal{T}_{\bowtie}$ with the lowest width of 3/2 has two bags $S_1 = \{A, B, C\}$ and $S_2 = \{A, D, E\}$, one for each clique (triangle) in the query. The application of Proposition 3 constructs the acyclic query $Q' = B_1(A, B, C) \bowtie B_2(A, D, E)$. The relations $B_1(A, B, C)$ and $B_2(A, D, E)$ are materializations of the two bags of $\mathcal{T}_{\bowtie}$. The database $\mathbf{D}' = \{B_1(A, B, C), B_2(A, D, E)\}$ is globally consistent with respect to Q', i.e., each tuple in B'_1 has at least one joinable tuple in B'_2 and vice versa. The decomposition $\mathcal{T}_{\bowtie}$ corresponds to a join tree of Q'.

3 Covers for Join Queries

In this section we introduce the notion of covers of join query results along with a characterization of their size bounds, the connection to minimal edge covers for hypergraphs of join query results, and the complexity for enumerating the tuples in the query result from a cover.

Let $(Q, \mathcal{T}, \mathbf{D})$ denote a triple of a natural join query Q, decomposition $\mathcal{T}$ of Q, and database $\mathbf{D}$. For an instance $(Q, \mathcal{T}, \mathbf{D})$, covers of the query result $Q(\mathbf{D})$ are relations that are minimal while preserving the information in the query result $Q(\mathbf{D})$ in the following sense.

▶ **Definition 5** (Result Preservation)**.** A relation K is *result-preserving with respect to* $(Q, \mathcal{T}, \mathbf{D})$ if its schema $\mathcal{S}(K)$ is $att(Q)$ and $\pi_B K = \pi_B Q(\mathbf{D})$ for each $B \in \mathcal{S}(\mathcal{T})$.

That is, for each bag B in the decomposition $\mathcal{T}$ of Q, both the relation K and the query result $Q(\mathbf{D})$ have the same projection onto B. This also means that the natural join of these projections of K is precisely $Q(\mathbf{D})$.

▶ **Proposition 6.** *Given $(Q, \mathcal{T}, \mathbf{D})$, a relation K with schema $att(Q)$ is result-preserving with respect to $(Q, \mathcal{T}, \mathbf{D})$ if and only if $\bowtie_{B \in \mathcal{S}(\mathcal{T})} \pi_B K = Q(\mathbf{D})$.*

We further say that the relation K is *minimal* result-preserving with respect to $(Q, \mathcal{T}, \mathbf{D})$ if it is result-preserving with respect to $(Q, \mathcal{T}, \mathbf{D})$, yet this is not the case for any strict subset of it. We can now define the notion of covers of query results.

▶ **Definition 7** (Covers)**.** Given $(Q, \mathcal{T}, \mathbf{D})$, a *cover of the query result* $Q(\mathbf{D})$ *over the decomposition* $\mathcal{T}$ is a minimal result-preserving relation with respect to $(Q, \mathcal{T}, \mathbf{D})$.

▶ **Example 8.** Figure 1 gives the decomposition $\mathcal{T}$ of a path query and one cover $rel(M)$ of the query result over $\mathcal{T}$. We give below four relations that are subsets of the query result. The relations K_1 and K_2 are covers, while the relations N_1 and N_2 are not covers:

K_1			
A	B	C	D
a_1	b_1	c_1	d_2
a_2	b_1	c_1	d_1
a_1	b_2	c_2	d_2
a_2	b_2	c_2	d_1

K_2			
A	B	C	D
a_1	b_1	c_1	d_2
a_2	b_1	c_1	d_1
a_1	b_2	c_2	d_1
a_2	b_2	c_2	d_2

N_1			
A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_1	c_1	d_2
a_1	b_2	c_2	d_1

N_2			
A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_1	c_1	d_2
a_2	b_1	c_1	d_1
a_1	b_2	c_2	d_2
a_1	b_2	c_2	d_1

To check the minimal result-preservation property, we take projections onto the bags $B_1 = \{A, B\}$, $B_2 = \{B, C\}$, and $B_3 = \{C, D\}$. The relation N_1 is not result-preserving, because $(a_2, b_2) \notin \pi_{B_1} N_1$. The same argument also applies to relation N_2.

Consider now the coarser decomposition $\mathcal{T}'$ with bags $B'_{1,2} = \{A, B, C\}$ and $B'_3 = \{C, D\}$. The covers over $\mathcal{T}$ discussed above are also covers over $\mathcal{T}'$. The query result is the only cover over the coarsest decomposition $\mathcal{T}''$ with only one bag.

▶ **Example 9.** A query result may admit exponentially many covers over the same decomposition. Consider for instance the product query $R_1(A) \bowtie R_2(B)$ with relations R_1 and R_2 of size two and respectively $n > 1$. The query result has size $2 \cdot n$. To compute a cover, we pair the first tuple in R_1 with any non-empty and strict subset of the n tuples in R_2, while the second tuple in R_1 is paired with the remaining tuples in R_2. There are $2^n - 2$ possible covers. The empty and the full sets are missing from the choice of a subset of R_2 as they would mean that one of the two tuples in R_1 would have to be paired with tuples in R_2 that are already paired with the other tuple in R_1 and that would violate the minimality criterion of the covers. All covers have size n and none is contained in another.

We next give a characterization of covers via the hypergraph of the query result.

▶ **Proposition 10.** *Given $(Q, \mathcal{T}, \mathbf{D})$, a relation K is a cover of the query result $Q(\mathbf{D})$ over $\mathcal{T}$ if and only if the hypergraph of $Q(\mathbf{D})$ over $\mathcal{S}(\mathcal{T})$ has a minimal edge cover M such that $rel(M) = K$.*

▶ **Example 11.** Figure 1 gives a minimal edge cover M and the cover $rel(M)$. By removing any edge from M, it is not anymore an edge cover. By removing the tuple corresponding to that edge from $rel(M)$, it is not anymore a cover since it is not result preserving. By adding an edge to M or the corresponding tuple to $rel(M)$, they are not anymore minimal.

We now turn our investigation to sizes and first note the following immediate property.

▶ **Proposition 12.** *Given $(Q, \mathcal{T}, \mathbf{D})$, each cover of $Q(\mathbf{D})$ over $\mathcal{T}$ is a subset of $Q(\mathbf{D})$.*

An implication of Proposition 12 is that the covers cannot be larger than the query result. However, they can be much more succinct. We first give size bounds for covers using the sizes of projections of the query result onto the bags of the underlying decomposition.

▶ **Proposition 13.** *Given $(Q, \mathcal{T}, \mathbf{D})$, the size of each cover K of $Q(\mathbf{D})$ over $\mathcal{T}$ satisfies the inequalities* $\max_{B \in \mathcal{S}(\mathcal{T})}\{|\pi_B Q(\mathbf{D})|\} \leq |K| \leq \Sigma_{B \in \mathcal{S}(\mathcal{T})} |\pi_B Q(\mathbf{D})|$.

We can now characterize the size of a cover using the width of the decomposition.

▶ **Theorem 14.** *Let Q be a natural join query and $\mathcal{T}$ a decomposition of Q.*
(i) For each database $\mathbf{D}$, each cover of the query result $Q(\mathbf{D})$ over $\mathcal{T}$ has size $\mathcal{O}(|\mathbf{D}|^{\mathit{fhtw}(\mathcal{T})})$.
(ii) There are arbitrarily large databases $\mathbf{D}$ such that each cover of the query result $Q(\mathbf{D})$ over $\mathcal{T}$ has size $\Omega(|\mathbf{D}|^{\mathit{fhtw}(\mathcal{T})})$.

The size gaps between query results and their covers can be arbitrarily large. For any join query Q and database $\mathbf{D}$, it holds that $|Q(\mathbf{D})| = \mathcal{O}(|\mathbf{D}|^{\rho^*(Q)})$ and there are arbitrarily large databases $\mathbf{D}$ for which $|Q(\mathbf{D})| = \Omega(|\mathbf{D}|^{\rho^*(Q)})$ [4]. For acyclic queries, the fractional edge cover number ρ^* can be as large as $|Q|$, while the fractional hypertree width is one. Section 4 shows that the same gap also holds for time complexity.

▶ **Example 15.** We continue Example 8. The decomposition $\mathcal{T}$ has width one, which is minimal. The covers over $\mathcal{T}$, such as K_1 and K_2, have sizes upper bounded by the input database size. The minimum size of a cover over $\mathcal{T}$ is the maximum size of a relation used in the query (assuming the relations are globally consistent). In contrast, there are arbitrarily large databases of size N for which the query result has size $\Omega(N^2)$.

Proposition 10 and Theorem 14 give alternative equivalent characterizations of the size of a cover of a query result. The former gives it as the size of a *minimal edge cover* of the hypergraph of the query result over the attribute sets given by the bags of a decomposition $\mathcal{T}$, while the latter states it using the fractional hypertree width of $\mathcal{T}$ or equivalently the *maximum fractional edge cover* number over all the bags of $\mathcal{T}$. Most notably, whereas the former is an *integral* number, the latter is a *fractional* number.

This size gap between query results and their covers is precisely the same as for query results and their factorized representations called d-representations [23]. In this sense, covers can be seen as relational encodings of factorized representations of query results. We can easily translate covers into factorized representations. Appendix A gives a brief introduction to d-representations and a translation example.

▶ **Proposition 16.** *Given $(Q, \mathcal{T}, \mathbf{D})$, each cover K of the query result $Q(\mathbf{D})$ over $\mathcal{T}$ can be translated into a d-representation of $Q(\mathbf{D})$ of size $\mathcal{O}(|K|)$ and in time $\widetilde{\mathcal{O}}(|K|)$.*

The above translation allows us to extend the applicability of covers to known workloads over factorized representations, such as in-database optimization problems [2] and in particular learning regression models [22]. Nevertheless, it is practically desirable to process such workloads directly on covers, since this would avoid the indirection via factorized representations that comes with extra space cost and non-relational data representation. Aggregates, which are at the core of such workloads, can be computed directly on covers by joint scans of the projections of the cover onto the bags of the decomposition; alternatively, they can be computed by expressing any cover as the natural join of its bag projections and then pushing the aggregates past the join.

▶ **Example 17.** We consider the query $Q = R(A, B) \bowtie S(B, C)$ and its decomposition $\mathcal{T}$ with bags $\{A, B\}$ and $\{B, C\}$. To compute aggregates over the join result $Q(\mathbf{D})$, we can use any cover K of $Q(\mathbf{D})$ over $\mathcal{T}$. The expression for counting the number of result tuples is $\sum_{b \in \mathrm{dom}(B)} \sum_{a \in \mathrm{dom}(A)} \sum_{c \in \mathrm{dom}(C)} \mathbf{1}_{R(a,b)} \cdot \mathbf{1}_{S(b,c)}$, where 1_E is the Kronecker delta that is evaluated to $\mathbf{1}$ if the event E is satisfied and $\mathbf{0}$ otherwise. We can compute it in one scan over K if K is sorted on (B, A, C) or (B, C, A). For each B-value b,

we multiply the distinct numbers of A-values and of C-values paired with b in K, and we sum up these products over all B-values. We can rewrite this expression as follows: $\sum_{b\in\mathrm{dom}(B)}(\sum_{a\in\mathrm{dom}(A)}\mathbf{1}_{(a,b)\in\pi_{\{A,B\}}K})(\sum_{c\in\mathrm{dom}(C)}\mathbf{1}_{(b,c)\in\pi_{\{B,C\}}K})$. This expression only uses the pairs (a,b) and (b,c) in K. The pairs (a,c), which make the difference among covers and are the culprits for the explosion in the size of the query result, are not needed.

Despite their succinctness over the explicit listing of tuples in a query result, any cover of the query result can be used to enumerate the result tuples with constant delay and extra space (data complexity) following linear-time pre-computation. In particular, the delay and the space are linear in the number of attributes of the query result which is as good as enumerating directly from the result. This complexity follows from Proposition 16 and the enumeration for factorized representations [23] with constant delay and extra space.

▶ **Corollary 18** (Proposition 16, Theorem 4.11 [23]). *Given $(Q,\mathcal{T},\mathbf{D})$, the tuples in the query result $Q(\mathbf{D})$ can be enumerated from any cover K of $Q(\mathbf{D})$ over $\mathcal{T}$ with $\widetilde{\mathcal{O}}(|K|)$ pre-computation time and $\mathcal{O}(1)$ delay and extra space.*

An alternative way to achieve constant-delay enumeration with $\widetilde{\mathcal{O}}(|K|)$ pre-computation is by noting that the acyclic join queries considered in this paper are free-connex and thus allow for enumeration with constant delay and $\widetilde{\mathcal{O}}(|\mathbf{D}|)$ pre-computation [5]. An acyclic conjunctive query is called free-connex if its extension by a new relation symbol covering all attributes of the result remains acyclic [25]. Moreover, given a cover K over a decomposition $\mathcal{T}$, the natural join of the projections of K onto the bags of $\mathcal{T}$ is an acyclic query that computes the original query result (Proposition 6).

4 Computing Covers for Join Queries using Cover-Join Plans

Given an arbitrary join query and database, we can compute covers using a *monolithic* algorithm akin to known algorithms for computing factorized representations of query results [22]. However, is it possible to compute covers in a *compositional* way, by computing covers for one join at a time? In this section, we answer this question in the affirmative for acyclic natural join queries Q and globally consistent databases $\mathbf{D}$ with respect to Q.

For a triple $(Q,\mathcal{J},\mathbf{D})$, where Q is an acyclic natural join query, $\mathcal{J}$ is a join tree of Q, and $\mathbf{D}$ is a database globally consistent with respect to Q, we use so-called cover-join plans to compute covers of the query result $Q(\mathbf{D})$ over the decomposition corresponding to the join tree $\mathcal{J}$. Such plans follow the structure of the join tree $\mathcal{J}$ and use a new binary join operator called cover-join. The cover-join of two relations yields a cover of their natural join. This approach is in the spirit of standard relational query evaluation. It is compositional in the sense that to compute a cover of the query result, it suffices to repeatedly compute a cover of the join of two relations. This is practical since it can be supported by existing query engines extended with the cover-join operator. We also show that, due to the binary nature of the cover-join operator, the cover-join plans cannot recover all possible covers of the query result. Furthermore, different plans may lead to different covers. Plans that do not follow the structure of a join tree may be unsound as they do not necessarily construct covers.

To compute covers for an arbitrary join query and database, we proceed in two stages. We first materialize the bags of a decomposition of the query so as to reduce it to an acyclic query Q over an extended database $\mathbf{D}$ that is now globally consistent with respect to Q (Proposition 3). We then use a cover-join plan to compute covers of $Q(\mathbf{D})$. The first step has a non-trivial time complexity overhead, whereas the second step is linearithmic. Overall, this strategy is worst-case optimal for computing covers for arbitrary join queries and databases.

4.1 The Cover-Join Operator

The building block of our approach to computing covers is the binary cover-join operator.

▶ **Definition 19** (Cover-Join)**.** The cover-join of two relations R_1 and R_2, denoted by $R_1 \bowtie R_2$, computes a cover of their join result over the decomposition with bags $\mathcal{S}(R_1)$ and $\mathcal{S}(R_2)$.

Following the alternative characterization of covers of a query result by minimal edge covers in the hypergraph of the query result (Proposition 10), the cover-join defines the relation $rel(M)$ of a minimal edge cover M of the hypergraph H of the result of the join $R_1 \bowtie R_2$ over the attribute sets $\mathcal{S}(R_1)$ and $\mathcal{S}(R_2)$. The hypergraph H is bipartite and consists of disjoint complete bipartite subgraphs. Since a cover is a minimal edge cover, it corresponds to a bipartite subgraph with the same number of nodes but a subset of the edges, where all paths can only have one or two edges. A cover cannot have unconnected nodes, since it would not be an edge cover. A path of three (or more) edges violates the minimality of the edge cover: Such a path $a_1 - b_1 - a_2 - b_2$ in a bipartite graph covers the four nodes, yet a minimal cover would only have the two edges $a_1 - b_1$ and $a_2 - b_2$.

We can compute a cover of a join of two relations R_1 and R_2 in time $\widetilde{\mathcal{O}}(|R_1| + |R_2|)$, since it amounts to computing a minimal edge cover in a collection of disjoint complete bipartite graphs that encode the join result. The smallest size of a cover is given by the edge cover number of the bipartite graph representing the join result, which is the maximum of the sizes of the two sets of nodes in the graph [19]. The largest size can be achieved in case one of the two node sets has size one, in which case this is paired with all nodes in the second set. In case both sets have more than one node, the largest size is achieved when we pair one node from one of the two node sets with all but one node in the second set and then the remaining node in the second set with all but the already used node in the first set.

For the analysis in this paper, we assume that our cover-join algorithm may return any cover of the natural join of two relations. In practice, however, it makes sense to compute a cover of minimum size. We choose this cover as follows: For each complete bipartite hypergraph in the join result with node sets V_1 and V_2 such that $|V_1| \leq |V_2|$, we choose a minimum edge cover by pairing each node in V_1 with one distinct node in V_2 and all remaining nodes in V_2 with one node in V_1.

▶ **Proposition 20.** *Given two consistent relations R_1 and R_2, the cover-join computes a cover K of their join result over the decomposition with bags $\mathcal{S}(R_1)$ and $\mathcal{S}(R_2)$ in time $\widetilde{\mathcal{O}}(|R_1| + |R_2|)$ and with size $\max\{|R_1|, |R_2|\} \leq |K| \leq |R_1| + |R_2|$.*

▶ **Example 21.** Consider again the product $R_1(A) \bowtie R_2(B)$ in Example 9, where $R_1 = [2]$ and $R_2 = [n]$ with $n > 1$. Examples of covers of size n over the decomposition $\mathcal{T}$ with bags $\{A\}$ and $\{B\}$ are: $\{(1, i) \mid i \in [n] - \{k\}\} \cup \{(2, k)\}$ for any $k \in [n]$; $\{(1, i) \mid i \in [k]\} \cup \{(2, j + k) \mid j \in [n - k]\}$ for any $k \in [n - 1]$. If $R_1 = [m]$ with $m > n$, then examples of covers over $\mathcal{T}$ of minimum size m are: $\{(i, i) \mid i \in [k - 1]\} \cup \{(k - 1 + i, k + i) \mid i \in [n - k]\} \cup \{(n - 1 + i, k) \mid i \in [m - n + 1]\}$ for any $k \in [n]$. A cover over $\mathcal{T}$ of maximal size $n + m - 2$ is: $\{(1, i) \mid i \in [n - 1]\} \cup \{(j + 1, n) \mid j \in [m - 1]\}$. Below are depictions of the complete bipartite graph corresponding to the query result for $n = 4$ and $m = 5$, where the edges in a minimal edge cover are solid lines and all other edges are dotted. The left minimal edge cover corresponds to a cover over $\mathcal{T}$ of minimum size $m = 5$, while the right minimal edge cover corresponds to a cover over $\mathcal{T}$ of maximum size $n + m - 2 = 7$.

4.2 Cover-join Plans

We now compose cover-join operators into so-called cover-join plans to compute covers for acyclic natural join queries. Before we define such plans, we need to introduce some notation.

For a join tree $\mathcal{J}$ of a query Q, we write $\mathcal{J} = \mathcal{J}_1 \circ \mathcal{J}_2$ if $\mathcal{J}$ can be split into two non-empty subtrees $\mathcal{J}_1$ and $\mathcal{J}_2$ that are connected by a single edge in $\mathcal{J}$. Any subtree $\mathcal{J}'$ of $\mathcal{J}$ defines the subquery of Q that is the natural join of all relation symbols that are nodes in $\mathcal{J}'$.

▶ **Definition 22** (Cover-Join Plan). Given $(Q, \mathcal{J}, \mathbf{D})$, a *cover-join plan φ over the join tree $\mathcal{J}$* is defined recursively as follows:

- If $\mathcal{J}$ consists of one node R, then $\varphi = R$. The plan φ returns R.
- If $\mathcal{J} = \mathcal{J}_1 \circ \mathcal{J}_2$ and φ_i is a cover-join plan over $\mathcal{J}_i$, then $\varphi = \varphi_1 \ltimes \varphi_2$. The plan φ returns the result of $R_1 \ltimes R_2$, where the relation R_i is returned by the plan φ_i ($i \in [2]$).

Lemma 23 states next that a cover-join plan computes a cover of the query result over the decomposition corresponding to a given join tree of the query.

▶ **Lemma 23.** *Given $(Q, \mathcal{J}, \mathbf{D})$ where $\mathbf{D} = \{R_i\}_{i\in[n]}$ is globally consistent with respect to Q, each cover-join plan over the join tree $\mathcal{J}$ computes a cover K of $Q(\mathbf{D})$ over the decomposition corresponding to $\mathcal{J}$ in time $\widetilde{\mathcal{O}}(|K|)$ and with size $\max_{i\in[n]}\{|R_i|\} \leq |K| \leq \sum_{i\in[n]} |R_i|$.*

Lemma 23 states three remarkable properties of cover-join plans. First, they compute covers compositionally: To obtain a cover of the entire query result it is sufficient to compute covers of the results for subqueries. More precisely, for a cover-join plan $\varphi_1 \ltimes \varphi_2$, the sub-plans φ_1 and φ_2 compute covers for the subqueries defined by the joins of the relations in the join trees $\mathcal{J}_1$ and respectively $\mathcal{J}_2$. Then, the plan $\varphi_1 \ltimes \varphi_2$ computes a cover for the join of the relations in the join tree $\mathcal{J} = \mathcal{J}_1 \circ \mathcal{J}_2$. Second, the output of a cover-join plan is always a cover, regardless which cover is picked at each cover-join operator in the plan. Third, it does not matter which cover-join plan we choose for a given join tree, the resulting covers are computed with the same time guarantee. Nevertheless, different plans for the same join tree may lead to different covers (Example 28).

These properties rely on the global consistency of the database and on the fact that the plans follow the structure of the join tree. For arbitrary databases, a cover-join operator may wrongly construct covers using dangling tuples at the expense of relevant tuples that are not anymore covered and therefore lost. Furthermore, plans that do not follow the structure of a join tree may be unsound (Example 26). Although each cover-join operator computes a cover of minimum size for the join of its input relations, the overall cover computed by a cover-join plan may not be a cover of minimum size of the query result (Example 35 in Appendix B).

▶ **Example 24.** A join tree that admits several splits can define many plans. For instance, the join tree for the query $R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(C, D)$ is the path $R_1 - R_2 - R_3$ and admits two possible splits that lead to the plans $\varphi_1 = (R_1(A, B) \ltimes R_2(B, C)) \ltimes R_3(C, D)$ and $\varphi_2 = R_1(A, B) \ltimes (R_2(B, C) \ltimes R_3(C, D))$. The relations are those in Figure 1, now calibrated. For this database, the covers computed by the sub-plans $R_1(A, B) \ltimes R_2(B, C)$ and $R_2(B, C) \ltimes R_3(C, D)$ correspond to full join results, since all join values only occur once in the relations. By taking

any possible cover at each cover-join operator in the plans, both plans yield the same four possible covers of the query result: One of them is $rel(M)$ in Figure 1 and two of them are K_1 and K_2 in Example 8. The last cover is not depicted: It is the same as K_1 with the change that the values d_1 and d_2 are swapped between the first two rows.

A corollary of Proposition 3 and Lemma 23 is that covers over decompositions of *arbitrary* natural join queries can be computed in time proportional to their sizes.

▶ **Theorem 25** (Proposition 3, Lemma 23)**.** *Given a natural join query Q, decomposition $\mathcal{T}$ of Q, and database $\mathbf{D}$, a cover of the query result $Q(\mathbf{D})$ over the decomposition $\mathcal{T}$ and with size $\mathcal{O}(|\mathbf{D}|^{fhtw(\mathcal{T})})$ can be computed in time $\widetilde{\mathcal{O}}(|\mathbf{D}|^{fhtw(\mathcal{T})})$.*

Given $(Q, \mathcal{T}, \mathbf{D})$ where Q is an arbitrary natural join query and $\mathbf{D}$ is an arbitrary database, we can compute a cover in four steps: construct $(Q', \mathcal{T}, \mathbf{D}')$ such that Q' is an acyclic natural join query, $\mathcal{T}$ corresponds to a join tree of Q' and $\mathbf{D}'$ consists of materializations of the bags of $\mathcal{T}$; turn $\mathbf{D}'$ into a globally consistent database $\mathbf{D}''$ with respect to Q'; turn $\mathcal{T}$ into a join tree $\mathcal{J}$ of Q' by replacing each bag by the corresponding relation symbol in Q'; and execute on $\mathbf{D}''$ a cover-join plan for Q' over $\mathcal{J}$. Since there are arbitrarily large databases for which the size bounds on covers are tight (Theorem 14), the cover-join plans, together with a worst-case optimal algorithm for materializing bags [21], represent a worst-case optimal algorithm for computing covers.

We conclude this section with three insights into the ability of cover-join plans to compute covers. We give an example of an unsound cover-join plan that does not follow the structure of a join tree. We then note the incompleteness of our cover-join plans due to the binary nature of the cover-join operator. We give an example of a cover that cannot be computed with our cover-join plans, but can be computed using a multi-way cover-join operator. Finally, we give an example showing that distinct cover-join plans over the same (or also distinct) join trees can yield incomparable sets of covers.

▶ **Example 26** (Unsound plan)**.** Consider the query $Q = R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(C, D)$, the following database with relations R_1, R_2, and R_3, and four relations computed by cover-joining two of the three relations:

R_1	R_2	R_3	$K_{1,3}$	$K'_{1,3}$	$K_{1,2}$	$K_{2,3}$
A B	B C	C D	A B C D	A B C D	A B C	B C D
a b_1	b_1 c_1	c_1 d	a b_1 c_1 d	a b_1 c_2 d	a b_1 c_1	b_1 c_1 d
a b_2	b_2 c_2	c_2 d	a b_2 c_2 d	a b_2 c_1 d	a b_2 c_2	b_2 c_2 d

Following Definition 22, the plan $(R_1(A, B) \ltimes R_3(C, D)) \ltimes R_2(B, C)$ would require a split $\mathcal{J}_{1,3} \circ \mathcal{J}_2$ of a join tree, where the join tree $\mathcal{J}_{1,3}$ has two nodes R_1 and R_3 while the join tree $\mathcal{J}_2$ has one node R_2. However, there is no join tree that allows such a split.

The cover-join $R_1(A, B) \ltimes R_3(C, D)$ computes one of the two covers $K_{1,3}$ and $K'_{1,3}$. The result of the join of $K'_{1,3}$ and R_2 is empty and so is the cover-join. This means that this plan does not always compute a cover, which makes it unsound.

This problem cannot occur with cover-join plans over join trees of Q. The only cover-join plans over join trees of Q are (up to commutativity) $(R_1(A, B) \ltimes R_2(B, C)) \ltimes R_3(C, D)$ and $R_1(A, B) \ltimes (R_2(B, C) \ltimes R_3(C, D))$. The only cover of $R_1(A, B) \ltimes R_2(B, C)$ is $K_{1,2}$ above, which can be cover-joined with R_3. The only cover of $R_2(B, C) \ltimes R_3(C, D)$ is $K_{2,3}$ above, which can be cover-joined with R_1.

▶ **Example 27** (Cover-Join Incompleteness)**.** Consider the product query $Q = R_1(A) \bowtie R_2(B) \bowtie R_3(C)$, the following database $\mathbf{D}$ with relations R_1, R_2, and R_3 and one cover K of the query result over the decomposition with bags $\{A\}$, $\{B\}$, and $\{C\}$:

R_1	R_2	R_3	K
A	B	C	$A\ B\ C$
a_1	b_1	c_1	$a_1\ b_1\ c_1$
a_2	b_2	c_2	$a_1\ b_2\ c_2$
			$a_2\ b_1\ c_2$

A decomposition of Q can have up to three bags which are not included in other bags.

In case of decompositions with three bags, each bag consists of exactly one attribute. These decompositions correspond to the join trees that are permutations of the three relation symbols. There are three possible cover-join plans (up to commutativity) over these join trees: $\varphi_1 = R_1(A) \ltimes (R_2(B) \ltimes R_3(C))$, $\varphi_2 = R_2(B) \ltimes (R_1(A) \ltimes R_3(C))$ and $\varphi_3 = R_3(C) \ltimes (R_1(A) \ltimes R_2(B))$. None of these plans can yield the cover K above. As discussed after Definition 19, a minimal edge cover corresponding to a cover computed by a *binary* cover-join operator can only have paths of one or two edges. For instance, $\pi_{\{A,B\}}K$, which should correspond to a cover of $R_1(A) \ltimes R_2(B)$, has the path of three edges $b_2 - a_1 - b_1 - a_2$. The cover-join $R_1(A) \ltimes R_2(B)$ would not create this path since it corresponds to a non-minimal edge cover. Similarly, $\pi_{\{A,C\}}K$ and $\pi_{\{B,C\}}K$ have paths of three edges.

For decompositions with two bags, two of the three attributes are in the same bag. Without loss of generality, assume A and B are in the same bag. Following Proposition 3, this bag is covered by a new relation $R_{1,2}$ that is the product of R_1 and R_2. This means that K has to be the cover of $R_{1,2}(A, B) \ltimes R_3(C)$, yet $\pi_{\{A,B\}}K$ is not $R_{1,2}$!

The decomposition with one bag consisting of all three attributes has this bag covered by a new relation that is the product of the three relations. This relation is the Cartesian product of the three relations that is the full query result and different from $K = \pi_{\{A,B,C\}}K$.

We conclude that K cannot be computed using (binary) cover-join plans.

▶ **Example 28** (Incomparable Sets of Covers)**.** Consider the product query $Q = R_1(A) \bowtie R_2(B) \bowtie R_3(C)$ and the following database $\{R_1, R_2, R_3\}$:

R_1	R_2	R_3	K	$K_{1,2}$	$K'_{1,2}$
A	B	C	$A\ B\ C$	$A\ B$	$A\ B$
a_1	b_1	c_1	$a_1\ b_1\ c_1$	$a_1\ b_1$	$a_1\ b_2$
a_2	b_2	c_2	$a_2\ b_2\ c_2$	$a_2\ b_2$	$a_2\ b_1$
		c_3	$a_1\ b_2\ c_3$		

Let us consider the join tree $\mathcal{J} = R_1 - R_2 - R_3$ of Q. There are (up to commutativity) two possible cover-join plans over $\mathcal{J}$: $\varphi_1 = R_1(A) \ltimes (R_2(B) \ltimes R_3(C))$ and $\varphi_2 = (R_1(A) \ltimes R_2(B)) \ltimes R_3(C)$. The above relation K is a cover of the result of Q and can be computed by φ_1, which cover-joins $R_1(A)$ and a cover of the join of $R_2(B)$ and $R_3(C)$. This cover cannot be computed by φ_2. Indeed, φ_2 first cover-joins $R_1(A)$ and $R_2(B)$, yielding $K_{1,2}$ or $K'_{1,2}$ as the only possible covers. Then, cover-joining any of them with $R_3(C)$ does not yield the cover K since $\pi_{\{A,B\}}K$ is different from both $K_{1,2}$ and $K'_{1,2}$. Similarly, φ_2 computes covers that cannot be computed by φ_1.

5 Covers for Functional Aggregate Queries

We first give a brief introduction to functional aggregate queries (FAQ) [18]. A detailed description can be found in the extended report [17].

Given an attribute set S, we use a_S to indicate that tuple a has schema S. For $S' \subseteq S$, we denote by $\mathsf{a}_{S'}$ the restriction of a to S'. A functional aggregate query has the following form (slightly adapted to our notation):

$$\varphi(\mathsf{a}_{\{A_1,\ldots,A_f\}}) = \bigoplus_{a_{f+1}\in\mathsf{dom}(A_{f+1})}^{(f+1)} \cdots \bigoplus_{a_n\in\mathsf{dom}(A_n)}^{(n)} \bigotimes_{S\in\mathcal{E}} \psi_S(\mathsf{a}_S), \text{ where:} \tag{1}$$

- $H = (\mathcal{V}, \mathcal{E})$ is the multi-hypergraph of the query with $\mathcal{V} = \{A_i\}_{i\in[n]}$.
- Dom is a fixed (output) domain, such as $\{\mathsf{true},\mathsf{false}\}$, $\{0,1\}$, or $\mathbb{R}^+$.
- $\mathcal{V}_{\text{free}} = \{A_1,\ldots,A_f\}$ is the set of result or free attributes; all other attributes are bound.
- For each attribute A_i with $i > f$, $\oplus^{(i)}$ is a binary (aggregate) operator on the domain Dom. Different bound attributes may have different aggregate operators.
- For each attribute A_i with $i > f$, either $\oplus^{(i)}$ is $\otimes$ or $(\mathsf{Dom}, \oplus^{(i)}, \otimes)$ forms a commutative semiring with the same additive identity $\mathbf{0}$ and multiplicative identity $\mathbf{1}$ for all semirings.
- For every hyperedge S in $\mathcal{E}$, $\psi_S : \prod_{A\in S} \mathsf{dom}(A) \to \mathsf{Dom}$ is an (input) function.

FAQs are a semiring generalization of aggregates over join queries, where the aggregates are the operators $\oplus^{(i)}$ and the natural join is expressed by $\bigotimes_{S\in\mathcal{E}} \psi_S(\mathsf{a}_S)$. The listing representation R_{ψ_S} of a function ψ_S is a relation over the schema $S \cup \{\psi_S(S)\}$ which consists of all input-output pairs for ψ_S where the output is non-zero, i.e., R_{ψ_S} contains a tuple $\mathsf{a}_{S\cup\{\psi_S(S)\}}$ if and only if $\psi_S(\mathsf{a}_S) = \mathsf{a}_{\psi_S(S)} \neq \mathbf{0}$. An input database for φ contains for each ψ_S its listing representation. We say that $\mathcal{T}$ is a decomposition of φ if $\mathcal{T}$ is a decomposition of the hypergraph H of φ. Given an FAQ φ and database $\mathbf{D}$, the FAQ-problem is to compute the query result $\varphi(\mathbf{D})$.

Each FAQ φ has an FAQ-width $\mathsf{faqw}(\varphi)$ which is defined similarly to the fractional hypertree width of the hypergraph of φ. For instance, in case where all attributes of φ are free, $\mathsf{faqw}(\varphi)$ is equal to the fractional hypertree width of the hypergraph of φ.

Given an FAQ φ and a database $\mathbf{D}$, the $\mathsf{InsideOut}$ algorithm [18] solves the FAQ-problem as follows. First, it eliminates all bound attributes along with their corresponding aggregate operators by performing equivalence-preserving transformations on φ. Then, it computes the listing representation of the remaining query. The algorithm runs in time $\widetilde{\mathcal{O}}(|\mathbf{D}|^{\mathsf{faqw}(\varphi)} + Z)$ where Z is the size of the output, i.e., the listing representation of φ.

We can compute a cover of the result of a given FAQ φ in time $\widetilde{\mathcal{O}}(|\mathbf{D}|^{\mathsf{faqw}(\varphi)})$, which does not depend on the size of the listing representation of φ. Our strategy is as follows. We first eliminate all bound attributes in φ by using $\mathsf{InsideOut}$ resulting in an FAQ φ'. We then take a decomposition $\mathcal{T}$ of φ' and compute bag functions β_B, $B \in \mathcal{S}(\mathcal{T})$, with $\varphi'(\mathsf{a}_{\mathcal{V}_{\text{free}}}) = \bigotimes_{B\in\mathcal{S}(\mathcal{T})} \beta_B(\mathsf{a}_B)$. Finally, we compute a cover of the join result of the listing representations of the bag functions over the extension of $\mathcal{T}$ that contains, for each bag B, the attribute $\beta_B(B)$ for the values of the function β_B. Keeping the $\beta_B(B)$-values of the bag functions in the cover is necessary for recovering the output values of φ when enumerating the result of φ from the cover.

▶ **Example 29.** We consider the following FAQ φ over the sum-product semiring $(\mathbb{N}, +, \cdot)$ (for simplicity we skip the explicit iteration over the domains of the attributes in φ):

$$\varphi(a,b,d) = \sum_{c,e,f,g,h} \psi_1(a,b,c)\cdot\psi_2(b,d,e)\cdot\psi_3(d,e,f)\cdot\psi_4(f,h)\cdot\psi_5(e,g), \text{ where}$$

φ, ψ_1, ψ_2, ψ_3, ψ_4 and ψ_5 are over $\{A,B,D\}$, $\{A,B,C\}$, $\{B,D,E\}$, $\{D,E,F\}$, $\{F,H\}$ and $\{E,G\}$, respectively. We first run InsideOut on φ to eliminate the bound attributes and obtain the following FAQ:

$$\varphi'(a,b,d) = \underbrace{\big(\sum_c \psi_1(a,b,c)\big)}_{\psi_6(a,b)} \cdot \underbrace{\sum_e \Big(\psi_2(b,d,e) \cdot \underbrace{\sum_f \big(\psi_3(d,e,f) \cdot \underbrace{\sum_h \psi_4(f,h)}_{\psi_7(f)}\big)}_{\psi_9(d,e)} \cdot \underbrace{\sum_g \psi_5(e,g)}_{\psi_8(e)}\Big)}_{\psi_{10}(b,d)}.$$

We consider the decomposition $\mathcal{T}$ of φ' with two bags $B_1 = \{A,B\}$ and $B_2 = \{B,D\}$ and bag functions ψ_6 and respectively ψ_{10}. Then, we execute the cover-join plan $R_{\psi_6} \bowtie R_{\psi_{10}}$ over the extended decomposition $\mathcal{T}'$ with bags $\{A,B,\psi_6(A,B)\}$ and $\{B,D,\psi_{10}(B,D)\}$. While the computation of the result of φ' can take quadratic time, the above cover-join plan takes linear time. We exemplify the computation of the cover-join plan. Assume the following tuples in ψ_6 and ψ_{10}, where $\gamma_1,\ldots,\gamma_4,\delta_1,\ldots,\delta_3 \in \mathbb{N}$:

ψ_6

A	B	$\psi_6(A,B)$
a_1	b_1	γ_1
a_2	b_1	γ_2
a_3	b_2	γ_3
a_4	b_2	γ_4

ψ_{10}

B	D	$\psi_{10}(B,D)$
b_1	d_1	δ_1
b_1	d_2	δ_2
b_2	d_3	δ_3

K

A	B	D	$\psi_6(A,B)$	$\psi_{10}(B,D)$
a_1	b_1	d_1	γ_1	δ_1
a_2	b_1	d_2	γ_2	δ_2
a_3	b_2	d_3	γ_3	δ_3
a_4	b_2	d_3	γ_4	δ_3

The relation K is a possible cover computed by the cover-join plan. The cover carries over the aggregates in columns $\psi_6(A,B)$ and $\psi_{10}(B,D)$, one per bag of $\mathcal{T}'$. The aggregate of the first tuple in K is $\gamma_1 \cdot \delta_1$ (or $\gamma_1 \otimes \delta_1$ under a semiring with multiplication $\otimes$).

The following theorem relies on Lemma 23 and Theorem 25 that give an upper bound on the time complexity for constructing covers of join results.

▶ **Theorem 30.** *For each FAQ φ and database $\mathbf{D}$, a cover of the query result $\varphi(\mathbf{D})$ can be computed in time $\widetilde{\mathcal{O}}(|\mathbf{D}|^{\mathsf{faqw}(\varphi)})$.*

Any enumeration algorithm for covers of join results can be used to enumerate the tuples of an FAQ result from one of its covers. We thus have the following corollary:

▶ **Corollary 31** (Corollary 18)**.** *Given a cover K of the result $\varphi(\mathbf{D})$ of an FAQ φ over a database $\mathbf{D}$, the tuples in the query result $\varphi(\mathbf{D})$ can be enumerated with $\widetilde{\mathcal{O}}(|K|)$ pre-computation time and $\mathcal{O}(1)$ delay and extra space.*

6 Conclusion

Results of join and functional aggregate queries entail redundancy in both their computation and representation. In this paper we propose the notion of covers of query results to reduce such redundancy. While covers can be more succinct than the query results, they nevertheless enjoy desirable properties such as listing representation and constant-delay enumeration of result tuples. For a given database and a join or functional aggregate query, the query result can be normalized as a globally consistent database over an acyclic schema. Covers represent one-relational, lossless, linear-size encodings of such normalized databases.

▶ **Definition 32.** **borged** `/bôrjd/` : Buy One Relation, Get Entire Database!

References

1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2 Mahmoud Abo-Khamis, Hung Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. In-database learning with sparse tensors. In *PODS*, 2018. To appear.
3 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
4 Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013.
5 Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *CSL*, pages 208–222, 2007.
6 Nurzhan Bakibayev, Tomás Kociský, Dan Olteanu, and Jakub Závodnỳ. Aggregation and ordering in factorised databases. *PVLDB*, 6(14):1990–2001, 2013.
7 Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30(3):479–513, 1983.
8 Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.
9 Hugh Darwen, C. J. Date, and Ronald Fagin. A normal form for preventing redundant tuples in relational databases. In *ICDT*, pages 114–126, 2012.
10 Ronald Fagin. Normal forms and relational database operators. In *SIGMOD*, pages 153–160, 1979.
11 Ronald Fagin, Phokion G. Kolaitis, and Lucian Popa. Data exchange: Getting to the core. *ACM Trans. Datab. Syst.*, 30(1):174–210, 2005.
12 Georg Gottlob. Computing cores for data exchange: New algorithms and practical solutions. In *PODS*, pages 148–159, 2005.
13 Georg Gottlob, Zoltán Miklós, and Thomas Schwentick. Generalized hypertree decompositions: NP-hardness and tractable variants. *J. ACM*, 56(6):30:1–30:32, 2009.
14 Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. *ACM Trans. Alg.*, 11(1):4, 2014.
15 Claudio Gutierrez, Carlos Hurtado, and Alberto O. Mendelzon. Foundations of semantic web databases. In *PODS*, pages 95–106, 2004.
16 Pavol Hell and Jaroslav Nešetřil. The core of a graph. *Discrete Mathematics*, 109(1):117–126, 1992.
17 Ahmet Kara and Dan Olteanu. Covers of query results. *CoRR*, abs/1709.01600, 2017. URL: `http://arxiv.org/abs/1709.01600`.
18 Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: questions asked frequently. In *PODS*, pages 13–28, 2016.
19 E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Dover Publications, 2001.
20 Dániel Marx. Approximating fractional hypertree width. *ACM Trans. Alg.*, 6(2):29:1–29:17, 2010.
21 Hung Q. Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Record*, 42(4):5–16, 2013.
22 Dan Olteanu and Maximilian Schleich. Factorized Databases. *SIGMOD Record*, 45(2):5–16, 2016.
23 Dan Olteanu and Jakub Závodnỳ. Size bounds for factorised representations of query results. *ACM Trans. Datab. Syst.*, 40(1):2:1–2:44, 2015.
24 Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning Linear Regression Models over Factorized Joins. In *SIGMOD*, pages 3–18, 2016.
25 Luc Segoufin. Constant delay enumeration for conjunctive queries. *SIGMOD Record*, 44(1):10–17, 2015.

26 Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed SQL database that scales. *PVLDB*, 6(11):1068–1079, 2013.

A From Covers to D-Representations

We next give a brief introduction to d-representations; for a detailed description, we refer the reader to the literature [23]. We then discuss a translation from covers to d-representations.

A.1 D-Representations in a Nutshell

D-representations are a lossless succinct representation for relational data. A d-representation is a set of relational algebra expressions $\{N_1 := E_1, \ldots, N_n := E_n\}$, where each N_i is a unique name and each E_i is a relational algebra expression with unions, Cartesian products, singleton relations, i.e., unary relations with one tuple, and name references in place of singleton relations. The size $|E|$ of a d-representation E is the number of its singletons.

We consider a special class of d-representations that encode results of join queries and whose nesting structure is given by so-called d-trees. In the literature, d-trees are defined as orderings on query variables. We give here an alternative, equivalent definition that is in line with our notion of fractional hypertree decomposition. Given a query Q, a d-tree of Q is a decomposition of Q where each bag is partitioned into one attribute A, called the *bag attribute*, and a set of attributes, called the *key* of A and denoted by $key(A)$. There is one bag per distinct attribute A in Q. Each decomposition $\mathcal{T}$ of a query Q can be translated into a d-tree $\mathcal{T}'$ of Q with $\mathsf{fhtw}(\mathcal{T}') \leq \mathsf{fhtw}(\mathcal{T})$ (Proposition 9.3 in [23]). Given a query Q, a d-tree $\mathcal{T}$ of Q, and a database $\mathbf{D}$, a d-representation E of $Q(\mathbf{D})$ over $\mathcal{T}$ with size $\mathcal{O}(|\mathbf{D}|^{\mathsf{fhtw}(\mathcal{T})})$ can be computed in time $\widetilde{\mathcal{O}}(|\mathbf{D}|^{\mathsf{fhtw}(\mathcal{T})})$ (Theorem 7.13 and Proposition 8.2 in [23]).

▶ **Example 33.** We consider the path query $Q = R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(C, D)$. Figure 2 depicts a database with relations R_1, R_2 and R_3 and the result of Q over the input database $\{R_1, R_2, R_3\}$. It also shows a decomposition $\mathcal{T}$ of Q and a cover K of the query result over $\mathcal{T}$. Finally, it depicts a d-tree $\mathcal{T}'$ (right below) derived from $\mathcal{T}$ by using the translation in the proof of Proposition 9.3 in [23].

D-representations can be encoded as parse graphs and sets of multi-maps. Figure 3 visualizes the two encodings for the d-representation of the query result from Figure 2 over the d-tree $\mathcal{T}'$. The parse graph follows the structure of the d-tree. At the top level we have a union of B-values. Then, given any B-value, the A-values are independent of the values for C and D. Therefore, under each B-value, the A-values are represented in a different branch than the values for C and D. Within the branches for C and D, the values are first grouped by C and then by D. The information on keys is used to share subtrees across branches. Since the key of attribute D is C, all C-nodes with the same value point to the same union of D-values. In our example, both c_1-nodes point to the same set $\{d_1, d_2\}$ of D-values.

The cover K from Figure 2 can be mapped immediately to the parse graph: Under each product node, we take a minimum number of combinations of its children to ensure that every value under the product node occurs in one of these combinations. To enumerate the tuples in the query result, it suffices to choose in turn one branch of each union node and all branches of each product node. For instance, the left product node represents the combinations of $\{a_1, a_2\}$ with $\{d_1, d_2\}$, together with the values b_1 and c_1. There are four combinations, so four tuples in the result. The first two tuples in the cover represent two of them, yet they are sufficient to recover all these tuples.

R_1	
A	B
a_1	b_1
a_2	b_1
a_3	b_2
a_4	b_2

R_2	
B	C
b_1	c_1
b_2	c_1

R_3	
C	D
c_1	d_1
c_1	d_2

$Q(\mathbf{D})$			
A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_1	c_1	d_2
a_2	b_1	c_1	d_1
a_2	b_1	c_1	d_2
a_3	b_2	c_1	d_1
a_3	b_2	c_1	d_2
a_4	b_2	c_1	d_1
a_4	b_2	c_1	d_2

$K \subseteq Q(\mathbf{D})$			
A	B	C	D
a_1	b_1	c_1	d_1
a_2	b_1	c_1	d_1
a_3	b_2	c_1	d_2
a_4	b_2	c_1	d_2

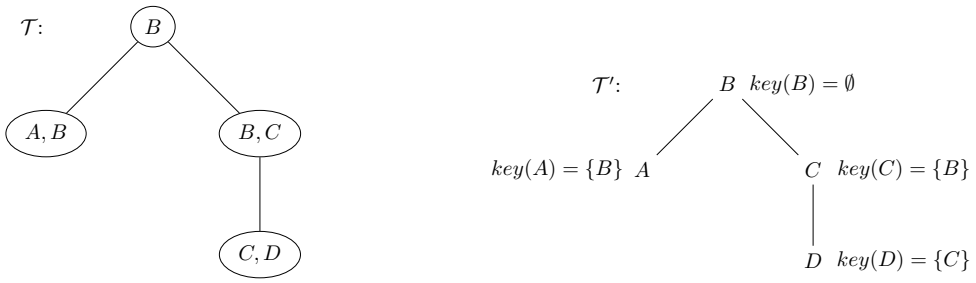
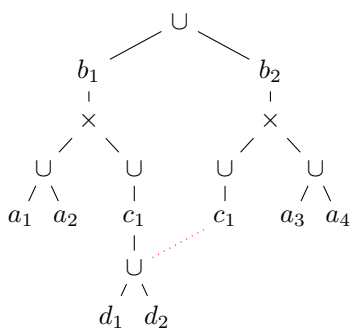


Figure 2 Top row: database $\mathbf{D} = \{R_1, R_2, R_3\}$, the result $Q(\mathbf{D})$ of the path query $Q = R_1 \bowtie R_2 \bowtie R_3$, and a cover $K \subseteq Q(\mathbf{D})$ over the decomposition $\mathcal{T}$; bottom row: decomposition $\mathcal{T}$ of Q and an equivalent d-tree $\mathcal{T}'$.



m_B	
$key(B)$	B
()	b_1
()	b_2

m_A	
$key(A)$	A
b_1	a_1
b_1	a_2
b_2	a_3
b_2	a_4

m_C	
$key(C)$	C
b_1	c_1
b_2	c_1

m_D	
$key(D)$	D
c_1	d_1
c_1	d_2

Figure 3 A d-representation encoded as a parse graph (left) and as a set of multimaps (right).

cover2factorization (cover K, decomposition $\mathcal{T}$)

convert $\mathcal{T}$ into an equivalent d-tree $\mathcal{T}'$ following Proposition 9.3 in [23];
let $\mathbf{V}$ be the set of attributes in $\mathcal{T}'$;
foreach attribute $A \in \mathbf{V}$ **do**
    create multi-map $m_A : \prod_{X \in key(A)} \mathrm{dom}(X) \mapsto \mathrm{dom}(A)$;
foreach tuple $t \in K$ **do**
    foreach attribute $A \in \mathbf{V}$ **do**
        insert assignment $\pi_{key(A)} t \mapsto \pi_A t$ into m_A;
return $\{m_A\}_{A \in \mathbf{V}}$;

Figure 4 Translating a cover K over a decomposition $\mathcal{T}$ into an equivalent d-representation.

The (multi-)map encoding of a d-representation consists of one map for each bag attribute: m_A maps tuples over the attributes in $key(A)$ to values of A. Figure 3 shows these maps as relations with columns for the key attributes (the map keys) and the column for the attribute A itself (the map payload). For instance, $m_A(b_1) = a_1$ and $m_A(b_1) = a_2$, while $m_C(b_1) = c_1$. Since $key(A) = \{B\}$ and there are two B-values in the d-representation leading to the sets $\{a_1, a_2\}$ and $\{a_3, a_4\}$, respectively, m_A maps the B-value b_1 to both A-values a_1 and a_2 and the B-value b_2 to both A-values a_3 and a_4.

A.2 Translating Covers into D-Representations

Figure 4 gives an algorithm that constructs an equivalent d-representation from a cover over a decomposition. Both the cover K and the output d-representation are for the same query result $Q(\mathbf{D})$ of a query Q. The decomposition $\mathcal{T}$ is for the query Q.

The algorithm creates a multi-map for each attribute A and populates it with assignments of tuples over the keys of A to the values of A as encountered in the tuples of the cover.

▶ **Example 34.** We consider the cover K over the decomposition $\mathcal{T}$ in Figure 2 and the d-tree $\mathcal{T}'$ equivalent to $\mathcal{T}$. The cover K is translated into a d-representation over $\mathcal{T}'$ as follows. On the first tuple (a_1, b_1, c_1, d_1), we add $() \mapsto b_1$ to m_B, $b_1 \mapsto a_1$ to m_A, $b_1 \mapsto c_1$ to m_C, and $c_1 \mapsto d_1$ to m_D, where $()$ means the empty tuple. On the second tuple (a_2, b_1, c_1, d_1), we only change m_A by adding $b_1 \mapsto a_2$ to m_A. On the third tuple (a_3, b_2, c_1, d_2), we add the following new assignments: $() \mapsto b_2$ to m_B, $b_2 \mapsto a_3$ to m_A, $b_2 \mapsto c_1$ to m_C, and $c_1 \mapsto d_2$ to m_D. On the last tuple (a_4, b_2, c_1, d_2), we add the new assignment $b_2 \mapsto a_4$ to m_A.

B Cover-Join Plans Computing Covers of Non-Minimum Size

▶ **Example 35.** We consider the acyclic natural join query $Q = R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(C, D)$, the database $\mathbf{D} = \{R_1, R_2, R_3\}$ globally consistent with respect to Q, and the join tree $\mathcal{J} = R_1 - R_2 - R_3$. The relations R_i are depicted below.

R_1		R_2		R_3		K				$K_{1,2}$			K'			
A	B	B	C	C	D	A	B	C	D	A	B	C	A	B	C	D
a_1	b_1	b_1	c_1	c_1	d_1	a_1	b_1	c_1	d_1	a_1	b_1	c_1	a_1	b_1	c_1	d_1
a_2	b_1	b_1	c_2	c_2	d_1	a_2	b_1	c_2	d_1	a_2	b_1	c_1	a_2	b_1	c_1	d_1
a_3	b_1			c_2	d_2	a_3	b_1	c_2	d_2	a_3	b_1	c_2	a_3	b_1	c_2	d_1
													a_3	b_1	c_2	d_2

The relation K is a cover of the query result $Q(\mathbf{D})$ over the decomposition $\mathcal{T}$ corresponding to $\mathcal{J}$. It follows from Proposition 13 that every cover of $Q(\mathbf{D})$ over $\mathcal{T}$ must have size at least three. Hence, K is a minimum-sized cover of $Q(\mathbf{D})$ over $\mathcal{T}$.

We take the cover-join plan $(R_1 \bowtie R_2) \bowtie R_3$ over $\mathcal{J}$ and assume that the cover-join operator computes for each two input relations R and R', a minimum-sized cover of $R \bowtie R'$ over the decomposition with bags $\mathcal{S}(R)$ and $\mathcal{S}(R')$. Then, a possible output of the sub-plan $R_1 \bowtie R_2$ is the relation $K_{1,2}$. A possible result of the cover-join of $K_{1,2}$ with R_3 is the relation K', which is a valid cover of $Q(\mathbf{D})$ over $\mathcal{T}$, but not a minimum-sized cover of $Q(\mathbf{D})$ over $\mathcal{T}$.

C Covers for Equi-Join Queries

In this section, we extend the class of queries from natural join queries to arbitrary equi-join queries, whose relation symbols may map to the same database relation.

Equi-join Queries. An equi-join query, aka full conjunctive query, has the form $Q = \sigma_\psi(R_1(S_1) \times \ldots \times R_n(S_n))$, where each R_i is a relation symbol with schema S_i and ψ is a conjunction of equalities of the form $A_1 = A_2$ with attributes A_1 and A_2. We require that all relation symbols in the query as well as all attributes occurring in the schemas of the relation symbols are distinct. We assume that each query comes with mappings $(\lambda, \{\mu_{R_i}\}_{i\in[n]})$, called the signature mappings of Q, where λ maps the relation symbols in Q to relation symbols in the schema of the database and each μ_{R_i} is a bijective mapping from the attributes of R_i to the attributes of $\lambda(R_i)$. Since we do not require λ to be injective, distinct relation symbols in Q might refer to the same relation in the database (cf. Example 36). The joins in equi-join queries are expressed by the equalities in ψ. The transitive closure ψ^+ of ψ under the equality on attributes defines the attribute equivalence classes: The equivalence class $\mathcal{A}$ of an attribute A is the set consisting of A and of all attributes equal to A in ψ^+. For a set S of attributes, S^+ denotes the set of attributes transitively equivalent to those in S.

The Hypergraph and the hypertree decompositions of Q are defined just like for natural join queries with the additional requirement that each hyperedge or bag is closed with respect to the equivalence classes in ψ^+. More formally, the hypergraph of Q consists of one node A for each attribute A in Q and one edge $\mathcal{S}(R)^+$ for each relation symbol $R \in \mathcal{S}(Q)$. Similarly, a hypertree decomposition $\mathcal{T}$ (of the hypergraph H) of Q is a pair (T, χ), where T is a tree and χ is a function mapping each node in T to a set V^+ where V is a subset of the nodes of H. All other notions and notations introduced in Section 2 as well as the definitions of result preservation and covers in Section 3 carry over to equi-join queries without any change.

▶ **Example 36.** We consider the equi-join query $Q = \sigma_\psi(R_1(A_1, A_2) \times R_2(A_3, A_4))$ where $\psi = \{A_2 = A_3\}$. Let $(\lambda, \{\mu_{R_1}, \mu_{R_2}\})$ be the signature mappings of Q. Assume that $\lambda(R_1) = \lambda(R_2) = R$, $\mu_{R_1}(A_1) = \mu_{R_2}(A_4) = A$ and $\mu_{R_1}(A_2) = \mu_{R_2}(A_3) = B$, i.e., both relation symbols are mapped to the same relation symbol R, attributes A_1 and A_4 are mapped to attribute A and attributes A_2 and A_3 are mapped to attribute B. Let $\mathbf{D} = \{R\}$ where R is defined as in Figure 5. The figure depicts in the top row the query result $Q(\mathbf{D})$, a cover K of the query result over the decomposition $\mathcal{T}$ depicted in the bottom row and two relations R'_1, R'_2 obtained from R by the application of Proposition 37 (given below). The bottom row shows the hypergraph of Q, the hypergraph H of $Q(\mathbf{D})$ over the attribute sets $\{\{A_1, A_2, A_3\}, \{A_2, A_3, A_4\}\}$, and a minimal edge cover M of H with $rel(M) = K$.

R
A B
a_1 b_1
a_2 b_1
a_1 b_2
a_2 b_2

$Q(\mathbf{D})$
A_1 A_2 A_3 A_4
a_1 b_1 b_1 a_1
a_1 b_1 b_1 a_2
a_2 b_1 b_1 a_1
a_2 b_1 b_1 a_2
a_1 b_2 b_2 a_1
a_1 b_2 b_2 a_2
a_2 b_2 b_2 a_1
a_2 b_2 b_2 a_2

$K = rel(M)$
A_1 A_2 A_3 A_4
a_1 b_1 b_1 a_1
a_2 b_1 b_1 a_2
a_1 b_2 b_2 a_2
a_2 b_2 b_2 a_1

R'_1
A_1 A_2 A_3
a_1 b_1 b_1
a_2 b_1 b_1
a_1 b_2 b_2
a_2 b_2 b_2

R'_2
A_2 A_3 A_4
b_1 b_1 a_1
b_1 b_1 a_2
b_2 b_2 a_1
b_2 b_2 a_2

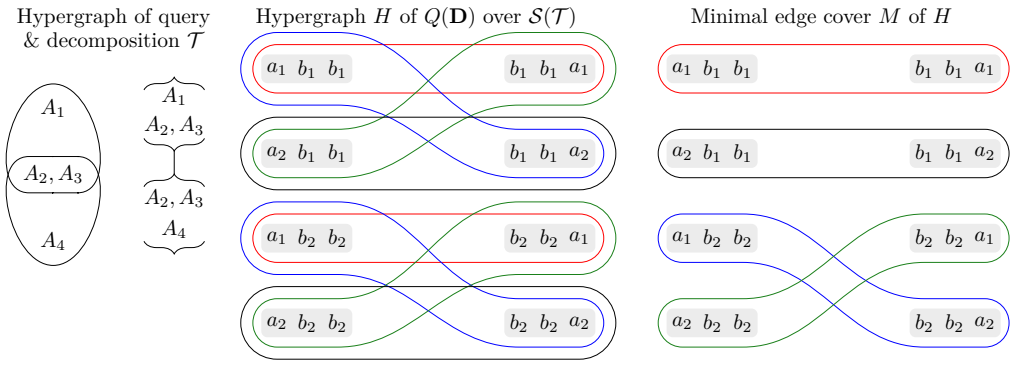


Figure 5 Top row: database $\mathbf{D} = \{R\}$, the result $Q(\mathbf{D})$ of the query Q in Example 36, a cover K of $Q(\mathbf{D})$ over $\mathcal{T}$, and relations R'_1, R'_2 obtained from R by the application of Proposition 37; bottom row: the hypergraph of Q, a decomposition $\mathcal{T}$ of Q, the hypergraph of $Q(\mathbf{D})$ over the attribute sets $\mathcal{S}(\mathcal{T})$, and a minimal edge cover M of this hypergraph.

Adaption of the results on covers to equi-join queries. Due to the following two propositions, all results on covers in Sections 3 and 4 carry over to equi-join queries.

▶ **Proposition 37.** *Given an equi-join query Q, a decomposition $\mathcal{T}$ of Q, and a database $\mathbf{D}$, there exist a natural join query Q' and a database $\mathbf{D}'$ such that: $Q'(\mathbf{D}') = Q(\mathbf{D})$, Q' has the decomposition $\mathcal{T}$ and can be constructed in time $\mathcal{O}(|Q|)$, and $\mathbf{D}'$ can be constructed in time $\mathcal{O}(|\mathbf{D}|)$.*

We briefly explain the construction. The query Q' is obtained from Q by replacing each relation symbol $R(S)$ in Q by a relation symbol $R'(S^+)$. The database $\mathbf{D}'$ contains, for each relation symbol $R'(S^+)$ in Q', a relation over the same schema that is obtained from relation $\lambda(R(S))$ as follows: for each attribute A contained in S^+ but not in S, $\lambda(R(S))$ is extended by a new A-column that is a copy of any B-column in $\lambda(R(S))$ such that A is equivalent to B. Figure 5 gives in the top row two relations R'_1 and R'_2 that result from relation R by the application of Proposition 37 in case Q is defined as in Example 36.

It follows from Proposition 37 that, since $Q'(\mathbf{D}') = Q(\mathbf{D})$, any relation K is a cover of $Q(\mathbf{D})$ over $\mathcal{T}$ if and only if K is a cover of $Q'(\mathbf{D}')$ over $\mathcal{T}$. Given the construction times for Q' and D', all our results on natural join queries in Sections 3 and 4, except the lower size bound on covers in Theorem 14(ii), hold for equi-join queries, too.

The following proposition is the counterpart of Theorem 14(ii) for equi-join queries.

▶ **Proposition 38.** *For each equi-join query Q and decomposition $\mathcal{T}$ of Q, there are arbitrarily large databases $\mathbf{D}$ such that each cover of $Q(\mathbf{D})$ over $\mathcal{T}$ has size $\Omega(|\mathbf{D}|^{\mathsf{fhtw}(\mathcal{T})})$.*

In Proposition 38, we first construct a natural join query Q' from Q as in Proposition 37. By Theorem 14(ii), there are arbitrarily large databases $\mathbf{D}'$ such that each cover of $Q'(\mathbf{D}')$ over $\mathcal{T}$ has size $\Omega(|\mathbf{D}'|^{\mathsf{fhtw}(\mathcal{T})})$. Given such a database $\mathbf{D}'$, it follows from Proposition 13, that $\Sigma_{B\in\mathcal{S}(\mathcal{T})}|\pi_B Q'(\mathbf{D}')| = \Omega(|\mathbf{D}'|^{\mathsf{fhtw}(\mathcal{T})})$, hence, $\max_{B\in\mathcal{S}(\mathcal{T})}\{|\pi_B Q'(\mathbf{D}')|\} = \Omega(|\mathbf{D}'|^{\mathsf{fhtw}(\mathcal{T})})$. We convert the database $\mathbf{D}'$ into a database $\mathbf{D}$ of size $\mathcal{O}(|\mathbf{D}'|)$ such that $|\pi_B Q(\mathbf{D})| \geq |\pi_B Q'(\mathbf{D}')|$ for each $B \in \mathcal{S}(\mathcal{T})$. By Proposition 13 (adapted to equi-join queries), each cover of $Q(\mathbf{D})$ over $\mathcal{T}$ must have size at least $\max_{B\in\mathcal{S}(\mathcal{T})}\{|\pi_B Q(\mathbf{D})|\}$. Since $\max_{B\in\mathcal{S}(\mathcal{T})}\{|\pi_B Q'(\mathbf{D}')|\} = \Omega(|\mathbf{D}'|^{\mathsf{fhtw}(\mathcal{T})})$ and $\max_{B\in\mathcal{S}(\mathcal{T})}\{|\pi_B Q(\mathbf{D})|\} \geq \max_{B\in\mathcal{S}(\mathcal{T})}\{|\pi_B Q'(\mathbf{D}')|\}$, we conclude that each cover of $Q(\mathbf{D})$ over $\mathcal{T}$ is of size $\Omega(|\mathbf{D}'|^{\mathsf{fhtw}(\mathcal{T})}) = \Omega(|\mathbf{D}|^{\mathsf{fhtw}(\mathcal{T})})$.

Distribution Policies for Datalog

Bas Ketsman[1]
Hasselt University, Hasselt, Belgium, and transnational University of Limburg, Belgium/The Netherlands

Aws Albarghouthi
University of Wisconsin-Madison, Madison, WI, USA

Paraschos Koutris
University of Wisconsin-Madison, Madison, WI, USA

Abstract

Modern data management systems extensively use parallelism to speed up query processing over massive volumes of data. This trend has inspired a rich line of research on how to formally reason about the parallel complexity of join computation. In this paper, we go beyond joins and study the parallel evaluation of recursive queries. We introduce a novel framework to reason about multi-round evaluation of Datalog programs, which combines implicit *predicate restriction* with *distribution policies* to allow expressing a combination of data-parallel and query-parallel evaluation strategies. Using our framework, we reason about key properties of distributed Datalog evaluation, including parallel-correctness of the evaluation strategy, disjointness of the computation effort, and bounds on the number of communication rounds.

2012 ACM Subject Classification Information systems → Query languages, Information systems → Parallel and distributed DBMSs

Keywords and phrases Datalog queries, Distributed evaluation, Distribution policies

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.17

1 Introduction

Modern data management systems – such as Spark [27, 33], Hadoop [16, 11], and others [17] – have extensively used parallelism to speed up query processing over massive volumes of data. Parallelism enables the distribution of computation into multiple machines, and thus significantly reduces the completion time for several critical data processing tasks. This trend has inspired a rich line of research on how to formally reason about the parallel complexity of join computation, one of the core tasks in massively parallel systems. Several papers [7, 8, 20, 19] have studied the tradeoff between synchronization (number of rounds) and communication cost, and have proposed and analyzed known and new parallel algorithms [4, 9]. Among these, the *Hypercube algorithm* [13, 4] can compute any multiway join query in one round by properly distributing the input data.

To reason about Hypercube-like algorithms, Ameloot et al. [6] recently introduced a framework that captures one-round evaluation of joins under different data distributions. Their framework implicitly describes a single-round parallel algorithm through a *distribution policy*, which specifies how the facts in the input relations are distributed among the machines. While for non-recursive queries a distribution policy defines a scalable parallel evaluation strategy, for Datalog programs this is typically not the case. For instance, a simple transitive

[1] PhD Fellow of the Research Foundation – Flanders (FWO)

closure query already requires for entire components of the input database that all facts must reside on the same server to ensure correctness of the computation.

To reason about Datalog evaluation in a distributed setting, we introduce a general theoretical framework that allows a combination of data and query parallelization strategies. The central concept in this framework is the notion of an *economic policy*. Our key observation is that, in order to deal with intensional predicates, we need to specify not only where a fact must be located to be *consumed* by a rule, but also where a fact must be *produced* by evaluating a rule of the program. An economic policy in our framework is defined as a pair of distribution policies: a *consumption policy*, which specifies the location of the facts that are used in the body of rules, and a *production policy*, which specifies the location of facts that appear in the head of a rule. The evaluation strategy that is implicitly defined by the data distribution must communicate any produced facts to the machines where they will be consumed, and thus can run over multiple rounds.

Our framework is inspired by a rich line of research on parallel evaluation strategies for Datalog programs from the early 90's [30, 13, 31, 34]. There, Datalog evaluation strategies are based on the idea of partitioning the instantiations of the program rules among machines by adding conditions to the bodies of the rules, called program restrictions. Some of the strategies proposed require no communication of intermediate (intensional) facts and thus can be completed in one round; other strategies require communication over multiple rounds. We show that an economic policy can capture several algorithms used for parallel evaluation of recursive and non-recursive queries, including the Hypercube algorithm [13, 4], and the decomposable strategies based on program restrictions [30].

In this framework we study several properties of economic policies. We first explore the property of *parallel-correctness*: when does an economic policy lead to a correct evaluation strategy? As can be expected, it is undecidable to show parallel-correctness for a general Datalog program, even for the simplest of economic policies. We therefore identify a sufficient condition: every minimal valuation of a rule must be *supported* by the policy. A rule valuation is supported if some machine consumes all the facts in the body, and produces the fact in the head. For unions of conjunctive queries, this condition is also necessary, recovering the result of Ameloot et al. [6]; however, we show that even for non-recursive programs with intermediate relations, the condition is no longer required. To overcome the undecidability of parallel-correctness, we identify a general family of economic policies, called *Generalized Hypercube Policies (GHPs)*, which are always parallel-correct, and further capture several commonly used parallel evaluation strategies.

Second, we study the property of *boundedness*: can we decide whether a given economic policy terminates in k rounds, independent of the input size? We show that there exists a sharp increase in complexity as we move from $k = 1$ to $k \geq 2$. For $k = 1$, we can succinctly characterize the structure of a policy that always terminates in one step. Additionally, given a GHP, we can do this in polynomial time in the description of the GHP. On the other hand, for $k \geq 2$ it is undecidable to determine whether it terminates in at most k steps, even for a GHP. We then ask which Datalog programs admit economic policies that are bounded by one round: we show that such programs are characterized by a syntactic property called *pivoting*, which was also identified by Wolfson and Silberschatz [32] in the context of decomposable programs.

2 Related Work

2.1 Parallel Complexity

The parallel complexity of Datalog was first investigated by Cosmadakis and Kanellakis [10, 18]. Later work used the complexity class NC to theoretically capture which Datalog programs are efficiently parallelizable. Since Datalog evaluation is P-complete, it is unlikely that every Datalog program belongs in NC, which implies that certain Datalog programs may not be significantly sped up through parallelism. Ullman and Van Gelder [28] showed that if a Datalog program has the *polynomial fringe property*, which says that every fact in the output has a proof tree of polynomial size, evaluation is in NC. Every linear Datalog program has the polynomial fringe property and is thus in NC. Afrati and Papadimitriou [3] showed that for simple chain queries (including non-linear queries) evaluation is either in NC or P-complete. Recently, Afrati and Ullman [5] studied the tradeoff between communication and number of rounds. They describe a very restricted class of Datalog programs where it is possible to reduce the number of recursion steps (to a number that is logarithmic in the size of the input) without significantly increasing the communication cost.

2.2 Decomposability

The concept of predicate decomposability was first introduced by Wolfson and Silberschatz [32]. A predicate T is decomposable if there are $r > 1$ restricted copies $P_1, P_2, \ldots, P_r$ of the Datalog program P (using arithmetic predicates) such that (i) the copies compute a partition of T for every input, and (ii) there exists an input instance where each copy will produce some input for T. The main result is that decomposability is equivalent to pivoting for sirups where there are no constants, no repeating variables, and the sirup is linear or a simple chain rule. Here, a *sirup* is a Datalog program with one IDB predicate S and two rules: (i) a base rule $S(\boldsymbol{x}) \leftarrow B(\boldsymbol{x})$, and (ii) a recursive rule with head predicate S. A sirup is *linear* if S appears exactly once in the body of the recursive rule.

Later works [30, 31] redefine the concept of decomposability semantically. A Datalog program is *decomposable* if it is possible to partition the output domain (to at least two blocks) such that for every instance I, every output fact has a proof tree where all the IDB facts belong in the same partition block. Wolfson and Ozen [31] show that deciding whether a given Datalog program is decomposable is undecidable. Cohen and Wolfson [30] provide necessary and sufficient syntactic conditions for decomposability for sirups where the arity of the IDB predicate is ≤ 2. They also define the notion of *strongly decomposable* sirups, where the partition must guarantee that, for some input, at least two blocks will produce a fact using the recursive rule of the sirup. Following the same line of work, Zhang et al. [34] present a more general framework that constructs partitionings of the rule instantiations.

2.3 Other Parallel Schemes

In addition to decomposability, several frameworks for parallel recursive processing were introduced in the early 90s [30, 13, 31]. Wolfson [30] generalizes decomposability to *load sharing schemes*, by allowing the output of a predicate to have overlap in the copies of the program P. Under a load sharing scheme, every linear program can be parallelized, even if it is not pivoting. In [13, 14, 31], general schemes are introduced that parallelize the evaluation by partitioning the set of rule instantiations, and allowing for communication among the machines (decomposable and load sharing schemes need no communication). Dewan et al. [12] proposes similar techniques with dynamic adjustments, to balance the load

of a computation. Our framework differs in that the set of rule instantiations is distributed implicitly among the servers, according to the production and consumption policies, and that the communication between servers is made explicit.

2.4 Systems

Recent work studies the implementation of Datalog (or fragments of Datalog) on modern shared-nothing distributed systems. Seo et al. [24] present a distributed version of a Datalog variant for social network analysis called Socialite; however, their framework requires that the user provides annotations to guide the distribution of data. Wang et al. [29] implement a variant of Datalog on the Myria system [17], focusing mostly on asynchronous evaluation and fault-tolerance. The BigDatalog system [26] describes an implementation of Datalog on Apache Spark, but focuses mostly on linear Datalog programs that use aggregation. The task of parallelizing Datalog has also been studied in the context of the popular MapReduce framework [2, 5, 25]. Motik et al. [22] provide an implementation of parallel Datalog in main-memory multicore systems.

3 Preliminaries

We assume an infinite domain **dom**. A database schema σ is a finite set of relation names $\{R_i\}_{i=1}^{n}$ with associated arities $ar(R_i)$. We shall write $R^{(k)}$ to denote a relation R with arity k. A fact $R(a_1, \ldots, a_k)$ over $U \subseteq \mathbf{dom}$ is a tuple consisting of a relation name and a sequence of values from U. We say that $R(a_1, \ldots, a_k)$ is *over* schema σ, if $R^{(k)} \in \sigma$. For a universe $U \subseteq \mathbf{dom}$ and schema σ, we denote by $\mathsf{facts}(\sigma, U)$ the complete set of facts over σ and U. An *instance* I over σ and U is defined as a finite subset of $\mathsf{facts}(\sigma, U)$. We write $I_{|\sigma}$ to denote the subset of I containing all facts in I that are over schema σ.

For $i \in \mathbb{N}$, we abbreviate the set $\{1, \ldots, i\}$ by $[i]$.

3.1 Datalog

We assume an infinite domain of variables **var**, disjoint from **dom**. An *atom* is a formula $R(t_1, \ldots, t_k)$ consisting of a relation name and a tuple of terms; a *term* t_i is either a variable from **var** or a constant from **dom**.

A *Datalog rule* τ is of the form $R(\boldsymbol{x}) \leftarrow S_1(\boldsymbol{y_1}), \ldots, S_n(\boldsymbol{y_n})$, where $R(\boldsymbol{x})$ is a single atom called the head of τ, denoted $head_\tau$, and all $S_i(\boldsymbol{y_i})$ are atoms called body atoms of τ, denoted $body_\tau$. We say that $S_i(\boldsymbol{y_i})$ is *over* schema σ, when $S_i \in \sigma$ and $k = ar(S_i)$. We say that τ is over schema σ if all its atoms are. We assume that Datalog rules are always *safe*, i.e., that all variables in the head occur in at least one body atom. By $vars(\tau)$ we denote the set of variables in rule τ.

A *Datalog program* P is a finite set of Datalog rules. A program P is said to be over schema σ if all its rules are. Particularly, by $\textsc{edb}(P) \subseteq \sigma$ we denote the relation names occurring only in the body of rules, and by $\textsc{idb}(P) \subseteq \sigma$ all other relation names occurring in P. We further distinguish the names in $\textsc{idb}(P)$ by calling some of them *output relations*, denoted $out(P) \subseteq \textsc{idb}(P)$; all other IDB relations are *auxiliary*. We write $\sigma(P)$ to denote $\textsc{edb}(P) \cup \textsc{idb}(P)$.

Consider the directed graph where each node is an IDB predicate, and there is an edge from S to S' if S' occurs in the head of some rule τ of P, and S in the body of τ. We say that P is *recursive* if the graph is cyclic; otherwise, we say it is *non-recursive*. A non-recursive Datalog program with only one rule is called a *conjunctive query* (CQ).

3.2 Evaluation Semantics

We define the evaluation semantics of Datalog programs as usual, through the immediate consequence operator. Let P be a Datalog program and I an instance over $\text{EDB}(P)$. A *valuation* v for rule $\tau \in P$ is a constant-preserving mapping of the terms in τ to values in $\mathbf{dom}$. For a rule $\tau \in P$ and valuation v, we say that τ derives fact $v(head_\tau)$ over instance I if $v(body_\tau) \subseteq I$. We refer to $v(\tau)$ as the instantiation of rule τ with valuation v.

We use T_P to denote the *immediate consequence operator* for P, which applies all rules in P exactly once over a given instance and adds all derived facts to that instance. Formally, $T_P(I) = I \cup \{v(head_\tau) \mid \tau \in P, \text{valuation } v \text{ s.t. } v(body_\tau) \subseteq I\}$. Then, $P(I)$ is defined as the fixpoint reached after iteratively applying the immediate consequence operator over I. It is not difficult to see that T_P is monotone, and thus always reaches a fixpoint after a finite number of iterations. Moreover, the output of the query that P computes is defined as $P(I)_{|out(P)}$. We refer to Abiteboul et al. [1] for a detailed description.

We call a fact $\boldsymbol{f}$ *P-derivable* if $\boldsymbol{f} \in P(I)$ for some instance I, and *P-consumable* if during the evaluation of P on some instance I a rule instantiation $v(\tau)$ fires that requires $\boldsymbol{f}$. Both notions naturally generalize to atoms and predicate symbols, e.g., predicate symbol R is said to be P-consumable if some P-consumable fact $\boldsymbol{f}$ exists with symbol R. Atom A is P-consumable if a rule instantiation as above exists, with $A \in body_\tau$.

3.3 Proof Theoretic Concepts

Let $T = (V, E)$ be a tree. By $fringe_T$ we denote its leaves and by $root_T$ its root vertex. All other vertices are called internal vertices. For a vertex $n \in V$ we denote by $children_T(n)$ the set of child vertices of n in T. We now recall the classical notion of proof tree [1]. A *proof tree* T for a fact $\boldsymbol{f}$ on instance I and Datalog program P is a tree T with vertices over $\mathsf{facts}(\sigma(P), \mathbf{dom})$, where $fringe_T \subseteq I$, $root_T = \boldsymbol{f}$, and for every internal vertex $\boldsymbol{g}$, there is a rule $\tau \in P$ and valuation v such that $\boldsymbol{g} = v(head_\tau)$ and $children_T(\boldsymbol{g}) = v(body_\tau)$. In this case, we shall say that T *uses* the instantiation of τ with valuation v. It is easy to see that $P(I)$ consists of exactly those facts $\boldsymbol{f}$ for which a proof tree for $\boldsymbol{f}$ on I and P exists. We say that a rule instantiation $v(\tau)$ is *useless* if $v(head_\tau) \in v(body_\tau)$; otherwise, we say that it is *useful*. W.l.o.g. we will consider only proof trees where all rule instantiations are useful.

We say that a proof tree T' is *entailed* by proof tree T for P, denoted $T' \sqsubseteq T$, if $fringe_{T'} \subseteq fringe_T$ and $root_{T'} = root_T$.

4 The Framework

Our framework considers a setting with p *servers* (or *machines*) that share no memory and can communicate only via messages – this is commonly referred to as a *shared-nothing* parallel architecture. The set of servers forms a *network* $[p]$ that we assume is fully connected. In order to define how computation is performed, we will use policies that specify how the data (input and output facts) are distributed over the network. We borrow the definition of a distribution policy from [6]:

▶ **Definition 4.1** (Distribution Policy). A *distribution policy* $\boldsymbol{P} = (U, \mathsf{facts}_{\boldsymbol{P}})$ over schema σ and network $[p]$ consists of a universe $U \subseteq \mathbf{dom}$ and a function $\mathsf{facts}_{\boldsymbol{P}} : [p] \to 2^{\mathsf{facts}(\sigma, U)}$ mapping servers to sets of facts over U and σ.

Distribution policies are instance independent, *i.e.*, they are oblivious of the specific database instance. Intuitively, a policy expresses on which servers a fact should reside if

the fact is in the network, but not whether the fact is in the network. Henceforth, we slightly abuse notation and write $\boldsymbol{P}(\boldsymbol{f})$ to denote the set of servers *responsible* for $\boldsymbol{f}$, *i.e.*, $\boldsymbol{P}(\boldsymbol{f}) = \{i \mid \boldsymbol{f} \in \mathsf{facts}_{\boldsymbol{P}}(i)\}$.

In contrast to [6], where the focus is on single-round query evaluation and policies that define only the initial data distribution over EDB facts, we consider a multi-round setting that allows the communication of intermediate facts.

▶ **Definition 4.2** (Economic Policy)**.** An *economic policy* $\boldsymbol{E}$ over schema σ and network $[p]$ is a pair $(\boldsymbol{P}, \boldsymbol{C})$ of distribution policies over the same universe U, where:

- $\boldsymbol{P}$ is defined over $\text{IDB}(P)$ and is called the *production policy*; and
- $\boldsymbol{C}$ is defined over $\text{EDB}(P) \cup \text{IDB}(P)$ and is called the *consumption policy*.

A production policy describes which machines have the responsibility of producing a certain IDB fact. A consumption policy describes which machines need an EDB or IDB fact to satisfy the body of a rule instantiation. We sometimes make universe U explicit, by writing $(\boldsymbol{P}, \boldsymbol{C}; U)$ rather than $(\boldsymbol{P}, \boldsymbol{C})$.[2] We say that a fact $\boldsymbol{f}$ is $\boldsymbol{C}$*-consumable* if $\boldsymbol{C}(\boldsymbol{f}) \neq \emptyset$.

A *family* of economic policies $\mathcal{F}$ is a set of economic policies over a common universe and schema. We say that a family $\mathcal{F}$ satisfies property $\mathcal{P}$ if all the policies in $\mathcal{F}$ satisfy $\mathcal{P}$.

4.1 Datalog Evaluation Modulo Policies

Instead of letting a server compute the full program over its local instance, we restrict the evaluation process based on a server's economic policy. That is, for economic policy $\boldsymbol{E} = (\boldsymbol{P}, \boldsymbol{C})$ and Datalog program P, the following sequential evaluation algorithm takes place on server i:

- First, every rule $\tau \in P$ is annotated with policy-predicates as follows. For the head $R(\boldsymbol{x})$, we add a predicate $\mathit{Policy}_R(\boldsymbol{x})$ to the body of τ. Here, predicate Policy_R refers to relation $\mathsf{facts}_{\boldsymbol{P}}(i)_{|\{R\}}$.
- Second, for every atom $S(\boldsymbol{y})$ in the body of τ, we add the predicate $\mathit{Policy}_S(\boldsymbol{y})$, where now Policy_S refers to the relation $\mathsf{facts}_{\boldsymbol{C}}(i)_{|\{S\}}$.

The added predicates may be infinitely large, but can be accessed through queries of the form "$\boldsymbol{t} \in \mathsf{facts}_{\boldsymbol{P}}(i)_{|\{R\}}$?" or "$\boldsymbol{t} \in \mathsf{facts}_{\boldsymbol{C}}(i)_{|\{S\}}$?".

Throughout the paper we assume the semi-naive evaluation strategy for Datalog programs. Semi-naive evaluation proceeds as usual over the annotated program: after each application of the fixpoint operator, the newly derived facts are added to a delta relation, and a rule instantiation is triggered only if at least one of its facts is in the delta relation from the previous iteration. We denote by $P_{\restriction \boldsymbol{E}}(I, J)$ the fixpoint instance when we execute P restricted to $\boldsymbol{E}$ on input I, with delta relations initialized with J.

4.2 Distributed Evaluation Strategy

We now present how an economic policy induces a parallel evaluation strategy. Our parallel model is the BSP-based Massively Parallel Communication Model (MPC) [21]. In this model, computation is performed over servers in a multi-round fashion. Each round has two distinct phases: a local computation phase, and a synchronized communication phase.

Consider a Datalog program P, a network $[p]$, and an economic policy $\boldsymbol{E} = (\boldsymbol{P}, \boldsymbol{C})$. Moreover, let I be the input instance, which we initially assume to be partitioned arbitrarily

[2] Notice that mentioning U is redundant, but allows a slightly simpler notation, since $\boldsymbol{P}$ and $\boldsymbol{C}$ need not be specified explicitly to reference their universe U.

over the p servers. Denote by I_i the initial local instance of machine i. Let local_i^k be the instance on machine i right after the k-th communication phase.

We consider the following procedure: Initially, we set $\mathsf{local}_i^0 \leftarrow I_i$. Then, at the k-th round (for $k \geq 1$), we perform the following:

1. *Communication:* Every machine sends its facts as defined by the consumption policy $\boldsymbol{C}$. That is, server i sends local fact $\boldsymbol{f} \in \mathsf{local}_i^{k-1}$ to server j if (and only if) $\boldsymbol{f} \in \mathsf{facts}_{\boldsymbol{C}}(j)$. Let rec_i^k be the facts received by machine i during the k-th communication phase.[3]
2. *Computation:* Every server computes the local fixpoint: if $k = 1$, then $\mathsf{local}_i^k = P_{\restriction \boldsymbol{E}}(\mathsf{rec}_i^k \cup \mathsf{local}_i^{k-1}, \mathsf{rec}_i^k \cup \mathsf{local}_i^{k-1})$, otherwise $\mathsf{local}_i^k = P_{\restriction \boldsymbol{E}}(\mathsf{rec}_i^k \cup \mathsf{local}_i^{k-1}, \mathsf{rec}_i^k \setminus \mathsf{local}_i^{k-1})$.

Intuitively, the algorithm terminates when, after a round is finished, for every server all locally derived facts that need to be sent to some other server according to the consumption policy, were already sent to these servers in an earlier round.

Formally, for server i, we define set $F_i = \{\boldsymbol{f} \mid \boldsymbol{C}(\boldsymbol{f}) \setminus i \neq \emptyset\}$. Intuitively, F_i represents all facts consumed by servers other than i itself. We say that a server has reached a *local fixpoint state* for $\boldsymbol{E}$ and P after round $k \geq 1$, if $\mathsf{local}_i^k \cap F_i \subseteq \mathsf{local}_i^{k-1}$. We say that the network $[p]$ has reached a *global fixpoint state* for $\boldsymbol{E}$ and P after round k, if all servers $i \in [p]$ have reached a local fixpoint state after round k. Notice that this condition is as desired, because every round goes into the communication phase first, then into the local computation phase.

One could imagine a smarter communication procedure that incorporates Datalog semantics as well. For example, a server does not need to send a local fact $\boldsymbol{f} \in \mathsf{facts}_{\boldsymbol{C}}(j)$ to server j if for every input I server j is guaranteed to already have $\boldsymbol{f}$ in its local instance. However, it is in general undecidable to make such a decision (see Lemma 5.3).

For instance I, let $[P, \boldsymbol{E}](I)$ denote the union of all facts over $out(P)$ found at any server after reaching the global fixpoint. Notice that the above evaluation strategy always reaches a fixpoint, due to monotonicity of Datalog.

▶ **Example 4.3.** Consider the left-linear Datalog program that computes transitive closure:

$$T(x, y) \leftarrow R(x, y). \qquad T(x, y) \leftarrow T(x, z), R(z, y).$$

For any function $h : \mathbf{dom} \to [p]$, we define the economic policy $(\boldsymbol{P}_1, \boldsymbol{C}_1)$, where $\boldsymbol{C}_1(R(a, b)) = [p]$, and $\boldsymbol{C}_1(T(a, b)) = \boldsymbol{P}_1(T(a, b)) = \{h(a)\}$ for all $a, b \in \mathbf{dom}$. This policy works as follows: it replicates the EDB facts everywhere, and then produces/consumes each fact $T(a, b)$ at machine $h(a)$. It is easy to see that the economic policy correctly computes the transitive closure. In fact, the evaluation always terminates in a single round.

Consider a different policy $(\boldsymbol{P}_2, \boldsymbol{C}_2)$, which again takes any function $h : \mathbf{dom} \to [p]$ and has $\boldsymbol{C}_2(R(a, b)) = \{h(a)\}$, $\boldsymbol{C}_2(T(a, b)) = \{h(b)\}$, and $\boldsymbol{P}_2(T(a, b)) = [p]$. This policy does not replicate the EDB facts, but it hash-partitions them according to the first attribute. Whenever a machine discovers a new fact, the new fact has to be consumed to the location determined by the hash of the second attribute. Observe that the production policy is $[p]$ because we do not know where each fact will be produced (in other words, each machine will produce as many IDB facts as possible from its local input without any restrictions).

We will see later in Section 6 that all the above economic policies belong in a specific family of policies that we call Generalized Hypercube Policies (GHPs). We notice that our framework supports evaluation strategies that are *oblivious* of the instance: each fact is

[3] We remark that from a practical viewpoint it makes no sense to communicate the same facts more than once. When $j = i$, no actual communication takes place.

communicated, consumed, and produced independent of whether other facts are in the same local instance or not. Lastly, we note that monotonicity of Datalog ensures monotonicity of economic policies for Datalog Programs.

▶ **Proposition 4.4.** *For every Datalog program P and economic policy $\boldsymbol{E}$ for P, $\boldsymbol{f} \in [P, \boldsymbol{E}](I')$ implies $\boldsymbol{f} \in [P, \boldsymbol{E}](I)$, for all $I' \subseteq I$. More specifically, if $\boldsymbol{f}$ is derived by $\boldsymbol{E}$ for I' in round i on server s, then $\boldsymbol{f}$ is derived by $\boldsymbol{E}$ for I in round $j \leq i$ on server s.*

5 Parallel-Correctness

An economic policy for a Datalog program does not necessarily lead to the desired output. For example, if the production policy maps every fact onto the empty set of servers, then the execution will generate only empty IDB relations. Henceforth, we are only interested in economic policies that generate the expected output.

▶ **Definition 5.1** (Parallel-correctness)**.** An economic policy $\boldsymbol{E} = (\boldsymbol{P}, \boldsymbol{C}; U)$ is *parallel-correct* for Datalog program P if $[P, \boldsymbol{E}](I) = P(I)|_{out(P)}$, for every instance $I \subseteq \mathsf{facts}(\text{EDB}(P), U)$.

Parallel-correctness is in general undecidable, even for simple classes of policies. For instance, consider the class of policies, where $\boldsymbol{P}(\boldsymbol{f}_1) = \boldsymbol{P}(\boldsymbol{f}_2)$ and $\boldsymbol{C}(\boldsymbol{f}_1) = \boldsymbol{C}(\boldsymbol{f}_2)$, whenever $\boldsymbol{f}_1, \boldsymbol{f}_2$ are facts with same relation symbol. We call this class of policies *value-independent*, denoted $\mathcal{E}_{indep}$, since the facts are mapped to machines only according to the relations they belong to. Value-independent policies allow a succinct representation by simply enumerating the IDB predicates of P and the subsets of $[p]$ where each relation is assigned.

We consider the following decision problem.

	PC$(\mathcal{L}, \mathcal{E})$
Input:	Program $P \in \mathcal{L}$, policy $\boldsymbol{E} \in \mathcal{E}$.
Question:	Is $\boldsymbol{E}$ parallel-correct for P?

▶ **Theorem 5.2.** PC$(Datalog, \mathcal{E}_{indep})$ *is undecidable.*

The proof is given in Appendix A.1. We next show an even stronger result:

▶ **Lemma 5.3.** *Let P be an arbitrary Datalog program and $\boldsymbol{E} = (\boldsymbol{P}, \boldsymbol{C}; U)$ an economic policy over σ that is parallel-correct for P. Now let $\boldsymbol{f} \in \mathsf{facts}(\sigma, U)$, and $\boldsymbol{C}'$ the consumption policy where $\boldsymbol{C}'(\boldsymbol{g}) = \boldsymbol{C}(\boldsymbol{g})$ for all $\boldsymbol{g} \in \mathsf{facts}(\sigma, U) \setminus \{\boldsymbol{f}\}$ and $\boldsymbol{C}'(\boldsymbol{f}) \subsetneq \boldsymbol{C}(\boldsymbol{f})$. It is still undecidable whether $\boldsymbol{E}'$ is parallel-correct for P.*

Despite the above results, we can present some syntactic conditions that are necessary for parallel-correctness, and some that are sufficient.

▶ **Definition 5.4** (Support)**.** An instantiation of rule τ with valuation v is *supported* by economic policy $\boldsymbol{E} = (\boldsymbol{P}, \boldsymbol{C})$ if there exists some machine $s \in [p]$ with $v(head_\tau) \in \mathsf{facts}_{\boldsymbol{P}}(s)$ and $v(body_\tau) \subseteq \mathsf{facts}_{\boldsymbol{C}}(s)$.

We consider various categories of economic policies based on which rule instantiations are supported for a given Datalog program P:

$\mathcal{N}_P^{all}$: the set of all rule instantiations of P.

$\mathcal{N}_P^{min}$: the set of all minimal rule instantiations of P. An instantiation of rule τ with valuation v is *minimal* if there is no rule τ' and valuation v' with $v'(head_{\tau'}) = v(head_\tau)$ and $v'(body_{\tau'}) \subsetneq v(body_\tau)$.

$\mathcal{N}_P^{use}$: the set of all rule instantiations of P that are useful. Recall that an instantiation of rule τ with valuation v is useful if $v(head_\tau) \notin v(body_\tau)$.

$\mathcal{N}_P^{ess}$: the set of all essential rule instantiations of P. An instantiation of rule τ with valuation v is *essential* if for some P-derivable fact $\boldsymbol{f}$ and instance I, every proof tree T for $\boldsymbol{f}$ on I and P has a vertex $\boldsymbol{g}$ with $\boldsymbol{g} = v(head_\tau)$ and $v(body_\tau) \subseteq children_T(\boldsymbol{g})$.

If the program is non-recursive, then $\mathcal{N}_P^{use} = \mathcal{N}_P^{all}$, since there will be no rule that contains the same relation in the head and the body. We also have:

▶ **Proposition 5.5.** *For every Datalog program P, we have $\mathcal{N}_P^{ess} \subseteq \mathcal{N}_P^{min} \cap \mathcal{N}_P^{use}$.*

The proof is in Appendix A.2. The following example demonstrates the different types of rule instantiations.

▶ **Example 5.6.** Let P be the left-linear transitive closure program from Example 4.3; consider a rule instantiation of the recursive rule: $T(a,b) \leftarrow T(a,c), R(c,b)$, for some (not necessarily different) constants a, b, c. We distinguish the following cases:

$c = a$: in this case, the instantiation is not minimal, since we can derive the same head fact from the instantiation $T(a,b) \leftarrow R(a,b)$ of the first rule.

$c = b$: in this case, the instantiation is useless, since $T(a,b)$ also belongs in the body. In some sense, this derivation is unnecessary, as we have already "discovered" the head fact.

$c \neq a, c \neq b$: in this case, the instantiation is minimal and useful; it is also essential. To show this, consider the instance $I = \{R(a,c), R(c,b)\}$, and the fact $\boldsymbol{f} = T(a,b)$. Because $c \notin \{a,b\}$, the only proof tree for $\boldsymbol{f}$ without "useless" rule instantiations is the one with root $\boldsymbol{f}$, children $T(a,c), R(c,b)$, where $T(a,c)$ has $R(a,c)$ as child.

Depending on which types of rule instantiations are supported by an economic policy, we can define different types of policies. An economic policy that supports all possible rule instantiations, that is, $\mathcal{N}_P^{all}$, is said to be *strongly supporting* for Datalog program P.

▶ **Proposition 5.7.** *Let P be a Datalog program and $\boldsymbol{E}$ an economic policy. If $\boldsymbol{E}$ supports all minimal and useful rule instantiations in P, then it is parallel-correct. If $\boldsymbol{E}$ is parallel-correct for P, then it supports all essential rule instantiations.*

▶ **Proposition 5.8.** *Let P be a Datalog program where each* IDB *predicate occurs only in the head of rules (i.e., P is a union of CQs). Then, $\mathcal{N}_P^{ess} = \mathcal{N}_P^{min} \cap \mathcal{N}_P^{use}$.*

Proofs are given in Appendix A.3 and A.4.

Together with Proposition 5.7, the above proposition implies that a Datalog program where the body of each rule contains only EDB relations is parallel-correct if and only if it supports every minimal rule instantiation, or equivalently if and only it supports every essential rule instantiation. Notice that this class of Datalog programs corresponds to a program that computes a set of UCQs, and thus the above result captures the characterization of parallel-correctness for CQs and UCQs in [6, 15]. We should emphasize here that [6, 15] consider only economic policies where $\boldsymbol{P}$ assigns every fact to every server, while a general economic policy can assign facts to any subset of servers.

For general Datalog programs, $\mathcal{N}_P^{ess} = \mathcal{N}_P^{min} \cap \mathcal{N}_P^{use}$ is not true anymore, and thus supporting essential instantiations is not a sufficient condition for parallel-correctness, even if P is non-recursive. (Recall that non-recursiveness is a syntactic condition, and that all such programs are straightforwardly rewritable to UCQs.)

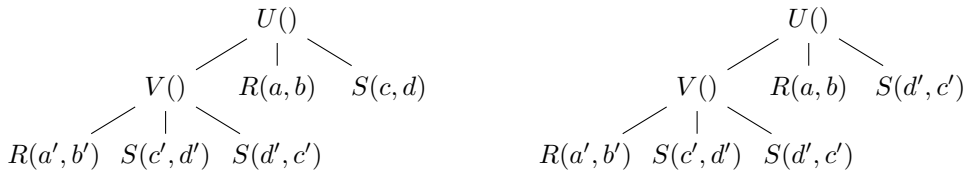


Figure 1 The two proof trees for the fact $\boldsymbol{f} = U(a, b)$.

▶ **Example 5.9.** Consider the following non-recursive Datalog program P:

$$V() \leftarrow R(x, y), S(z, w), S(w, z). \qquad U() \leftarrow V(), R(x, y), S(z, w).$$

and take the rule instantiation with head $U()$ and body $\{V(), R(a, b), S(c, d)\}$. Assume that $c \neq d$. This rule instantiation is minimal, but we will show that it is not essential.

For the sake of contradiction, assume that it is essential. Then, for some instance I there exists a proof tree T for $U()$ on I and P such that there exists a vertex $U()$ with $\{V(), R(a, b), S(c, d)\} \subseteq \mathit{children}_T(U())$. Since the proof tree contains the fact $V()$, it also contains a rule instantiation that derives the fact $V()$ with body $\{R(a', b'), S(c', d'), S(d', c')\}$ for some constants a', b', c', d'. We can now construct two proof trees for $U()$ on the same instance, as seen in Figure 1. Because $c \neq d$, one of the facts $S(c', d'), S(d', c')$ must be different from $S(c, d)$ (In Figure 1 we assume this fact is $S(d', c')$). Thus, for one of the two trees, the children of $U()$ will not be a subset of $\{V(), R(a, b), S(c, d)\}$. This implies that the rule instantiation we considered is indeed not essential.

▶ **Example 5.10.** This example shows that $\mathcal{N}_P^{ess} = \mathcal{N}_P^{min} \cap \mathcal{N}_P^{use}$ can hold for recursive programs. Consider Example 5.6. Notice that every rule instantiation of the base rule, $T(x, y) \leftarrow R(x, y)$, is trivially minimal, useful and essential. As for the recursive rule, we showed in Example 5.6 that an instantiation that is minimal and useful is also essential. Observe that if this instantiation is only minimal but not useful, or only useful and not minimal, it is not essential. Thus, both properties are necessary to guarantee essentiality.

We conclude this section by commenting on whether it is computationally feasible to test the different properties of rule instantiations. It is easy to see that given an instantiation, it is possible to check whether it is useful in polynomial time. The complexity for checking the minimality of a rule instantiation is coNP-complete [6]. Unfortunately, testing essentiality of a rule instantiation is undecidable.

▶ **Proposition 5.11.** *Testing essentiality of rule instantiations is undecidable.*

6 Generalized Hypercube Policies

In this section, we present a general class of economic policies, called *Generalized Hypercube Policies (GHP)*, which encompass a broad variety of evaluation strategies.

We first give an intuitive explanation. The formalism of GHPs relies on the Hypercube partitioning for CQs [4], which has been shown to provide nice guarantees on the communication-cost for CQ evaluation [7]. Let $P = \{\tau\}$ be a CQ with k distinct variables. Hypercube conceptually orders the p servers as a hypercube $\mathcal{P} = [p_1] \times [p_2] \times \cdots \times [p_k]$, with $\prod_i p_i = p$, where every dimension $p_i \geq 0$ corresponds to a variable x_i from the query; every server is assigned a unique coordinate in space $\mathcal{P}$; and every variable x_i is associated to a hash function $h_{x_i} : \mathbf{dom} \mapsto [p_i]$. Then, a fact $R(a_1, \ldots, a_r)$, matching with atom

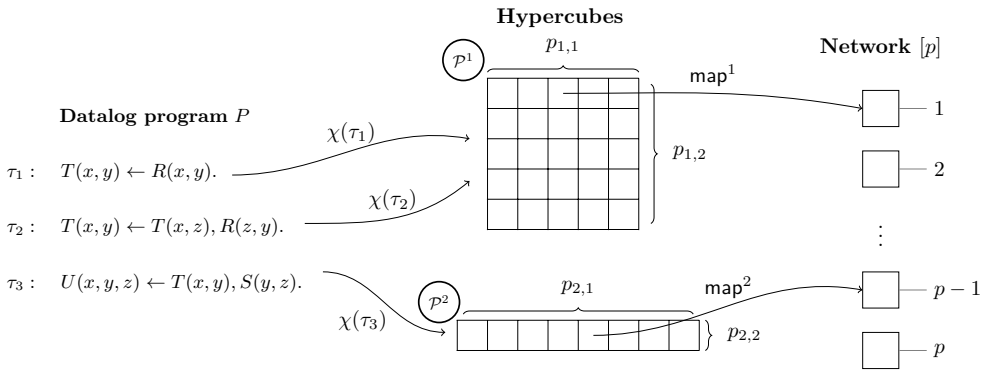


Figure 2 Example of a GHP policy for the Datalog program P with three rules.

$R(y_1, \ldots, y_r) \in body_\tau$, is sent to all servers whose coordinate agrees, for all $j \in [r]$, with position $h_{y_j}(a_j)$ on the dimension of $\mathcal{P}$ where y_j is associated with.

For GHPs we associate to every rule a hypercube over the full p-server network, and intuitively define the consumption policy so that "a fact is consumed at server i if and only if one of the considered Hypercube specifications would send it to server i"; for the specification of the production policy, we rely on a similar mechanism.

GHP parameters

Let P be a Datalog program, and assume we have a network $[p]$. A *GHP for* P defines a finite set of k-dimensional *hypercubes* $\mathcal{P}^1, \ldots, \mathcal{P}^\ell$, for some parameter k.[4] The size of the dimensions of the hypercubes are parametrized by a matrix $\mathbf{P}$ of dimensions $\ell \times k$, such that $\prod_{i=1}^{k} p_{j,i} = p$, for each $j \in [\ell]$. Each hypercube is then defined as $\mathcal{P}^j = [p_{j,1}] \times [p_{j,2}] \times \ldots [p_{j,k}]$. For each hypercube $\mathcal{P}^j$, we also define a bijective mapping map^j that assigns to every point in $\mathcal{P}^j$ a machine $s \in [p]$. The latter thus provides the mapping between conceptual machines in the cube and real machines in the considered network.

A GHP policy next assigns each rule τ to exactly one of the hypercubes: let $\chi : P \to [\ell]$ be the function that encodes this assignment. Given this assignment, a GHP defines a mapping $\rho^\tau : [k] \to 2^{vars(\tau)}$ that maps each dimension of the hypercube $\mathcal{P}^{\chi(\tau)}$ to a subset of the variables that appear in τ, such that the following condition holds:

> If $\chi(\tau) = \chi(\tau')$, then $|\rho^\tau(i)| = |\rho^{\tau'}(i)|$ for every dimension i. In other words, the mappings of variables of different rules to the same hypercube must be consistent.

Finally, the GHP defines for each dimension $i \in [k]$ and each hypercube $\mathcal{P}^j$ a *hash function* h_i^j that maps sets of size $\leq \alpha$ (α is the size of the set $\rho^\tau(i)$ for any τ such that $\chi(\tau) = j$) to a value in the i-th dimension. We require hash functions to be surjective. Notice that our concept of hash-function is a generalization of the hash-functions used in, e.g., the Hypercube algorithm, where $\alpha = 1$. Further, we notice that, by definition, rules that use the same hypercube, also use the same hash function for each dimension of that hypercube.

[4] We assume w.l.o.g. that each hypercube has the same number of dimensions, but we can also define it such that different rules have a different number of dimensions.

GHP semantics

Let $\boldsymbol{f}$ be a fact and suppose that $\boldsymbol{f} = v(A)$, for some valuation v and atom $A = R(\boldsymbol{y})$ that appears in rule τ.[5] We define the following set of machines:

$$S^{\tau}_{\boldsymbol{f},A} = \{\mathsf{map}^{\chi(\tau)}(\mathbf{q}) \mid \mathbf{q} \in \mathcal{P}^{\chi(\tau)} \text{ such that } \forall i \text{ with } \emptyset \subsetneq \rho^{\tau}(i) \subseteq \boldsymbol{y} : \mathbf{q}_i = h_i^{\chi(\tau)}(v(\rho^{\tau}(i)))\}.$$

Intuitively, $S^{\tau}_{\boldsymbol{f},A}$ denotes the set of machines whose coordinate $\boldsymbol{q}$ is consistent with the hash mappings specified for τ. Notice that if the atom $R(\boldsymbol{y})$ has only a part of the variables that correspond to some dimension i, then facts are broadcast over dimension i, as it happens if none of these variables are in $\boldsymbol{y}$.

The consumption policy $\boldsymbol{C}(\boldsymbol{f})$ is defined as the union over all sets $S^{\tau}_{\boldsymbol{f},A}$ for rules τ and atoms $A \in \mathit{body}_{\tau}$ with instantiation $\boldsymbol{f}$. The production policy $\boldsymbol{P}(\boldsymbol{f})$ is similarly defined as the union over all sets $S^{\tau}_{\boldsymbol{f},A}$ for rules τ and atom head_{τ} with instantiation $\boldsymbol{f}$.

▶ **Example 6.1.** Consider the Datalog program depicted in Figure 2. We choose two hypercubes $\mathcal{P}^1, \mathcal{P}^2$ ($\ell = 2$) with dimension $k = 2$. The first two rules τ_1, τ_2 are mapped to the hypercube $\mathcal{P}^1$, and the third rule τ_3 is mapped to $\mathcal{P}^2$. We choose the dimensions of the hypercubes such that $p_{1,1} \cdot p_{1,2} = p$, $p_{2,1} = p$, and $p_{2,2} = 1$. The two functions $\mathsf{map}^1, \mathsf{map}^2$ map the points of $\mathcal{P}^1, \mathcal{P}^2$ respectively to $\{1, \ldots, p\}$ in a one-to-one fashion. Finally, the mapping of variables to dimensions is:

$$\rho^{\tau_1}(1) = \{x\}, \rho^{\tau_1}(2) = \{y\}, \rho^{\tau_2}(1) = \{x\}, \rho^{\tau_2}(2) = \{z\}, \rho^{\tau_3}(1) = \{y\}, \rho^{\tau_3}(2) = \{\}$$

Consider the first two rules (which form the left-linear TC example), and assume that $p_{1,1} = 1$ and $p_{1,2} = p$. Then, the resulting GHP is equivalent to the hash partitioning policy that we described in Example 4.3. Notice that since we use the same hypercube for both rules, the EDB relation R will be hash partitioned only once. If we now change the dimensions to $p_{1,1} = p, p_{1,2} = 1$, we obtain the decomposable policy of Example 4.3 that broadcasts the EDB R to every machine and can terminate in a single round. Apart from the above two GHPs, we can also define other GHPs by configuring different dimensions of the hypercube $\mathcal{P}^1$. For example, we can choose $p_{1,1} = p_{1,2} = \sqrt{p}$.

We next show that GHPs are strongly supporting policies. A proof is in Appendix A.5.

▶ **Proposition 6.2.** *Let P be a Datalog program. Every GHP $\boldsymbol{E}$ for P is strongly supporting for P and, as a consequence, parallel-correct for P.*

GHP Families

Since we do not want to consider an encoding mechanism for hash functions – which is necessary to formally reason about properties for GHPs – we introduce the concept of GHP families. Given a Datalog program P and network $[p]$, a *GHP family* $\mathcal{H}$ is defined as the set of GHPs over P and $[p]$ that all have the same parametrization for $\mathbf{P}, \mathsf{map}^j, \chi, \rho^{\tau}$. In other words, policies in $\mathcal{H}$ can differ only with respect to the choice of hash functions, and for every choice of hash functions, the associated GHP is in the family. By $\mathcal{F}_{\text{GHP}}$ we denote the class of all GHP families.

[5] Notice that either v does not exist, or is unique for the variables in atom A.

7 Bounded & Disjoint Evaluation

In this section, we we ask two main questions: First, can we reason about the number of rounds that an economic policy needs to compute a Datalog program? Second, can we constrain the number of machines that derive a copy of the same fact? We start with a formal definition of boundedness.

▶ **Definition 7.1** (Boundedness). An economic policy $\boldsymbol{E}$ for Datalog program P is *bounded* if some constant k exists such that, for every instance I, the network reaches a global fixpoint for $\boldsymbol{E}$ and P, when round k is finished. We say $\boldsymbol{E}$ is ℓ-bounded if $k \leq \ell$.

One should not confuse the number of rounds in the parallel computation with the number of iterations of semi-naive evaluation. Nevertheless, as the following proposition shows, boundedness of the Datalog program implies boundedness of the evaluation.

▶ **Proposition 7.2.** *If P is a bounded Datalog program, then every parallel-correct economic policy $\boldsymbol{E}$ for P is k-bounded, for some constant k that depends on P.*

Surprisingly, there exist economic policies for bounded Datalog programs that are not bounded. However, due to Proposition 7.2, such policies cannot be parallel-correct.

▶ **Example 7.3.** Consider the following bounded program.

$$T(x) \leftarrow A(x). \qquad T(x) \leftarrow B(x), T(y).$$

We construct a network with $p > 1$ machines. Consider a policy that consumes $T(i)$ and $B(i)$ at machine $(i \mod p)+1$, and produces $T(i)$ at machine $(i \mod p)$. Every tuple in A is consumed at machine 1. Now, consider the following input instance: $\{A(0), B(1), B(2), \ldots, B(p-1)\}$. It is easy to see that $T(0)$ is produced at machine 1 at round 1, $T(1)$ is produced at machine 2 at round 2, and so on, until $T(p-1)$ is produced at round p at machine p.

In the remainder of this section, we focus on pure Datalog (denoted *PureDatalog*). We call a Datalog program *pure* if its variables occur at most once in every atom and it has no constants [23]. We consider the following decision problems.

k-BOUNDEDNESS$(\mathcal{L}, \mathcal{E})$	
Input:	Program $P \in \mathcal{L}$, policy $\boldsymbol{E} \in \mathcal{E}$.
Question:	Is $\boldsymbol{E}$ k-bounded for P?

BOUNDEDNESS$_F(\mathcal{L}, \mathcal{W})$	
Input:	Program $P \in \mathcal{L}$, family $\mathcal{F} \in \mathcal{W}$.
Question:	Is there a k s.t. $\mathcal{F}$ is k-bounded for P?

k-BOUNDEDNESS$_F(\mathcal{L}, \mathcal{W})$	
Input:	Program $P \in \mathcal{L}$, family $\mathcal{F} \in \mathcal{W}$.
Question:	Is $\mathcal{F}$ k-bounded for P?

▶ **Theorem 7.4.**

1. BOUNDEDNESS$_F(\mathit{PureDatalog}, \mathcal{F}_{\text{GHP}})$ *is undecidable;*
2. k-BOUNDEDNESS$(\mathit{PureDatalog}, \mathcal{E}_{indep})$ *and* k-BOUNDEDNESS$_F(\mathit{PureDatalog}, \mathcal{F}_{\text{GHP}})$ *are undecidable for $k \geq 2$; and*
3. k-BOUNDEDNESS$_F(\mathit{PureDatalog}, \mathcal{F}_{\text{GHP}})$ *is in* PTIME *if $k = 1$.*

Result (3) follows from the syntactical characterization shown in the next subsection. Towards this characterization, we first give a general characterization of 1-boundedness for strongly supporting policies.

Let P be a Datalog program and $\boldsymbol{E} = (\boldsymbol{P}, \boldsymbol{C})$ an economic policy. We denote by $\boldsymbol{P}^*$ the policy obtained by removing from every $\boldsymbol{P}(\boldsymbol{f})$ any server s for which no rule instantiation $v(\tau)$ exists with $v(\mathit{head}_\tau) = \boldsymbol{f}$, $v(\mathit{body}_\tau) \subseteq \mathsf{facts}_{\boldsymbol{C}}(s)$, and $v(\mathit{body}_\tau)$ being all P-derivable. Intuitively, $\boldsymbol{P}^*(\boldsymbol{f})$ removes those servers that are allowed to produce $\boldsymbol{f}$, but cannot due to limitations of the consumption policy $\boldsymbol{C}$. Notice that if $\boldsymbol{E} = (\boldsymbol{P}, \boldsymbol{C})$ is strongly supporting for P, then so is $\boldsymbol{E} = (\boldsymbol{P}^*, \boldsymbol{C})$, since we have not removed the support of any rule instantiation.

▶ **Proposition 7.5.** *Let P be a Datalog program and $\boldsymbol{E} = (\boldsymbol{P}, \boldsymbol{C})$ a strongly supporting economic policy for P. $\boldsymbol{E}$ is 1-bounded if and only if for every P-derivable* IDB *fact $\boldsymbol{f}$: (1) $|\boldsymbol{C}(\boldsymbol{f})| \leq 1$; and (2) $|\boldsymbol{C}(\boldsymbol{f})| = 1$ implies $\boldsymbol{C}(\boldsymbol{f}) = \boldsymbol{P}^*(\boldsymbol{f})$.*

7.1 Weakly Pivoting GHPs

We present a necessary and sufficient syntactic condition for 1-boundedness of GHP families. Here, for atom A and set of variables $X \subseteq \mathit{vars}(A)$, we denote by $\mathit{pos}_A(X)$ the positions in A having variables from X.

▶ **Definition 7.6** (Pivoting Relation)**.** A relation R is *pivoting for GHP family $\mathcal{H}$* if for every two atoms A_1, A_2 (in rules τ_1, τ_2 respectively) over R, and for all dimensions i of cube $\chi(\tau_1)$ with $p_{\chi(\tau_1),i} > 1$: $\emptyset \subsetneq \rho^{\tau_1}(i) \subseteq \mathit{vars}(A_1)$; $\chi(\tau_1) = \chi(\tau_2)$; and $\mathit{pos}_{A_1}(\rho^{\tau_1}(i)) = \mathit{pos}_{A_2}(\rho^{\tau_2}(i))$.

Intuitively, if R is pivoting, then every rule that sends R tuples will send each R tuple to exactly one machine, and the rules agree on this machine.

▶ **Example 7.7.** For example, take the program

$$\tau_1 : T(x,y) \leftarrow R(x,y). \qquad \tau_2 : T(x,y) \leftarrow T(x,z), R(z,y). \qquad \tau_3 : O(y) \leftarrow T(x,y), S(x).$$

and the GHP over the single one-dimensional cube (*cube* 1). We define $\chi(\tau_1) = \chi(\tau_2) = \chi(\tau_3) = 1$ and $\rho^{\tau_1}(1) = \rho^{\tau_2}(1) = \rho^{\tau_3}(1) = \{x\}$. Let map^1 be the identity mapping. Here, S and T are pivoting relations; O and R are not pivoting.

▶ **Definition 7.8** (Pivoting/Weakly pivoting)**.** We say that a GHP family is *pivoting* (*weakly pivoting*, resp.) for P if all (all P-consumable, resp.) IDB relations are pivoting.

The program from Example 7.7 is weakly pivoting. We can test whether a GHP family is weakly pivoting in polynomial time, since we need to go over all P-consumable IDB relations, and then for each such relation R test all pairs of atoms over R. This observation, along with the proposition below – that shows that weakly pivoting is a necessary and sufficient condition for 1-boundedness – implies that deciding 1-boundedness for GHP families is indeed in PTIME.

▶ **Proposition 7.9.** *Let P be a pure Datalog program, and $\mathcal{H}$ a GHP family. Then, $\mathcal{H}$ is 1-bounded for P* if and only if *it is weakly pivoting for P.*

We remark that Proposition 7.9 cannot be easily generalized. For example, one cannot replace GHP families by strongly supporting policies, since then facts that are not P-consumable may still be $\boldsymbol{C}$-consumable (*i.e.*, $\boldsymbol{C}(\boldsymbol{f}) \neq \emptyset$). Reasoning about the latter requires a concrete representation mechanism. Further, it is unclear what the complexity becomes for testing 1-boundedness under general (not necessarily pure) Datalog, since then it is required to reason about P-derivability of facts. An example is given in Appendix A.6.

7.2 Weakly Pivoting Datalog

We have so far looked at whether a given GHP family is 1-bounded. In this section, we ask: *which Datalog programs admit a 1-bounded policy?*

If $A = R(\boldsymbol{x})$ is an atom, we use $A[i]$ to denote the variable/constant in atom A in position i. We naturally extend $A[\cdot]$ to map tuples of positions (that take values from the set $\{1, \ldots, ar(R)\}$) onto tuples of variables/constants. For example, if $A = R(x_1, x_2, x_3)$ and $\boldsymbol{b} = (1, 3)$, then $A[\boldsymbol{b}] = (A[1], A[3]) = (x_1, x_3)$.

▶ **Definition 7.10** (Pivot Base)**.** Let P be a Datalog program, and let $\sigma \subseteq \text{IDB}(P)$. Let $\boldsymbol{\beta}$ be a function that takes as input some $R \in \sigma$ and outputs a non-empty tuple with values in $[ar(R)]$. We say that $\boldsymbol{\beta}$ is a *pivot base* for σ if:

- For every rule $\tau \in P$ and for every pair of atoms $R(\boldsymbol{x})$, $S(\boldsymbol{y})$ in $\{head_\tau\} \cup body_\tau$, such that $R, S \in \sigma$, we have $R(\boldsymbol{x})[\boldsymbol{\beta}(R)] = S(\boldsymbol{y})[\boldsymbol{\beta}(S)]$.

A Datalog program P is *pivoting* (*weakly pivoting*, resp.) if it has a pivot base for all relations in $\text{IDB}(P)$ (for all relations in $\text{IDB}(P)$ that occur in the body of some rule in P).

An example is given in Appendix A.7.

The concept pivoting Datalog was first introduced for single rule programs [30] and then generalized to full Datalog [23] where it is called *generalized pivoting*. The latter definition is based on a rather complex argument over fractional weight-mappings, but relates to pivoting in that every generalized pivoting Datalog program is pivoting for *all* IDB relations. For pure Datalog these notions are equivalent. The proposition below shows that for pure Datalog, a weakly pivoting program admits a weakly pivoting (and thus 1-bounded) GHP family.

▶ **Proposition 7.11.** *Let P be a pure Datalog program and $p \geq 2$. There is a 1-bounded GHP family if and only if P is weakly pivoting.*

7.3 Bounded and Disjoint Evaluation

Sometimes we want to guarantee that, at the end of computation, no two copies of the same fact have been derived at different machines. We call this property disjointness.

▶ **Definition 7.12** (Disjointness)**.** Let P be a Datalog program, and R an IDB predicate of P. We call an economic policy $\boldsymbol{D}$ for P *R-disjoint* if for every instance, every fact of R is produced in at most one server.

We study economic policies that are both 1-bounded *and* disjoint. For this, let P be a Datalog program and $\boldsymbol{D}$ a strongly supporting economic policy for P over $[p]$. We call $s \in [p]$ a *straggler* if $s \in \boldsymbol{C}(\boldsymbol{f})$ or $s \in \boldsymbol{P}^*(\boldsymbol{f})$ for all facts $\boldsymbol{f}$ of some IDB relation where P is defined over. Intuitively, a straggler is a server that consumes or produces an entire relation.

▶ **Proposition 7.13.** *Let $P \in PureDatalog$ and $\mathcal{H}$ a GHP family for P. Then, $\mathcal{H}$ is 1-bounded, disjoint for P, and without stragglers for IDB relations, if and only if, $\mathcal{H}$ is pivoting.*

Next, we show which programs admit a 1-bounded, disjoint policy.

▶ **Proposition 7.14.** *Let $P \in PureDatalog$. Then P is pivoting if, and only if, P admits a 1-bounded, strongly supporting, disjoint economic policy without stragglers for IDB relations.*

▶ Remark. The reader may wonder how the above concepts relate to the class of decomposable programs [32, 31]. A *decomposable* program is a (single rule) Datalog program that admits an evaluation strategy (via predicate restrictions) that is parallel-correct, 1-bounded, disjoint,

and non-trivial. (Here non-triviality means that all servers do part of the work.) We did not consider the non-triviality property, but instead require the absence of stragglers. Nevertheless, for GHPs, non-triviality is implied – at least for pure Datalog – by the use of surjective hash functions).

8 Conclusion

We introduce a theoretical framework to reason about multi-round Datalog evaluation in a distributed setting. In this framework we study three properties: parallel-correctness, boundedness, and disjointness. There are many interesting questions left open. For example, it would be interesting to come up with restrictions on Datalog programs and economic policies, for which the mentioned properties are not undecidable. Another interesting direction for future work would be to define a relevant fairness condition for economic policies, e.g., an instance independent notion of load-balancing; and to study bounds on the amount of communication needed to evaluate Datalog programs. Another direction is to consider smarter algorithms for local Datalog evaluation than semi-naive, by, for example, allowing to express unique-decomposition conditions (c.f., [5]) in the economic policy.

References

1 Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

2 Foto N. Afrati, Vinayak R. Borkar, Michael J. Carey, Neoklis Polyzotis, and Jeffrey D. Ullman. Map-reduce extensions and recursive queries. In Anastasia Ailamaki, Sihem Amer-Yahia, Jignesh M. Patel, Tore Risch, Pierre Senellart, and Julia Stoyanovich, editors, *EDBT 2011, 14th International Conference on Extending Database Technology, Uppsala, Sweden, March 21-24, 2011, Proceedings*, pages 1–8. ACM, 2011. doi:10.1145/1951365.1951367.

3 Foto N. Afrati and Christos H. Papadimitriou. The parallel complexity of simple chain queries. In Moshe Y. Vardi, editor, *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 23-25, 1987, San Diego, California, USA*, pages 210–213. ACM, 1987. doi:10.1145/28659.28682.

4 Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In Ioana Manolescu, Stefano Spaccapietra, Jens Teubner, Masaru Kitsuregawa, Alain Léger, Felix Naumann, Anastasia Ailamaki, and Fatma Özcan, editors, *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, volume 426 of *ACM International Conference Proceeding Series*, pages 99–110. ACM, 2010. doi:10.1145/1739041.1739056.

5 Foto N. Afrati and Jeffrey D. Ullman. Transitive closure and recursive datalog implemented on clusters. In Elke A. Rundensteiner, Volker Markl, Ioana Manolescu, Sihem Amer-Yahia, Felix Naumann, and Ismail Ari, editors, *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, pages 132–143. ACM, 2012. doi:10.1145/2247596.2247613.

6 Tom J. Ameloot, Gaetano Geck, Bas Ketsman, Frank Neven, and Thomas Schwentick. Parallel-correctness and transferability for conjunctive queries. In Tova Milo and Diego Calvanese, editors, *Proceedings of the 34th ACM Symposium on Principles of Database Systems, PODS 2015, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 47–58. ACM, 2015. doi:10.1145/2745754.2745759.

7 Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In Richard Hull and Wenfei Fan, editors, *Proceedings of the 32nd*

ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013, pages 273–284. ACM, 2013. doi:10.1145/2463664.2465224.

8 Paul Beame, Paraschos Koutris, and Dan Suciu. Skew in parallel query processing. In Richard Hull and Martin Grohe, editors, *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, pages 212–223. ACM, 2014. doi:10.1145/2594538.2594558.

9 Shumo Chu, Magdalena Balazinska, and Dan Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 63–78. ACM, 2015. doi:10.1145/2723372.2750545.

10 Stavros S. Cosmadakis and Paris C. Kanellakis. Parallel evaluation of recursive rule queries. In Avi Silberschatz, editor, *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 24-26, 1986, Cambridge, Massachusetts, USA*, pages 280–293. ACM, 1986. doi:10.1145/6012.15421.

11 Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI '04*, pages 137–150, 2004. URL: http://www.usenix.org/events/osdi04/tech/dean.html.

12 Hasanat M. Dewan, Salvatore J. Stolfo, Mauricio A. Hernández, and Jae-Jun Hwang. Predictive dynamic load balancing of parallel and distributed rule and query processing. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994.*, pages 277–288. ACM Press, 1994. doi:10.1145/191839.191893.

13 Sumit Ganguly, Abraham Silberschatz, and Shalom Tsur. A framework for the parallel processing of datalog queries. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990.*, pages 143–152. ACM Press, 1990. doi:10.1145/93597.98724.

14 Sumit Ganguly, Abraham Silberschatz, and Shalom Tsur. Parallel bottom-up processing of datalog queries. *J. Log. Program.*, 14(1&2):101–126, 1992. doi:10.1016/0743-1066(92)90048-8.

15 Gaetano Geck, Bas Ketsman, Frank Neven, and Thomas Schwentick. Parallel-correctness and containment for conjunctive queries with union and negation. *CoRR*, abs/1512.06246, 2015. arXiv:1512.06246.

16 Hadoop. http://hadoop.apache.org/.

17 Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. Demonstration of the myria big data management service. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 881–884. ACM, 2014. doi:10.1145/2588555.2594530.

18 Paris C. Kanellakis. Logic programming and parallel complexity. In Giorgio Ausiello and Paolo Atzeni, editors, *ICDT'86, International Conference on Database Theory, Rome, Italy, September 8-10, 1986, Proceedings*, volume 243 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 1986. doi:10.1007/3-540-17187-8_27.

19 Bas Ketsman and Dan Suciu. A worst-case optimal multi-round algorithm for parallel computation of conjunctive queries. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 417–428. ACM, 2017. doi:10.1145/3034786.3034788.

20 Paraschos Koutris, Paul Beame, and Dan Suciu. Worst-case optimal algorithms for parallel query processing. In Wim Martens and Thomas Zeume, editors, *19th International Conference on Database Theory, ICDT 2016, Bordeaux, France, March 15-18, 2016*, volume 48 of *LIPIcs*, pages 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.ICDT.2016.8.

21 Paraschos Koutris and Dan Suciu. Parallel evaluation of conjunctive queries. In Maurizio Lenzerini and Thomas Schwentick, editors, *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12-16, 2011, Athens, Greece*, pages 223–234. ACM, 2011. doi:10.1145/1989284.1989310.

22 Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks, and Dan Olteanu. Parallel materialisation of datalog programs in centralised, main-memory RDF systems. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 129–137. AAAI Press, 2014. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8505.

23 Jürgen Seib and Georg Lausen. Parallelizing datalog programs by generalized pivoting. In Daniel J. Rosenkrantz, editor, *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 29-31, 1991, Denver, Colorado, USA*, pages 241–251. ACM Press, 1991. doi:10.1145/113413.113435.

24 Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. Distributed socialite: A datalog-based language for large-scale graph analysis. *PVLDB*, 6(14):1906–1917, 2013. URL: http://www.vldb.org/pvldb/vol6/p1906-seo.pdf.

25 Marianne Shaw, Paraschos Koutris, Bill Howe, and Dan Suciu. Optimizing large-scale semi-naïve datalog evaluation in hadoop. In Pablo Barceló and Reinhard Pichler, editors, *Datalog in Academia and Industry - Second International Workshop, Datalog 2.0, Vienna, Austria, September 11-13, 2012. Proceedings*, volume 7494 of *Lecture Notes in Computer Science*, pages 165–176. Springer, 2012. doi:10.1007/978-3-642-32925-8_17.

26 Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1135–1149. ACM, 2016. doi:10.1145/2882903.2915229.

27 Apache spark. http://spark.apache.org/.

28 Jeffrey D. Ullman and Allen Van Gelder. Parallel complexity of logical query programs. *Algorithmica*, 3:5–42, 1988. doi:10.1007/BF01762108.

29 Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *PVLDB*, 8(12):1542–1553, 2015. URL: http://www.vldb.org/pvldb/vol8/p1542-wang.pdf.

30 Ouri Wolfson. Sharing the load of logic-program evaluation. In Sushil Jajodia, Won Kim, and Abraham Silberschatz, editors, *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems, Austin, Texas, USA, December 5-7, 1988*, pages 46–55. IEEE Computer Society, 1988. doi:10.1109/DPDS.1988.675001.

31 Ouri Wolfson and Aya Ozeri. A new paradigm for parallel and distributed rule-processing. *SIGMOD Rec.*, 19(2):133–142, 1990. doi:10.1145/93605.98723.

32 Ouri Wolfson and Avi Silberschatz. Distributed processing of logic programs. *SIGMOD Rec.*, 17(3):329–336, 1988. doi:10.1145/971701.50242.

33 Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: SQL and rich analytics at scale. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 13–24. ACM, 2013. doi:10.1145/2463676.2465288.

34 Weining Zhang, Ke Wang, and Siu-Cheung Chau. Data partition and parallel evaluation of datalog programs. *IEEE Trans. Knowl. Data Eng.*, 7(1):163–176, 1995. doi:10.1109/69.368511.

A Appendix

A.1 Proof for Theorem 5.2

▶ **Theorem 5.2.** PC$(Datalog, \mathcal{E}_{indep})$ *is undecidable.*

Proof. The proof is by reduction from the Datalog containment problem, which is well-known to be undecidable [1]. Let P_1 and P_2 be two arbitrary Datalog programs given as input for the containment problem. As usual, we assume that both are over the same output predicate, say O.

We first denote by P_i^* an indexed version of program P_i; particularly we define P_i^* as P_i in which all IDB predicates are annotated with index i. We now construct a program P by taking all rules from P_1^* and P_2^*, and adding the rules $O(\boldsymbol{x}) \leftarrow O_i(\boldsymbol{x})$, for $i \in \{1, 2\}$. We note that $edb(P) = edb(P_1^*) \cup edb(P_2^*)$ and $out(P) = \{O\}$. As economic policy we take $\boldsymbol{E} = (\boldsymbol{P}, \boldsymbol{C})$ over the 2-node network $\{1, 2\}$. The consumption policy maps all facts with index i to server i. The production policy maps all facts with index i to server i, and all O-facts to server 2. The EDB facts are consumed on all servers.

Intuitively, programs P_1^* and P_2^* are computed locally on server 1 and server 2. It thus follows from the construction that (†) $P_1(I) \cup P_2(I) \subseteq [P, \boldsymbol{E}](I)$, for every instance I. Notice that rule $O(\boldsymbol{x}) \leftarrow O_1(\boldsymbol{x})$ is never used, since server 2 cannot consume facts over predicates with index 1.

It remains to show that $\boldsymbol{E}$ is parallel-correct for P if and only if $P_1 \subseteq P_2$. Indeed, if $P_1 \subseteq P_2$, then $O(I) = P_2(I)$ for every instance I, which implies that the policy will compute the correct result for O. The other direction follows from monotonicity of P. From (†) it follows that this condition is satisfied if and only if all facts over the O relation produced by $P(I)$ are also produced by $[P, \boldsymbol{E}](I)$, which is the case only if every fact $O(\boldsymbol{a}) \in P(I)$ implies a fact $O_2(\boldsymbol{a}) \in P(I)$. The latter is equivalent to saying $O(\boldsymbol{a}) \in P_1(I)$ implies $O(\boldsymbol{a}) \in P_2(I)$ for every instance I, which means that $P_1 \subseteq P_2$. ◀

A.2 Proof for Proposition 5.5

We first show the following Lemma.

▶ **Lemma A.1.** *For every proof tree T of depth d, there exists a proof tree $T' \sqsubseteq T$ of depth at most d that uses only minimal and useful rule instantiations.*

Proof. The proof is by induction on the depth of T, which we denote d. We show that there exists a proof tree $T' \sqsubseteq T$ with depth $\leq d$ that uses only minimal rule instantiations.

For the base case, let $d = 1$. Then, T corresponds to a single rule instantiation (τ, v) for P where all the facts in $v(body_\tau)$ are EDB facts. By definition, there is also a minimal rule instantiation (τ', v'), with $v'(head_{\tau'}) = v(head_\tau)$ and $v'(head_{\tau'}) \subseteq v(body_\tau)$, which admits the desired proof tree.

As induction hypothesis we take the statement of the lemma. Now for the induction step, suppose T has depth $d > 1$. Then, the root of T, together with its children, defines a rule instantiation (τ, v) for P. Now take an entailed minimal instantiation (τ', v) such that $v'(head_{\tau'}) = v(head_\tau)$ and $v'(body_{\tau'}) \subseteq v(body_\tau)$. For every fact $\boldsymbol{f} \in v'(head_{\tau'})$, let $T_{\boldsymbol{f}}$ be the subtree of T with root $\boldsymbol{f}$ (child of $root_T$). By the induction hypothesis, there is a proof

tree $T'_{\boldsymbol{f}} \sqsubseteq T_{\boldsymbol{f}}$ with depth $\leq d-1$ that uses only minimal rule instantiations. The proof tree that combines instantiation (τ', v') with $T'_{\boldsymbol{f}}$ for all $\boldsymbol{f} \in v'(\tau')$ is as desired. ◀

▶ **Proposition 5.5.** *For every Datalog program P, we have $\mathcal{N}_P^{ess} \subseteq \mathcal{N}_P^{min} \cap \mathcal{N}_P^{use}$.*

Proof. The containment $\mathcal{N}_P^{ess} \subseteq \mathcal{N}_P^{use}$ is straightforward, since a proof tree does not use any useless rule instantiations. We next show that $\mathcal{N}_P^{ess} \subseteq \mathcal{N}_P^{min}$. Suppose that we have an instantiation of rule τ with valuation v that is essential. Then, there exists some fact $\boldsymbol{f}$ and instance I for which every proof tree T has a vertex $\boldsymbol{g}$ with $\boldsymbol{g} = v(\mathit{head}_\tau)$ and $v(\mathit{body}_\tau) \subseteq \mathit{children}_T(\boldsymbol{g})$. By Lemma A.1, we can pick this tree such that it uses only minimal rule instantiations. This implies that the rule instantiation with head $\boldsymbol{g}$ and body $\mathit{children}_T(\boldsymbol{g})$ is minimal. Hence, the instantiation with head $v(\mathit{head}_\tau)$ and body $v(\mathit{body}_\tau)$ is also minimal. ◀

A.3 Proof for Proposition 5.7

We first show the following lemma.

▶ **Lemma A.2.** *Let P be a Datalog program and $\boldsymbol{E}$ an economic policy. If a proof tree T for P is supported by $\boldsymbol{E}$, then for every instance I, with $\mathit{fringe}_T \subseteq I$, we have $\mathit{root}_T \in [P, \boldsymbol{E}]$.*

Proof. The proof is by induction on the depth d of T. Particularly we show using a simple inductive argument that $\mathit{root}_T \in \mathsf{local}_i^k$, for some server i and $k \leq d$, which implies $\mathit{root}_T \in [P, \boldsymbol{E}]$. Recall that local_i^k denotes the facts residing locally on server i after the k-th computation round.

As base case let $d = 1$, meaning that T describes a single rule instantiation. After the first communication round, all servers j have $\mathsf{local}_j^0 \cup \mathsf{rec}_j^1 \subseteq I \cap \mathsf{facts}_{\boldsymbol{C}}(j)$. By the assumption that $\boldsymbol{E}$ supports T, it follows that $\mathit{children}_T(\mathit{root}_T) \subseteq \mathsf{facts}_{\boldsymbol{C}}(i) \cap I \subseteq \mathsf{local}_i^1$ and $\mathit{root}_T \in \mathsf{facts}_{\boldsymbol{P}}(i)$, for some server i, thus after the first computation round, $\mathit{root}_T \in \mathsf{local}_i^1$.

For $d > 1$ we observe that root_T and its children in T define a rule instantiation (τ, v), and, by the assumptions of the lemma, this rule instantiation is supported by $\boldsymbol{E}$. More specifically, some server i exists where $\mathit{root}_T \in \mathsf{facts}_{\boldsymbol{P}}(i)$ and $\mathit{children}_T(\mathit{root}_T) \subseteq \mathsf{facts}_{\boldsymbol{C}}(i)$. Further, for all facts $\boldsymbol{f} \in \mathit{children}_T(\mathit{root}_T)$, the respective subtree $T_{\boldsymbol{f}}$ of T with root $\boldsymbol{f}$ is supported by $\boldsymbol{E}$ and with depth $d-1$. By the induction hypothesis it follows that for all these facts $\boldsymbol{f}$ there is a server j and $k \leq d-1$, where $\boldsymbol{f} \in \mathsf{local}_j^k$. Therefore $\mathit{children}_T(\mathit{root}_T) \subseteq \mathsf{local}_i^{k^*} \cup \mathsf{rec}_i^{k^*}$, where k^* denotes the maximal k, and consequently, $\mathit{root}_T \in \mathsf{local}_i^{k^*+1} \subseteq \mathsf{local}_i^d$. ◀

We say that an economic policy $\boldsymbol{E}$ *supports a proof tree* T if all the rule instantiations in T are supported.

▶ **Lemma A.3.** *Let P be a Datalog program. An economic policy $\boldsymbol{E} = (\boldsymbol{P}, \boldsymbol{C}; U)$ is parallel-correct for P if and only if for every proof tree for P with fringe over $\mathsf{facts}(\sigma(P), U)$, an entailed supported proof tree exists.*

Proof. *(If).* Let I be an arbitrary instance, we show $P(I) = [P, \boldsymbol{E}](I)$. By monotonicity, $[P, \boldsymbol{E}](I) \subseteq P(I)$, thus we focus on completeness. For this, let $\boldsymbol{f} \in P(I)$, which means that a proof tree T exists with $\mathit{fringe}_T \subseteq I$ and $\mathit{root}_T = \boldsymbol{f}$. Particularly, by the assumption of the lemma we can choose T so that it is also supported by $\boldsymbol{E}$. It now follows from Lemma A.2 that $\boldsymbol{f} \in [P, \boldsymbol{E}]$.

(Only if). We assume $(\boldsymbol{P}, \boldsymbol{C})$ is parallel-correct for P. Let T be an arbitrary proof tree. The proof is by construction following the derivation of root_T using $\boldsymbol{E}$. First, from parallel-correctness it follows that $P(I) = [P, \boldsymbol{E}](I)$, for any instance I. Here we take $I = \mathit{fringe}_T$,

implying $root_T \in [P, \boldsymbol{E}](I)$. The proof now continues by induction on the number of rounds needed for $\boldsymbol{E}$ to derive $root_T$.

The induction hypothesis is that if k rounds are needed to derive $root_T$, then a supported proof-tree of depth k entailed by T exists.

As a base case suppose $k = 1$. That is, $root_T \in \mathsf{local}_i^1$, meaning that $root_T \in P_{\uparrow \boldsymbol{E}}(\mathsf{local}_j^0 \cup \bigcup_j \mathsf{rec}_j^1)$ for some server j. Particularly, a valuation v and rule $\tau \in P$ existed with $v(body_\tau) \subseteq \mathsf{facts}_{\boldsymbol{C}}(j) \cap I$ and $v(head_\tau) = root_T$, which means that the corresponding rule instantiation is supported by $\boldsymbol{E}$. Here, the proof tree admitted by (τ, v) is as desired.

For $k > 1$ the proof is analogous, but now we take as proof tree the tree obtained by concatenating the rule instantiation with the proof trees for each child. Existence of the latter follows from the induction hypothesis. As the number of rounds decreases by one in each inductive step, and the fringes of the obtained trees cannot have other facts than does in I, the constructed proof tree is as again as desired. ◀

▶ **Proposition 5.7.** *Let P be a Datalog program and $\boldsymbol{E}$ an economic policy. If $\boldsymbol{E}$ supports all minimal and useful rule instantiations in P, then it is parallel-correct. If $\boldsymbol{E}$ is parallel-correct for P, then it supports all essential rule instantiations.*

Proof. The first item follows from Lemma A.3 and Lemma A.1. For the second item, consider a parallel-correct policy $\boldsymbol{E}$ and an essential instantiation of rule τ with valuation v. By the definition of essential, for some fact $\boldsymbol{f}$ and instance I, every proof tree T for $\boldsymbol{f}$ on I and P has a vertex $\boldsymbol{g}$ with $\boldsymbol{g} = v(head_\tau)$ and $v(body_\tau) \subseteq children_T(\boldsymbol{g})$. By Lemma A.3, there must exist such a tree T that is supported. This implies that there exists server s with $v(head_\tau) = \boldsymbol{g} \in \mathsf{facts}_{\boldsymbol{P}}(s)$ and $v(body_\tau) \subseteq children_T(\boldsymbol{g}) \subseteq \mathsf{facts}_{\boldsymbol{C}}(s)$. Hence, the essential rule instantiation is indeed supported. ◀

A.4 Proof for Proposition 5.8

▶ **Proposition 5.8.** *Let P be a Datalog program where each* IDB *predicate occurs only in the head of rules (i.e., P is a union of CQs). Then, $\mathcal{N}_P^{ess} = \mathcal{N}_P^{min} \cap \mathcal{N}_P^{use}$.*

Proof. Because P is not recursive, $\mathcal{N}_P^{use} = \mathcal{N}_P^{all}$; hence, because of Proposition 5.5 it suffices to show that $\mathcal{N}_P^{min} \subseteq \mathcal{N}_P^{ess}$. Indeed, consider a minimal instantiation for rule τ with valuation v, and consider the instance $I = v(body_\tau)$ and fact $\boldsymbol{f} = v(head_\tau)$. Take any proof tree T for $\boldsymbol{f}$ on I and P; T must have depth one. Because of the minimality of the rule instantiation, it must be that $children_T(\boldsymbol{f}) = v(body_\tau)$, which proves the essentiality. ◀

A.5 Proof for Proposition 6.2

▶ **Proposition 6.2.** *Let P be a Datalog program. Every GHP $\boldsymbol{E}$ for P is strongly supporting for P and, as a consequence, parallel-correct for P.*

Proof. To show that $\boldsymbol{E}$ is supporting, consider some rule $\tau \in P$, and its instantiation w.r.t. some valuation v. Consider some atom $A = R(\boldsymbol{y})$ in the body of τ; then the consumption policy says that its instantiation $\boldsymbol{f} = v(A)$ will be consumed in the set $S^\tau_{\boldsymbol{f},A}$, as defined in Section 6. Similarly if A is the head, the fact $\boldsymbol{f}$ will be produced in $S^\tau_{\boldsymbol{f},A}$. Now we can write the intersection $\bigcap_{A \in \tau} S^\tau_{\boldsymbol{f},A}$ as:

$$\bigcap_{A \in \tau} \{\mathsf{map}^{\chi(\tau)}(\mathbf{q}) \mid \forall i : \emptyset \subsetneq \rho^\tau(i) \subseteq vars(A) \Rightarrow \mathbf{q}_i = h_i^{\chi(\tau)}(v(\rho^\tau(i)))\}$$

$$\supseteq \{\mathsf{map}^{\chi(\tau)}(\mathbf{q}) \mid \forall i : \mathbf{q}_i = h_i^{\chi(\tau)}(v(\rho^\tau(i)))\} \supsetneq \emptyset$$

In other words, there will be at least one machine in $\bigcap_{A \in \tau} S^{\tau}_{A}$, which means that every instantiation of the rule τ will be strongly supported. ◀

A.6 Example A.4

▶ **Example A.4.** For an example showing that not every 1-bounded GHP is weakly pivoting, consider the following *non-pure* Datalog program P:

$$R(x,x) \leftarrow S(x,x). \qquad T(x,y) \leftarrow R(x,y). \qquad T(x,y) \leftarrow T(z,x), R(z,y).$$

and GHP family $\mathcal{H}$ over a single one-dimensional cube 1. Let map^1 be the identity mapping, $\chi(\tau) = 1$ and $\rho^{\tau}(1) = \{x\}$ for all rules τ. Clearly, $\mathcal{H}$ is not weakly pivoting. Nevertheless, it can be shown that $\mathcal{H}$ is 1-bounded, which follows from the observation that only single-valued rule instantiations can satisfy under P.

A.7 Example A.5

▶ **Example A.5.** Consider the left-linear TC example, and let $\sigma = \{T\}$. Suppose we choose $\boldsymbol{\beta}(T) = (1)$. Then $\boldsymbol{\beta}$ is a pivot base for σ, since for the recursive rule and the only pair of T-atoms $T(x,y), T(x,z)$ we have $T(x,y)[\boldsymbol{\beta}(T)] = T(x,y)[1] = (x)$, and $T(x,z)[\boldsymbol{\beta}(T)] = T(x,z)[1] = (x)$. Since T is the only IDB relation, left-linear TC is pivoting.

Next, consider the left-linear TC with an extra rule:

$$T(x,y) \leftarrow R(x,y). \qquad T(x,y) \leftarrow T(x,z), R(z,y). \qquad U(y) \leftarrow T(x,y).$$

Here, there are two IDB relations, but only T occurs in the body of a rule. The pivot base $\boldsymbol{\beta}$ from before is still a pivot base for $\{T\}$; hence the program is weakly pivoting. However, there is no pivot base for to $\{T, U\}$, which means that the program is not pivoting.

Parallel-Correctness and Transferability for Conjunctive Queries under Bag Semantics

Bas Ketsman[1]
Hasselt University, Hasselt, Belgium, and transnational University of Limburg, Belgium
bas.ketsman@uhasselt.be

Frank Neven
Hasselt University, Hasselt, Belgium, and transnational University of Limburg, Belgium
frank.neven@uhasselt.be

Brecht Vandevoort[2]
Hasselt University, Hasselt, Belgium, and transnational University of Limburg, Belgium
brecht.vandevoort@uhasselt.be

Abstract

Single-round multiway join algorithms first reshuffle data over many servers and then evaluate the query at hand in a parallel and communication-free way. A key question is whether a given distribution policy for the reshuffle is adequate for computing a given query. This property is referred to as parallel-correctness. Another key problem is to detect whether the data reshuffle step can be avoided when evaluating subsequent queries. The latter problem is referred to as transfer of parallel-correctness. This paper extends the study of parallel-correctness and transfer of parallel-correctness of conjunctive queries to incorporate bag semantics. We provide semantical characterizations for both problems, obtain complexity bounds and discuss the relationship with their set semantics counterparts. Finally, we revisit both problems under a modified distribution model that takes advantage of a linear order on compute nodes and obtain tight complexity bounds.

2012 ACM Subject Classification Information systems → Query languages, Information systems → Parallel and distributed DBMSs

Keywords and phrases Conjunctive queries, distributed evaluation, bag semantics

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.18

1 Introduction

The rise of parallel data management systems like, for instance, Spark [18] and Hadoop [11], inspired a line of research on the foundations of parallel complexity of query evaluation. Several papers investigate trade-offs between the number of rounds and the amount of communication of parallel algorithms for join queries (e.g., [1–3, 6, 13, 14]). Among these, the Hypercube algorithm [3, 6, 9] is a single-round algorithm that works in two phases. The first phase is a distribution phase (where data is repartitioned or reshuffled over the servers) that is followed by a computation phase, where each server contributes to the query answer in isolation, by evaluating the query at hand over the local data without any further communication.

[1] PhD Fellow of the Research Foundation - Flanders (FWO)
[2] PhD Fellow of the Research Foundation - Flanders (FWO)

Ameloot et al. [5] introduced a framework for reasoning about generic one-round Hypercube-style algorithms for the evaluation of join queries. In this model, the distribution phase is modeled through a distribution policy specifying how the facts in the input relations are distributed among the machines. They defined two problems:

- **Parallel-Correctness:** Given a distribution policy and a query, can we be sure that the corresponding generic one-round algorithm will always compute the query result correctly, no matter the actual data?
- **Parallel-Correctness Transfer:** Given two queries $\mathcal{Q}$ and $\mathcal{Q}'$, can we infer from the fact that $\mathcal{Q}$ is computed correctly under the current distribution policy, that $\mathcal{Q}'$ is computed correctly as well?

Ameloot et al. [5] obtained tight complexity bounds for (unions of) conjunctive queries (with disequalities) for the above problems. In addition, they considered subcases that lower the complexity by either restricting the structure of queries or restricting the family of allowed distribution policies. Furthermore, it was shown (in the journal version and also in [4]) that transferability of parallel-correctness for conjunctive queries is incomparable with query containment. Geck et al. [10] consider the complexity of parallel-correctness for (unions of) conjunctive queries with negation. As a by-product it is shown that the containment problem for conjunctive queries with negation is coNEXPTIME-complete. Finally, Ketsman, Albarghouthi and Koutris [12] introduce a framework to reason about multi-round evaluation of Datalog programs and consider parallel-correctness for Datalog programs. Understanding the optimization of single-round algorithms is still important as every multi-round algorithm is a sequence of single-round steps and results from the single-round case can be transferred to or used as inspiration for studying multi-round algorithms.

Whereas the bulk of the research related to conjunctive queries focuses on set semantics, a more accurate approximation of SQL semantics is the bag semantics where multiplicities of the same tuples are taken into account. Moreover, bag semantics is particularly relevant for aggregate operators. In this paper, we therefore revisit parallel-correctness and parallel-correctness transfer under bag semantics.

As in [5], we consider conjunctive queries (CQs), allowing disequalities. Parallel-correctness under set semantics is characterized in terms of a property of minimal valuations. In brief, a CQ is parallel-correct with respect to a distribution policy if and only if for every *minimal* valuation for that query there is *at least one* compute node containing all the facts required for that valuation. Using the latter characterization, Ameloot et al. [5] obtained that testing parallel-correctness for CQs is Π_2^p-complete. In Section 3, we prove the Highlander Lemma stating that under bag semantics a CQ is parallel-correct with respect to a distribution policy if and only if for *every valuation* (not only the minimal ones) there is *exactly one* compute node containing all facts required for that valuation. Using the latter characterization, we obtain that testing for parallel-correctness under bag semantics is coNP-complete. While parallel-correctness under bag semantics implies parallel-correctness under set semantics, the converse is not true. We obtain that when CQs are strongly minimal and distribution policies are non-replicating, parallel-correctness coincides for set and bag semantics.

In a setting where multiple queries need to be evaluated, it is relevant to study whether parallel-correctness carries over from one query to another. That is, whether two queries can be evaluated after another *without* an intermediate reshuffling of the data. The latter can be relevant w.r.t. ordering of queries to improve query evaluation. For instance, in the setting of automatic data partitioning, an optimizer tries to automatically partition the base data across multiple nodes to achieve overall optimal performance for a given workload of queries (see, e.g., [15, 16]). In this setting, partitionings are thus instance dependent and not known in advance.

We say that parallel-correctness transfers from a query $\mathcal{Q}$ to a query $\mathcal{Q}'$ when $\mathcal{Q}'$ is parallel-correct under every distribution policy $\boldsymbol{P}$ under which $\mathcal{Q}$ is parallel-correct. We prove the Sandwich Lemma that provides a semantical characterization for parallel-correctness transfer under bag semantics in terms of a sandwich property for valuations. Like in the case for parallel-correctness, when comparing to set semantics, the characterization considers all valuations instead of only the minimal ones. On the other hand, as a consequence of the Highlander Lemma, the structure of queries can put additional requirements on distribution policies that are bag-parallel-correct. Therefore, our semantical characterization takes into account facts that are *implied* by a valuation w.r.t. a given query. Using the latter characterization, we obtain a decision procedure in EXPTIME for testing parallel-correctness transfer under bag semantics. In addition, we show that transferability under set and bag semantics is incomparable in general but coincides for strongly minimal conjunctive queries and non-replicating distribution policies.

The setting we have considered up to now allows every (distributed) compute node to contribute to the query result. Indeed, as is the case for the Hypercube algorithm, the result of the distributed query evaluation is the union of the results over *all* compute nodes. In this setting and under bag-semantics, the Highlander Lemma of Section 3 implies that the space of valuation for a conjunctive query should be perfectly partitioned over all compute nodes. That is, every valuation should occur in exactly one compute node. The latter can lead to situations where for particular queries the only bag-parallel-correct distribution policies are those that assign all facts to one single node. To remedy this situation, we consider the setting of ordered networks where every compute node is assigned a number and for every valuation only the node with the smallest number containing all facts required for that valuation can contribute to the query result. While both settings do not differ under set semantics, the new setting is more natural for bag semantics. We characterize parallel-correctness as well as transferability under bag semantics in this new setting and obtain tight complexity bounds.

In this paper, we make the following contributions:

1. The Highlander Lemma provides a semantical characterization of bag-parallel correctness. We obtain tight bounds for the complexity of deciding bag-parallel-correctness. We show that bag-parallel-correctness always implies set-parallel-correctness but not vice-versa and obtain that they coincide for strongly minimal queries and non-replicating distribution policies.
2. The Sandwich Lemma provides a semantical characterization of bag-parallel correctness transfer. We obtain an EXPTIME upper bound for deciding bag-parallel correctness transfer. We show that transfer of parallel-correctness under bag and set semantics is incomparable. In addition, we show that they coincide for strongly minimal queries and non-replicating distribution policies.
3. We introduce the ordered network model and again provide tight complexity bounds for parallel-correctness and transfer.

Outline

This paper is structured as follows. In Section 2, we introduce the necessary definitions. In Section 3 and Section 4, we consider parallel-correctness and parallel-correctness transfer under bag semantics. We revisit both problems under a modified distribution model that takes advantage of a linear order on compute nodes in Section 5. Finally, we conclude in Section 6.

2 Definitions

2.1 Queries and instances

We assume an infinite set **dom** of data values that are representable by strings over a fixed alphabet. A *database schema* $\mathcal{D}$ is a finite set of relation names R where every R has arity $ar(R)$. A *fact* $R(d_1, \ldots, d_k)$ is over a database schema $\mathcal{D}$ and a universe $U \subseteq \mathbf{dom}$ where $R \in \mathcal{D}$, $k = ar(R)$ and $d_1, \ldots, d_k \in U$. We use $Facts(\mathcal{D}, U)$ to denote the set of all facts over database schema $\mathcal{D}$ and universe $U \subseteq \mathbf{dom}$. We note that U can be infinite. We sometimes abbreviate $Facts(\mathcal{D}, \mathbf{dom})$ as $Facts(\mathcal{D})$.

An *annotated fact* $\boldsymbol{f_a}$ is a tuple $(\boldsymbol{f}, m)$ with $\boldsymbol{f}$ a fact and $m \in \mathbb{N}^+$ the multiplicity of $\boldsymbol{f}$. Here $\mathbb{N}^+$ denotes the set of strictly positive integers. A *bag of facts* F is a set of annotated facts. Every fact f may appear at most once as an annotated fact in B. That is, $(\boldsymbol{f}, m) \in B$ and $(\boldsymbol{f}', m') \in B$ implies $\boldsymbol{f} \neq \boldsymbol{f}'$. Intuitively, the multiplicity m of a fact $\boldsymbol{f}$ indicates the number of times $\boldsymbol{f}$ appears in the bag. We denote the set of facts appearing in F by $Facts(F)$ and the multiplicity of a fact $\boldsymbol{f}$ in the bag F by $mul_F(\boldsymbol{f})$. For convenience, we abuse notation and extend $mul_F(\boldsymbol{f})$ to arbitrary facts by setting $mul_F(\boldsymbol{f}) = 0$ when $\boldsymbol{f} \notin Facts(F)$. We next define the notion of bag union and subbag. We overload notation by using the same symbols as for set union and subset. It should always be clear from the context whether we refer to bags or to sets. For two bags of facts F and G, the *bag union*, denoted $F \cup G$, is defined as $Facts(F) \cup Facts(G)$ and $mul_H(\boldsymbol{f}) = mul_F(\boldsymbol{f}) + mul_G(\boldsymbol{f})$ for each fact $\boldsymbol{f} \in Facts(H)$. Furthermore, F is a *subbag* of G, denoted $F \subseteq G$, if $mul_F(\boldsymbol{f}) \leq mul_G(\boldsymbol{f})$ for each fact $\boldsymbol{f} \in Facts(F)$. By $|F|$, we denote the number of facts in F, that is, $\sum_{\boldsymbol{f} \in Facts(F)} mul_F(\boldsymbol{f})$.

A *database instance* I, instance for short, over a database schema $\mathcal{D}$ is a bag of facts, with $Facts(I) \subseteq Facts(\mathcal{D})$. We use $adom(I)$ to denote the set of data values occurring in I.

A *query* $\mathcal{Q}$ *over input schema* $\mathcal{D}_1$ *and output schema* $\mathcal{D}_2$ is a generic mapping from instances over $\mathcal{D}_1$ to instances over $\mathcal{D}_2$. A query $\mathcal{Q}$ is *monotone* if $\mathcal{Q}(I') \subseteq \mathcal{Q}(I)$ for every pair of instances I and I' with $I' \subseteq I$.

2.2 Conjunctive queries

Assume an infinite set of variables **var**, disjoint from **dom**. An *atom* over a database schema $\mathcal{D}$ is of the form $R(\mathbf{x})$, with $R \in \mathcal{D}$ and $\mathbf{x} = (x_1, \ldots, x_k)$ a tuple of variables in **var** with $k = ar(R)$.

A *conjunctive query* $\mathcal{Q}$ over input schema $\mathcal{D}$ is an expression of the form

$$T(\mathbf{x}) \leftarrow R_1(\mathbf{y_1}), \ldots, R_m(\mathbf{y_m}), \beta_1, \ldots, \beta_p$$

where every $R_i(\mathbf{y_i})$ is an atom over $\mathcal{D}$, $T(\mathbf{x})$ is an atom, called the *head atom*, with $T \notin \mathcal{D}$, and every β_i is a disequality of the form $z \neq z'$ (with z a variable different from z'). Every variable $x \in \mathbf{x}$ needs to appear in at least one $\mathbf{y_i}$. We require that every variable occurring in a disequality occurs in at least one $\mathbf{y_i}$. Furthermore, we refer to $T(\mathbf{x})$ as $head_{\mathcal{Q}}$, to the set $\{R_1(\mathbf{y_1}), \ldots, R_m(\mathbf{y_m})\}$ as $body_{\mathcal{Q}}$ and to the set of all variables occurring in $\mathcal{Q}$ as $vars(\mathcal{Q})$.

We denote by $\mathbf{CQ}^{\neq}$ the set of all conjunctive queries (allowing disequalities) and by $\mathbf{CQ}$ the set of conjunctive queries without disequalities. A conjunctive query with disequalities is *without self-joins* if all of its atoms have distinct relation names. A conjunctive query with disequalities $\mathcal{Q}$ is *full* if every variable occurring in $\mathcal{Q}$ appears in the head atom.

A *valuation* for a conjunctive query $\mathcal{Q} \in \mathbf{CQ}^{\neq}$ is a total function $V : vars(\mathcal{Q}) \to \mathbf{dom}$ that is consistent with the disequalities in $\mathcal{Q}$. More specifically: for every $z \neq z'$ in $\mathcal{Q}$ it holds

that $V(z) \neq V(z')$. Valuations naturally extend to atoms and sets of atoms. We refer to $V(body_{\mathcal{Q}})$ as the set of facts *required* by V.

A valuation V *satisfies* a conjunctive query $\mathcal{Q} \in \mathbf{CQ}^{\neq}$ on instance I if $V(body_{\mathcal{Q}}) \subseteq Facts(I)$. In that case, V *derives the annotated fact* $\boldsymbol{f_a} = (V(head_{\mathcal{Q}}), m)$, with

$$m = \prod_{\boldsymbol{f} \in V(body_{\mathcal{Q}})} mul_I(\boldsymbol{f}).$$

For convenience, we also say that V *derives* the fact $\boldsymbol{f} = V(head_{\mathcal{Q}})$ if V satisfies $\mathcal{Q}$ on I. The result of V on an instance I, denoted $[\mathcal{Q}, V](I)$, is the bag of annotated facts derived by V on instance I. This bag is empty when V does not satisfy $\mathcal{Q}$ on I. When V does satisfy $\mathcal{Q}$ on I, the set $Facts([\mathcal{Q}, V](I))$ is always a singleton. The *result* $\mathcal{Q}(I)$ of a conjunctive query $\mathcal{Q} \in \mathbf{CQ}^{\neq}$ on I is defined as the bag union over all results of satisfying valuations for $\mathcal{Q}$ on I:

$$\mathcal{Q}(I) = \bigcup_{V \in \mathcal{V}} [\mathcal{Q}, V](I)$$

with $\mathcal{V}$ the set containing all valuations that satisfy $\mathcal{Q}$ on I.

2.3 Networks, data distribution and policies

A *network* $\mathcal{N}$ is a nonempty finite set of values from **dom**, called *nodes*.

A distribution policy specifies how a database, possibly already distributed, is reshuffled by determining which fact is sent to which server. Formally, a *distribution policy* $\boldsymbol{P} = (U, rfacts_{\boldsymbol{P}})$ for a database schema $\mathcal{D}$ and a network $\mathcal{N}$ consists of a universe U and a total function $rfacts_{\boldsymbol{P}} : \mathcal{N} \to 2^{Facts(\mathcal{D},U)}$ mapping each node $\kappa \in \mathcal{N}$ onto a set of facts from $Facts(\mathcal{D}, U)$. A node $\kappa \in \mathcal{N}$ is *responsible* for a fact $\boldsymbol{f} \in Facts(\mathcal{D}, U)$ under $\boldsymbol{P}$ if $\boldsymbol{f} \in rfacts_{\boldsymbol{P}}(\kappa)$. For an instance I, the function $loc\text{-}inst_{\boldsymbol{P},I}$ maps each node $\kappa \in \mathcal{N}$ to the bag of facts it is responsible for. More formally, $(\boldsymbol{f}, m) \in loc\text{-}inst_{\boldsymbol{P},I}(\kappa)$ iff $(\boldsymbol{f}, m) \in I$ and $\boldsymbol{f} \in rfacts_{\boldsymbol{P}}(\kappa)$. We refer to I as the *global instance* and to $loc\text{-}inst_{\boldsymbol{P},I}(\kappa)$ as the *local instance at node* κ.

As distribution policies are defined on facts, either all copies of a certain fact are sent to a specific server or none are. The latter happens for instance when using hash functions to define distribution policies as is the case for instance for Hypercube [3, 6, 9].

Next, we define the one-round distributed evaluation induced by $\boldsymbol{P}$. Query $\mathcal{Q}$ is evaluated at each node κ separately, after which the bag union of all results is taken:

$$[\mathcal{Q}, \boldsymbol{P}](I) = \bigcup_{\kappa \in \mathcal{N}} \mathcal{Q}(loc\text{-}inst_{\boldsymbol{P},I}(\kappa)).$$

2.4 Classes of distribution policies

To reason about the complexity of problems involving distribution policies (which are just defined as functions), we need to consider a representation mechanism for these policies. For this, we first discuss the classes $\mathcal{P}_{fin}$ and $\mathfrak{P}_{nondet}$ as introduced by Ameloot et al. [5] and then describe the class $\mathfrak{P}_{det}$.

The class $\mathcal{P}_{fin}$ is defined over distribution policies with a finite universe. Intuitively, $\mathcal{P}_{fin}$ allows to express all distribution policies over a finite universe, but uses the most naive and exhaustive representation mechanism: explicit enumeration. Formally, a policy $\boldsymbol{P} = (U, rfacts_{\boldsymbol{P}})$ belongs to $\mathcal{P}_{fin}$ if U is a finite set. Such policies are represented by an explicit enumeration of the data values in U and an explicit enumeration of all pairs $(\kappa, \boldsymbol{f})$ where $\boldsymbol{f} \in rfacts_{\boldsymbol{P}}(\kappa)$.

A more general way to describe classes of distribution policies by an arbitrarily succinct representation is by means of a "test algorithm" that allows to decide $\boldsymbol{f} \in \mathit{rfacts}_{\boldsymbol{P}}(\kappa)$ with time bound ℓ^k, where ℓ is the length of the input and k a constant. We call this class $\mathfrak{P}_{nondet}$. More precisely, a policy $\boldsymbol{P} = (U, \mathit{rfacts}_{\boldsymbol{P}})$ over network $\mathcal{N}$ is in $\mathcal{P}^k_{nondet}$ if it is specified by a pair $(n, \mathcal{A}_P)$, with n a natural number in unary representation and $\mathcal{A}_P$ a non-deterministic algorithm. The value n is used to give an upper bound to the length of data values in universe U and on the names of nodes in $\mathcal{N}$. More specifically, the universe U consists of all data values representable by a string of length at most n and the network $\mathcal{N}$ consists of all nodes representable by strings of length at most n. A fact f is in $\mathit{rfacts}_{\boldsymbol{P}}(\kappa)$ for a given node κ if $\mathcal{A}_P$ has an accepting run of at most $|(\kappa, \boldsymbol{f})|^k$ steps on input $(\kappa, \boldsymbol{f})$. We define $\mathfrak{P}_{nondet}$ as the set $\{\mathcal{P}^k_{nondet} \mid k \geq 2\}$. We remark that each policy in $\mathcal{P}_{fin}$ can thus be described in $\mathcal{P}^2_{nondet}$.

The complexity of deciding set-parallel-correctness is so high that complexity bounds are retained even when considering policies in $\mathcal{P}^k_{nondet}$. For bag-parallel-correctness this is not the case and considering policies from $\mathcal{P}^k_{nondet}$ artificially increases the complexity of the decision problem. Therefore, for bag-parallel-correctness, we use the class $\mathcal{P}^k_{det}$, which is defined next. A policy $\boldsymbol{P} = (U, \mathit{rfacts}_{\boldsymbol{P}})$ is in $\mathcal{P}^k_{det}$ if it can be specified by a tuple $(\mathcal{N}, n, \mathcal{A}_P)$ where $\mathcal{N}$ is an explicit enumeration of the nodes in the network, n is a natural number in unary representation and $\mathcal{A}_P$ is a *deterministic* algorithm. The universe U of $\boldsymbol{P}$ is the set of values representable by strings of length at most n. Given a fact $\boldsymbol{f}$ and node κ, algorithm $\mathcal{A}_P$ decides in at most $|(\kappa, \boldsymbol{f})|^k$ steps whether $\boldsymbol{f} \in \mathit{rfacts}_{\boldsymbol{P}}(\kappa)$. We define $\mathfrak{P}_{det}$ as the set of policies $\{\mathcal{P}^k_{det} \mid k \geq 2\}$.

Since each distribution policy implicitly induces a network and each query implicitly defines a database schema, we often omit the explicit notation for networks and schemas.

3 Parallel-correctness

Intuitively, the notion of parallel-correctness relates to whether the distributed execution of a query with relation to a specific distribution policy produces the correct result. That is, whether the distributed execution produces the same result as when the query was evaluated on the global instance.

3.1 Definition and results for set-parallel-correctness

We distinguish between parallel-correctness under the set and under the bag semantics. The former was introduced in [5] and we refer to it as set-parallel-correctness. We next generalize the notion to bag semantics and call it bag-parallel-correctness. Recall that $\mathit{Facts}(F)$ denotes the set of facts occurring in the bag F.

▶ **Definition 3.1.** Let $\mathcal{Q}$ be a query and $\boldsymbol{P}$ a distribution policy. Then,

- $\mathcal{Q}$ is *bag-parallel-correct on instance I under $\boldsymbol{P}$* if $\mathcal{Q}(I) = [\mathcal{Q}, \boldsymbol{P}](I)$;
- $\mathcal{Q}$ is *set-parallel-correct on instance I under $\boldsymbol{P}$* if $\mathit{Facts}(\mathcal{Q}(I)) = \mathit{Facts}([\mathcal{Q}, \boldsymbol{P}](I))$; and,
- $\mathcal{Q}$ is *bag-parallel-correct (resp., set-) under $\boldsymbol{P}$* if $\mathcal{Q}$ is bag-parallel-correct (resp., set-) on all instances I under $\boldsymbol{P}$.

We now formally define the decision problems related to parallel-correctness. In the following, $\mathcal{C}$ denotes a query class, $\mathcal{P}$ denotes a class of distribution policies, and $x \in \{\text{set}, \text{bag}\}$. Then, define the following problem definitions:

	$\mathbf{PCI}^x(\mathcal{C}, \mathcal{P}, \mathbf{I})$
Input:	Query $\mathcal{Q} \in \mathcal{C}$, distribution policy $\boldsymbol{P} \in \mathcal{P}$, instance I
Question:	Is $\mathcal{Q}$ x-parallel-correct on I under $\boldsymbol{P}$?

	$\mathbf{PC}^x(\mathcal{C}, \mathcal{P})$
Input:	Query $\mathcal{Q} \in \mathcal{C}$, distribution policy $\boldsymbol{P} \in \mathcal{P}$
Question:	Is $\mathcal{Q}$ x-parallel-correct under $\boldsymbol{P}$?

We recall the following result by Ameloot et al. [5]:

▶ **Theorem 3.2** ([5]). *Problems* $\mathbf{PCI}^{set}(\mathcal{C}, \mathcal{P})$ *and* $\mathbf{PC}^{set}(\mathcal{C}, \mathcal{P})$ *are* Π_2^p*-complete for every query class* $\mathcal{C} \in \{\mathbf{CQ}, \mathbf{CQ}^{\neq}\}$ *and for every policy class* $\mathcal{P} \in \{\mathcal{P}_{fin}\} \cup \mathfrak{P}_{nondet}$.

The upper bounds given by the above theorem follow rather directly from the semantical characterization given in the next lemma. To this end, we need the notion of minimal valuations. For $\mathcal{Q}$ in $\mathbf{CQ}^{\neq}$, a valuation V is *minimal* if there is *no valuation* V' for $\mathcal{Q}$ that derives the same head fact with a strict subset of body facts, that is, such that $V'(body_{\mathcal{Q}}) \subsetneq V(body_{\mathcal{Q}})$ and $V'(head_{\mathcal{Q}}) = V(head_{\mathcal{Q}})$. Recall from the definitions that $V(body_{\mathcal{Q}})$ always refers to a set of facts, regardless of the considered semantics.

▶ **Lemma 3.3** ([5]). *Let* $\mathcal{Q}$ *be in* $\mathbf{CQ}^{\neq}$. *Then* $\mathcal{Q}$ *is set-parallel-correct under distribution policy* $\boldsymbol{P} = (U, rfacts_{\boldsymbol{P}})$ *if and only if for every minimal valuation* V *for* $\mathcal{Q}$ *over* U, *there is a node* $\kappa \in \mathcal{N}$ *such that* $V(body_{\mathcal{Q}}) \subseteq rfacts_{\boldsymbol{P}}(\kappa)$.

3.2 Bag-parallel-correctness

We now discuss the problem of deciding bag-parallel-correctness. To start, we obtain a property that characterizes bag-parallel-correctness in direct analogy to Lemma 3.3. The characterization for bag-parallel-correctness is again related to valuations but is more strict than the condition of Lemma 3.3 in two different ways. First, the condition should now hold for *all* valuations not just the minimal ones. Second, the condition requires that, for each valuation, *there can be only one* node harboring all the required facts for that valuation.

To prove the next lemma, we introduce the notion of support. For $\mathcal{Q} \in \mathbf{CQ}^{\neq}$ and distribution policy $\boldsymbol{P}$, we say that node κ *supports* valuation V for $\mathcal{Q}$, if $V(body_{\mathcal{Q}}) \subseteq rfacts_{\boldsymbol{P}}(\kappa)$. By $Sup_{\boldsymbol{P}}(\mathcal{Q}, V)$, we denote the set of all nodes that support V under $\boldsymbol{P}$.

▶ **Lemma 3.4** (Highlander Lemma[3]). *For* $\mathcal{Q} \in \mathbf{CQ}^{\neq}$ *and a distribution policy* $\boldsymbol{P} = (U, rfacts_{\boldsymbol{P}})$ *over* $\mathcal{N}$, $\mathcal{Q}$ *is bag-parallel-correct under* $\boldsymbol{P}$ *if and only if* $|Sup_{\boldsymbol{P}}(\mathcal{Q}, V)| = 1$, *for every valuation* V *for* $\mathcal{Q}$.

Proof sketch. *(If).* Since every valuation V for $\mathcal{Q}$ is supported by exactly one node, bag-parallel-correctness follows trivially.

(Only-if). Let $\mathcal{Q}$ be bag-parallel-correct for $\boldsymbol{P}$. We claim that for all valuations V for $\mathcal{Q}$, $|Sup_{\boldsymbol{P}}(\mathcal{Q}, V)| \leq 1$. Indeed, if there is a valuation V with $|Sup_{\boldsymbol{P}}(\mathcal{Q}, V)| > 1$ and $\boldsymbol{f} = V(head_{\mathcal{Q}})$, it follows that the multiplicity of $\boldsymbol{f}$ in $[\mathcal{Q}, \boldsymbol{P}](I)$ with $Facts(I) = V(body_{\mathcal{Q}})$ is too high.

It remains to argue that there cannot be a valuation V for $\mathcal{Q}$ with $|Sup_{\boldsymbol{P}}(\mathcal{Q}, V)| = 0$. Assume towards a contradiction that such a valuation V exists, and let $\boldsymbol{f} = V(head_{\mathcal{Q}})$. For

[3] "There can be only one." `https://en.wikipedia.org/wiki/Highlander_(film)`

an arbitrary instance I with $Facts(I) = V(body_{\mathcal{Q}})$, the resulting multiplicity of $\boldsymbol{f}$ in $[\mathcal{Q}, \boldsymbol{P}](I)$ is too low, unless there is a valuation W for $\mathcal{Q}$ with $\boldsymbol{f} = W(head_{\mathcal{Q}})$ and $|Sup_{\boldsymbol{P}}(\mathcal{Q}, W)| > 1$. But this contradicts our earlier claim. ◀

We next obtain the complexity of deciding bag-parallel-correctness. The upper bound follows rather directly from Lemma 3.4. The lower bound is a reduction from the complement of 3-SAT.

▶ **Theorem 3.5.** $\mathbf{PC}^{bag}(\mathcal{C}, \mathcal{P})$ *is* coNP*-complete for every query class* $\mathcal{C} \in \{\mathbf{CQ}, \mathbf{CQ}^{\neq}\}$ *and every policy class* $\mathcal{P} \in \{\mathcal{P}_{fin}\} \cup \mathfrak{P}_{det}$*, even over networks with only two nodes.*

3.3 Relationship between set- and bag-parallel-correctness

We next address the relationship between set- and bag-parallel-correctness. The implication in the following proposition follows immediately from Lemma 3.3 and Lemma 3.4. A counterexample for the converse is given in Example 3.7.

▶ **Proposition 3.6.** *Bag-parallel-correctness implies set-parallel-correctness for queries in* $\mathbf{CQ}^{\neq}$*, but not vice-versa.*

▶ **Example 3.7.** For an example showing that the reverse direction of Proposition 3.6 does not hold, consider query $\mathcal{Q}$: $T(x) \leftarrow R(x)$. Let $\boldsymbol{P} = (U, rfacts_{\boldsymbol{P}})$ be a distribution policy over network $\mathcal{N} = \{\kappa_1, \kappa_2\}$, with $rfacts_{\boldsymbol{P}}(\kappa_1) = rfacts_{\boldsymbol{P}}(\kappa_2) = \{R(a), R(b)\}$, and $U = \{a, b\}$.

We observe that $\mathcal{Q}$ has only two valuations under U which in addition are minimal: $V_a = \{x \mapsto a\}$ and $V_b = \{x \mapsto b\}$. Since $Sup_{\boldsymbol{P}}(\mathcal{Q}, V_1) = Sup_{\boldsymbol{P}}(\mathcal{Q}, V_2) = \{\kappa_1, \kappa_2\}$ it follows immediately from Lemma 3.3 and Lemma 3.4 that $\mathcal{Q}$ is set-parallel-correct, but not bag-parallel-correct, under $\boldsymbol{P}$. ◀

Interestingly, we can identify a class of $\mathbf{CQ}^{\neq}$-queries and a class of distribution policies for which the notions of set- and bag-parallel-correctness coincide. First, we introduce the necessary definitions.

A query in $\mathbf{CQ}^{\neq}$ is *strongly minimal* if all its valuations are minimal. We consider the family of non-replicating distribution policies that do not replicate any fact onto multiple nodes. More formally, a distribution policy $\boldsymbol{P} = (U, rfacts_{\boldsymbol{P}})$ over a network $\mathcal{N}$ is *non-replicating* if and only if $rfacts_{\boldsymbol{P}}(\kappa_1) \cap rfacts_{\boldsymbol{P}}(\kappa_2) = \emptyset$ for every pair of nodes $\kappa_1, \kappa_2 \in \mathcal{N}$ with $\kappa_1 \neq \kappa_2$.

▶ **Theorem 3.8.** *For a strongly minimal query* $\mathcal{Q}$ *in* $\mathbf{CQ}^{\neq}$ *and a non-replicating distribution policy* $\boldsymbol{P}$*,* $\mathcal{Q}$ *is bag-parallel-correct under* $\boldsymbol{P}$ *iff* $\mathcal{Q}$ *is set-parallel-correct under* $\boldsymbol{P}$*.*

Proof. It follows from Proposition 3.6 that bag-parallel-correctness of $\mathcal{Q}$ under $\boldsymbol{P}$ implies set-parallel-correctness. We show the reverse direction through Lemma 3.4. For this, let V be an arbitrary valuation for $\mathcal{Q}$. Since $\mathcal{Q}$ is set-parallel-correct under $\boldsymbol{P}$, and V is minimal (due to strong minimality of $\mathcal{Q}$), it follows from Lemma 3.3 that $|Sup_{\boldsymbol{P}}(\mathcal{Q}, V)| \geq 1$. Since $\boldsymbol{P}$ is non-replicating, the latter implies $|Sup_{\boldsymbol{P}}(\mathcal{Q}, V)| = 1$. ◀

Notice that in the constructed counterexample from Example 3.7, the query $\mathcal{Q}$ is strongly minimal, but $\boldsymbol{P}$ is replicating. In the following example we show that, for Theorem 3.8, the condition that $\mathcal{Q}$ is strongly minimal can not be dropped.

▶ **Example 3.9.** Consider query $\mathcal{Q}$: $T(x) \leftarrow R(x), R(y)$, and network $\mathcal{N} = \{\kappa_1, \kappa_2\}$. Let $\boldsymbol{P} = (U, rfacts_{\boldsymbol{P}})$ be a distribution policy over $U = \{a, b\}$ and $\mathcal{N}$, with $rfacts_{\boldsymbol{P}}(\kappa_1) = \{R(a)\}$ and $rfacts_{\boldsymbol{P}}(\kappa_2) = \{R(b)\}$. Notice that $\boldsymbol{P}$ is non-replicating.

We observe that $\boldsymbol{P}$ is set-parallel-correct for $\mathcal{Q}$. Indeed, there are only two minimal valuations for $\mathcal{Q}$ over U: $V_a = \{x \mapsto a, y \mapsto a\}$ and $V_b = \{x \mapsto b, y \mapsto b\}$. Furthermore, V_a is supported by κ_1 while V_b is supported by κ_2. The result then follows from Lemma 3.3.

For non-minimal valuation $V = \{x \mapsto a, y \mapsto b\}$, we observe that $|Sup_{\boldsymbol{P}}(\mathcal{Q}, V)| = \emptyset$. Thus $\boldsymbol{P}$ cannot be bag-parallel-correct for $\mathcal{Q}$ (due to Lemma 3.4). ◀

4 Transferability

Parallel-correctness *transfers* from a query $\mathcal{Q}$ to a query $\mathcal{Q}'$ when $\mathcal{Q}'$ is parallel-correct under every distribution policy $\boldsymbol{P}$ under which $\mathcal{Q}$ is parallel-correct. This means in particular that query $\mathcal{Q}'$ can *always* be evaluated after query $\mathcal{Q}$ *without* an intermediate, possibly expensive, reshuffling of the data. The present section studies parallel-correctness transfer under bag semantics.

4.1 Definition and results for transferability under set semantics

The notion of parallel-correctness transfer was introduced by Ameloot et al. [5]. We next distinguish between transferability under set and bag semantics.

▶ **Definition 4.1.** For two queries $\mathcal{Q}$ and $\mathcal{Q}'$ over the same input schema, *bag-parallel-correctness transfers from $\mathcal{Q}$ to $\mathcal{Q}'$* if $\mathcal{Q}'$ is bag-parallel-correct under every distribution policy for which $\mathcal{Q}$ is bag-parallel-correct. In this case, we write $\mathcal{Q} \xrightarrow{\text{bag}} \mathcal{Q}'$. Set-parallel-correctness transferability is defined similarly and denoted by $\mathcal{Q} \xrightarrow{\text{set}} \mathcal{Q}'$.

▶ **Lemma 4.2** ([5]). *For queries $\mathcal{Q}, \mathcal{Q}' \in \mathbf{CQ}^{\neq}$, set-parallel-correctness transfers from $\mathcal{Q}$ to $\mathcal{Q}'$ if for each minimal valuation V' for $\mathcal{Q}'$ there is a minimal valuation V for $\mathcal{Q}$ where $V'(body_{\mathcal{Q}'}) \subseteq V(body_{\mathcal{Q}})$ and $adom(V'(body_{\mathcal{Q}'})) = adom(V(body_{\mathcal{Q}}))$.*

4.2 Transferability under bag semantics

The following example highlights how, depending on the structure of the query, different valuations must be supported by the *same* compute node for distribution policies under which the query is bag-parallel-correct. In particular, the example shows that the assignment of a fact to a particular node can *imply* that other facts should be assigned to that same node as well.

▶ **Example 4.3.** Consider the query $\mathcal{Q} : H(x) \leftarrow R(x, y), R(x, z)$. Let $\boldsymbol{P}$ be a distribution policy under which $\mathcal{Q}$ is bag-parallel-correct. Assume $R(a, a) \in rfacts_{\boldsymbol{P}}(\kappa)$ for some node κ. Then, by Lemma 3.4, every fact of the form $R(a, c)$ for any c should belong to $rfacts_{\boldsymbol{P}}(\kappa)$ as well. Furthermore, denoting the valuation $\{x \mapsto a, y \mapsto b, z \mapsto c\}$ by $W_{a,b,c}$, the following set of valuations $\{W_{a,b,c} \mid b, c \in U\}$ for a fixed a have to be supported by the same node. ◀

We formally define the set of facts that are implied by a valuation w.r.t. a given query.

▶ **Definition 4.4.** Let V be a valuation for $\mathcal{Q} \in \mathbf{CQ}^{\neq}$. A fact $\boldsymbol{f}$ is *implied* by V w.r.t. $\mathcal{Q}$ if for every distribution policy $\boldsymbol{P} = (U, rfacts_{\boldsymbol{P}})$, with $adom(V(body_{\mathcal{Q}})) \subseteq U$ under which $\mathcal{Q}$ is bag-parallel-correct, and for every node κ in the network of $\boldsymbol{P}$: $V(body_{\mathcal{Q}}) \subseteq rfacts_{\boldsymbol{P}}(\kappa)$ implies $\boldsymbol{f} \in rfacts_{\boldsymbol{P}}(\kappa)$. We denote the set of facts implied by V w.r.t. $\mathcal{Q}$ by $ImpFacts(V, \mathcal{Q})$.

Notice that $ImpFacts(V, \mathcal{Q})$ is well-defined as there is always a distribution policy under which Q is bag-parallel-correct: namely, the policy which is defined over a single-node network

and maps all facts to a single node. Furthermore, $\mathit{ImpFacts}(V, \mathcal{Q}) \subseteq \mathit{rfacts}_{\boldsymbol{P}}(\kappa)$ whenever $V(\mathit{body}_{\mathcal{Q}}) \subseteq \mathit{rfacts}_{\boldsymbol{P}}(\kappa)$ for every distribution policy $\boldsymbol{P}$ under which $\mathcal{Q}$ is bag-parallel-correct.

We are now ready to characterize bag-parallel-correctness transfer. The lemma plays a role similar to the Highlander Lemma and requires that every valuation for the second query is sandwiched between a valuation for the first query and the implied facts.

▶ **Lemma 4.5** (Sandwich lemma). *Bag-parallel-correctness transfers from $\mathcal{Q}$ to $\mathcal{Q}'$ if and only if for each valuation V' for $\mathcal{Q}'$ there is a valuation V for $\mathcal{Q}$ such that $V(\mathit{body}_{\mathcal{Q}}) \subseteq V'(\mathit{body}_{\mathcal{Q}'}) \subseteq \mathit{ImpFacts}(V, \mathcal{Q})$.*

Proof. *(If).* Let $\boldsymbol{P} = (U, \mathit{rfacts}_{\boldsymbol{P}})$ be an arbitrary distribution policy such that $\mathcal{Q}$ is bag-parallel-correct under $\boldsymbol{P}$. Let V' be an arbitrary valuation for $\mathcal{Q}'$ over U. We argue that $|\mathit{Sup}_{\boldsymbol{P}}(\mathcal{Q}', V')| = 1$ which by Lemma 3.4 implies that $\mathcal{Q}'$ is bag-parallel-correct under $\boldsymbol{P}$ as well. By assumption there is a valuation V for $\mathcal{Q}$ over U such that $V(\mathit{body}_{\mathcal{Q}}) \subseteq V'(\mathit{body}_{\mathcal{Q}'}) \subseteq \mathit{ImpFacts}(V, \mathcal{Q})$. Then, by Lemma 3.4, $\mathit{Sup}_{\boldsymbol{P}}(\mathcal{Q}, V) = \{\kappa\}$ for some node κ and $\mathit{ImpFacts}(V, \mathcal{Q}) \subseteq \mathit{rfacts}_{\boldsymbol{P}}(\kappa)$. Therefore, $V'(\mathit{body}_{\mathcal{Q}'}) \subseteq \mathit{rfacts}_{\boldsymbol{P}}(\kappa)$. So, $|\mathit{Sup}_{\boldsymbol{P}}(\mathcal{Q}', V')| \geq 1$. However, as $V(\mathit{body}_{\mathcal{Q}}) \subseteq \mathit{rfacts}_{\boldsymbol{P}}(\kappa)$ and $\mathit{Sup}_{\boldsymbol{P}}(\mathcal{Q}, V) = \{\kappa\}$, $|\mathit{Sup}_{\boldsymbol{P}}(\mathcal{Q}', V')| = 1$.

(Only-If). The proof is by contraposition. In particular, we show that bag-parallel-correctness does not transfer from $\mathcal{Q}$ to $\mathcal{Q}'$ if the condition of the lemma fails for some valuation V' for $\mathcal{Q}'$. We distinguish two cases: the case when no valuation V for $\mathcal{Q}$ exists with $V(\mathit{body}_{\mathcal{Q}}) \subseteq V'(\mathit{body}_{\mathcal{Q}'})$, and the case when for each valuation V, with $V(\mathit{body}_{\mathcal{Q}}) \subseteq V'(\mathit{body}_{\mathcal{Q}'})$, we have that $V'(\mathit{body}_{\mathcal{Q}'}) \not\subseteq \mathit{ImpFacts}(V, \mathcal{Q})$.

Case 1: there is no valuation V with $V(\mathit{body}_{\mathcal{Q}}) \subseteq V'(\mathit{body}_{\mathcal{Q}'})$. We construct the policy $\boldsymbol{P}$ over a two-node network $\{\kappa_1, \kappa_2\}$ and universe U consisting of all domain values used by V', with $\mathit{rfacts}_{\boldsymbol{P}}(\kappa_1) = \mathit{Facts}(\mathcal{D}, U)$ and $\mathit{rfacts}_{\boldsymbol{P}}(\kappa_2) = V'(\mathit{body}_{\mathcal{Q}'})$. Then, $\mathit{Sup}_{\boldsymbol{P}}(Q', V') = \{\kappa_1, \kappa_2\}$ and Lemma 3.4 implies that $\boldsymbol{P}$ is not bag-parallel-correct for $\mathcal{Q}'$. In contrast, every valuation for $\mathcal{Q}$ is supported only on node κ_1 (as none of them are included in $V'(\mathit{body}_{\mathcal{Q}'})$) which implies that $\boldsymbol{P}$ is bag-parallel-correct for $\mathcal{Q}$. We conclude that bag-parallel-correctness does not transfer from $\mathcal{Q}$ to $\mathcal{Q}'$.

Case 2: for each valuation V, $V(\mathit{body}_{\mathcal{Q}}) \subseteq V'(\mathit{body}_{\mathcal{Q}'})$ implies $V'(\mathit{body}_{\mathcal{Q}'}) \not\subseteq \mathit{ImpFacts}(V, \mathcal{Q})$. From the previous case, we can assume the existence of a valuation V with $V(\mathit{body}_{\mathcal{Q}}) \subseteq V'(\mathit{body}_{\mathcal{Q}'})$. Then, by definition of $\mathit{ImpFacts}(V, \mathcal{Q})$, $V'(\mathit{body}_{\mathcal{Q}'}) \not\subseteq \mathit{ImpFacts}(V, \mathcal{Q})$ implies that there must be a policy $\boldsymbol{P}$ (over some network $\mathcal{N}$) such that $\mathcal{Q}$ is bag-parallel-correct under $\boldsymbol{P}$ and $\boldsymbol{P}$ has a node κ with $V(\mathit{body}_{\mathcal{Q}}) \subseteq \mathit{rfacts}_{\boldsymbol{P}}(\kappa)$ and $V'(\mathit{body}_{\mathcal{Q}'}) \not\subseteq \mathit{rfacts}_{\boldsymbol{P}}(\kappa)$. From Lemma 3.4, it follows that for all other nodes κ', that is $\kappa' \in \mathcal{N} \setminus \{\kappa\}$, $V(\mathit{body}_{\mathcal{Q}}) \not\subseteq \mathit{rfacts}_{\boldsymbol{P}}(\kappa')$, and thus $V'(\mathit{body}_{\mathcal{Q}'}) \not\subseteq \mathit{rfacts}_{\boldsymbol{P}}(\kappa')$. Hence, $\boldsymbol{P}$ is not bag-parallel-correct for $\mathcal{Q}'$ and, consequently, bag-parallel-correctness does not transfer from $\mathcal{Q}$ to $\mathcal{Q}'$. ◀

Notice that the inclusion between $V(\mathit{body}_{\mathcal{Q}})$ and $V'(\mathit{body}_{\mathcal{Q}'})$ in Lemma 4.5 is in the opposite direction as in Lemma 4.2, since the inclusion now asserts that V' is supported by at most one node instead of at least one.

We formally define the respective decision problems for $x \in \{\text{set}, \text{bag}\}$. By $\mathcal{C}$ and $\mathcal{C}'$ we denote query classes.

	PC-Trans$^x(\mathcal{C}, \mathcal{C}')$
Input:	Query $\mathcal{Q} \in \mathcal{C}$, query $\mathcal{Q}' \in \mathcal{C}'$
Question:	Does x-parallel-correctness transfer from $\mathcal{Q}$ to $\mathcal{Q}'$?

Algorithm 1 MAX-PROOF-FOREST$(\mathcal{Q}, U)$.

Let $\mathcal{I}$ be the set of single-node IF-proof-trees, one for each set $V(body_\mathcal{Q})$, where V is a valuation for $\mathcal{Q}$ over U.
while Distinct $\mathbf{T}_1, \mathbf{T}_2 \in \mathcal{I}$ and V for $\mathcal{Q}$ over U exist, with $V(body_\mathcal{Q}) \subseteq Inst_{\mathbf{T}_1}(n_1) \cap Inst_{\mathbf{T}_2}(n_2)$, with n_1, n_2 the roots of $\mathbf{T}_1, \mathbf{T}_2$ respectively **do**
  Remove $\mathbf{T}_1$ and $\mathbf{T}_2$ from $\mathcal{I}$
  Insert new node n with children $\mathbf{T}_1$ and $\mathbf{T}_2$ to $\mathcal{I}$
  $Inst_\mathbf{T}(n) = Inst_{\mathbf{T}_1}(n_1) \cup Inst_{\mathbf{T}_2}(n_2)$;
end while
return $\mathcal{I}$

Algorithm 2 MAX-PROOF-TREE$(V, \mathcal{Q}, U)$.

Compute MAX-PROOF-FOREST$(\mathcal{Q}, U)$.
return The unique tree $\mathbf{T}$, with $V(body_\mathcal{Q}) \subseteq Inst_\mathbf{T}(n)$, where n is the root of $\mathbf{T}$.

Recall that under set semantics $\mathbf{PC\text{-}Trans}^{set}(\mathbf{CQ}^{\neq}, \mathbf{CQ}^{\neq})$ is Π^p_3-complete [5]. In the remainder of this section, we obtain the following result:

▶ **Theorem 4.6.** $\mathbf{PC\text{-}Trans}^{bag}(\mathbf{CQ}^{\neq}, \mathbf{CQ}^{\neq})$ *is in* EXPTIME.

We introduce IF-proof-trees as a means for reasoning on implied facts.

▶ **Definition 4.7.** For a query $\mathcal{Q}$ and universe $U \subseteq \mathbf{dom}$, an *IF-proof-tree* $\boldsymbol{T}$ *for* $\mathcal{Q}$ *over* U is a binary tree in which all nodes n have an instance $Inst_\mathbf{T}(n)$ as label with the following conditions:

1. If n is a leaf, then $Inst_\mathbf{T}(n) = V(body_\mathcal{Q})$ for some valuation V for $\mathcal{Q}$ over U;
2. If n is an intermediate node with children n_1 and n_2, then $Inst_\mathbf{T}(n) = Inst_\mathbf{T}(n_1) \cup Inst_\mathbf{T}(n_2)$, and some valuation V for $\mathcal{Q}$ over U exists with $V(body_\mathcal{Q}) \subseteq Inst_\mathbf{T}(n_1) \cap Inst_\mathbf{T}(n_2)$.

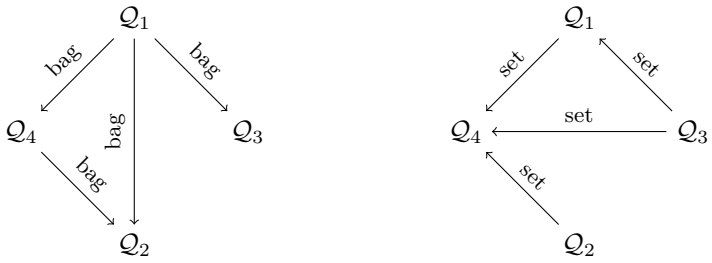
In the next lemma, we relate IF-proof-trees and bag-parallel-correct distribution policies. In particular, the lemma says that all facts occurring together in an IF-proof-tree for a given query have to be assigned to exactly one compute node by every distribution policy that is bag-parallel-correct for that query.

▶ **Lemma 4.8.** *Let* $\mathcal{Q} \in \mathbf{CQ}^{\neq}$ *and* $\boldsymbol{T}$ *an IF-proof-tree over universe* U'*. For every distribution policy* $\boldsymbol{P} = (U, rfacts_\boldsymbol{P})$ *with* $U' \subseteq U$ *(over some network* $\mathcal{N}$*) that is bag-parallel-correct for* $\mathcal{Q}$*, there is exactly one node* $\kappa \in \mathcal{N}$*, with* $Inst_\boldsymbol{T}(n) \subseteq rfacts_\boldsymbol{P}(\kappa)$*, for every* n *in* $\boldsymbol{T}$*.*

Algorithm 1 is a procedure that constructs all maximal IF-proof-trees. We notice that at each point during the evaluation of MAX-PROOF-FOREST$(\mathcal{Q}, U)$, all trees in $\mathcal{I}$ are valid IF-proof-trees for $\mathcal{Q}$ and U, by construction. In particular, the output of Algorithm 1 contains for every valuation V a unique tree with $V(body_\mathcal{Q}) \subseteq Inst_\mathbf{T}(n)$. Indeed, if two such trees would exist, they would have been combined into a new tree by construction. Algorithm 2 then selects the unique tree w.r.t. a given valuation. We notice that MAX-PROOF-TREE$(V, \mathcal{Q}, U)$ is well-defined, since, if V is a valuation for $\mathcal{Q}$ over U, then the desired tree $\mathbf{T}$ indeed exists.

The next lemma shows that MAX-PROOF-TREE$(V, \mathcal{Q}, U)$ computes precisely the facts that are implied by V and $\mathcal{Q}$.

▶ **Lemma 4.9.** *For a query* $\mathcal{Q}$ *and valuation* V *for* $\mathcal{Q}$*,* $\boldsymbol{f} \in ImpFacts(V, \mathcal{Q})$ *if and only if* $\boldsymbol{f} \in Inst_\boldsymbol{T}(n)$*, with* n *being the root of* $\boldsymbol{T} =$MAX-PROOF-TREE$(V, \mathcal{Q}, U)$.



(a) Bag-parallel-correctness transfer. **(b)** Set-parallel-correctness transfer.

Figure 1 Relationship between the queries of Section 4.3 with respect to (a) bag-parallel-correctness transfer and (b) set-parallel-correctness transfer.

Observe that when U is finite, MAX-PROOF-TREE$(V, \mathcal{Q}, U)$ runs in time exponential in the size of $\mathcal{Q}$ and U. The next lemma says that we can restrict attention to finite universes of size bounded by the number of variables in the queries.

▶ **Lemma 4.10.** *Let $\mathcal{Q}, \mathcal{Q}' \in \mathbf{CQ}^{\neq}$ and $\boldsymbol{dom}_k = \{1, \ldots, k\}$ be a subset of $\boldsymbol{dom}$, where $k = max(|Vars(\mathcal{Q})|, |Vars(\mathcal{Q}')|)$. The following conditions are equivalent:*

(1) *For each valuation V' for $\mathcal{Q}'$ over $U \subseteq \boldsymbol{dom}$, there exists a valuation V for $\mathcal{Q}$ over U such that $V(body_{\mathcal{Q}}) \subseteq V'(body_{\mathcal{Q}'}) \subseteq ImpFacts(V, \mathcal{Q})$.*

(2) *For each valuation W' for $\mathcal{Q}'$ over $U_k \subseteq \boldsymbol{dom}_k$, there exists a valuation W for $\mathcal{Q}$ over U_k such that $W(body_{\mathcal{Q}}) \subseteq W'(body_{\mathcal{Q}'}) \subseteq ImpFacts(W, \mathcal{Q})$.*

We are now ready to prove Theorem 4.6.

Proof. *(of Theorem 4.6)* The proof is by a naive verification of condition (2) of Lemma 4.10. More specifically, for every universe $U \subseteq \mathbf{dom}_k$ and every valuation V for $\mathcal{Q}$ over U, we compute $ImpFacts(V, \mathcal{Q})$ through MAX-PROOF-TREE$(V, \mathcal{Q}, U)$ (cf. Lemma 4.9). Then, for every valuation V' for $\mathcal{Q}'$ over U and every valuation V for $\mathcal{Q}$ over U we test condition $V(body_{\mathcal{Q}}) \subseteq V'(body_{\mathcal{Q}'}) \subseteq ImpFacts(V, Q)$. If for some V' no V is found that satisfies the condition, then the algorithm returns *false*, otherwise it returns *true*.

Correctness of the algorithm follows directly from Lemma 4.10 and Lemma 4.5. It remains to show that this algorithm proceeds in exponential time in the size of $\mathcal{Q}$ and $\mathcal{Q}'$. For this, we recall that $\mathbf{dom}_k$ is linear in $\mathcal{Q}$ and $\mathcal{Q}'$ by construction, and thus that there are only exponentially many universes $U \subseteq \mathbf{dom}_k$ (w.r.t $\mathcal{Q}$ and $\mathcal{Q}'$). The set of implied facts for a given V and $\mathcal{Q}$, restricted to U, is computable in exponential time and itself is of at most exponential size. Since only exponentially many valuations for $\mathcal{Q}$ and $\mathcal{Q}'$ exist over U, and the test condition itself proceeds in a linear run over the set of implied facts, the result follows. ◀

4.3 Relationship between transferability under set and bag semantics

We argue that set-parallel-correctness transfer is orthogonal to bag-parallel-correctness transfer. Indeed, consider the following queries:

$\mathcal{Q}_1 : H() \leftarrow R(x, y), R(z, w).$

$\mathcal{Q}_2 : H() \leftarrow R(x, x), R(y, y), R(z, z), x \neq y, y \neq z, x \neq z.$

$\mathcal{Q}_3 : H() \leftarrow R(x, y), R(x, z), y \neq z.$

$\mathcal{Q}_4 : H() \leftarrow R(x, y), R(y, z), R(x, x).$

Figure 1 shows the directions in which set-parallel-correctness transfer and bag-parallel-correctness transfer hold. In particular, when an edge is missing, there is no set- or bag-parallel-correctness transfer between the two queries.

The next lemma follows directly from Theorem 3.8.

▶ **Lemma 4.11.** *For strongly minimal queries $\mathcal{Q}, \mathcal{Q}' \in \mathbf{CQ}^{\neq}$ and non-replicating distribution policies, we have that $\mathcal{Q} \xrightarrow{bag} \mathcal{Q}'$ if and only if $\mathcal{Q} \xrightarrow{set} \mathcal{Q}'$.*

5 Modifying the distribution model

As already hinted upon in the Introduction, the Highlander Lemma of Section 3 implies that the space of valuations for a conjunctive query should be perfectly partitioned over all compute nodes. That is, every valuation should occur in exactly one compute node. We next give a simple example query for which the distribution policies that are bag-parallel-correct for it, have to map all facts to a single node.

▶ **Example 5.1.** Consider the query $\mathcal{Q} : H(x, z) \leftarrow R(x, y), R(y, z)$. We argue that distribution policies that map all facts to a single node are the only distribution policies that are bag-parallel-correct. Indeed, let $\boldsymbol{P}$ be a distribution policy that is bag-parallel-correct for $\mathcal{Q}$. Assume $R(a, a) \in \mathit{rfacts}_{\boldsymbol{P}}(\kappa)$ for some node κ. Then, the valuation $\{x \mapsto a, y \mapsto a, z \mapsto b\}$ (for every b) together with Lemma 3.4, implies that every fact of the form $R(a, b)$ for any b should belong to $\mathit{rfacts}_{\boldsymbol{P}}(\kappa)$ as well. Furthermore, the valuation $\{x \mapsto a, y \mapsto b, z \mapsto c\}$ (for every b and c) together with Lemma 3.4, implies that every fact of the form $R(b, c)$ for any b and any c should belong to $\mathit{rfacts}_{\boldsymbol{P}}(\kappa)$ as well. Consequently, $\boldsymbol{P}$, to be bag-parallel-correct for $\mathcal{Q}$, maps all facts to node κ. ◀

The previous example shows that there are queries where the demand for bag-parallel-correctness effectively prohibits parallel computation. We note that this is not the case for all queries. See for instance Example 4.3.

In this section, we consider the setting of *ordered networks* where every compute node is assigned a number and for every valuation only the node with the smallest number containing all facts required for that valuation can contribute to the query result. While both settings do not differ under set semantics, the new setting is more natural for bag semantics and alleviates the problem put forward in Example 5.1.

We associate a total order $<_{\mathcal{N}}$ to every network $\mathcal{N}$. We refer to these networks as *ordered networks*. The definition of a distribution policy $\boldsymbol{P} = (U, \mathit{rfacts}_{\boldsymbol{P}})$ seamlessly carries over to ordered networks. Let $\mathcal{Q}$ be a query and V be a valuation over U for $\mathcal{Q}$. Then, we say that a node $\kappa \in \mathcal{N}$ is *responsible for V (of $\mathcal{Q}$)* if $V(\mathit{body}_{\mathcal{Q}}) \subseteq \mathit{rfacts}_{\boldsymbol{P}}(\kappa)$ and there is no node $\kappa' \in \mathcal{N}$ with $\kappa' <_{\mathcal{N}} \kappa$ and $V(\mathit{body}_{\mathcal{Q}}) \subseteq \mathit{rfacts}_{\boldsymbol{P}}(\kappa')$. Intuitively, the node responsible for a valuation V is the smallest node in the ordered network containing all the facts for $V(\mathit{body}_{\mathcal{Q}})$.

We redefine the one-round distributed evaluation induced by $\boldsymbol{P}$ and $<_{\mathcal{N}}$ as follows:

$$[\mathcal{Q}, \boldsymbol{P}, <_{\mathcal{N}}](I) = \bigcup_{\kappa \in \mathcal{N}, V \in \mathcal{V}_{\kappa}} [\mathcal{Q}, V](\mathit{loc\text{-}inst}_{\boldsymbol{P},I}(\kappa))$$

with $\mathcal{V}_{\kappa}$ the set of valuations for which κ is responsible.

The notions of set- and bag-parallel-correctness carry over directly to the setting of ordered networks. Notice that under set-semantics it does not matter whether the ordering of nodes is taken into account.

▶ **Proposition 5.2.** *For each query $\mathcal{Q}$, distribution policy $\boldsymbol{P}$, and ordered network $(\mathcal{N}, <_{\mathcal{N}})$, the following hold for all instances I:*

1. $[\mathcal{Q}, \boldsymbol{P}, <_{\mathcal{N}}](I) \subseteq [\mathcal{Q}, \boldsymbol{P}](I)$;
2. $[\mathcal{Q}, \boldsymbol{P}, <_{\mathcal{N}}](I) \subseteq \mathcal{Q}(I)$; *and,*
3. $\mathit{Facts}([\mathcal{Q}, \boldsymbol{P}](I)) = \mathit{Facts}([\mathcal{Q}, \boldsymbol{P}, <_{\mathcal{N}}](I))$;

In particular, Proposition 5.2(3) implies that Theorem 3.2 and Lemma 3.3 carry over to ordered networks. The next lemma provides characterizations of bag-parallel-correctness and transferability over ordered networks.

▶ **Lemma 5.3.** *Let $\mathcal{Q}$ and $\mathcal{Q}'$ be in $\mathbf{CQ}^{\neq}$. Let $\boldsymbol{P} = (U, \mathit{rfacts}_{\boldsymbol{P}})$ be a distribution policy over an ordered network $\mathcal{N}$. Then the following characterizations hold true:*

1. *$\mathcal{Q}$ is bag-parallel-correct under $\boldsymbol{P}$ if and only if for every valuation V for $\mathcal{Q}$ over U there is a node κ with $V(\mathit{body}_{\mathcal{Q}}) \subseteq \mathit{rfacts}_{\boldsymbol{P}}(\kappa)$; and,*
2. *bag-parallel-correctness transfers from $\mathcal{Q}$ to $\mathcal{Q}'$ over ordered networks if and only if for each valuation V' for $\mathcal{Q}'$ over a universe U' there is a valuation V for $\mathcal{Q}$ over U' such that $V'(\mathit{body}_{\mathcal{Q}'}) \subseteq V(\mathit{body}_{\mathcal{Q}})$.*

Notice the similarity with Lemma 3.3 and Lemma 4.2. In particular, the inclusion between $V(\mathit{body}_{\mathcal{Q}})$ and $V'(\mathit{body}_{\mathcal{Q}'})$ now is in the same direction as in Lemma 4.2. The only difference is that in the above lemma *all* valuations are considered rather than only the minimal ones. The latter is reflected in the complexity of the associated decision problems.

We formally define the respective decision problems. By $\mathcal{C}$ and $\mathcal{C}'$ we denote query classes, by $\mathcal{P}$ a class of distribution policies.

	$\mathbf{PC}^{bag}{}_{<_{\mathcal{N}}}(\mathcal{C}, \mathcal{P})$
Input:	Query $\mathcal{Q} \in \mathcal{C}$, distribution policy $\boldsymbol{P} \in \mathcal{P}$
Question:	Is $\mathcal{Q}$ bag-parallel-correct under $\boldsymbol{P}$?

	$\mathbf{PC\text{-}Trans}^{bag}{}_{<_{\mathcal{N}}}(\mathcal{C}, \mathcal{C}')$
Input:	Query $\mathcal{Q} \in \mathcal{C}$, query $\mathcal{Q}' \in \mathcal{C}'$
Question:	Does bag-parallel-correctness transfer from $\mathcal{Q}$ to $\mathcal{Q}'$?

Using the characterizations in Lemma 5.3, we obtain the following results.

▶ **Theorem 5.4.** 1. $\mathbf{PC}^{bag}{}_{<_{\mathcal{N}}}(\mathbf{CQ}, \mathcal{P}_{fin})$ *is* coNP*-hard and* $\mathbf{PC}^{bag}{}_{<_{\mathcal{N}}}(\mathbf{CQ}^{\neq}, \mathcal{P})$ *is in* coNP *for all $\mathcal{P} \in \{\mathcal{P}_{fin}\} \cup \mathfrak{P}_{det}$; and*
2. $\mathbf{PC\text{-}Trans}^{bag}{}_{<_{\mathcal{N}}}(\mathbf{CQ}^{\neq}, \mathbf{CQ}^{\neq})$ *and* $\mathbf{PC\text{-}Trans}^{bag}{}_{<_{\mathcal{N}}}(\mathbf{CQ}^{\neq}, \mathbf{CQ})$ *are Π_2^p-complete; and*
3. $\mathbf{PC\text{-}Trans}^{bag}{}_{<_{\mathcal{N}}}(\mathbf{CQ}, \mathbf{CQ}^{\neq})$ *and* $\mathbf{PC\text{-}Trans}^{bag}{}_{<_{\mathcal{N}}}(\mathbf{CQ}, \mathbf{CQ})$ *are* NP*-complete.*

Proof sketch. (1) We first argue that $\mathbf{PC}^{bag}{}_{<_{\mathcal{N}}}(\mathbf{CQ}^{\neq}, \mathcal{P})$ is in coNP for $\mathcal{P} \in \{\mathcal{P}_{fin}\} \cup \mathfrak{P}_{det}$. The required algorithm follows from Lemma 5.3(1). It suffices to guess a valuation V and a node κ and verify that $V(\mathit{body}_{\mathcal{Q}}) \not\subseteq \mathit{rfacts}_{\boldsymbol{P}}(\kappa)$ to check whether $\mathcal{Q}$ is not bag-parallel-correct under a given distribution policy $\boldsymbol{P}$.

To show that $\mathbf{PC}^{bag}{}_{<_{\mathcal{N}}}(\mathbf{CQ}, \mathcal{P}_{fin})$ is coNP-hard, we use a reduction from the problem that asks whether a given graph is *not* 3-colorable. Let G be an arbitrary undirected graph with n edges. We construct a query $\mathcal{Q}$ and policy $\boldsymbol{P}$ over a network with n nodes and a universe $U = \{r, g, b\}$ as follows: For every edge $e = (u, v)$ in G, we add the atom $E_e(x_u, x_v)$ to $\mathit{body}_{\mathcal{Q}}$ and define $\mathit{rfacts}_{\boldsymbol{P}}(e) = \{\boldsymbol{f} \mid \boldsymbol{f} \in \mathit{Facts}(\{E_i\}, U), e \neq i\} \cup \{E_e(r, r), E_e(g, g), E_e(b, b)\}$. Intuitively, each valuation for $\mathcal{Q}$ corresponds to a coloring of G, and only valuations related to an invalid coloring are supported by at least one node.

(2) The algorithm to show that $\mathbf{PC\text{-}Trans}^{bag}{}_{<_{\mathcal{N}}}(\mathbf{CQ}^{\neq}, \mathbf{CQ}^{\neq})$ is in Π_2^p follows from Lemma 5.3(2). The lower bound is by a non-trivial reduction from the quantified boolean satisfiability problem for the respective level of the hierarchy that is inspired by a technique used in [17].

(3) It can be shown that for a CQ $\mathcal{Q}$, bag-parallel-correctness transfers from $\mathcal{Q}$ to $\mathcal{Q}'$ over ordered networks if and only if a mapping θ for $\mathcal{Q}$ over $adom(body_{\mathcal{Q}'})$ exists such that $body_{\mathcal{Q}'} \subseteq \theta(body_{\mathcal{Q}})$. The required algorithm to show that $\mathbf{PC\text{-}Trans}^{bag}{}_{<_{\mathcal{N}}}(\mathbf{CQ}, \mathbf{CQ}^{\neq})$ is in NP now follows.

To prove NP-hardness, we provide a reduction from graph 3-colorability. Let G be an arbitrary graph with n edges. We first introduce the following sets of atoms:

$$\texttt{invalidE} = \{E_i(x_i, x_i), E_i(y_i, y_i), E_i(z_i, z_i) \mid i \in [n]\}.$$
$$\texttt{surplusE} = \{E_i(\_, \_), E_i(\_, \_), E_i(\_, \_), E_i(\_, \_), E_i(\_, \_) \mid i \in [n]\ \}.$$

We now define $\mathcal{Q}$ and $\mathcal{Q}'$ as follows:

$$body_{\mathcal{Q}} = \{E_i(x_u, x_v) \mid E(u, v) \in G \text{ having label } i\} \cup \texttt{invalidE} \cup \texttt{surplusE},$$
$$body_{\mathcal{Q}'} = \{E_i(x, y) \mid i \in [n] \text{ and } x, y \in \{x_r, x_g, x_b\}\}.$$

Intuitively, $body_{\mathcal{Q}'} \subseteq \theta(body_{\mathcal{Q}})$ implies that for every edge all colorings can be partitioned into three sets: one valid coloring that participates in the 3-coloring of the graph; the invalid colorings; and, the rest or the surplus of the colorings. ◀

6 Discussion

In this paper, we revisited the framework of [5] under bag semantics. The latter represents a more accurate semantics for real world queries and is a necessary step towards aggregate queries. We obtained semantical characterizations for parallel-correctness as well as transferability under bag semantics. For bag-parallel-correctness we provide tight complexity bounds whereas for transferability we provide an upper bound in EXPTIME. In addition, we show correspondences and incomparabilities with the analog problems under set semantics. We also introduced an ordered network setting that could be more natural for capturing bag semantics and in this setting obtained tight complexity bounds for both decision problems. We mention that all our results can be naturally extended to unions of conjunctive queries. The latter does not need any additional ideas but clutters notation.

There are quite a number of directions for follow-up work. We did not prove a lower bound for transfer of bag-parallel-correctness. Actually, we suspect the upper bound can be improved by coming up with a more efficient algorithm to compute the set of implied facts.

A motivation for the ordered model presented in Section 5 is that bag-parallel-correctness under the previous model can prohibit parallelization. Indeed, Example 5.1 shows a query that can not be parallelized while retaining bag-parallel-correctness. A natural question is whether this class of queries for which no efficient policy exists can be characterized.

Whereas the focus in this paper is on set and bag semantics, it could be interesting to consider parallel-correctness and parallel-correctness transfer under bag-set [7] or combined semantics [8]. Similarly, another direction of future work would be to consider parallel-correctness in the context of aggregate operators.

References

1. Foto N. Afrati, Manas R. Joglekar, Christopher Ré, Semih Salihoglu, and Jeffrey D. Ullman. GYM: A multiround distributed join algorithm. In *ICDT*, pages 4:1–4:18, 2017.
2. Foto N. Afrati, Anish Das Sarma, Semih Salihoglu, and Jeffrey D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *PVLDB*, 6(4):277–288, 2013.
3. Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.
4. Tom J. Ameloot, Gaetano Geck, Bas Ketsman, Frank Neven, and Thomas Schwentick. Data partitioning for single-round multi-join evaluation in massively parallel systems. *SIGMOD Record*, 45(1):33–40, 2016.
5. Tom J. Ameloot, Gaetano Geck, Bas Ketsman, Frank Neven, and Thomas Schwentick. Parallel-correctness and transferability for conjunctive queries. *J. ACM*, 64(5):36:1–36:38, 2017.
6. Paul Beame, Paraschos Koutris, and Dan Suciu. Skew in parallel query processing. In *PODS*, pages 212–223. ACM, 2014.
7. Surajit Chaudhuri and Moshe Y. Vardi. Optimization of *Real* conjunctive queries. In *PODS*, pages 59–70. ACM Press, 1993.
8. Sara Cohen. Equivalence of queries that are sensitive to multiplicities. *VLDB J.*, 18(3):765–785, 2009.
9. Sumit Ganguly, Abraham Silberschatz, and Shalom Tsur. A framework for the parallel processing of datalog queries. In *SIGMOD*, pages 143–152. ACM Press, 1990.
10. Gaetano Geck, Bas Ketsman, Frank Neven, and Thomas Schwentick. Parallel-correctness and containment for conjunctive queries with union and negation. In *ICDT*, pages 9:1–9:17, 2016.
11. Hadoop. URL: `https://hadoop.apache.org/`.
12. Bas Ketsman, Aws Albarghouthi, and Paraschos Koutris. Distribution policies for datalog. In *ICDT*, pages 17:1–17:22, 2018.
13. Bas Ketsman and Dan Suciu. A worst-case optimal multi-round algorithm for parallel computation of conjunctive queries. In *PODS*, pages 417–428, 2017.
14. Paraschos Koutris and Dan Suciu. Parallel evaluation of conjunctive queries. In *PODS*, pages 223–234, 2011.
15. Rimma Nehme and Nicolas Bruno. Automated partitioning design in parallel database systems. In *SIGMOD*, pages 1137–1148. ACM, 2011.
16. Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. Automating physical database design in a parallel database. In *SIGMOD*, pages 558–569, 2002.
17. Ron van der Meyden. The complexity of querying indefinite data about linearly ordered domains. In *PODS*, pages 331–345, 1992.
18. Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD*, pages 13–24, 2013.

Evaluation and Enumeration Problems for Regular Path Queries

Wim Martens
University of Bayreuth, Bayreuth, Germany
wim.martens@uni-bayreuth.de

Tina Trautner
University of Bayreuth, Bayreuth, Germany
tina.trautner@uni-bayreuth.de

Abstract

Regular path queries (RPQs) are a central component of graph databases. We investigate decision- and enumeration problems concerning the evaluation of RPQs under several semantics that have recently been considered: *arbitrary paths*, *shortest paths*, and *simple paths*.

Whereas arbitrary and shortest paths can be enumerated in polynomial delay, the situation is much more intricate for simple paths. For instance, already the question if a given graph contains a simple path of a certain length has cases with highly non-trivial solutions and cases that are long-standing open problems. We study RPQ evaluation for simple paths from a parameterized complexity perspective and define a class of *simple transitive expressions* that is prominent in practice and for which we can prove a dichotomy for the evaluation problem. We observe that, even though simple path semantics is intractable for RPQs in general, it is feasible for the vast majority of RPQs that are used in practice. At the heart of our study on simple paths is a result of independent interest: the two disjoint paths problem in directed graphs is W[1]-hard if parameterized by the length of one of the two paths.

2012 ACM Subject Classification Information systems → Query languages for non-relational engines, Theory of computation → Database query languages (principles), Theory of computation → Regular languages

Keywords and phrases Graph databases, regular path queries, regular languages, parameterized complexity

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.19

Acknowledgements We are grateful to Phokion Kolaitis for suggesting us to study enumeration problems on simple paths matching RPQs. We are also grateful to Holger Dell for pointing us to Theorem 9 and providing us with a proof sketch. We acknowledge the many useful comments of the anonymous reviewers for ICDT 2018 that helped us to significantly improve the structure and presentation of the paper.

1 Introduction

Regular path queries (RPQs) are a crucial feature of graph database query languages, since they allow us to pose queries about arbitrarily long paths in graphs. Essentially, RPQs are regular expressions that are matched against labeled directed paths in graph databases. Currently, the openCypher project [33] and the World Wide Web Consortium (W3C) [39] are considering how RPQ evaluation can be formally defined for the development of Neo4j's Cypher [31, 34] and SPARQL 1.1 [38], respectively. Several popular candidates that are being considered for the semantics of RPQs are *arbitrary paths*, *shortest paths*, and *simple paths* ([3, Section 4.4], [34]).

We briefly explain these semantics. Given a graph, an RPQ r considers directed paths for which the labels on the edges form a word in the language of r. We call such paths *candidate matches*. The different semantics restrict the kind of paths that *match* the RPQ, i.e., can be returned as answers. *Arbitrary paths* imposes no restriction and returns every candidate match. *Shortest paths*, on the other hand, only returns the shortest candidate matches and *simple paths* only returns candidate matches that do not have duplicate nodes.

Under *arbitrary paths*, the number of matches may be infinite if the graph is cyclic. This may pose a challenge for designing the query language, even if one does not choose to return all matching paths. Indeed, a well-known semantics of RPQs is to return *node pairs* (x, y) such that there exists a matching path from x to y. Under bag semantics for node pairs,[1] where each (x, y) is returned as often as the number of matches from x to y, one needs to deal with the case where this number is infinite.

Under *shortest paths* and *simple paths*, the number of matching paths is always finite, which simplifies the aforementioned design challenge. However, these two versions face other challenges. *Simple paths* may present complexity issues. Two fundamental problems are that

- counting the number of simple paths between two nodes is #P-complete [37] and
- deciding if there exists a simple path of even length between two given nodes is NP-complete [23].

Indeed, the first problem implies that evaluating the RPQ a^* under bag semantics is #P-complete and the second one implies that deciding if the RPQ $(aa)^*$ returns at least one answer is NP-complete.[2] *Shortest paths* does not have these complexity issues, but it is unclear if its semantics is very natural. For instance, under shortest paths semantics, if we ask how many paths exist from x to y, then this number may decrease if a new, shorter, path is added.[3] This may seem counter-intuitive to users.

Since it seems that there is no one-size-fits-all solution, the openCypher project team recently proposed to support several kinds of semantics for Cypher [34]. This situation motivated us to shed more light on evaluation of RPQs and enumerating the answers, focusing on the following aspects:

- Our goal is to better understand *enumerating the paths* that match RPQs. That is, we study problems where the task is to enumerate all matching paths without duplicates. We are interested in which situations it is possible to answer queries in *polynomial delay*, i.e., such that the time between consecutive answers is polynomial. To reach this goal, we must also improve our understanding for some decision problems related to RPQ evaluation.
- We take into account a recent study that investigated the structure of about 250K RPQs gathered from a wide range of SPARQL query logs [8]. It turns out that all these RPQs have a relatively simple structure, which is remarkable because their syntax is not restricted by the SPARQL recommendation.

Our contributions are the following.

1. We first observe that enumeration of arbitrary or shortest paths that match a given RPQ can be done in polynomial delay (Section 3).
2. We then turn to simple paths and study RPQ evaluation as a decision problem. This problem is challenging because it contains subproblems that are quite non-trivial. One

[1] SPARQL 1.1 uses such a bag semantics approach.

[2] It is also known that answering the RPQ a^*ba^* under simple path semantics is at least as difficult as the Two Disjoint Paths problem [29].

[3] Notice that each semantics only returns or counts the number of paths that match.

such subproblem is testing if there exists a directed simple path of length $\log n$ between two given nodes in a graph with n nodes, which was shown to be in PTIME by Alon et al., using their color coding technique [2]. The question if it can be decided in PTIME if there is a simple path of length $\log^2 n$ [2] is an open problem since two decades. Notice that these two problems are special cases of RPQ evaluation under simple path semantics (i.e., evaluate the RPQs $a^{\log n}$ and $a^{\log^2 n}$ in a graph where every edge has label a).

We therefore investigate RPQ evaluation from the angle of parameterized complexity (Section 4). We introduce the class of *simple transitive expressions (STEs)* that capture over 99% of the RPQs that were found in SPARQL query logs in a recent study [8]. We identify a property of STEs that we call *cuttability* and prove a dichotomy, showing that the parameterized complexity for evaluating STEs $\mathcal{R}$ is in FPT if $\mathcal{R}$ is cuttable and W[1]-hard otherwise. Examples of cuttable classes of expressions are $\{a^k a^* \mid k \in \mathbb{N}\}$ and $\{(a+b)^k a^* \mid k \in \mathbb{N}\}$. Examples of non-cuttable classes are $\{a^k b^* \mid k \in \mathbb{N}\}$, $\{a^k b a^* \mid k \in \mathbb{N}\}$, and $\{a^k (a+b)^* \mid k \in \mathbb{N}\}$.

3. At the core of the dichotomy are two results of independent interest (Section 5). The first is by the authors of [16], who showed that it can be decided in FPT if there is a simple path of length *at least* k between two nodes in a graph (Theorem 9). The second shows that the Two Disjoint Paths problem is W[1]-hard when parameterized by the length of one of the two paths (Theorem 11).
4. We then turn to enumeration of simple paths and prove that the dichotomy on STEs carries over to the enumeration setting. We also study the data complexity and show that Bagan et al.'s dichotomy for deciding the existence of a simple path that matches an RPQ [5] carries over to enumeration problems (Section 6).

Putting everything together, we see that, although simple path semantics leads to high complexity in general, its complexity for RPQs that have been found in SPARQL query logs is reasonable. We discuss this in the conclusions.

Related Work

RPQs on graph databases have been studied since the end of the 80's [10, 11, 40]. Given a graph database G, an RPQ r, and two nodes s and t, there are several natural fundamental problems associated to RPQ evaluation.

- The *decision problem*: Does r match a path from s to t in G?
- The *counting problem*: How many paths from s to t does r match?
- The *computation problem*: Compute the set of paths from s to t for which r matches.

The decision problem is well known to be tractable for arbitrary and shortest paths by standard automata techniques. Mendelzon and Wood [29] studied the problem for simple paths. They observed that the problem is NP-complete for a^*ba^* and $(aa)^*$. These two results heavily rely on the work of Fortune et al. [17], who showed NP-completeness of the two disjoint paths problem, and Lapaugh and Papadimitriou [23], who showed that the even length simple path problem is NP-complete.

Bagan et al. [5] provided a dichotomy for the *data complexity* of the decision problem. They defined a class $\mathsf{C}_{\text{tract}}$ such that the problem is in PTIME for each language in $\mathsf{C}_{\text{tract}}$ and NP-complete otherwise.

The *counting problem* for arbitrary paths is #P-complete in general [21].[4] However, if the RPQ is represented by a deterministic automaton (or even an unambiguous one), the counting problem is in PTIME [26], since it reduces to counting the number of paths in a graph. The complexity results for arbitrary paths can easily be extended to shortest paths. Indeed, all words have equal length in Kannan et al.'s #P-hardness proof [21] and the PTIME algorithm also works if we need to count the words of a given length n.

Concerning simple paths, we know from the classical result of Valiant [37] that counting the number of simple paths between two given nodes in a graph is #P-complete. This immediately implies that counting is already #P-hard for the RPQ a^*.

Concerning the *computation problem*, Ackermann and Shallit [1] proved that one can enumerate the words accepted by a given NFA in polynomial delay. This is easily extended to RPQ evaluation w.r.t. arbitrary paths and shortest paths, as we observe in Section 3. Concerning simple paths, Yen's algorithm [41] is a method to enumerate all simple paths between two given nodes in polynomial delay. We build on this result in Section 6.

Yen's algorithm was generalized by Lawler [24] and Murty [30] to a tool for designing general algorithms for enumeration problems. Lawler-Murty's procedure has been used for solving enumeration problems in databases in various contexts [18, 20, 22].

Further related work concerning RPQs on graph databases are studies about the complexity of SPARQL 1.1 property paths [4, 26], which are relevant because property paths extend RPQs. Their semantics is a mixture between arbitrary path and simple path semantics. The relative expressive power of graph query languages using transitive closures, data value comparisons, and branching was investigated in [25, 36]. Finally, we refer to [3, 6] for general overviews of the wide literature on graph databases.

2 Preliminaries

By Σ we always denote an *alphabet*, that is, a finite set. A (Σ-)*symbol* is an element of Σ. A *word* (over Σ) is a finite sequence $w = a_1 \cdots a_n$ of Σ-symbols. The *length* of w, denoted by $|w|$, is its number of symbols n. We denote the empty word by ε.

We assume familiarity with regular expressions and finite automata. The regular expressions we use in this paper are defined as follows: $\emptyset$, ε and every Σ-symbol is a regular expression; and when r and s are regular expressions, then (rs), $(r + s)$, $(r?)$, (r^*), and (r^+) are also regular expressions. From now on, we use the usual precedence rules to omit parentheses. The *size* $|r|$ of a regular expression is the number of occurrences of Σ-symbols in r. For example, $|aba^*| = 3$. We define the *language* $L(r)$ of r as usual. Since it is easy to test if $L(r) = \emptyset$ for a given expression r, we assume in this paper that $L(r) \neq \emptyset$ for all expressions, unless mentioned otherwise. For $n \in \mathbb{N}$, we use r^n to abbreviate the n-fold concatenation $r \cdots r$ of r. We abbreviate $(r?)^n$ by $r^{\leq n}$. In the context of graph databases, *regular path queries (RPQs)* are regular expressions that can be evaluated on graphs and return an output. In this paper, we will blur the distinction between them (language acceptors vs. queries) and use "regular expression" and RPQ as synonyms.

A *non-deterministic finite automaton (NFA)* N over Σ is a tuple $(Q, \Sigma, \Delta, Q_I, Q_F)$, where Q is a finite set of states, Σ is a finite alphabet, $\Delta : Q \times \Sigma \times Q$ is the transition relation, $Q_I \subseteq Q$ is the set of initial states, and Q_F is the set of accepting states. By $\delta^*(w)$ we denote the set of states reachable by N after reading w, that is, $\delta^*(\varepsilon) = Q_I$ and, for every word w

[4] Kannan et al. proved that counting the number of words accepted by a non-deterministic automaton for a finite language is #P-complete. This result trivially extends to RPQ evaluation.

and symbol a, we define $\delta^*(wa) = \{\delta(q, a) \mid q \in \delta^*(w)\}$. The *size* of an NFA is $|Q|$, i.e., its number of states. We define the *language* $L(N)$ of N as usual.

2.1 Graph Databases

We use edge-labeled directed graphs as abstractions for graph databases. A graph G (with labels in Σ) will be denoted as $G = (V, E)$, where V is the finite set of *nodes* of G and $E \subseteq V \times \Sigma \times V$ is the set of *edges*. We say that edge $e = (u, a, v)$ goes *from node u to node v* and *has label a*. We use *a-edge* to refer to an edge with label a. Sometimes we write an edge as $(u, v) \in V \times V$ if the label does not matter. In this paper, we assume that graphs are directed, unless mentioned otherwise. Notice that our definition allows graphs to have self-loops and multi-edges. The *size* of a graph G, denoted by $|G|$ is $|V| + |E|$.

We assume familiarity with basic terminology on graphs. A *path* from node u to node v in G is a sequence $p = (v_0, a_1, v_1)(v_1, a_2, v_2) \cdots (v_{n-1}, a_n, v_n)$ of edges in G such that $u = v_0$ and $v = v_n$. For $0 \leq i \leq n$, we denote by $p[i, i]$ (or $p[i]$) the node v_i and, for $0 \leq i < j \leq n$, we denote by $p[i, j]$ the subpath $(v_i, a_{i+1}, v_{i+1}) \dots (v_{j-1}, a_j, v_j)$. A path p is *simple* if all nodes $v_0, \dots, v_n$ are pairwise different.[5] The *length* of p, denoted $|p|$, is the number n of edges in p. By definition of paths, we consider two paths to be different if they are different sequences of edges. In particular, two paths going through the same nodes in the same order, but using different edge labels are different.

The set of *nodes of path p* is $V(p) = \{v_0, \dots, v_n\}$. The *word of p* is $a_1 \cdots a_n$ and is denoted by $\text{lab}(p)$. Path p *matches* a regular expression r (resp., NFA N) if $\text{lab}(p) \in L(r)$ (resp., $\text{lab}(p) \in L(N)$). The *concatenation* of paths $p_1 = (v_0, a_1, v_1) \cdots (v_{n-1}, a_n, v_n)$ and $p_2 = (v_n, a_{n+1}, v_{n+1}) \cdots (v_{n+m-1}, a_{n+m}, v_{n+m})$ is simply the concatenation $p_1 p_2$ of the two sequences.

We will often consider a graph $G = (V, E)$ together with a *source node s* and a *target node t*, for example, when considering paths from s to t. We denote such a graph with source s and target t as (G, s, t) and define its size $|(G, s, t)|$ as $|G|$.

The *product* of graph (G, s, t) and NFA $N = (Q, \Sigma, \Delta, Q_I, Q_F)$ is a graph (V', E') with $V' = (V \times Q)$ and $E' = \{((u_1, q_1), a, (u_2, q_2)) \mid (u_1, a, u_2) \in E \text{ and } (q_1, a, q_2) \in \Delta\}$. We denote this product by $(G, s, t) \times N$. Notice that simple paths in $(G, s, t) \times N$ may use nodes $(u, q_1) \neq (u, q_2)$ and may therefore correspond to non-simple paths in G.

2.2 Decision and Enumeration Problems

We consider the following problems, where G is always a graph, s and t are nodes in G, and r is an RPQ.

- Path: Given (G, s, t) and r, is there a path from s to t that matches r?
- SimPath: Given (G, s, t) and r, is there a simple path from s to t that matches r?

An *enumeration problem* P is a (partial) function that maps each input i to a finite or countably infinite set of *outputs for i*, denoted by $\mathsf{P}(i)$. Terminologically, we say that, given i, the task is to *enumerate* $\mathsf{P}(i)$. We consider the following enumeration problems:

- EnumPaths: Given (G, s, t) and r, enumerate the paths in G from s to t that match r.
- EnumShortPaths: Given (G, s, t) and r, enumerate the shortest paths in G from s to t that match r.

[5] We focus on *node-distinct paths* in this paper, but one can also consider *edge-distinct paths*. We come back to this in the conclusions.

- EnumSimPaths: Given (G, s, t) and r, enumerate the simple paths in G from s to t that match r.

An *enumeration algorithm* for P is an algorithm that, given input i, writes a sequence of answers to the output such that every answer in $\mathsf{P}(i)$ is written precisely once. If A is an enumeration algorithm for an enumeration problem P, we say that A runs in *polynomial delay* if the time before writing the first answer and the time between writing every two consecutive answers is polynomial in $|i|$.

For a class $\mathcal{R}$ of regular expressions, we denote by $\mathsf{Path}(\mathcal{R})$ the problem Path where we always assume that $r \in \mathcal{R}$. We use the same convention for all other decision- and enumeration problems. We assume familiarity with the notions *combined* and *data* complexity. In our decision problems, (G, s, t) is the data and r is the query.

3 Enumerating All Regular Paths and Shortest Regular Paths

It is well known that $\mathsf{Path}(\mathcal{R})$ is in PTIME for the complete class $\mathcal{R}$ of RPQs. Indeed, one only needs to construct the product of the graph and an NFA N for the RPQ and test if (t, q_f) is reachable from (s, q_0), where q_0 and q_f are an initial and an accepting state of N, respectively. We note that this favorable complexity carries over to EnumPaths and EnumShortPaths. At the core lies the following result by Ackerman and Shallit.

▶ **Theorem 1** (Theorem 3 in [1]). *Given an NFA N, enumerating the words in $L(N)$ can be done in polynomial delay.*

This result generalizes a result of Mäkinen [27], who proved that the words in $L(N)$ can be enumerated in polynomial delay if N is deterministic. Ackermann and Shallit genereralized his algorithm for nondeterministic N and proved that, for a given length n (which they call *cross-section*), the lexicographically smallest word in $L(N)$ can be found in time $O(|Q|^2 n^2)$ ([1], Theorem 1). They then prove that the set of all words of length n can be computed in time $O(|Q|^2 n^2 + |\Sigma||Q|^2 x)$, where x is the sum of the lengths of the words that were written to the output ([1], Theorem 2). A closer inspection of their algorithm actually shows that it has delay $O(|\Sigma||Q|^2|w|)$ where $|w|$ is the size of the next output. In fact, Ackermann and Shallit prove that the words in $L(N)$ can be enumerated in *radix order*.[6]

It is easy to extend the algorithm of Ackerman and Shallit to solve EnumPaths in polynomial delay as follows. We construct an NFA N_r for r and take the product with (G, s, t). The product automaton therefore has states (u, q) where u is a node from G and q a state from N_r. In the resulting automaton, we replace every transition $[(u_1, q_1), a, (u_2, q_2)]$ with $[(u_1, q_1), (u_1, a, u_2), (u_2, q_2)]$. Enumerating the words from the resulting automaton corresponds to enumerating the paths from s to t that match r. Using Theorem 1, we have the following corollary.

▶ **Corollary 2.** *EnumPaths and EnumShortPaths can be solved in polynomial delay.*

For completeness, we note that counting the number of paths from s to t that match a given regular expression r is #P-complete in general, even if G is acyclic, see [26, Theorem 4.8(1)] and [4, Theorem 6.1].[7] The same holds for counting the number of shortest paths, since all paths in the proof of [26, Theorem 4.8(1)] have equal length.

---

[6] That is, $w_1 < w_2$ in radix order if $|w_1| < |w_2|$ or $|w_1| = |w_2|$ and w_1 is lexicographically before w_2.

[7] Arenas et al. [4] actually prove that the problem is SPANL-complete. Although it is not known if SPANL = #P, they are equal under Cook reductions.

4 Deciding Existence of Simple Paths

We now turn to *simple paths*, which will require much more effort. First, we focus on the decision problem SimPath, where our main result will be a dichotomy for *simple transitive expressions (STEs)*, a very restricted class of RPQs.

We will investigate SimPath from a parametrized complexity perspective. The main reason is that the size of the regular expression has a drastic effect on the complexity of the problem. Indeed, if G is a graph with n nodes and only a-edges, then asking if there is a simple path that matches the expression a^{n-1} is the NP-complete Hamilton Path problem. On the other hand, Alon et al. [2] proved that SimPath for graphs with n nodes is in PTIME for the language $a^{\log n}$. It is open[8] since 1995 whether SimPath is in PTIME for $a^{\log^2 n}$ [2].

So, even very elementary RPQs of the form a^k can behave very differently depending on the relationship between k and the size of the graph. This motivates us to study the problem from the angle of parameterized complexity.

4.1 Parameterized Complexity

We first give a quick overview of some notions in parameterized complexity. We follow the exposition of Cygan et al. [12] and refer to their work for further details. A *parameterized problem* is a language $L \subseteq \Sigma^* \times \mathbb{N}$ where, as before, Σ is a fixed, finite alphabet. For an instance $(x, p) \in \Sigma^* \times \mathbb{N}$, we call p the *parameter*. The *size* $|(x, p)|$ of an instance (x, p) is defined as $|x| + p$. A parameterized problem L is called *fixed-parameter tractable* if there exists an algorithm $\mathcal{A}$, a computable function $f : \mathbb{N} \to \mathbb{N}$, and a constant c such that, given $(x, p) \in \Sigma^* \times \mathbb{N}$, the algorithm $\mathcal{A}$ correctly decides whether $(x, p) \in L$ in time bounded by $f(p) \cdot |(x, p)|^c$. The complexity class containing exactly the fixed-parameter tractable problems is called FPT.

In the remainder of this section, we will study the parameterized complexity of SimPath. The instances (x, p) of this problem will always be such that x encodes the graph G and regular expression r, and the parameter p is $|r|$. For this reason, we overload notation and also denote the parameterized problem as SimPath.

4.2 Some Illustrations of the Dichotomy

Before we present our main dichotomy, we illustrate a few of its implications to give the reader some intuition about the result. In the following, we abbreviate the class of regular expressions $\{a^k \mid k \in \mathbb{N}\}$ simply by "a^k", and similar for $a^{\leq k}$, $a^k a^*$, $a^k b^*$, $a^k b a^*$, $b a^k a^*$, etc.

In the following Theorem, we consider problems SimPath($\mathcal{R}$), where $\mathcal{R}$ is one of the abovementioned classes.

▶ **Theorem 3.**
(a) *SimPath*(a^k), *SimPath*($a^{\leq k}$), *SimPath*($a^k a^*$), *and* *SimPath*($b a^k a^*$) *are in FPT.*
(b) *SimPath*($b^k a^*$) *and* *SimPath*($a^k b a^*$) *are W[1]-hard.*

[8] Recently, Björklund et al. [7] showed that, under the Exponential Time Hypothesis, there is no PTIME algorithm that can decide if there exists a simple path of length $\Omega(f(n) \log^2 n)$ between two nodes in a graph of size n for any nondecreasing polynomial time computable function f that tends to infinity.

Table 1 Structure of the 247,404 SPARQL property paths that were also used in the query logs investigated by Bonifati et al. [8]. The structure is sometimes in terms of a variable $\ell \in \mathbb{N}$, for which the second column indicated the values that were found in the logs. *Relative* indicates which percentage of the 247,404 property paths have this structure.

Expression Type	ℓ	*Relative*	*STE?*	*Expression Type*	ℓ	*Relative*	*STE?*
$(a_1 + \cdots + a_\ell)^*$	2–4	29.10%	yes	$a^*b?$		$< 0.01\%$	yes
$.$		25.48%	yes$^{(*)}$	abc^*		$< 0.01\%$	yes
a^*		19.66%	yes	$A_1 \cdots A_\ell$	2,6	$< 0.01\%$	yes
$a_1 \cdots a_\ell$	2–6	8.66%	yes	$.^*$		$< 0.01\%$	yes$^{(*)}$
a^*b		7.73%	yes	$(a_1 + a_2)?$		$< 0.01\%$	yes
$(a_1 + \cdots + a_\ell)$	1–6	6.61%	yes	$.?$		$< 0.01\%$	yes$^{(*)}$
$(a_1 + \cdots + a_\ell)^+$	1,2	1.54%	yes	$a^* + b$		$< 0.01\%$	no
$a_1?a_2? \cdots a_\ell?$	1–3,5	1.15%	yes	$a + b^+$		$< 0.01\%$	no
$a(b_1 + b_2)?$		0.01%	yes	$a^+ + b^+$		$< 0.01\%$	no
$a_1a_2? \cdots a_\ell?$	2,3	0.01%	yes	$(ab)^*$		$< 0.01\%$	no
$(ab^*) + c$		$< 0.01\%$	no				

We see that, even though all classes of regular expressions in Theorem 3 are similar, the complexities are drastically different (assuming FPT $\neq$ W[1]). The intuition is twofold:

1. "Short" paths can be dealt with using Color Coding [2] (or Bagan et al.'s extension thereof that incorporates finite regular languages [5, Theorem 6]). This explains why $\mathsf{SimPath}(a^k)$ and $\mathsf{SimPath}(a^{\leq k})$ are in FPT.
2. If paths can become arbitrarily long, the complexity depends on the interplay between the symbol in the *transitive closure* (which is always a here) and the rest. The intuition is that symbols that are "incompatible" with a (which is always b here) should only occur on positions that are a constant distance away from the beginning or end of words in the language. This explains why SimPath is in FPT for the classes a^ka^* (no incompatible symbols) and ba^ka^* (b is always on position one). Likewise, for the classes b^ka^* and a^kba^*, the symbol b can occur at positions arbitrarily far away from the beginning and end of words in the languages.

4.3 Dichotomy for Simple Transitive Expressions

We now aim at generalizing the results in Theorem 3 to more general RPQs which we call *simple transitive expressions (STEs)*. Although STEs are very restricted, we feel that they are relevant and important from a practical perspective since they constitute more than 99.99% of the *SPARQL property paths* found in query logs in an extensive recent study [8]. Notice that SPARQL property paths strictly extend RPQs. Their syntax is not restricted to a subset of regular expressions as, e.g., in Cypher patterns for "variable length pattern matching" [32, Section 3.2.7.7].

In the following definition, we use sets $A = \{a_1, \ldots, a_n\} \subseteq \Sigma$ to abbreviate expressions $(a_1 + \cdots + a_n)$. We allow $A = \emptyset$, in which case $L(A) = \emptyset$.

▶ **Definition 4.** An *atomic expression* is of the form $A \subseteq \Sigma$. A *bounded expression* is a regular expression of the form $A_1 \cdots A_k$ or $A_1? \cdots A_k?$, where $k \geq 0$ and each A_i is an atomic expression. Finally, a *simple transitive expression (STE)* is a regular expression

$$B_{\text{pre}} T^* B_{\text{suff}},$$

where B_{pre} and B_{suff} are bounded expressions and T is an atomic expression.

The central idea for STEs is that they can first perform some local navigation in B_{pre}, then an optional transitive part, followed by a second step of local navigation in B_{suff}. The local navigation steps allow to test paths of length exactly k or at most k, for some $k \in \mathbb{N}$. The transitive part is optional, since one can take $T = \emptyset$, so that T^* only matches ε.

We believe that STEs capture many RPQs that users ask in practice. Bonifati et al. [8] investigated the structure of 247,404 SPARQL property paths from query logs. Table 1 presents a classification of their raw data that facilitates comparison to RPQs. SPARQL property paths can express wildcard tests, which we denote by "." (similar to regexes). Furthermore, SPARQL uses reverse edges ("$\hat{}a$" means "follow an a-edge in reverse direction"), which we treat the same as a normal label test. Under *Expression Type*, the table summarizes which types of expressions are in Bonifati et al.'s data set, sometimes parameterized by a number ℓ for which the next column describes the values that were found. *Relative* describes which percentage of the 247,404 expressions fall into this expression type, and *STE?* indicates whether the expression is an STE. Here, we write "yes$^{(*)}$" to indicate that the expression is an STE if a wildcard is treated the same as a set of labels A. (Our algorithms indeed can be generalized to incorporate wildcards.)

In total, we saw that only 20 property paths are not STEs or trivially equivalent to an STE (by taking $T = \emptyset$ in the definition of STEs, for example).[9] For instance, the expression type $a_1 a_2? \cdots a_\ell?$ is equivalent to an STE where $B_{\text{pre}} = a_1$, $T = \emptyset$, and $B_{\text{suff}} = a_2? \cdots a_\ell?$. In summary, 99.992% of the property paths in Table 1 correspond to STEs.

We now define the notions that we need for the dichotomy.

▶ **Definition 5.** Let $r = B_{\text{pre}} T^* B_{\text{suff}}$ be an STE with $L(r) \neq \emptyset$. If $B_{\text{pre}} = A_1 \cdots A_{k_1}$, then the *left cut border* c_1 *of* r is the largest value such that $T \not\subseteq A_{c_1}$ if it exists and zero otherwise. If $B_{\text{pre}} = A_1? \cdots A_{k_1}?$, then the left cut border is zero. Symmetrically, if $B_{\text{suff}} = A'_{k_2} \cdots A'_1$, then the *right cut border* c_2 *of* r is the largest value such that $T \not\subseteq A'_{c_2}$ if it exists and zero otherwise. (Notice that the indices in B_{suff} are reversed.) If $B_{\text{suff}} = A'_{k_2}? \cdots A'_1?$, then the right cut border is zero.

We explain the intuition behind cut borders in Figure 1. For $c \in \mathbb{N}$, an expression is *c-bordered* if the maximum of its left and right cut borders is c. We call a class $\mathcal{R}$ of STEs *cuttable* if there exists a constant $c \in \mathbb{N}$ such that each expression in $\mathcal{R}$ is c'-bordered for some $c' \leq c$.

We can now prove a dichotomy on the complexity of $\mathsf{SimPath}(\mathcal{R})$ for classes of STEs $\mathcal{R}$, if $\mathcal{R}$ satisfies the following mild condition. We say that $\mathcal{R}$ *can be sampled* if there exists an algorithm that, given $k \in \mathbb{N}$, returns an expression in $\mathcal{R}$ that is k'-bordered with $k' \geq k$, and "no" if there is no such expression. We need the condition that $\mathcal{R}$ can be sampled to prove the W[1]-hardness. For this reason, this condition is no longer needed in Theorem 15.

▶ **Theorem 6.** *Let $\mathcal{R}$ be a class of STEs that can be sampled.*
(a) *If $\mathcal{R}$ is cuttable, then $\mathsf{SimPath}(\mathcal{R})$ is in FPT and*
(b) *otherwise, $\mathsf{SimPath}(\mathcal{R})$ is W[1]-hard.*

Proof idea. The main techniques will be presented in Section 5. We can attack case (a) using Theorem 7, Observation 8, and the techniques for proving Theorem 9. In short, if $\mathcal{R}$ is cuttable and we need to deal with arbitrarily long paths, then we can use exhaustive search to enumerate all possible pre- and suffixes of length at most c. We then use a variation of the representative sets technique [16] to obtain an FPT algorithm. In case (b), it is possible to adapt the reduction in the proof of Theorem 11. ◀

[9] In fact, *all* expressions except for $(ab)^*$ can be handled with the techniques we present here. For instance, the FPT algorithm can trivially be extended to unions of STEs by testing each STE separately. For the expression $(ab)^*$, even the data complexity of SimPath is NP-complete.

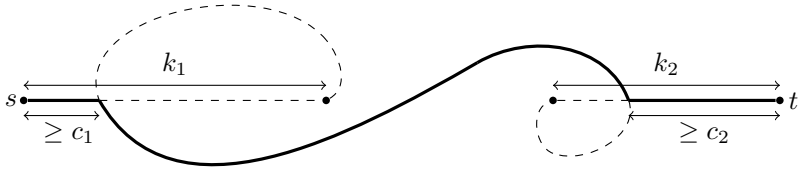


Figure 1 Assume $r = A_1 \cdots A_{k_1} T^* A'_{k_2} \cdots A'_1$ has left and right cut borders c_1 and c_2, respectively. Assume that an arbitrary path from s to t matches r such that its length k_1 prefix and length k_2 suffix are node-disjoint. If, after removing all loops, (1) the length c_1 prefix and length c_2 suffix are still the same and (2) the path still has length at least $k_1 + k_2$, then it matches r.

Notice that the difference between cuttable and non-cuttable classes of STEs can be quite subtle. For instance, $b^k a^*$ and $a^k(a+b)^*$ are non-cuttable, but $(a+b)^k a^*$ is cuttable. Looking back at Table 1, we see that abc^* is 2-bordered and all other STEs are either 0-bordered or 1-bordered. It therefore seems that cut borders in practice are small and over 99% of the expressions fall on the tractable side of Theorem 6.

5 Technical Core: Simple Paths With Length Constraints

In this section we investigate the parameterized complexity of problems that involve simple paths with length constraints. The problems we consider here are the core of the RPQ evaluation problems in Section 4.

5.1 One Path

We consider the following parameterized problems.

- PSimPath: Given an instance $((G, s, t), k)$ with parameter $k \in \mathbb{N}$, is there a simple path from s to t of length exactly k in G?
- PSimPath$^{\leq}$ and PSimPath$^{\geq}$: These two problems are defined analogously to PSimPath but ask if there is a simple path of length at most k and at least k, respectively.

These three problems are in FPT, but the techniques to obtain these results are quite different. For PSimPath, membership in FPT follows from the famous color coding technique [2].

▶ **Theorem 7** (Alon et al. [2]). *PSimPath is in FPT.*

PSimPath$^{\leq}$ is trivially in FPT because the shortest path problem is in PTIME.

▶ **Observation 8.** *PSimPath$^{\leq}$ is in PTIME (and therefore in FPT).*

Finally, PSimPath$^{\geq}$ can be shown to be in FPT by adapting methods from Fomin et al. [16]. They proved that finding simple cycles of length at least k is in FPT for cycles and discovered that their technique also works for paths [13]. The following theorem is therefore due to the authors of [16]. (Fomin et al. [16] already showed FPT membership for PSimPath$^{\geq}$ on *undirected* graphs, but the techniques needed on directed graphs are quite different.)

▶ **Theorem 9.** *(Similar to Theorem 5.3 in [16]) PSimPath$^{\geq}$ is in FPT.*

5.2 Two Disjoint Paths

We consider variants of the TwoDisjointPaths problem [17]. A *two-colored graph* is a directed graph in which every edge is labeled a or b. An *a-path* is a path consisting of only a-edges. We consider the following parameterized problems.

- PTwoDisjointPaths: Given a graph G, nodes s_1, t_1, s_2, t_2, and parameter $k \in \mathbb{N}$, are there simple paths p_1 from s_1 to t_1 and p_2 from s_2 to t_2 such that p_1 and p_2 are node-disjoint and p_1 has length k?
- PTwoColorDisjointPaths: Given a two-colored graph G, nodes s_a, t_a, s_b, t_b, and parameter $k \in \mathbb{N}$, is there a simple a-path p_a from s_a to t_a and a simple b-path p_b from s_b to t_b such that p_a and p_b are node-disjoint and p_a has length k?

It is well-known that TwoDisjointPaths, the non-parameterized version of PTwoDisjointPaths, is NP-complete [17]. In terms of parameterized complexity, Downey and Fellows [14] introduced the W-hierarchy, where FPT = W[0] and W[i] $\subseteq$ W[j] for all $i \leq j$. A famous complete problem for W[1] (under so-called *fpt-reductions*) is k-Clique with parameter k [15]. Therefore, k-Clique not being fixed-parameter tractable is equivalent to FPT $\neq$ W[1], which is a standard assumption in parameterized complexity.

Cai and Ye [9] proved that PTwoDisjointPaths is in FPT for *undirected graphs*, both for the cases where one wants node-disjoint or edge-disjoint paths. They left the cases for directed graphs as open problems [9, Problem 2]. We solve one of the cases by showing in Theorem 11 that PTwoDisjointPaths is W[1]-hard. We also prove that PTwoColorDisjointPaths is W[1]-hard – the proof for PTwoDisjointPaths relies on it.

▶ **Theorem 10.** *PTwoColorDisjointPaths is W[1]-hard.*

Proof idea. This result follows from a slight adaptation of a proof of Slivkins [35, Theorem 2.1]. Slivkins proved that k-Edge-Disjoint-Paths with parameter k is W[1]-hard in directed acyclic graphs. More precisely, given an instance of k-Clique, Slivkins constructs a DAG G and nodes s_i, t_i (with $1 \leq i \leq k$) and s_{ij}, t_{ij} (with $1 \leq i < j \leq k$) such that the input graph has a k-clique if and only if G has paths from each s_i to the corresponding t_i and from each s_{ij} to the corresponding t_{ij}, all edge-disjoint.

The main idea for our reduction is to take Slivkins' construction and

- connect each t_i to s_{i+1} with a b-edge;
- connect each t_{ik} to $s_{(i+1)(i+2)}$ with an a-edge;
- connect each t_{ij} with $i < j < k$ to $s_{i(j+1)}$ with an a-edge; and
- label all edges intended for "verifiers" with a and all edges intended for "selectors" with b. (Some edges in Slivkins' proof are intended for both verifiers and selectors. Here we can add two parallel edges, one labeled a and one labeled b.)

Then, it can be shown that the original instance is in k-Clique if and only if there exists an a-path from s_{12} to $t_{(k-1)k}$ and a b-path from s_1 to t_k. Moreover, the a-path, if it exists, has length $k' \in O(k^2)$. ◀

The two colors in the proof of Theorem 10 play a central role: since the a-path cannot use any b-edges and vice versa, we have much control over where the two paths can be. The following Theorem shows that the construction in Theorem 10 can be strengthened so that we do not need the two colors.

This is a non-trivial change. Very roughly, one can think of the graph G in the proof of Theorem 10 as a grid with k rows and n columns, where the a-edges are "vertical" and the b edges are "horizontal". (In reality, each coordinate in this grid is another gadget with $4k$ nodes.) The task for Theorem 11 is to change the proof so that paths with mixed a-edges and b-edges do not lead to a solution. For doing this, we use two ideas:

- We use the idea of "control nodes" by Grohe and Grüber [19, Lemma 16], who showed that Slivkins' construction can be used to show that k-Disjoint-Cycles is W[1]-hard.
- We replace each b-edge by a path p_b of length k', ensuring that each path that has p_b as subpath is too long.

▶ **Theorem 11.** *PTwoDisjointPaths is W[1]-hard.*

We provide a proof sketch in Appendix A.

For completeness, we mention the complexity of other variants of PTwoDisjointPaths, some of which can be shown by extending the technique from Theorem 11. We define TwoDisjointPaths$^{\leq}$ and TwoDisjointPaths$^{\geq}$ analogously to PTwoDisjointPaths by requiring that p_1 has length $\leq k$ and $\geq k$, respectively.

▶ **Theorem 12.**
- *TwoDisjointPaths$^{\leq}$ is W[1]-hard.*
- *TwoDisjointPaths$^{\geq}$ is NP-complete for every constant $k \in \mathbb{N}$ ([17]).*
- *PTwoColorDisjointPaths, PTwoDisjointPaths, and TwoDisjointPaths$^{\leq}$ are in W[P].*

Here, being in W[P] implies that the problems are in PTIME for each fixed k.

6 Enumerating Simple Regular Paths

We now turn to the question of enumerating simple paths with polynomial delay. A starting point is Yen's algorithm [41] for finding simple paths from a source s to target t. Yen's algorithm usually takes another parameter K and returns the K shortest simple paths, but we present a version here for enumerating all simple paths, see Algorithm 1.

We give a high-level explanation. First, observe that each shortest path in a graph is also a simple one. Therefore, the first solution is obtained by finding a shortest path p. The next shortest path must differ in some edge from p. So we search (if it exists), for all i, the shortest path that shares the first i edges with p, but not the $(i+1)$th edge. One of the shortest paths found this way is the next solution, which we again store in p. The next shortest path must again differ in some edge from the paths we already found. So we search again, for all i, for a shortest path that shares the first i edges with the new p, but not the $(i+1)$th edge. To avoid rediscovering an old path, we also forbid other edges to appear in the new path (lines 9–11). Correctness is proved in [41].

▶ **Theorem 13** (Implicit in [41])**.** *Given a graph G and nodes s, t, Algorithm 1 enumerates all simple paths from s to t in polynomial delay.*

Proof sketch. The original algorithm of Yen [41] finds, for a given G, s, t, and $K \in \mathbb{N}$ the K shortest simple paths from s to t in G. Its only difference to Algorithm 1 is that it stops when K paths are returned.

Yen does not prove that the algorithm has polynomial delay, but instead shows that the delay is $O(KN + N^3)$, where N is the number of nodes in G.[10] Unfortunately, K can be exponential in $|G|$ in general. However, the reason why the algorithm has K in the complexity is line 9, which iterates over all paths in A. If we do not store A as a linked list as in [41] but as a prefix tree of paths instead, the algorithm only needs $O(N^2)$ steps to complete the entire for-loop on line 9 (without any optimizations). Indeed, if paths p and p' share the first i edges, they will share a path of length i from the root node in the prefix tree. So we can find all forbidden $i+1$th edges by forbidding all edges that start at the end of this path. We therefore obtain delay $O(N^3)$ from Yen's analysis. ◀

[10] In [41], Section 5, he notes that computing path number k in the output costs, in his terminology, $O(KN)$ time in Step I(a) and $O(N^3)$ in Step I(b).

Algorithm 1 Yen's algorithm.

Input: Graph $G = (V, E)$, nodes s, t
Output: The simple paths from s to t in G

1: $A \leftarrow \emptyset$ ▷ A is the set of paths already written to output
2: $B \leftarrow \emptyset$ ▷ B is a set of paths from s to t
3: $p \leftarrow$ a shortest path from s to t in G
4: **while** $p \neq$ null **do** ▷ As long as we find a path p
5: **output** p
6: Add p to A
7: **for** $i = 1$ to $|p|$ **do**
8: $G' \leftarrow (V', E')$, where $V' = V \setminus V(p[0, i-1])$ and $E' = E \cap (V' \times V')$
9: **for** every path p_1 in A with $p_1[0, i-1] = p[0, i-1]$ **do**
10: Delete the edge $p_1[i-1, i]$ in G'
11: **end for** ▷ G' now no longer has paths already in A
12: Find a shortest path p_2 from $p[i, i]$ to t in G'
13: Add $p[0, i] \cdot p_2$ to B
14: **end for**
15: $p \leftarrow$ a shortest path in B ▷ $p \leftarrow$ null if $B = \emptyset$
16: Remove p from B
17: **end while**

6.1 Enumeration for Downward Closed Languages

Yen's algorithm immediately shows that EnumSimPaths can be solved in polynomial delay for languages that are closed under taking subsequences. Formally, we say that a language L is *downward closed* if, for every word $w = a_1 \cdots a_n \in L$ and every sequence $0 < i_1 < \cdots < i_k < n + 1$, we have that $a_{i_1} \cdots a_{i_k} \in L$.

▶ **Proposition 14.** *EnumSimPaths is in polynomial delay for regular expressions r such that $L(r)$ is downward closed.*

Proof sketch. Assume that (G, s, t) and r is an input for EnumSimPaths such that $L(r)$ is downward closed. Let $N = (Q, \Sigma, \delta, Q_I, Q_F)$ be an NFA for r.

We change Algorithm 1 as follows. In line 3, instead of finding a shortest path p in G, we first find a shortest path p in $(G, s, t) \times N$. We then replace every node of the form $(u, q) \in V \times Q$ in p by u.

In line 12 we need to find a shortest path in a product between $(G', p[i, i], t)$ and N. More precisely, let $J = \delta^*(\text{lab}(p[0, i]))$ and denote by N_J the NFA with initial state set J, that is, $(Q, \Sigma, \delta, J, Q_F)$. Then, in line 12 we first find a shortest path p_2 from any node in $\{(p[i, i], q_i) \mid q_i \in \delta^*(\text{lab}(p[0, i]))\}$ to any node in $\{(t, q_F) \mid q_F \in Q_F\}$ in $(G', p[i, i], t) \times N_J$. We then replace every node of the form $(u, q) \in V \times Q$ in p_2 by u. ◀

6.2 Enumeration for STEs

We show that Theorem 6(a) – the FPT part – can be extended to enumeration problems. We note that we do not need to show hardness, since the W[1]-hardness in Theorem 6(b) already holds for the decision problems.

To this end, a *parameterized enumeration problem* is defined analogously as an enumeration problem, but its input is of the form $(x, k) \in \Sigma^* \times \mathbb{N}$. It is in *FPT delay* if there exists an

algorithm that enumerates the output such that the time between two consecutive outputs is bounded by $f(k) \cdot |(x,k)|^c$ for a constant c and a computable function f. Notice that each problem in polynomial delay is also in FPT delay.

▶ Remark. Yen's algorithm makes two important calls to a black box algorithm for computing a shortest path, namely on lines 3 and 12. (There is another call to "shortest path" on line 15, but this one is only important for the ordering of the outputs and not for the correctness of the algorithm.) We can show that the algorithm is also correct if these two calls simply return a *simple path* instead of a shortest one. The main reason why this works is that no simple path from s to t is a subpath of another simple path from s to t. Therefore, we do not "lose" a path by enumerating them in this different order. Therefore, working on $(G,s,t) \times N$ as in the proof of Proposition 14, Yen's algorithm can be applied to any class of RPQs for which we can compute simple paths on lines 3 and 12 sufficiently efficiently.

Using this idea, we can show the following.

▶ **Theorem 15.** *Let $\mathcal{R}$ be a cuttable class of STEs. Then* EnumSimPaths($\mathcal{R}$) *is in FPT delay.*

To prove Theorem 15, we also need to show that the enumeration versions of PSimPath, PSimPath$^{\leq}$, and PSimPath$^{\geq}$ (from Section 5.1) are in FPT delay.

▶ **Theorem 16.** *PEnumSimPaths, PEnumSimPaths$^{\leq}$, and PEnumSimPaths$^{\geq}$ are in FPT delay.*

The proofs of the results in Theorem 16 are all along the same lines. We observe that the FPT algorithms for the decision versions of the problems can be trivially adjusted to also return a matching path if it exists. We also need to show that we can find simple paths matching *suffixes*[11] in the language (for the adapted line 12 of Yen's algorithm). This can also be done here, essentially because the suffixes of the languages we need to consider again can be solved with our FPT algorithms.

6.3 Data Complexity of Enumeration

Finally, we consider the *data complexity* of simple path enumeration. Bagan et al. [5] studied the data complexity of SimPath and discovered a dichotomy w.r.t. a class $\mathsf{C}_{\text{tract}}$ of regular languages.[12] More precisely, although SimPath(r) can be NP-complete in general, it is in PTIME if $L(r) \in \mathsf{C}_{\text{tract}}$ and NP-complete otherwise [5, Theorem 2]. Here, $\mathsf{C}_{\text{tract}}$ is defined as follows.

▶ **Definition 17** (Similar to [5], Theorem 4)**.** For $i \in \mathbb{N}$, we say that a regular language L *can be i-loop abbreviated* if, for all $w_\ell, w, w_r \in \Sigma^*$ and $w_1, w_2 \in \Sigma^+$ we have that, if $w_\ell w_1^i w w_2^i w_r \in L$, then $w_\ell w_1^i w_2^i w_r \in L$. We define $\mathsf{C}_{\text{tract}}$ as the set of regular languages L such that there exists an $i \in \mathbb{N}$ for which L can be i-loop abbreviated.

We show that Bagan et al.'s classification also leads to a dichotomy w.r.t. polynomial delay enumeration in terms of data complexity.

[11] More precisely, we need *language derivatives*, sometimes also called *Brzozowski derivatives*.

[12] They actually proved that there is a trichotomy: the third characterization is that SimPath is in AC^0 if $L(r)$ is finite.

▶ **Theorem 18.** *In terms of data complexity,*

(a) *EnumSimPaths*(r) *can be solved in polynomial delay if* $L(r) \in \mathsf{C}_{tract}$ *and*

(b) *SimPath*(r) *is NP-complete otherwise.*

Proof sketch. Part (b) is immediate from [5, Theorem 1]. For (a), our plan is to use Bagan et al.'s algorithm for simple paths (which we call BBG algorithm) as a subroutine in Yen's algorithm. We call BBG in lines 3 and 12, so that the algorithm receives

(i) a simple path from s to t that matches r in line 3 and

(ii) a simple path p_2 from $p[i]$ to t such that $p[0,i] \cdot p_2$ matches r in line 12,

respectively. Change (i) to Yen's algorithm is trivial. Change (ii) can be done by calling BBG with $(G', p[i], t)$ for the language of the automaton N_J in the proof of Proposition 14. ◀

The algorithm for Theorem 18(a) can even be adapted to output paths in increasing length or radix order.

▶ Remark (STEs versus $\mathsf{C}_{\mathsf{tract}}$). Notice that every STE is in $\mathsf{C}_{\mathsf{tract}}$. Therefore, the data complexity of their evaluation problem is in PTIME (and in polynomial delay for the enumeration version). Since $\mathsf{C}_{\mathsf{tract}}$ is a much bigger class than STEs, it is remarkable that, in Table 1, all expressions in $\mathsf{C}_{\mathsf{tract}}$ are *unions* of STEs.

7 Conclusions

Our main result shows a dichotomy on the parameterized complexity of evaluating *simple transitive expressions (STEs)*, which are a class of regular expressions powerful enough to capture over 99% of the RPQs occurring in a recent practical study [8].

The central property that we require for a class of expressions so that evaluation is in FPT is *cuttability*, i.e., constant *cut borders* (also see Figure 1). Looking at Table 1, we see that the cut borders for expressions in practice are indeed very small: it is one for a^*b, two for abc^*, and zero in all other cases.

Therefore, although the *simple path* semantics of RPQs is known to be hard in general, it seems that the RPQs that users actually ask are much less harmful. In fact, since the vast majority (over 99%) of expressions in Table 1 has cut borders of at most two, our FPT result in Theorem 6 implies that evaluation for this majority of expressions is in polynomial time combined complexity. Furthermore, matching paths can be enumerated in polynomial delay. (Recall that, if P $\neq$ NP, this is impossible even for fixed expressions: evaluation for a^*ba^* or $(aa)^*$ under simple path semantics is NP-complete.)

Finally, we note that this paper investigated evaluation for *node-distinct paths*. Preliminary work shows that our techniques can also be applied for *edge-distinct paths* [28]. More precisely, there is a similar dichotomy for edge-distinct paths, with subtle differences. For instance, evaluation for a^kb^* under edge-distinct path semantics is in FPT, whereas it is W[1]-hard under node-distinct (i.e., simple) path semantics.

We also noticed that our techniques extend beyond the class of STEs. For instance, we can also prove that, for every constant c and word w with $|w| = c$, the problem SimPath$(a^kw?a^*)$ with parameter k is in FPT. We believe that it would be very interesting to understand to which extent cuttability can be used to obtain FPT results for larger classes of RPQs (such as unions of STEs).

References

1 Margareta Ackerman and Jeffrey Shallit. Efficient enumeration of words in regular languages. *Theoretical Compututer Science (TCS)*, 410(37):3461–3470, 2009.

2 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *Journal of the ACM*, 42(4):844–856, 1995.

3 Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50(5):68:1–68:40, 2017.

4 Marcelo Arenas, Sebastián Conca, and Jorge Pérez. Counting beyond a yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *International Conference on World Wide Web (WWW)*, pages 629–638, 2012.

5 Guillaume Bagan, Angela Bonifati, and Benoît Groz. A trichotomy for regular simple path queries on graphs. In *Symposium on Principles of Database Systems (PODS)*, pages 261–272, 2013.

6 Pablo Barceló. Querying graph databases. In *Symposium on Principles of Database Systems (PODS)*, pages 175–188, 2013.

7 Andreas Björklund, Thore Husfeldt, and Sanjeev Khanna. Approximating longest directed paths and cycles. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 222–233, 2004.

8 Angela Bonifati, Wim Martens, and Thomas Timm. An analytical study of large SPARQL query logs. *Proceedings of the VLDB Endowment (PVLDB)*, 11, 2017.

9 Leizhen Cai and Junjie Ye. Finding two edge-disjoint paths with length constraints. In *International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, pages 62–73, 2016.

10 Mariano P. Consens and Alberto O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *Symposium on Principles of Database Systems (PODS)*, pages 404–416, 1990.

11 Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 323–330, 1987.

12 Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Daniel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.

13 Holger Dell. Personal communication, 2017.

14 Rodney G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness I: basic results. *SIAM Journal on Computing*, 24(4):873–921, 1995.

15 Rodney G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness II: on completeness for W[1]. *Theoretical Computer Science (TCS)*, 141(1):109–131, 1995.

16 Fedor V. Fomin, Daniel Lokshtanov, Fahad Panolan, and Saket Saurabh. Efficient computation of representative families with applications in parameterized and exact algorithms. *Journal of the ACM*, 63(4):29:1–29:60, 2016.

17 Steven Fortune, John Hopcroft, and James Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science (TCS)*, 10(2):111–121, 1980.

18 Konstantin Golenberg, Benny Kimelfeld, and Yehoshua Sagiv. Optimizing and parallelizing ranked enumeration. *Proceedings of the VLDB Endowment (PVLDB)*, 4(11):1028–1039, 2011.

19 Martin Grohe and Magdalena Grüber. Parameterized approximability of the disjoint cycle problem. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 363–374, 2007.

20 Oren Kalinsky, Yoav Etsion, and Benny Kimelfeld. Flexible caching in trie joins. In *International Conference on Extending Database Technology (EDBT)*, pages 282–293, 2017.

21 Sampath Kannan, Z. Sweedyk, and Stephen R. Mahaney. Counting and random generation of strings in regular languages. In *Symposium on Discrete Algorithms (SODA)*, pages 551–557, 1995.

22 Benny Kimelfeld and Yehoshua Sagiv. Extracting minimum-weight tree patterns from a schema with neighborhood constraints. In *International Conference on Database Theory (ICDT)*, pages 249–260, 2013.

23 Andrea S. LaPaugh and Christos H. Papadimitriou. The even-path problem for graphs and digraphs. *Networks*, 14(4):507–513, 1984.

24 Eugene L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7):401–405, 1972.

25 Leonid Libkin, Wim Martens, and Domagoj Vrgoč. Querying graphs with data. *Journal of the ACM*, 63(2):14:1–14:53, 2016.

26 Katja Losemann and Wim Martens. The complexity of regular expressions and property paths in SPARQL. *ACM Transactions on Database Systems*, 38(4):24:1–24:39, 2013.

27 Erkki Mäkinen. On lexicographic enumeration of regular and context-free languages. *Acta Cybernetica*, 13(1):55–62, 1997.

28 Wim Martens and Tina Trautner. Enumeration problems for regular path queries. *CoRR*, abs/1710.02317, 2017. URL: `https://arxiv.org/abs/1710.02317`.

29 Alberto O. Mendelzon and Peter T. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6):1235–1258, 12 1995.

30 Katta G. Murty. An algorithm for ranking all the assignments in order of increasing cost. *Operations Research*, 16(3):682–687, 1968.

31 Neo4j. Intro to cypher. `https://neo4j.com/developer/cypher-query-language/`, 2017.

32 Neo4j. The neo4j developer manual v3.3. `https://neo4j.com/docs/developer-manual/3.3/`, 2017.

33 Opencypher. `www.opencypher.org`. Visited on Sept. 14, 2017.

34 Stefan Plantikow, Mats Rydberg, and Petra Selmer. CIP2017-01-18 – configurable pattern matching semantics. `https://github.com/boggle/openCypher/blob/isomatch/cip/1.accepted/CIP2017-01-18-configurable-pattern-matching-semantics.adoc`. Visited on Aug. 08, 2017.

35 Aleksandrs Slivkins. Parameterized tractability of edge-disjoint paths on directed acyclic graphs. *SIAM Journal on Discrete Mathematics*, 24(1):146–157, 2010.

36 Dimitri Surinx, George H. L. Fletcher, Marc Gyssens, Dirk Leinders, Jan Van den Bussche, Dirk Van Gucht, Stijn Vansummeren, and Yuqing Wu. Relative expressive power of navigational querying on graphs using transitive closure. *Logic Journal of the IGPL*, 23(5):759–788, 2015.

37 Leslie G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science (TCS)*, 8(2):189–201, 1979.

38 SPARQL 1.1 query language. `https://www.w3.org/TR/sparql11-query/`, 2013. World Wide Web Consortium.

39 World wide web consortium. `www.w3.org`. Visited on Sept. 14, 2017.

40 Mihalis Yannakakis. Graph-theoretic methods in database theory. In *Symposium on Principles of Database Systems (PODS)*, pages 230–242, 1990.

41 Jin Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.

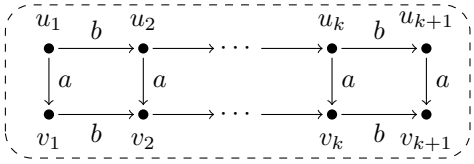


Figure 2 Internal structure of each of the gadgets $G_{i,j}$.

A Appendix

We present a proof sketch for Theorem 11. We will do an *fpt-reduction*, which we define next. If L and L' are two parameterized problems, an *fpt-reduction* from L to L' is an algorithm $\mathcal{R}$ that, given an instance (x, k) of L, outputs an instance (x', k') of L' such that

- (x, k) is a yes-instance of L if and only if (x', k') is a yes-instance of L',
- $k' \leq g(k)$ for some computable function g, and
- the running time of $\mathcal{R}$ is $f(k) \cdot |x|^{O(1)}$ for some computable function f.

▶ **Theorem 11.** *PTwoDisjointPaths is W[1]-hard.*

Proof sketch. We reduce from k-Clique, which is well known to be W[1]-complete [15, Corollary 3.2]. Let $G = (V, E)$ be an *undirected* graph and assume w.l.o.g. that $V = \{1, \ldots, n\}$.

The reduction consists of two steps. In the first step, we will construct a two-colored graph G', nodes s_a, t_a, s_b, t_b, and parameter $k' \in \Theta(k^2)$ such that G has a k-clique if and only if $(G', s_a, t_a, s_b, t_b, k') \in$ PTwoColorDisjointPaths. The graph G' will have $O(k^2 n)$ nodes. In the second step, we will construct a graph G'' and nodes s_1, t_1, s_2, t_2 such that $(G', s_a, t_a, s_b, t_b, k') \in$ PTwoColorDisjointPaths if and only if $(G'', s_1, t_1, s_2, t_2, k') \in$ PTwoDisjointPaths. The graph G'' will have $O(k^4 n)$ nodes.

We now explain the construction of G'. It contains kn gadgets $G_{i,j}$ with $i = 1, \ldots, k$ and $j = 1, \ldots, n$, each consisting of $2(k+1)$ nodes. Gadgets will be ordered in k rows, where row i has the gadgets $G_{i,1}, \ldots, G_{i,n}$. Furthermore, G' contains $k+1$ additional nodes $r_1, \ldots, r_{k+1}$ that link the rows together, and $k + 1 + k(k-1)/2$ control nodes $c_1, \ldots c_{k+1}$ and $c_{i_1 i_2}$ with $1 \leq i_1 < i_2 \leq k$ that will limit the number of disjoint paths from row $i-1$ to row i or from row i_1 to i_2, respectively. (To be fair, c_1 and c_{k+1} do not link rows together but just serve as start and end-nodes.) We define $s_a = c_1$, $t_a = c_{k+1}$, $s_b = r_1$, and $t_b = r_{k+1}$.

We will now explain how the nodes are connected in G'. We will denote by $u \xrightarrow{a} v$ that there is an a-edge from u to v (similar for b-edges). Each gadget contains a disjoint copy of $2(k+1)$ nodes which we call $u_1, u_2, \ldots, u_{k+1}$ and $v_1, v_2, \ldots, v_{k+1}$. To simplify notation, we sometimes give these nodes the same name (e.g., in Figures 3, 4, and 5), even though they are different. One such gadget is depicted in Figure 2. To avoid ambiguity, we may also refer to node u_ℓ in gadget $G_{i,j}$ by $G_{i,j}[u_\ell]$. Each gadget contains edges $u_\ell \xrightarrow{a} v_\ell$ (for every $\ell = 1, \ldots, k+1$) and $u_\ell \xrightarrow{b} u_{\ell+1}$ and $v_\ell \xrightarrow{b} v_{\ell+1}$ (for every $\ell = 1, \ldots, k$).

We now explain how the gadgets $G_{i,j}$ are connected within the same row, see Figure 3. In each row $i \in \{1, \ldots, k\}$, node r_i has two outgoing edges $r_i \xrightarrow{b} G_{i,1}[u_1]$ and $r_i \xrightarrow{b} G_{i,2}[v_1]$. We also have two incoming edges for r_{i+1}, namely $G_{i,n-1}[u_{k+1}] \xrightarrow{b} r_{i+1}$ and $G_{i,n}[v_{k+1}] \xrightarrow{b} r_{i+1}$. Furthermore, we have the edges $G_{i,j}[u_{k+1}] \xrightarrow{b} G_{i,j+1}[u_1]$ and $G_{i,j}[v_{k+1}] \xrightarrow{b} G_{i,j+1}[v_1]$ for every $j = 1, \ldots, n-1$. We also add edges $G_{i,j}[u_{k+1}] \xrightarrow{b} G_{i,j+2}[v_1]$ for every $j = 1, \ldots, n-2$.

We now explain how the gadgets $G_{i,j}$ are connected in different rows via the control nodes c_i and $c_{i_1 i_2}$ (Figure 4). We first consider the edges from row i to $i+1$. In each

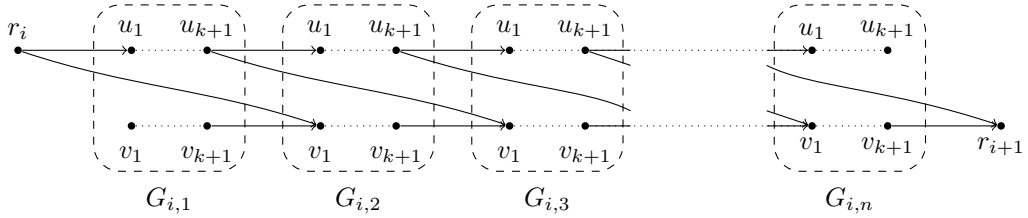


Figure 3 The b-edges in row i. The internal structure of the $G_{i,j}$ is as in Figure 2.

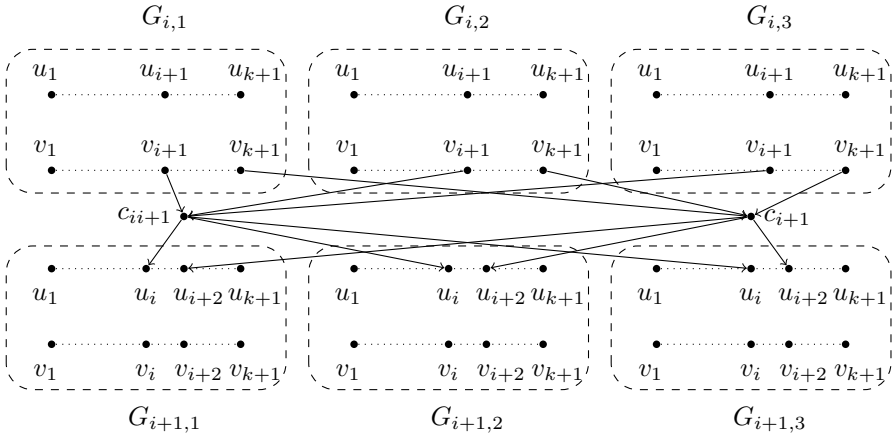


Figure 4 The a-edges from row i to row $i+1$. (We assume $n = 3$ in the picture).

row $i = 1, \ldots, k-1$, and every $j = 1, \ldots, n$, we add the edges $G_{i,j}[v_{k+1}] \xrightarrow{a} c_{i+1}$ and $c_{i+1} \xrightarrow{a} G_{i+1,j}[u_{i+2}]$. Furthermore, we add the edges $c_1 \xrightarrow{a} G_{1,j}[u_2]$ and $G_{k,j}[v_{k+1}] \xrightarrow{a} c_{k+1}$. We connect two rows i_1, i_2, with $1 \leq i_1 < i_2 \leq k$, by adding the edges $G_{i_1,j}[v_{i_2}] \xrightarrow{a} c_{i_1 i_2}$, and $c_{i_1 i_2} \xrightarrow{a} G_{i_2,j}[u_{i_1}]$ for all $j = 1, \ldots, n$.

The edges in G are modeled in G' by adding the edge $G_{i_2,x}[v_{i_1}] \xrightarrow{a} G_{i_1,y}[u_{i_2+1}]$ if and only if $1 \leq i_1 < i_2 \leq k$, $x \neq y$, and $(x, y) \in E$. This is illustrated in Figure 5.

Finally, we define $k' = k(k-1)/2 \cdot 5 + 3k$.

This concludes the construction of G'. We denote by G'_a the subgrapn of G' that contains only the a-edges.

▶ **Lemma 19.** *The graph G'_a has the following properties:*

(a) *G'_a is a DAG. Moreover, there is a strict total order $<_c$ on all control nodes C such that, for every path from a node $v \in C$ to another node $v' \in C$ where no intermediate vertex is in C, node v' is the successor of v in $<_c$.*

(b) *Each path in G'_a has length exactly k' if and only if it is from c_1 to c_{k+1}.*

(c) *Each path in G'_a of length k' visits all control nodes, i.e., it contains all c_i and $c_{i_1 i_2}$, with $i \in \{1, \ldots, k+1\}$ and $1 \leq i_1 < i_2 \leq k$.*

(d) *Each path in G'_a of length k' has at least one edge $u_\ell \xrightarrow{a} v_\ell$ in every row of G'_a.*

We omit the proof of Lemma 19 due to space constraints.

We prove that $(G, k) \in k$-Clique if and only if $(G', k') \in$ PTwoColorDisjointPaths. Let us first assume that the undirected graph G has a k-clique with nodes $\{n_1, \ldots, n_k\}$. Then an a-path can go from c_1 to c_{k+1} using only the gadgets G_{i,n_i} with $i = 1, \ldots, k$. The reason is that, since $(n_{i_1}, n_{i_2}) \in E$, the edges $G_{i_2,n_{i_2}}[v_{i_1}] \xrightarrow{a} G_{i_1,n_{i_1}}[u_{i_2+1}]$ exist for all $i_1 \leq i_2$. Due to Lemma 19(b), this path has exactly k' edges. The b-path, on the other hand, can go from

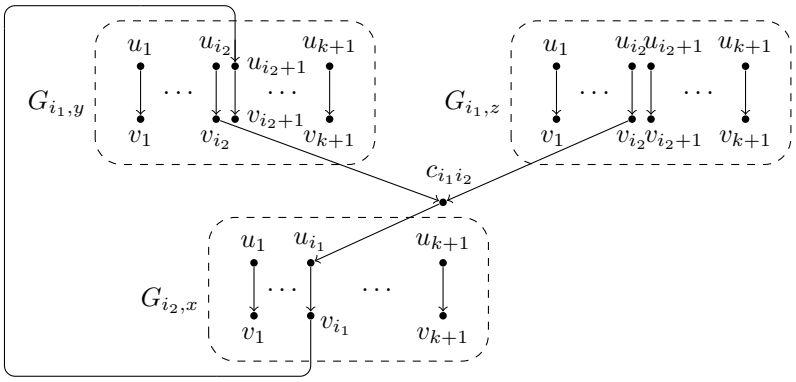


Figure 5 The a-edges in the gadgets and between gadgets $G_{i_1,y}$, $G_{i_1,z}$ and $G_{i_2,x}$, with $i_1 < i_2 - 1$, under the assumption that $(x, y) \in E$ and $(x, z) \notin E$.

r_1 to r_{k+1} and skip exactly G_{i,n_i} for all $i = 1, \ldots, k$ (using the diagonal edges in Figure 3). Since it skips these G_{i,n_i}, it is node-disjoint from the a-path and therefore we have a solution for PTwoColorDisjointPaths.

For the other direction let us assume that there exists a simple a-path p_a from c_1 to c_{k+1} and a simple b-path p_b from r_1 to r_{k+1} in G' such that p_a and p_b are node-disjoint and p_a has length k'. We show that G has a k-clique. Since every b-path from r_1 to r_{k+1} goes through each row, that is, from r_i to r_{i+1} for all $i = 1, \ldots, k$, this is also the case for p_b. By construction p_b must also skip exactly one gadget in each row, using the diagonal edges in Figure 3. Furthermore, for each gadget $G_{i,j}$ that p_b visits, it must be the case that it either visits all nodes $u_1, \ldots, u_{k+1}$ or all nodes $v_1, \ldots, v_{k+1}$. (This is immediate from Figure 2, showing all internal edges of a gadget.) Therefore, since p_a and p_b are node-disjoint, the p_a cannot visit any gadget $G_{i,j}$ already visited by p_b. Therefore, p_a, which goes from c_1 to c_{k+1}, can only do so through the k skipped gadgets, call them G_{i,n_i} for $i = 1, \ldots, k$. Recall that the edges between the gadgets $G_{i_2,n_{i_2}}$ and $G_{i_1,n_{i_1}}$ only exist if $(n_{i_1}, n_{i_2}) \in E$. As these edges are necessary for the existence of the a-path from c_1 to c_{k+1} that uses only the skipped gadgets, all nodes n_i must be pairwise adjacent in G. That is, they form a clique of size k in G. This completes the first step of the reduction.

We now explain the second and final step. We construct the graph G'' from G' by replacing each b-edge with a b-path of length k'. (Even though PTwoDisjointPaths does not care about a-edges or b-edges, we keep them to simplify the reasoning in the remainder of the proof.) We define $s_1 = s_a$, $t_1 = t_a$, $s_2 = s_b$, and $t_2 = t_b$.

▶ **Observation 20.** *In G'', we have that*

(a) *every path from c_1 to c_{k+1} has length at least k' and*

(b) *every path from c_1 to c_{k+1} has length exactly k' if and only if it is an a-path.*

We prove the observation using Lemma 19(b). For part (a) we have two cases. If a path from c_1 to c_{k+1} is an a-path, the result is immediate from Lemma 19(b). If it uses at least one b-edge, then it uses at least k' b-edges by construction. Thus, the path will have length at least k'.

For part (b), if a path from c_1 to c_{k+1} has length exactly k', it uses at least one a-edge since c_{k+1} only has incoming a-edges. If it used at least one b-edge, it would therefore use at least $k' + 1$ edges which contradicts that the length is k'. The converse direction is immediate from Lemma 19(b). This concludes the proof of Observation 20.

We show that $(G', s_a, t_a, s_b, t_b, k') \in$ PTwoColorDisjointPaths if and only if $(G'', s_1, t_1, s_2, t_2, k') \in$ PTwoDisjointPaths. If $(G', s_a, t_a, s_b, t_b, k') \in$ PTwoColorDisjointPaths, then we can use the corresponding paths in G'' (where we follow the longer b-paths in G'' instead of the b-edges in G').

Conversely, if $(G'', s_1, t_1, s_2, t_2, k') \in$ PTwoDisjointPaths, it follows from Observation 20 that p_1 can only use a-edges. We now show that the path p_2 from r_1 to r_{k+1} can only use b-edges, that is, we show that it cannot use a-edges. There are three types of a-edges in G'': (i) the ones from and to control nodes, (ii) "upward" edges that connect row i_2 to row i_1 with $i_1 < i_2$, and (iii) edges from u_ℓ to v_ℓ in one gadget.

Notice that, by construction, p_2 must visit nodes in row 1 and later also nodes in row k. To do so, p_2 cannot use edges from or to control nodes (type (i)), since, due to Lemma 19(c), p_1 already visits all control nodes. So p_2 cannot go from row i to a row j with $i < j$ via a-edges. This means that, if $i < j$, then p_2 can only go from row i to row j through r_{i+1} (and through nodes in row $i + 1$), since every remaining path from row i to a larger row goes through r_{i+1}. So, in order to go from row 1 to row k, path p_2 needs to visit all nodes $r_2, \ldots, r_k$, in that order. This means that it is also impossible for p_2 to use edges of type (ii). Indeed, if p_2 were to use an edge from row j to row i with $j > i$, then it would need to visit r_{i+1} a second time to arrive back in row j. Finally, if p_2 used an edge of type (iii) in row i, then, by construction, it would have to visit every gadget in this row. But since p_1 already uses at least one edge from u_ℓ to v_ℓ in each row, see Lemma 19(d), this means that p_2 cannot be node-disjoint with p_1. This completes the proof. ◀

Massively Parallel Entity Matching with Linear Classification in Low Dimensional Space

Yufei Tao
Chinese University of Hong Kong, Shatin, Hong Kong
taoyf@cse.cuhk.edu.hk

Abstract

In *entity matching classification*, we are given two sets R and S of objects where whether r and s form a match is known for each pair $(r, s) \in R \times S$. If R and S are subsets of domains $D(R)$ and $D(S)$ respectively, the goal is to discover a *classifier function* $f : D(R) \times D(S) \to \{0, 1\}$ from a certain class satisfying the property that, for every $(r, s) \in R \times S$, $f(r, s) = 1$ if and only if r and s are a match.

Past research is accustomed to running a learning algorithm directly on all the labeled (i.e., match or not) pairs in $R \times S$. This, however, suffers from the drawback that even reading through the input incurs a quadratic cost. We pursue a direction towards removing the quadratic barrier. Denote by T the set of matching pairs in $R \times S$. We propose to accept R, S, and T as the input, and aim to solve the problem with cost proportional to $|R| + |S| + |T|$, thereby achieving a large performance gain in the (typical) scenario where $|T| \ll |R||S|$.

This paper provides evidence on the feasibility of the new direction, by showing how to accomplish the aforementioned purpose for *entity matching with linear classification*, where a classifier is a linear multi-dimensional plane separating the matching and non-matching pairs. We actually do so in the MPC model, echoing the trend of deploying massively parallel computing systems for large-scale learning. As a side product, we obtain new MPC algorithms for three geometric problems: linear programming, batched range counting, and dominance join.

2012 ACM Subject Classification Theory of computation → Massively parallel algorithms

Keywords and phrases Entity Matching, Linear Programming, Range Counting, Dominance Join, Massively Parallel Computation

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.20

Funding This work was partially supported by a direct grant (Project Number: 4055079) from CUHK and by a Faculty Research Award from Google.

1 Introduction

Entity matching has garnered considerable attention from the database community (representative works: [7, 13, 15, 21, 22, 23]). Generally speaking, the objective is to determine whether two objects o_1 and o_2 belong to the same entity, e.g., "are the two voice recordings by the same person?". For this purpose, we are given training sets R and S such that, each pair $(r, s) \in R \times S$ carries a *label*, indicating whether r matches s. The goal of learning is to produce a *classifier* of a certain type that correctly decides the matching results for all $(r, s) \in R \times S$. This classifier will then be applied to new, unknown, object pairs $(o_1, o_2) \notin R \times S$.

The literature on entity matching is accustomed to applying a learning algorithm directly on the labeled $R \times S$. This suffers from the drawback that even reading $R \times S$ itself forces a quadratic cost. This issue has been recognized; for instance, [21] stated that "*the resulting quadratic complexity is inefficient for large datasets even on cloud infrastructures.*" In practice, it is commonly dealt with using the so-called "blocking technique" (e.g., [13, 21, 22, 23]),

which relies on heuristic rules to divide R and S into *blocks*, such that only objects from the same block can possibly match each other (e.g., a block includes the male recordings, while the female recordings form another block). This technique, however, offers no provable improvement in theory.

We will pursue a different direction towards removing the quadratic pitfall. The fundamental observation is that often times the non-matching pairs in $R \times S$ by far out-number the matching ones. So, why not supply only R, S, and the set T of matching pairs in $R \times S$? Conceptually, nothing is lost because if $(r, s) \in R \times S$ does not appear in T, it must be a non-matching pair. The hope is that we can design algorithms that produce precisely the same learning result, but with cost proportional to $|R| + |S| + |T|$, thus harvesting a significant gain when $|T| \ll |R||S|$. Algorithmically, this is essentially asking: is it possible to carry out the learning *without* enumerating $R \times S$?

This paper will establish theoretical evidence on the feasibility of this direction. We will look at a specific form of entity matching—called *entity matching with linear classification*—where the classifier is a multi-dimensional plane separating matching and non-matching pairs. We will achieve the purpose on coarse-grained parallel computing platforms (e.g., MapReduce [10] and Spark [29]), responding to the need of deploying such platforms for large-scale learning. Our algorithmic endeavor will require us to attack several fundamental geometric problems as well. Next, after introducing the computation model, we will formally define all the relevant problems, and then present our results.

1.1 The Computation Model

We will work under the *massively parallel computation* (MPC) model due to its popularity in the database area [1, 3, 4, 5, 19, 20, 24, 25]. In this model, p machines are interconnected in a network. An algorithm runs in *rounds*, each of which has two phases. In the first phase, every machine carries out computation locally, whereas in the second phase, the machines exchange information over the network. Specifically, a machine u can send messages to arbitrary machines in the second phase, as long as u has prepared those messages in the first phase (of the same round). This means, for example, that u is not allowed to decide what to send based on what is received in this round.

The performance of an algorithm is measured by (i) the amount of network communication, and (ii) The number of rounds executed. The first metric, specifically, is measured in *load*. Formally, the *load of a round* equals $\max_{i=1}^{p} \lambda_i$ where λ_i is the number of words communicated (sending and receiving combined) by the i-th machine in this round. The *load* of the algorithm equals the maximum load of all the rounds executed. In the early development of the model, research concentrated on algorithms that perform an exceedingly small number (e.g., 1 or 2) of rounds [2, 3, 25], because old MapReduce implementations suffered from expensive system-level overhead in executing a round. Such overhead has been considerably reduced in today's systems [14, 29], making it feasible for algorithms to perform more (but preferably $O(1)$) rounds [1, 4, 19, 20, 24, 28]. This is especially true when additional rounds help to reduce the load by significant factors.

Several remarks are in order. By default, at the beginning of an MPC algorithm, the input is distributed on the p machines at the will of an adversary, but in a balanced manner, i.e., each machine stores $O(N/p)$ elements, where N is the input size. Some of our algorithms drop the "balanced" requirement, and instead allow an arbitrarily uneven distribution of the N elements onto the p machines. All the previous work on the MPC model considers the value of p to be less than the input size N by a polynomial factor. Our analysis will assume $p \leq N^{0.9}$. The constant 0.9 is not mandatory, and can be replaced by any constant less than 1

by adjusting the constants in our analysis accordingly. We assume that every number (from either the input or intermediate computation) fits in a word. All our algorithms transmit a number always as a whole word, while we allow the transmission of individual bits in proving lower bounds. Regarding CPU calculation, our algorithms will use only comparisons, $+$, $-$, $\cdot$, and $/$. Finally, when we say that a randomized algorithm succeeds "with high probability" (w.h.p.), we mean that its failure probability is $O(1/N^2)$.

1.2 Definition of the Main Problem

We consider entity matching between two sets R, S of objects whose similarity is reflected by absolute coordinate differences in a multi-dimensional space (the dimensions are the extracted features based on which learning is performed). Formally, let R and S each be a set of points in $\mathbb{R}^d$. Denote the coordinate of a point $q \in \mathbb{R}^d$ on dimension $i \in [1, d]$ as $q[i]$. Each pair $(r, s) \in R \times S$ defines a d-dimensional point $q_{(r,s)}$ whose coordinate on dimension $i \in [1, d]$ is $q_{(r,s)}[i] = |r[i] - s[i]|$ that is, $q_{(r,s)}$ captures the similarity of r and s on the d dimensions. Let $Q_{R,S}$ be the (perhaps multi-) set of points thus obtained, namely: $Q_{(R,S)} = \{q_{(r,s)} \mid (r, s) \in R \times S\}$. Each $q_{(r,s)} \in Q_{(R,S)}$ carries a *label* of either 1 or 0, indicating whether r and s are a match. Define:

$$T_{(R,S)} = \{(r, s) \in R \times S \mid \text{label of } q_{(r,s)} = 1\},$$

that is, $T_{(R,S)}$ contains all and only the matches between R and S. We are now ready to clarify the input and output of the *entity matching with linear classification* (EMLC) problem:

- Input: R, S, and $T_{(R,S)}$.
- Output: A linear plane π such that the label-1 points of $Q_{(R,S)}$ fall on one side of π, while the label-0 points on the other. If such a separation plane does not exist, the output is NULL.

1.3 Our Results: The Main and Accompanying Problems

The main result of the paper is:

▶ **Theorem 1.** *Set $N = |R| + |S| + |T_{(R,S)}|$ for the EMLC problem in $\mathbb{R}^d$ with $d = O(\text{polylog}\, N)$. There is a $d^{O(1)}$-round MPC algorithm that solves the problem w.h.p. with load $d^{O(d)} \cdot N\sqrt{\log N}/p$.*

The result is most appealing when the dimensionality d is a fixed constant, in which case our algorithm performs constant rounds with load $O(N\sqrt{\log N}/p)$. The theorem confirms the superiority of accepting R, S, and $T_{(R,S)}$ as the input to EMLC (rather than $Q_{(R,S)}$) when $|T_{(R,S)}| \ll |R||S|$. Indeed, even just "uploading" the labeled $Q_{(R,S)}$ onto the p machines will entail a network cost of $\Omega(|R||S|/p)$ on at least one machine[1]. The algorithm in Theorem 1 combines new MPC solutions to several geometric problems, as explained below.

Linear Programming (LP). Define a *half-space* h in $\mathbb{R}^d$ as the set $\{q \in \mathbb{R}^d \mid \sum_{i=1}^{d} \alpha_i \cdot q[i] \geq \beta\}$, where real numbers $\alpha_1, \alpha_2, ..., \alpha_d$ and β are the *coefficients* of h. The input to LP is a set H of N half-spaces. Let $I(H)$ be the intersection of all the half-spaces in H. The objective is to (i) determine whether $I(H)$ is empty, and (ii) if not, report a point $q \in I(H)$ with the smallest $q[1]$ (i.e., minimize the coordinate on dimension 1).

[1] Remember that $Q_{(R,S)}$ is a set of *labeled* points; and the labeling depends on the underlying application, and does not need to follow any geometric patterns. Thus, $Q_{(R,S)}$ cannot be computed from R and S.

We are not aware of existing results in the MPC model, but the problem has been well studied in PRAM (see [11, 17] for a collection of results). The focus of those algorithms, however, is to minimize "work"[2], rather than communication. In particular, for constant d, when adapted to finish within constant rounds under MPC, they all have a load of $\Omega(\frac{N/p}{\text{polylog } N})$. We will prove:

▶ **Theorem 2.** *Suppose that the dimensionality d is $O(\text{polylog } N)$. Fix any constants ϵ, δ satisfying $\delta > \epsilon > 0$ and $p \leq N^{\delta-\epsilon}$. There is a $d^{O(1)}$-round MPC algorithm which solves the LP problem w.h.p. with load $O(N^{\delta}/p)$. Furthermore, this is true even if the half-spaces are distributed on the p machines arbitrarily at the beginning (i.e., perhaps unevenly).*

The theorem may appear a bit unusual such that it is worth offering a guided tour, assuming $d = O(1)$ for simplicity:

- First, look at the realistic scenario where $p = O(\text{polylog } N)$. In this case, the theorem essentially says that LP can be settled in constant rounds with load $O(N^{\delta}/p)$, for arbitrarily small constant $\delta > 0$ (simply set $\epsilon = \delta/2$).
- Consider now the less realistic scenario where $p = \Theta(N^c)$ for some constant $c > 0$. The theorem tells us that the load is $O(N^{c+\epsilon}/p) = O(N^{\epsilon})$ for arbitrarily small constant $\epsilon > 0$ (set $\delta = c + \epsilon$).
- In all circumstances, Theorem 2 asserts that LP can be settled with a load *polynomially* less than N/p. Recall that $\Omega(\frac{N/p}{\log N})$ is a load lower bound for numerous problems under the MPC model, e.g., sorting [16], set intersection [27], equi-joins [19], etc. It is thus interesting to see that a problem as sophisticated as LP requires far less communication.
- The theorem does not require a balanced distribution of the input on the p machines. This is a nice property when the input is produced by a preceding operator (e.g., a join) which may leave a huge number—in the worst case $\Omega(N)$—of half-spaces on one machine.

Batched Range Counting (BRC). In $\mathbb{R}^d$, define R as a set of axis-parallel rectangles, and S as a set of points. The objective of BRC is to report, for every rectangle $r \in R$, the number $|r \cap S|$ (i.e., how many points of S are covered in r).

Set $N = |R| + |S|$. For constant d, Hu et al. [19] recently presented a constant-round algorithm with load $O((N/p)\log^{d-1} p)$. We improve their result to optimality:

▶ **Theorem 3.** *There is an $O(d)$-round deterministic MPC algorithm that solves the BRC problem with load $d^{O(d)} \cdot N/p$. Furthermore, any constant-round deterministic algorithm must entail a load of $\Omega(N/p)$ when $d = 2$.*

Dominance Join (DJ). In $\mathbb{R}^d$, let R and S be two sets of points. The objective of DJ is to report all pairs $(r, s) \in R \times S$ such that s dominates r (namely, $s[i] \geq r[i]$ for all $i \in [1, d]$). An algorithm is *α-out-balanced* if each machine produces at most $\alpha \cdot \text{OUT}/p$ result pairs, where OUT is the total number of result pairs.

Let $N = |R| + |S|$. For constant d, Hu et al. [19] proposed a constant-round algorithm with load $O((N/p)\log^{d-1} p + \sqrt{\text{OUT}/p})$ (there was no discussion on α-out-balance in [19]). We prove:

[2] The *work* of a PRAM algorithm is the product of (i) the number of steps performed and (ii) the number of processors deployed.

▶ **Theorem 4.** *For the DJ problem, there is a $d^{O(d)}$-out-balanced deterministic MPC algorithm that performs $O(d^2)$ rounds, and incurs load $d^{O(d)} \cdot (N + OUT)/p$.*

For constant d, our algorithm does not always improve the load of [19], but it does whenever $\text{OUT} = o(N \log^{d-1} p)$. This is especially useful when OUT is small and has no dependence on d, as is indeed what we rely on to prove a "dimensionality-oblivious" load of $O(N\sqrt{\log N}/p)$ for EMLC under any constant d (see Theorem 1). Adopting the algorithm of [19] instead will introduce a multiplicative factor of $\log^{O(d)} p$.

2 Preliminaries

ϵ-Nets and Its Application to LP. We will review some useful results on LP. Let H be a set of d-dimensional planes. An *ϵ-net* of H is a subset $X \subseteq H$ with the following property: any point $q \in \mathbb{R}^d$ that appears in at least $\epsilon|H|$ half-spaces of H, must be in at least one half-space of X.

▶ **Lemma 5** (Theorem 5.7.3 of [26])**.** *Consider an integer $r \in [4d, N]$ and a value $\Delta \in (0, 1/2]$ satisfying $r \leq 1/\Delta$. Let X be a set of $O(d \cdot r \log(1/\Delta))$ samples from H taken uniformly at random with replacement. Then, X is a $(1/r)$-net of H with probability at least $1 - O(\Delta)$.*

The next corollary follows directly from the above lemma and the definition of ϵ-net (by considering the complement of each half-space in H and X):

▶ **Corollary 6.** *The following property holds with probability at least $1 - O(\Delta)$ on the sample set X in Lemma 5: any point $q \in \mathbb{R}^d$ inside $I(X)$ (i.e., the intersection of the half-spaces in X). can be outside less than N/r half-spaces in H.*

The next result we review concerns an elegant framework for solving the LP problem. At the beginning, take an arbitrary subset X_0 of H. In general, given a subset X_i of H (for $i \geq 0$), we perform an *iteration* as follows. First, solve the LP problem on X_i. If $I(X_i)$ is empty, stop the algorithm and return empty. Otherwise, let ϕ_i be the solution point found. If $i = d + 1$, return ϕ_{d+1} (for the original LP problem). If $i < d + 1$, collect the set Y_i of half-spaces in H that do *not* contain ϕ_i. Launch the next iteration with $X_{i+1} = X_i \cup Y_i$. The lemma below states an interesting fact:

▶ **Lemma 7** ([8])**.** *Regardless of X_0, the above algorithm always solves the problem correctly.*

The previous two lemmas constitute the core idea behind a number of LP algorithms (e.g., [8, 6, 17]). The version below is based an original proposition from [8], but includes optimization from [6]. Simply choose X_0 to be a random sample set (with replacement) of size $O(d \cdot \sqrt{N \log N})$. Applying Lemma 5 with $r = \Theta(\sqrt{N/\log N})$ and $\Delta = 1/N^3$, we know that X_0 has the property stated in Corollary 6 with probability at least $1 - 1/N^3$. The fact $X_0 \subseteq X_i$ $(i \geq 1)$ implies that every X_i also has this propertiy w.h.p. Hence, w.h.p., $|Y_i| = O(N/r) = O(\sqrt{N \log N})$ holds for every $i \in [1, d]$, which in turn tells us that $|X_i| \leq |X_0| + i \cdot N/r = O(d \cdot \sqrt{N \log N})$ for all $i \in [1, d + 1]$.

The above description essentially reduces an LP problem of size N to $d + 1$ smaller problems each with size $O(d \cdot \sqrt{N \log N})$. Standard analysis shows that, when d is a constant, the running time is $O(N)$ w.h.p. in the RAM model.

Broadcasting in MPC. Suppose that the machines carry ids $1, 2, ..., p$. In designing an algorithm, we often need to send information from one or more machines to several machines in a *continuous* id range. The lemma below explains the cost when there is one source machine:

▶ **Lemma 8** ([18]). *If a machine needs to broadcast x words to y machines in a continuous id range, this can be done in constant rounds with load*
- $O(x)$ *if* $x \geq y^c$ *for an arbitrarily small constant* $c > 0$*;*
- $O(y^c)$ *otherwise, where* $c > 0$ *is an arbitrarily small constant.*

This implies the next result for the case when there are multiple source machines:

▶ **Corollary 9.** *Suppose that z machines are holding in total x words of messages. In constant rounds, these x words can be broadcast to y machines in a continuous id range with load $O(x + y^c)$, where $c > 0$ is an arbitrarily small constant (this holds regardless of the value of z).*

Proof. In one round, instruct all the z machines to send their messages to one machine, which requires load $O(x)$. Then, the recipient machine broadcasts all the words to the y machines. The previous lemma indicates that this can be done with constant rounds and load $O(x + y^c)$. ◀

3 Linear Programming in MPC

We will prove Theorem 2 in this section. As before, let H be the input set of N half-spaces in $\mathbb{R}^d$, and $I(H)$ be their intersection. The goal is to find a point $q \in I(H)$ with the smallest coordinate on dimension 1. We will give a $d^{O(1)}$-round MPC algorithm with load $O(N^\epsilon)$ which is $O(N^\delta/p)$, for the selection of δ and ϵ as in Theorem 2. Our algorithm for LP can be thought of as an efficient implementation of the framework described in Section 2. The main idea is to *do away with* the default requirement of MPC that, the input should be distributed onto the machines evenly. Indeed, the requirement does not even allow us to distribute $X_1 = X_0 \cup Y_0$ (as defined in Section 2) for recursion, because this would lead to a load of $\Omega(d\sqrt{N \log N}/p)$. We circumvent this obstacle by sampling in an "in-place" manner (keeping the samples on the original machines), and performing the recursion anyway in spite of an unbalanced input distribution.

3.1 In-Place Sampling

Suppose that there is a set X of elements that are distributed on p machines in an arbitrary manner (not necessarily balanced). We want to take t samples from X uniformly at random with replacement. If an element $e \in X$ is sampled t_e times, we indicate the fact by storing the count t_e on the machine where e is stored. No elements need to be moved across the machines.

The above purpose can be fulfilled in $O(c)$ rounds with load $O(p^{1/c})$, for any constant $c \geq 1$. This holds regardless of $|X|$ and t. We can organize the p machines arbitrarily into a tree where each internal node, except possibly the root, has $\Theta(p^{1/c})$ child nodes. The height of the tree is $O(c)$. The *level* of a node is the number of edges on its path to the root (the root is at level 0).

Let $X(u)$ be the set of elements stored on node (i.e., a machine) u. In the first step, each node (i.e., a machine) u obtains the number—denoted as x_u—of elements of X that are stored in the subtree rooted at u. This is trivial if u is a leaf because x_u is simply $|X(u)|$. Inductively, after a node v at level $i \geq 1$ has acquired x_v, it sends x_v to its parent u (at level $i - 1$). Node u then sums up the numbers from its child nodes, together with $|X_u|$, to obtain x_u. This bottom-up process takes $O(c)$ rounds to finish, and has a load equal to the internal fanout $O(p^{1/c})$.

In the second step, each node u decides the number t_u of samples to be taken from X_u. This can be done in a top-down manner. Suppose, inductively, that a node u at level $i \geq 0$ has been informed that y_u samples should be from the subtree rooted at u (for the base case of induction, u is the root, and $y_u = t$). Then, u generates y_u independent integers uniformly at random from $[1, x_u]$. Remember that, from the previous step, u has already obtained x_v for every child v, and that $|X_u| + \sum_v x_v = x_u$. Hence, from the y_u random integers, u can generate t_u, as well as y_v for each child v. This step also has $O(c)$ rounds and load $O(p^{1/c})$.

Now that every machine u knows t_u, it performs the sampling locally at X_u, which completes the in-place sampling.

3.2 The LP Algorithm

The core of our algorithm can be summarized as:

▶ **Lemma 10.** *Fix a sufficiently large integer N, an integer $d = O(\text{polylog}\, N)$, and a constant satisfying $0 < \epsilon < 1$. Consider that we are given an LP problem of size $n \in [N^\epsilon, N]$.*

Available to us is an algorithm $\mathcal{A}_1$ that is capable of solving any LP problem of a smaller size $m = O(d\sqrt{n \log N})$ with probability at least $1 - \Delta_{\mathcal{A}_1}$, even if the input is distributed on the p machines arbitrarily. Suppose that $\mathcal{A}_1$ performs $\rho(m)$ rounds, and incurs load $\lambda(m)$.

Then, for the LP problem on size n, there is an $O(d) + (d+1) \cdot \rho(m)$-round algorithm $\mathcal{A}_2$ that, with probability at least $1 - (1/N^3 + O(d \cdot \Delta_{\mathcal{A}_1}))$, solves the problem with load $\max\{O(d + p^\epsilon), \lambda(m)\}$, regardless of how the input is distributed on the machines initially.

Before proving the lemma, let us first explain how it leads to Theorem 2. Here, set N to the input size of the original LP problem, and d to the problem's dimensionality, and take the value of ϵ from Theorem 2. Given an LP problem with input size $n \leq N$, we do the following:

- If $n \leq N^\epsilon/d$, simply send all the half-spaces to one machine, and solve the problem there (with probability 1). Requiring apparently only $\rho(n) = 1$ round and load $\lambda(n) = O(N^\epsilon)$, this serves as the base algorithm $\mathcal{A}_1$ for the recursion Lemma 10 implies.
- Otherwise, apply the algorithm $\mathcal{A}_2$ in Lemma 10, which performs $\rho(n) = O(d) + (d+1) \cdot \rho(m)$ rounds, and requires load $\lambda(n) = \max\{O(d + p^\epsilon), \lambda(m)\}$. Effectively, the algorithm solves $d + 1$ smaller problems of size at most m.

To settle the original LP problem, simply run the above algorithm on H. For performance analysis, we need to solve the above recurrences in order to obtain $\rho(N)$ and $\lambda(N)$.

Notice that $m = O(d\sqrt{n \log N}) \leq n^{0.51}$ for N larger than a certain constant, applying $d = O(\text{polylog}\, N)$ and $n \geq N^\epsilon$. We compute $\rho(n)$ by recursively computing $d + 1$ copies of $\rho(m)$ until $n \leq N^\epsilon/d$. The recursion tree triggered by $\rho(N)$ has a depth of $O(\log(1/\epsilon))$, and contains $(d+1)^{O(\log(1/\epsilon))} = d^{O(1)}$ nodes. As a result, $\rho(N) = O(d) \cdot d^{O(1)} = d^{O(1)}$. The analysis of $\lambda(N)$ is much simpler: by (i) $d = \text{polylog}(N)$, and (ii) $\lambda(n) = O(N^\epsilon)$ in the base case ($n \leq N^\epsilon$), we know that $\max\{O(d + p^\epsilon), \lambda(m)\}$ is always $\lambda(m)$. Therefore, we have $\lambda(N) = O(N^\epsilon)$.

Our algorithm *fails* if, in any application of Lemma 10, $\mathcal{A}_2$ fails. The above analysis indicates that Lemma 10 is applied $d^{O(1)}$ times in total. Hence, we guarantee that the algorithm fails with probability $d^{O(1)} \cdot O(d/N^3) = o(1/N^2)$, as desired in Theorem 2.

Proof of Lemma 10. It suffices to combine in-place sampling with the ideas reviewed in Section 2. No half-spaces of H need to be moved from one machine to another. For each machine u and any subset $X \subseteq H$, we denote by $X(u)$ the set of half-spaces in X that are stored at u.

To solve the given LP problem of size n, first perform in-place sampling to obtain a sample set X_0 of size $O(d\sqrt{n \log N})$. It follows from Section 3.1 that the sampling takes $O(1/\epsilon) = O(1)$ rounds and incurs load $O(p^\epsilon)$. Applying Lemma 5 with $\Delta = 1/N^3$ and $r = \sqrt{n/\log(1/\Delta)}$, we know that X_0 has the property in Corollary 6 with probability at least $1 - 1/N^3$.

In general, given a subset X_i of H ($i \geq 0$) which is not necessarily evenly distributed on the machines, an *iteration* invokes algorithm $\mathcal{A}_1$ to solve the LP problem on X_i with probability at least $1 - \Delta_{\mathcal{A}_1}$. If $I(X_i)$ is empty, stop the algorithm and return empty. Otherwise, let ϕ_i be the solution point found. If $i = d+1$, return ϕ_i. Now consider $i < d+1$, the machine holding ϕ_i broadcasts the point to all other machines in constant rounds with load $O(d + p^\epsilon)$ (Lemma 8). Each machine u identifies the set $Y_i(u)$ of half-spaces that are in $H(u)$ and do not contain ϕ_i. This, conceptually, defines $Y_i = \bigcup_u Y_i(u)$ and $X_{i+1} = X_i \cup Y_i$. The next iteration is then launched with X_{i+1}.

When X_0 has the property in Corollary 6, by the reasoning mentioned in Section 2, we know that $X_i \leq |X_0| + i \cdot n/r = O(d\sqrt{n \log N})$ for all $i \in [1, d+1]$. Therefore, each iteration requires $O(1) + \rho(m)$ rounds and incurs load $\max\{O(d + p^\epsilon), \lambda(m)\}$. The number of rounds of all iterations is $O(d) + (d+1) \cdot \rho(m)$. The algorithm correctly solves the LP problem with probability at least $1 - (1/N^3 + (d+1)\Delta_{\mathcal{A}_1})$. This completes the proof of Lemma 10.

4 Batched Range Counting

This section will be mainly concerned with:

Batched Dominance Counting (BDC). In $\mathbb{R}^d$, let R and S be two sets of points. The objective is to report, for each point $r \in R$, the number of points $s \in S$ such that r dominates s (namely, $r[i] \geq s[i]$ for all $i \in [1, d]$).

Set $N = |R| + |S|$. We will give an MPC algorithm that solves the above problem in $O(d)$ rounds with load $d^{O(d)} \cdot N/p$. Our technique is bootstrapping in nature. First, design a base algorithm that performs constant rounds and has load $O(p^d + d^2 \cdot N/p)$. This load is acceptable only when p is small, but otherwise can be rather expensive (e.g., for $p = N^{0.9}$). To fix this, we recursively partition the problem, until the number of machines in each subproblem is small enough to apply the base algorithm with an acceptable load. The details will be presented in Sections 4.1 and 4.2. Load $O(N/p)$ is asymptotically optimal for constant d, as shown in Section 4.3 via a reduction from *sparse set disjointness*. These BDC results will then imply Theorem 3, as explained in Section 4.4.

4.1 A Base Algorithm with Load $O(p^d + d^2 \cdot N/p)$

For each dimension $i \in [1, d]$, divide the space $\mathbb{R}^d$ into p *slabs* using $p-1$ planes perpendicular to the dimension, such that each slab has $\Theta(N/p)$ points of the input $R \cup S$. These slabs are said to be *on dimension* i. The slab boundaries of all dimensions together define a grid of p^d cells. Count the number of points of S in each cell. We want to give every machine a copy of all the p^d counts (i.e., one per cell). It is fundamental to do so in constant rounds with load $O(p^d + dN/p)$.[3] A point $s \in S$ dominated by a point $r = (r[1], ..., r[d]) \in R$ must appear

[3] The slab boundaries on each dimension can be decided by sorting, which takes constant rounds and load $O(N/p)$ [16]. Carrying this out for all dimensions simultaneously increases the load by a factor of d, without changing the number of rounds. By Corollary 9, the $d \cdot p^d$ boundaries can be broadcast to

- (**Cat. 1**) Either in a cell completely covered by $(-\infty, r[1]] \times (-\infty, r[2]] \times ... \times (-\infty, r[d]]$;
- (**Cat. 2**) Or in the same slab as r on at least one dimension.

The number of points in the first category can be calculated from the p^d counts already available on the machine where r is stored. It remains to count the pairs (r, s) in the second category.

Since a slab has $O(N/p)$ points, we can send all those points to one machine, which then locally performs the counting of Category 2 for every $r \in R$ in the slab. Specifically, for each $i \in [1, d]$, we send the points of each slab on dimension i to a distinct machine, which can be done in a single round with load $O(dN/p)$. Doing so for all dimensions simultaneously still in one round increases the load to $O(d^2 \cdot N/p)$. A slight complication is that, if r and s are in the same slab on more than one dimension, s may be counted multiple times for r. This can be easily avoided by introducing a consistent de-duplication policy: e.g., count s for r in a slab on dimension i only if they are not in the same slab on any dimension $j < i$. We thus have obtained a constant-round algorithm of load $O(p^d + d^2 \cdot N/p)$.

4.2 An Algorithm with Load $d^{O(d)} \cdot N/p$

Next, we describe an alternative algorithm for BDC. More generally, given an arbitrary $f \in (0, 1]$, we discuss how to deploy $p' = p^f$ machines to tackle a BDC problem of input size $N' = O(p' \cdot N/p)$; this will be referred to as an *f-BDC problem*. The original BDC problem corresponds to $f = 1$. Our algorithm will perform $\rho(f)$ rounds, and incur load $\lambda(f)$, where $\rho(f)$ and $\lambda(f)$ are functions to be resolved in the analysis. Notice that $N'/p' = O(N/p)$.

Base Case $f \leq 0.1/d$. Simply invoke the base algorithm of Section 4.1. It finishes in $\rho(f) = O(1)$ rounds, and incurs load $\lambda(f) = O(p'^d + d^2 \cdot N'/p') = O(p^{df} + d^2 \cdot N/p) = O(p^{0.1} + d^2 \cdot N/p) = O(d^2 \cdot N/p)$, where the last step used $p \leq N^{0.9}$.

Inductive Case $f > 0.1/d$. Along each dimension, divide the space into $p^{0.1/d}$ slabs each with $\Theta(N'/p^{0.1/d})$ points of $R \cup S$. As in Section 4.1, the slabs of all dimensions together define a grid, which here has $p^{0.1}$ cells. Count the number of points of S in each cell, and give every machine a copy of the $p^{0.1}$ counts (one count per cell). This can be done in constant rounds with load $O(p^{0.1} + dN'/p') = O(dN/p)$.

We now proceed in the same way as in the base algorithm except that Category 2 is now handled by solving $(f - 0.1/d)$-BDC problems recursively. Focusing on an arbitrary dimension, let us assign $\Theta(p'/p^{0.1/d})$ machines to each slab on the dimension. Since a slab has $\Theta(N'/p^{0.1/d})$ points, counting the number of pairs (r, s) of Category 2 within the slab is an $(f - 0.1/d)$-BDC problem. The $(f - 0.1/d)$-BDC problems of all slabs on the dimension can be solved in parallel using $\alpha(f - 0.1/d)$ rounds and load $\lambda(f - 0.1/d)$. Doing so for all d dimensions together increases the load by a factor of d, without changing the number of

all the p' machines in constant rounds with load $O(d \cdot p^d)$. Every machine can now locally calculate the boundary faces of all cells. We then count the number of points of S in each cell using the sum-by-key algorithm of [19] (for each point in S, its "key" is the id of the cell containing it; this algorithm counts, for every key k, the number of points having k as the key), using constant rounds with load $O(N/p)$. Broadcasting the p^d counters to all the machines can be done in constant rounds and load $O(p^d)$ by Corollary 9.

rounds. This indicates the following recurrences:

$$\begin{aligned}\rho(f) &= O(1) + \rho(f - 0.1/d)\\ \lambda(f) &= d \cdot \lambda(f - 0.1/d) + O(d \cdot N/p)\end{aligned}$$

Resolving them gives $\rho(1) = O(d)$, and $\lambda(1) = d^{O(d)} \cdot N/p$. We thus have obtained an $O(d)$-round BDC algorithm with load $d^{O(d)} \cdot N/p$.

4.3 Lower Bound

We can prove that any constant-round deterministic BDC algorithm must incur a load of $\Omega(N/p)$ when $d = 2$. For this purpose, let us revisit the *sparse set disjointness* problem in communication complexity. Alice and Bob each take a size-k subset of $\{1, 2, ..., U\}$; denote their subsets as X_A and X_B, respectively. They want to determine whether $X_A \cap X_B = \emptyset$ by communicating the least number of bits. It is known that any deterministic algorithm requires sending $\Omega(k \log(2U/k))$ bits between the two parties [27].

Fix $U = k^2$ and set the word length to $\log_2 U$ bits. Suppose that we are given a constant-round MPC algorithm $\mathcal{A}$ that solves the BDC problem with load $o(N/p)$ (measured in words). We can use $\mathcal{A}$ for sparse set disjointness as follows. Alice constructs a 2D point set $R = \{(x, U - x) \mid x \in X_A\}$, and likewise, Bob constructs $S = \{(x, U - x) \mid x \in X_B\}$. It is easy to verify that the BDC problem returns a non-zero count for at least one point in R, if and only if $X_A \cap X_B \neq \emptyset$. We ask Alice and Bob to simulate $\mathcal{A}$, by asking each of them to be in charge of $p/2$ machines. At the beginning, Alice (or Bob) distributes her (or his, resp.) elements on her (or his, resp.) $p/2$ "machines" evenly. They then do the communication as instructed by $\mathcal{A}$. Since each round of $\mathcal{A}$ has load $o(k/p)$, every "machine" from Alice can send $o(k/p)$ words over to Bob, and vice versa. Hence, a round necessitates only $o(k)$ words—namely $o(k \log U)$—bits of communication. As $\mathcal{A}$ finishes in constant rounds, we have obtained an algorithm solving sparse set disjointness with $o(k \log U)$ bits, contradicting the $\Omega(k \log(2U/k)) = \Omega(k \log U)$ lower bound.

4.4 Proving Theorem 3 and Beyond

Batched range counting (BRC) can be reduced to BDC elegantly [12]. For each rectangle $r \in R$, the number $|r \cap S|$ can be aggregated from the "dominance counts" of the 2^d corners of r, where the *dominance count* of a corner q equals how many points in S are dominated by q. Hence, we can convert BRC into 2^d instances of BDC of the same size, solve all of them simultaneously, and then aggregate the 2^d dominance counts for each $r \in R$. This gives an algorithm that settles the BRC problem in $O(d)$ rounds with load $2^d \cdot d^{O(d)} \cdot N/p = d^{O(d)} \cdot N/p$. Finally, the lower bound on BDC clearly applies to BRC. This completes the proof of Theorem 3.

A Semi-Group Remark. Our solution actually settles the more general problem of batched range *sum*. Specifically, let R and S be defined as in BRC. However, each point $s \in S$ is now associated with a *weight* from a semi-group domain. The goal now is to report, for every $r \in R$, the sum of the weights of all the points in $r \cap S$. The proposed algorithm solves this problem in $O(d)$ rounds with load $d^{O(d)} \cdot N/p$, which is optimal for constant d.

5 Dominance Join

This section will establish Theorem 4 on the DJ problem. Recall that we have two sets of points in $\mathbb{R}^d$: R and S. The objective is to find all $(r, s) \in R \times S$ such that s dominates r. Set $N = |R| + |S|$, and denote by OUT the number of pairs in the result. We will solve the problem with an $d^{O(d)}$-out-balanced algorithm that runs in $O(d^2)$ rounds with load $d^{O(d)} \cdot (N + \text{OUT})/p$.

Our solution benefits from the techniques of Section 4 in two ways. First, it deploys BDC (batched dominance counting) as an intermediate operation. Second, following the bootstrapping approach of Section 4, we will first develop a base algorithm for DJ that runs in constant rounds but with load $O(p^d + d^2 \cdot N/p + d \cdot \text{OUT}/p)$, and then attain the target performance guarantees with recursion. Attention, however, should be paid to how we avoid a $\log^{O(d)} p$ term in the load. For this purpose, we abandon the range-tree-styled "canonical decomposition" approach in [19], which is inherently penalized by a multiplicative $O(\log p)$ factor per dimension. Instead, we will show how to use the grid partitioning approach in Section 4 for reporting, by integrating it with new cell-pairing and machine-assignment ideas.

To facilitate discussion, in the following presentation, we will (conceptually) color the points of R in *red*, and those of S in *blue*.

5.1 A Base Algorithm with Load $O(p^d + d^2 \cdot N/p + d \cdot \text{OUT}/p)$

Divide the space into p slabs on each dimension, such that each slab has $\Theta(N/p)$ points of $R \cup S$. Number these slabs as $1, 2, ..., p$ in ascending order (call those numbers *slab ids*). The slab boundaries of all d dimensions define a grid of p^d cells. For each cell in the grid, count the number of red points and number of blue points. This gives a total of $2p^d$ counts. Broadcast a copy of all these counts to the p machines. As in Section 4.1, all the above can be done in constant rounds with load $O(p^d + dN/p)$. We say that a cell c_1 *dominates* another c_2 if the slab id of c_1 is larger than or equal to that of c_2 on every dimension.

Run the algorithm of Section 4.1 to obtain the value of OUT, which requires constant rounds and load $O(p^d + dN/p)$. Recall from Section 4.1 that the set of result pairs (r, s) can be classified into two categories. We slightly rephrase the description of those categories below:

- **(Cat. 1)** The cell containing r dominates the cell containing s;
- **(Cat. 2)** r and s are in the same slab on at least one dimension.

Next, we will explain how to produce the result pairs of each category, in such a way that $O(d \cdot \text{OUT}/p)$ pairs are produced on each machine.

Reporting Category 1

Let $k = O(p^{2d})$ be the number of *dominance cell pairs* (c, c') satisfying (i) $c \neq c'$, and (ii) c dominates c'. Let us order all such pairs lexicographically: $(c_1, c'_1), (c_2, c'_2), ..., (c_k, c'_k)$. For each $i \in [1, k]$, denote by $F(i)$ the set of red points in c_i, and by $G(i)$ the set of blue points in c'_i. We say that (c_i, c'_i) is *heavy* if $|F(i)| \cdot |G(i)| \geq \text{OUT}/p$, or *light* if $0 < |F(i)| \cdot |G(i)| < \text{OUT}/p$. Note that, importantly, a light pair must produce at least one result pair. Next, we will report (r, s) contributed by heavy and light pairs separately.

Dealing with Heavy Pairs. There are at most p heavy dominance cell pairs; denote them as: (c_{i_1}, c'_{i_1}), (c_{i_2}, c'_{i_2}), ..., $(c_{i_{k_1}}, c'_{i_{k_1}})$ where $k_1 \leq p$. For each $j \in [1, k_1]$, assign $p_j = \Theta(\frac{|F(i_j)||G(i_j)|}{\text{OUT}} p)$ machines with continuous ids, making sure $\sum_j p_j = p$. Ask these p_j

machines to collect (from other machines) the points of $F(i_j)$ and $G(i_j)$ evenly, so that each machines gets $O(1 + \frac{|F(i_j)|+|G(i_j)|}{p_j})$ points. Doing so for all $j \in [1, k_1]$ in parallel takes constant rounds and entails load

$$\begin{aligned} O\left(dN/p + d \cdot \max_{j=1}^{k_1} \frac{|F(i_j)| + |G(i_j)|}{p_j}\right) &= O\left(dN/p + d \cdot \max_{j=1}^{k_1} \frac{|F(i_j)| + |G(i_j)|}{|F(i_j)||G(i_j)|} \frac{\text{OUT}}{p}\right) \\ &= O(d(N + \text{OUT})/p) \end{aligned}$$

where the first term $O(dN/p)$ is because every machine may send out all of the $O(dN/p)$ points in its local storage. Then, for each $j \in [1, k_1]$, compute $F(i_j) \times G(i_j)$ (i.e., the cartesian product) on the p_j machines assigned. The algorithm of [3] can do this in constant rounds with load $O(d\frac{|F(i_j)|+|G(i_j)|}{p_j} + d\sqrt{\frac{|F(i_j)||G(i_j)|}{p_j}}) = O(d(N + \text{OUT})/p)$. Their algorithm is $O(1)$-out-balanced, with each machine producing $O(\frac{|F(i_j)||G(i_j)|}{p_j}) = O(\text{OUT}/p)$ result pairs.

Dealing with Light Pairs. Let us redefine $(c_{i_1}, c'_{i_1}), (c_{i_2}, c'_{i_2}), ..., (c_{i_{k_2}}, c'_{i_{k_2}})$ as the light pairs in lexicographic order where $k_2 = O(p^{2d})$ is the number of such pairs. Partition the range $[1, k_2]$ into p intervals $I_1, I_2, ..., I_p$ such that $O(\text{OUT}/p)$ result pairs are produced from every I_i, namely: for any $j \in [1, p]$, it holds that $\sum_{x \in I_j} |F(i_x)||G(i_x)| = O(\text{OUT}/p)$. Such a partitioning definitely exists because, by definition of light pair, $|F(i_x)||G(i_x)| < \text{OUT}/p$ for all $x \in [1, k_2]$. Assign one machine to every I_j ($j \in [1, p]$). This machine collects the $F(i_x)$ and $G(i_x)$ for all $x \in I_j$, and then locally produces all the results pairs from them. This requires constant rounds and load

$$\begin{aligned} O\Big(dN/p + d \cdot \max_{j=1}^{p} \sum_{x \in I_j} |F(i_x)| + |G(i_x)|\Big) &= O\Big(dN/p + d \cdot \max_{j=1}^{p} \sum_{x \in I_j} |F(i_x)||G(i_x)|\Big) \\ &= O(d(N + \text{OUT})/p). \end{aligned}$$

Reporting Category 2

Let $\sigma_1, \sigma_2, ..., \sigma_p$ be the slabs of a certain dimension. The objective of *processing* this dimension is to report all result pairs (r, s) such that r and s are both in an identical slab. We achieve the purpose with constant rounds and $O(dN/p)$ load as follows. For $i \in [1, p]$, let R_i and S_i be the set of red and blue points in σ_i, respectively. To each σ_i, use a machine to obtain (in parallel) locally the number OUT_i of result pairs that are produced by R_i and S_i. Then, re-assign $p_i = \Theta(\lceil \frac{\text{OUT}_i}{\text{OUT}} p \rceil)$ machines to σ_i, making sure $\sum_i p_i = p$. For every σ_i, we carry out the following steps in parallel. First, broadcast R_i and S_i to all the p_i assigned machines, which by Corollary 9 can be done in constant rounds with load $O(d(|R_i| + |S_i|) + p_i) = O(dN/p + p)$. Each of the p_i machines then produces $O(\text{OUT}/p)$ distinct result pairs from R_i and S_i. This distinctness requirement can be fulfilled by, e.g., resorting to the lexicographic order of the result pairs.

We process all the dimensions simultaneously using the above strategy. The number of rounds remains unchanged, but the load is increased d times to $O(dp + d^2 \cdot N/p) = O(p^d + d^2 \cdot N/p)$. Each machine now reports at most $O(d \cdot \text{OUT}/p)$ result pairs. The algorithm is therefore $O(d)$-out-balanced. Duplicate reporting can be avoided by the same approach described in Section 4.1. Note that this approach can only decrease the number of result pairs reported on a machine, and therefore, will not affect the $O(d)$-out-balance.

Thus, we have obtained a constant-round $O(d)$-out-balanced algorithm that solves the DJ problem with a load bounded by $O(p^d + d^2 \cdot N/p + d \cdot \text{OUT}/p)$.

5.2 An Algorithm with Load $d^{O(d)} \cdot (N + \text{OUT})/p$

We defer the algorithm to Appendix A, which also serves as a proof for Theorem 4.

6 Entity Matching with Linear Classification

We are now ready to solve EMLC. Recall that the input includes two sets R, S of points in $\mathbb{R}^d$, and the set $T_{(R,S)}$ of matching pairs in $R \times S$. Every pair $(r, s) \in T_{(R,S)}$ defines a point $q_{(R,S)} \in Q_{(R,S)}$ with label 1, whereas every $(r, s) \in R \times S \setminus T_{(R,S)}$ defines a point $q_{(R,S)} \in Q_{(R,S)}$ with label 0. We want to find a plane π that separates the 1-label points from the 0-label points in $Q_{(R,S)}$, or assert that such a plane does not exist.

Section 6.1 will explain the core of our technique: a reduction from EMLC to linear programming and dominance join. We will first present the reduction without having any computation models in mind. Then, Section 6.2 will give a fast implementation in MPC to establish Theorem 1.

6.1 Reduction to LP and DJ

A plane π can be described as $\{q \in \mathbb{R}^d \mid \sum_{i=1}^{d} \alpha_i \cdot q[i] = \beta\}$ where α_i ($i \in [1, d]$) and β are the coefficients. It defines two half-spaces: $\pi^+ : \{q \in \mathbb{R}^d \mid \sum_{i=1}^{d} \alpha_i \cdot q[i] \geq \beta\}$, and $\pi^- : \{q \in \mathbb{R}^d \mid \sum_{i=1}^{d} \alpha_i \cdot q[i] < \beta\}$. If a separation plane π exists, then π^+ contains all and only either the label-1 points, or the label-0 points. Due to symmetry, it suffices to discuss the former situation.

The space $\mathbb{R}^d$ where the points of $Q_{(R,S)}$ distribute is the *primal* space. We instead look at the corresponding LP problem in the *dual* $(d+1)$-dimensional space that consists of all the possible $(\alpha_1, \alpha_2, ..., \alpha_d, \beta)$. Every point $q \in Q_{(R,S)}$ defines a half-space $h_q = \{(\alpha_1, \alpha_2, ..., \alpha_d, \beta) \in \mathbb{R}^{d+1} \mid \sum_{i=1}^{d} \alpha_i \cdot q[i] \vee \beta\}$ where the operator $\vee$ is $\geq$ if q has label 1, or $<$ otherwise. This spawns a set H of $|R||S|$ half-spaces. The LP problem on H returns a solution point if and only if π exists for the original EMLC problem. To avoid the quadratic pitfall, however, we cannot afford to materialize H, which precludes directly invoking an LP algorithm on H. Next, we show that materialization is unnecessary.

Set $n = |R| + |S|$; it must hold that $|H| \leq n^2$. Take a random sample set Σ (with replacement) of H with size $|\Sigma| = \Theta(d \cdot n\sqrt{\log n})$. By Lemma 5, Σ is a $(1/r)$-net of H with probability $1 - 1/n^2$ where $r = \Theta(n\sqrt{\log n})$. Then, we carry out the LP algorithmic framework in Section 2, by setting $X_0 = \Sigma$. By reviewing the framework, one can see that it suffices to implement two key steps:

- **(LP Step)** In the i-th ($i \geq 0$) iteration, solve the LP problem on X_i.
- **(DJ Step)** If a solution point ϕ_i is found in the LP step, find the set Y_i of half-spaces in H that do not contain ϕ_i.

Note that X_i, Y_i, and ϕ_i are as defined in Section 2. The details of each step are clarified below (after which it will become clear why the second step is named as such).

The LP Step. The analysis in Section 2 tells us that $|X_i| = O(|R||S|/r) = O(n^2/r) = O(n\sqrt{\log n})$ when Σ is a $(1/r)$-net. This means that w.h.p. we are able to afford to materialize X_i and apply an LP algorithm on it.

The DJ Step. Divide Y_i into:

- Y^1: the set of half-spaces $h_q \in Y_i$ that are defined by a point $q \in Q_{(R,S)}$ with label 1;
- Y^0: the set of half-spaces $h_q \in Y_i$ that are defined by a point $q \in Q_{(R,S)}$ with label 0.

As all the matching pairs have been given in $T_{(R,S)}$, we can easily produce Y^1 by scanning $T_{(R,S)}$ once: for each $(r,s) \in T_{(R,S)}$, include $h_{q_{(r,s)}}$ into Y^1 if it does not contain ϕ_i.

The computation of Y^0 is less trivial. Let us write out the coordinates of ϕ_i as $(\alpha_1^*, \alpha_2^*, ..., \alpha_d^*, \beta^*)$. The next lemma gives a crucial observation:

▶ **Lemma 11.** *Define $Z(\phi_i)$ as the set of all $(r,s) \in R \times S$ satisfying*

$$\sum_{i=1}^{d} \alpha_i^* \cdot |r[i] - s[i]| \quad \geq \quad \beta^*. \tag{1}$$

Then, $Y^0 = Z(\phi_i) \setminus T_{(R,S)}$.

Proof. See Appendix B. ◀

Motivated by the above, we turn our attention to computing $Z(\phi_i)$, after which Y^0 can then be obtained with a set difference with $T_{(R,S)}$. It turns out that, as elaborated below, this can be achieved by solving 2^d instances of DJ, each of which is between two sets of $(d+1)$-dimensional points with sizes $|R|$ and $|S|$, respectively.

Fix an arbitrary point $r \in R$. To compute $T_{(R,S)}$, we want to find all $s \in S$ satisfying (1). There are 2^d different ways to remove the absolute signs in (1) because, on each dimension $i \in [1,d]$, we should independently distinguish $r[i] \geq s[i]$ and $r[i] < s[i]$. Due to symmetry, it suffices to consider

$$r[i] \geq s[i] \text{ for all } i \in [1,d] \tag{2}$$

in which case (1) can be written as:

$$\sum_{i=1}^{d} \alpha_i^* \cdot r[i] \geq \beta^* + \sum_{i=1}^{d} \alpha_i^* \cdot s[i]. \tag{3}$$

Motivated by this, we construct two sets R', S' of $(d+1)$-dimensional points:

$$R' = \left\{ \left(r[1], r[2], ..., r[d], \sum_{i=1}^{d} \alpha_i^* \cdot r[i] \right) \middle| \, r \in R \right\}$$

$$S' = \left\{ \left(s[1], s[2], ..., s[d], \beta^* + \sum_{i=1}^{d} \alpha_i^* \cdot s[i] \right) \middle| \, s \in S \right\}$$

The DJ between the two sets returns all $(r', s') \in R' \times S'$ such that r' dominates s'. Every (r', s') corresponds to a unique pair $(r,s) \in R \times S$ satisfying (2) and (3), and vice versa. Performing all the 2^d DJs will produce the desired $Z(\phi_i)$. It is possible that a pair $(r,s) \in R \times S$ is reported multiple, but apparently at most 2^d times.

A final remark concerns the size of $Z(\phi_i)$. When Σ (i.e., the sample set on H) is a $(1/r)$-net (recall that $r = \Theta(n\sqrt{\log n})$, we know from Section 2 that $|Y_i| = O(d \cdot n\sqrt{\log n})$. As $Y^0 \subseteq Y_i$, Lemma 11 implies that w.h.p. $Z(\phi_i)$ has size at most $O(d \cdot n\sqrt{\log n}) + |T_{(R,S)}|$.

6.2 Proof of Theorem 1

To implement the reduction of Section 6.1, we first need to obtain the sample set Σ. This is equivalent to sampling $t = O(d \cdot n\sqrt{\log n})$ pairs from $R \times S$ with replacement. For this purpose, we will take a size-t sample set Σ_1 of R, a size-t sample set Σ_2 of S, randomly permute both, and then produce Σ by paring the i-th element (after permutation) of Σ_1 with that of Σ_2.

To execute the strategy, first apply the in-place sampling algorithm of Section 3 to obtain Σ_1 and Σ_2, respectively. The fact $p \leq n^{0.9}$ assures that w.h.p. every machine produces $O(t/p)$ samples from Σ_1 and Σ_2. To see why, notice that a machine holds $O(n/p)$ points of $R \cup S$ initially, and hence, produces $\Theta(\frac{n}{p}\frac{t}{n}) = \Theta(t/p)$ samples in expectation. By Chernoff, it produces $O(t/p)$ samples with probability at least $1 - e^{-\Theta(t/p)} = 1 - e^{-\Omega(n^{0.1})}$, which is $1 - o(1/(p \cdot n^2))$. Applying the union bound on p machines gives the desired high probability result. Randomly permuting Σ_1 (similarly for Σ_2) can be done by first associating each sample with an independent random integer in some domain $[1, U]$, and then, sorting all the samples by their random integers. The sorted list is a random permutation of Σ_1, as long as no two samples' random integers collide, which can be ensured with probability at least $1 - 1/n^2$ as long as U is chosen to be n^c for a sufficiently large constant c. Finally, pairing up Σ_1 and Σ_2 can also be achieved with sorting. Therefore, w.h.p. the above algorithm returns Σ in constant rounds with load $O(d \cdot t/p)$.

Implementing the rest of the reduction is a simple matter. Carry out the LP step with Theorem 2, which has $d^{O(1)}$ rounds and requires load $o(|X_i|/p) = o(n\sqrt{\log n}/p)$ w.h.p. Regarding the DJ step, recall that we need to produce Y^1 and Y^0. The former can be obtained in constant rounds with load $O(d \cdot (N + |T_{(R,S)}|)/p)$ by broadcasting ϕ_i to all machines, and then, redistributing Y^1 evenly. To obtain Y^0, we first prepare the 2^d instances of DJ locally on the p machines. These instances can then be solved in parallel with Theorem 4 in $O(d^2)$ rounds and load $d^{O(d)} \cdot (n + \text{OUT})/p$ where $\text{OUT} = |Z(\phi_i)|$, which is $O(d \cdot n\sqrt{\log n}) + |T_{(R,S)}|$ w.h.p. Since the algorithm of Theorem 4 is $d^{O(d)}$-out-balanced, each machine holds $d^{O(d)} \cdot |Z(\phi_i)|/p$ elements of $Z(\phi_i)$. This allows the set difference $Z(\phi_i) \setminus T_{(R,S)}$ to be implemented again by sorting in constant rounds and load $d^{O(d)} \cdot |Z(\phi_i)|/p + O(d \cdot |T_{(R,S)}|/p)$. Putting everything together, we have obtained an algorithm solving the EMLC problem w.h.p. using $d^{O(1)}$ rounds and load $d^{O(d)} \cdot (n\sqrt{\log n} + |T_{(R,S)}|)/p$.

It is worth mentioning that the algorithm of Theorem 3, although not explicitly invoked above, is required in the DJ algorithm of Theorem 4.

7 Conclusions

This paper pursues a new direction to perform entity matching classification with a cost that is sub-quadratic to the number of objects involved in the training. The novelty lies in receiving only the *matching* pairs, as opposed to the traditional approach of accepting all of the matching and non-matching pairs. We have demonstrated the first success of the direction, by proving its feasibility for *entity matching with linear classification* (EMLC) on the massively parallel computation (MPC) model. Specifically, for small d (our results are most appealing for constant d, and could also be interesting for $d = O(\log \log N / \log \log \log N)$), we have designed an MPC algorithm that performs EMLC in a small number of rounds, with a load that is (almost) as low as the communication cost of uploading the input to a parallel system. In doing so, we have also obtained new MPC results on several geometric problems: linear programming, batched range counting, and dominance join.

The limitations of the new direction are still yet to be understood when the classifier functions are more sophisticated. On the positive side, we want to develop algorithms with similar performance guarantees for various forms of classifiers (e.g., decision trees and support vector machines), perhaps leveraging recent advances on sparse matrices such as [9]. On the negative side, it is an intriguing question to ask when the direction actually does *not* work, namely, when it is essentially the best approach to work directly with a labeled cartesian product.

References

1 Foto N. Afrati, Manas R. Joglekar, Christopher Re, Semih Salihoglu, and Jeffrey D. Ullman. GYM: A multiround distributed join algorithm. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 4:1–4:18, 2017.

2 Foto N. Afrati, Paraschos Koutris, Dan Suciu, and Jeffrey D. Ullman. Parallel skyline queries. *Theory Comput. Syst.*, 57(4):1008–1037, 2015.

3 Foto N. Afrati and Jeffrey D. Ullman. Optimizing multiway joins in a map-reduce environment. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 23(9):1282–1298, 2011.

4 Pankaj K. Agarwal, Kyle Fox, Kamesh Munagala, and Abhinandan Nath. Parallel algorithms for constructing range and nearest-neighbor searching data structures. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 429–440, 2016.

5 Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 273–284, 2013.

6 Timothy M. Chan and Eric Y. Chen. Multi-pass geometric algorithms. *Discrete & Computational Geometry*, 37(1):79–102, 2007.

7 Xu Chu, Ihab F. Ilyas, and Paraschos Koutris. Distributed data deduplication. *Proceedings of the VLDB Endowment (PVLDB)*, 9(11):864–875, 2016.

8 Kenneth L. Clarkson. Las vegas algorithms for linear and integer programming when the dimension is small. *Journal of the ACM (JACM)*, 42(2):488–499, 1995.

9 Kenneth L. Clarkson and David P. Woodruff. Low rank approximation and regression in input sparsity time. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 81–90, 2013.

10 Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

11 Martin E. Dyer and Sandeep Sen. Fast and optimal parallel multidimensional search in PRAMs with applications to linear programming and related problems. *SIAM Journal of Computing*, 30(5):1443–1461, 2000.

12 Herbert Edelsbrunner and Mark H. Overmars. On the equivalence of some rectangle problems. *Information Processing Letters (IPL)*, 14(3):124–127, 1982.

13 Vasilis Efthymiou, George Papadakis, George Papastefanatos, Kostas Stefanidis, and Themis Palpanas. Parallel meta-blocking for scaling entity resolution over big heterogeneous data. *Inf. Syst.*, 65:137–157, 2017.

14 Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang, Zeyu Zheng, Bohan Zhang, Yang Cao, and Chao Tian. Parallelizing sequential graph computations. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 495–510, 2017.

15 Chaitanya Gokhale, Sanjib Das, AnHai Doan, Jeffrey F. Naughton, Narasimhan Rampalli, Jude W. Shavlik, and Xiaojin Zhu. Corleone: hands-off crowdsourcing for entity matching. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 601–612, 2014.

16 Michael T. Goodrich. Communication-efficient parallel sorting. *SIAM Journal of Computing*, 29(2):416–432, 1999.

17 Michael T. Goodrich and Edgar A. Ramos. Bounded-independence derandomization of geometric partitioning with applications to parallel fixed-dimensional linear programming. *Discrete & Computational Geometry*, 18(4):397–420, 1997.

18 Xiao Hu, Paraschos Koutris, and Ke Yi. The relationships among coarse-grained parallel models. *Technical report, HKUST*, 2016.

19 Xiao Hu, Yufei Tao, and Ke Yi. Output-optimal parallel algorithms for similarity joins. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 79–90, 2017.

20 Bas Ketsman and Dan Suciu. A worst-case optimal multi-round algorithm for parallel computation of conjunctive queries. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 417–428, 2017.

21 Lars Kolb, Andreas Thor, and Erhard Rahm. Load balancing for mapreduce-based entity resolution. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 618–629, 2012.

22 Hanna Köpcke and Erhard Rahm. Frameworks for entity matching: A comparison. *Data Knowl. Eng.*, 69(2):197–210, 2010.

23 Hanna Köpcke, Andreas Thor, and Erhard Rahm. Evaluation of entity resolution approaches on real-world match problems. *Proceedings of the VLDB Endowment (PVLDB)*, 3(1):484–493, 2010.

24 Paraschos Koutris, Paul Beame, and Dan Suciu. Worst-case optimal algorithms for parallel query processing. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 8:1–8:18, 2016.

25 Paraschos Koutris and Dan Suciu. Parallel evaluation of conjunctive queries. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 223–234, 2011.

26 Ketan Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice-Hall, 1994.

27 Mert Saglam and Gábor Tardos. On the communication complexity of sparse set disjointness and exists-equal problems. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 678–687, 2013.

28 Yufei Tao, Wenqing Lin, and Xiaokui Xiao. Minimal MapReduce algorithms. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 529–540, 2013.

29 Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI*, pages 15–28, 2012.

A Proof of Theorem 4

Next, we present an algorithm for the DJ problem that will establish Theorem 4. More generally, given an arbitrary $f \in (0, 1]$, we discuss how to deploy $p' = p^f$ machines to tackle a DJ problem of input size $N' = O(p' \cdot N/p)$ and output size $\text{OUT}' = O(p' \cdot \frac{OUT}{p})$; this will be referred to as an *f-DJ problem*. The original DJ problem corresponds to $f = 1$. Our algorithm will be $\alpha(f)$-out-balanced, perform $\rho(f)$ rounds, and incur load $\lambda(f)$, where $\alpha(f), \rho(f)$, and $\lambda(f)$ are functions that are the target of analysis. Note that $N'/p' = O(N/p)$

and $\text{OUT}'/p' = O(\text{OUT}/p)$. We will enforce the invariant that the value of OUT' is already known (at $f = 1$, this can be ensured by running the BDC algorithm of Section 4).

Base Case $f \leq 0.1/d$. Simply invoke the base algorithm. From the previous subsection, we know that the algorithm finishes in $\rho(f) = O(1)$ rounds, and incurs load $\lambda(f) = O(d^2 \cdot N/p + d \cdot \text{OUT}/p)$. The algorithm is $O(d)$-out-balanced, i.e., $\alpha(f) = O(d)$.

Inductive Case $f > 0.1/d$. Impose a grid by dividing $\mathbb{R}^d$ along each dimension into $p^{0.1/d}$ slabs. For each cell, count the number of red points and number of blue points therein. Send a copy of these counts to all machines. All these can be accomplished in constant rounds with load $O(dN/p)$.

The result pairs can be divided into Categories 1 and 2 as explained in Section 5.1. We will report the two categories separately.

Category 1. This category can be produced in the same way as described in the base algorithm (replacing N, p, and OUT with N', p', and OUT' respectively), which performs constant rounds and requires load $O(d(N + \text{OUT})/p)$.

Category 2. The procedure for this category also follows the framework illustrated in the base algorithm. Consider the *processing* of a dimension with slabs $\sigma_1, \sigma_2, ..., \sigma_{p^{0.1/d}}$ (recall that the objective is to find result pairs (r, s) such that r and s are both in some slab). For each $i \in [1, p^{0.1/d}]$, let R_i and S_i be the set of red and blue points in σ_i, respectively.

To compute the number OUT_i of result pairs produced by R_i and S_i, we apply the BDC algorithm of Section 4, by assigning $\Theta(p'/p^{0.1/d})$ machines to σ_i. Run an instance of algorithm in parallel for each slab. All instances finish in $O(d)$ rounds, and incur load $d^{O(d)} \cdot \frac{N'/p^{0.1/d}}{p'/p^{0.1/d}} = d^{O(d)} \cdot N/p$.

For each i, we will compute the result pairs from R_i and P_i with

$$p_i = \Theta\left(\frac{p'}{p^{0.1/d}} + \frac{\text{OUT}_i}{\text{OUT}'} \cdot p'\right)$$

machines, making sure that $\sum_i p_i = p'$. For this purpose, distribute R_i and P_i onto the p_i machines evenly. Doing so for all i simultaneously takes constant rounds and incurs load

$$O\left(\frac{dN'}{p'} + \max_{i=1}^{p^{0.1/d}} \frac{d \cdot (|R_i| + |P_i|)}{p_i}\right) = O\left(\frac{dN'}{p'} + \frac{dN'/p^{0.1/d}}{p'/p^{0.1/d}}\right) = O(dN/p).$$

Now, producing the result pairs from R_i and P_i becomes an $(f - 0.1/d)$-DJ problem. Solving it recursively in parallel for all i. This requires $\rho(f - 0.1/d)$ rounds with load $\lambda(f - 0.1/d)$. Each of the p_i machines produces at most $\alpha(f - 0.1/d) \cdot \text{OUT}_i/p_i = \alpha(f - 0.1/d) \cdot O(\text{OUT}/p)$ result pairs.

We carry out the above processing on all the dimensions simultaneously. This does not affect the number of rounds, but increases the load by d times. Each machine now produces at most $d \cdot \alpha(f - 0.1/d) \cdot O(\text{OUT}/p)$ result pairs.

Summarizing the discussion on Categories 1 and 2 gives the following recurrences:

$$\begin{aligned}
\alpha(f) &= d \cdot \alpha(f - 0.1/d) \\
\rho(f) &= O(d) + \rho(f - 0.1/d) \\
\lambda(f) &= d \cdot \lambda(f - 0.1/d) + d^{O(d)} \cdot N/p + O(d \cdot \text{OUT}/p).
\end{aligned}$$

To analyze the performance of our algorithm on the initial DJ problem, we need to solve the above recurrences to obtain $\rho(1), \lambda(1)$, and $\alpha(1)$. It is straightforward to verify that $\alpha(f) = d^{O(d)}$ and $\rho(1) = O(d^2)$. To compute $\lambda(f)$, recursively break it into d copies of $\lambda(f - 0.1/d)$ until $f \leq 0.1/d$. The recursion tree triggered by $\lambda(1)$ has a depth $O(d)$ with $d^{O(d)}$ nodes in total. Each leaf contributes $O(d^2 \cdot N/p + d \cdot \text{OUT}/p)$ to $\lambda(f)$, while each internal node contributes $d^{O(d)}N/p + O(d \cdot \text{OUT}/p)$. Hence:

$$\lambda(1) = d^{O(d)} \left(d^{O(d)} \cdot N/p + d \cdot \text{OUT}/p \right) = d^{O(d)} \cdot (N + \text{OUT})/p.$$

This establishes the claim in Theorem 4.

B Proof of Lemma 11

Consider an arbitrary non-matching pair $(r, s) \in R \times S \setminus T_{(R,S)}$. Since $q_{(r,s)}$ has label 0, $h_{q_{(r,s)}} = \{(\alpha_1, \alpha_2, ..., \alpha_d, \beta) \in \mathbb{R}^{d+1} \mid \sum_{i=1}^{d} \alpha_i \cdot q_{(r,s)}[i] < \beta\}$. By definition of $q_{(r,s)}$, (1) can be written as $\sum_{i=1}^{d} \alpha_i^* \cdot q_{(r,s)}[i] \geq \beta^*$, which indicates that $h_{q_{(r,s)}}$ does not contain ϕ_i. Therefore, (r, s) satisfies (1) if and only if $q_{(r,s)} \in Y^0$. The correctness of the lemma then follows.